Refinement Types

- Extend verification for functions (contracts, pre/post) to a functional language, where functions can be partially applied and passed as arguments

```
add :: Int -> Int -> Int        add 0 :: Int -> Int
```

- Idea: Base types like Int and Bool get an extra logical predicate called refinement.
- The refinement restricts the values inhabitants of a type may take.
- Example: `type nat ≜ int{v : v ≥ 0}`

  restricts the base-type int to non-negative numbers.
- Example:

```
size :: x : array(α) -> nat{v : v = length(x)}
```

(Refinement refers to argument)

```
get :: x : array(α) -> nat{v < length(x)} -> α
```

(α: polymorphism)

  - size returns the length of the array
  - get take array and in-bounds index and returns an element.

- refinements may refer to arguments

⇒ called **dependent types**

- the functions are **polymorphic** in the type α of array elements.

Running Example:

$$(\lambda x.\ \text{add}\ x\ 1) :: (x : \text{nat} \to \text{int}\{v \mid x < v\}) \quad v > 0$$

- Increment input x
- We'll check that the return value is larger than the input.

- Our language is the **lambda calculus**.

Terms:

$$\begin{aligned} e ::= \quad & c && \text{-- constant} \\ \mid\ & x && \text{-- variable} \\ \mid\ & \text{let } x = e \text{ in } e && \text{-- let binding} \\ \mid\ & \lambda x.e && \text{-- function} \end{aligned}$$

$(\lambda x.e_1)\ e_2 \rightsquigarrow$

$e_1\langle e_2/x\rangle$

| e x -- application

| e : t -- type annotation

Types

- Next, we'll define a language of types.

Base $b ::= int$ -- integers

Refinements $r ::= v : p$ -- v refined with p (p: formula in first-order theory)

Types $t, s ::= b\{r\}$ -- refined base type

$| \; x : t \to t$ -- dependent function type

Typing Environment $\Gamma ::= \emptyset$ -- empty

$| \; \Gamma, x : t$ -- binding

Entailment:

- We define a judgement $\Gamma \vdash c$ saying that under typing environment Γ,

logical predicate c is valid.

Example:

(1) $x : int \{0 \leq x\} \vdash \underbrace{\forall y : int.\ y = x + 1 \Rightarrow 0 < y}_{c}$

This judgement says that under the typing environment where x is non-negative, constraint c is valid.

Rules Entailment: $\Gamma \vdash c$

$$\frac{SMTValid(c)}{\emptyset \vdash c}\ \text{ENT-EMP}$$

(SMT solver says c is valid)

$$\frac{\Gamma \vdash \forall x : b.\ p \Rightarrow c}{\Gamma ; x : b\{x : p\} \vdash c}\ \text{ENT-EXT}$$

- Rule ENT-EMP uses an SMT solver to check validity, if the environment is empty.

- Rule ENT-EXT extends the constraint using a binding from the environment.

Example:

- We can check (1) by applying rule ENT-EX which yields

$\emptyset \vdash \forall x : int.\ 0 \leq x \Rightarrow \forall y : int.\ y = x + 1 \Rightarrow y > 0$

- Applying rule ENT-EMP, we can query an SMT solver, which returns that the formula is valid.

Subtyping:

- Next, we define a judgement $\Gamma \vdash t_1 <: t_2$ (subtype) saying that t_1 is a subtype of t_2, under Γ.

- Subtyping answers the question:
  If, we expect type t_2, is it okay to use type t_1 instead?

Example:

(2) $x : int\{0 \le x\} \vdash int\{y : y = x+1\} <: int\{v : v > 0\}$

Rules:

$\subseteq$

$\Gamma \vdash t_1 <: t_2$

$$\frac{\Gamma \vdash \forall v_1 : b.\ p_1 \Rightarrow p_2[v_1/v_2]}{\Gamma \vdash b\{v_1 : p_1\} <: b\{v_2 : p_2\}}$$ SUB-Base

$s_1 <: s_2$ — not a typo

$$\frac{\Gamma \vdash s_2 <: s_1 \qquad \Gamma; x_2 : s_2 \vdash t_1[x_2/x_1] <: t_2}{\Gamma \vdash x_1 : s_1 \to t_1 <: x_2 : s_2 \to t_2}$$ SUB-FUN

- The requirement on input types in SUB-FUN says its okay to use a function $f_1 : s_1 \to t_1$ instead of a function $f_2 : s_2 \to t_2$, if s_1 accepts more arguments than f_2.

- If we use a function that accepts fewer arguments, it may happen that we call it with an argument it cannot handle.

Example:

- Applying rule SUB-Base to (2) gives us (1) whose validity we derived before.

Example:

Applying rule SUB-FUN to

$\emptyset \vdash x : int \to int\{y : y = x+1\} <: x:\{0 \le x\} \to int\{v : 0 < v\}$

gives us two constraints

a) $\emptyset \vdash x : int\{0 \le x\} <: x : int$ and

b) $x : int\{0 \le x\} \vdash int\{y : y = x+1\} <: int\{v : 0 < v\}$

Judgement a) reduces to the constraint

$\forall x.\, int\ x \ge 0 \Rightarrow \top$, which is trivially valid.

We derived judgement b) in (2).

Type checking:

- We define two judgements for type-checking
  - $\Gamma \vdash e \Rightarrow t$ (input: e; output: t), which says we can synthesize type t for e, under environment Γ.
  - $\Gamma \vdash e \Leftarrow t$ (input: e; input: t), which says we can check e to have type t.
- Called bidirectional type checking.

Synthesis:

- Basic functions and constants have an associated type $prim(c)$.
- $prim(0) \triangleq int\{v : v = 0\}$
- $prim(1) \triangleq int\{v : v = 1\}$
- $prim(add) \triangleq x : int \to y : int \to int\{v : v = x + y\}$

Rules:

$\Gamma \vdash e \Rightarrow t$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x \Rightarrow t}\ \text{SYN VAR} \qquad \frac{prim(c) = t}{\Gamma \vdash c \Rightarrow t}\ \text{SYN CON}$$

$$\frac{\Gamma \vdash e \Leftarrow t}{\Gamma \vdash e : t \Rightarrow t}\ \text{SYN-ANN} \qquad \frac{\Gamma \vdash e \Rightarrow x : s \to t \quad \Gamma \vdash y \Leftarrow s}{\Gamma \vdash e\ y \Rightarrow t[y/x]}\ \text{SYN-APP}$$

- SYN-VAR and SYN-CON synthesize types from the environment and from primitive types.
- SYN-ANN synthesizes types for annotated terms, after checking the annotation.
- SYN-APP synthesizes the type of an application, if the type of the function can be inferred.

Example:

Let $\Gamma_0 \triangleq x : nat ; one : \{ one = 1 \}$ then

$\Gamma_0 \vdash one \Rightarrow \{ v \mid v = 1 \}$

$\Gamma_0 \vdash add \Rightarrow x : int \to y : int \to \{ v : v = x + y \}$

using SYN-VAR and SYN-CON.

We can show the judgement

$\Gamma_0 \vdash add\ x \Rightarrow y : int \to int \{ v : v = x + y \}$

using rule SYN-APP, if we can show

$\Gamma \vdash x \Leftarrow int$.

Similarly, we can show the judgement

(3) $\Gamma_0 \vdash (add\ x)\ one \Rightarrow int \{ v : v = x + one \}$

by using rule SYN-APP ; if we can show $\Gamma_0 \vdash one \Leftarrow int$.

Checking: $\Gamma \vdash e \Leftarrow t$

$$\frac{\Gamma \vdash e \Rightarrow s \quad \Gamma \vdash s <: t}{\Gamma \vdash e \Leftarrow t}\ \text{CHK SYN}$$

$$\frac{\Gamma, x:t_1 \vdash e \Leftarrow t_2}{\Gamma \vdash \lambda x.e \Leftarrow x:t_1 \to t_2}\ \text{CHK-LAM}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow t_1 \quad \Gamma, x:t_1 \vdash e_2 \Leftarrow t_2}{\Gamma \vdash let\ x = e_1\ in\ e_2 \Leftarrow t_2}\ \text{CHK-LET}$$

- Rule CHK-SYN lets us check e against type t, if we can synthesize a type s for e, such that s is a subtype

of t — and therefore we can use it in a place where we expect type t.

- Rule CHK-LAM checks a lambda expression by extending the environment with the argument and checking the body.
  - Let synthesizes a type for the bound expression, and checks the body after extending the environment.

Example:

To check $\Gamma_0 \vdash x \Leftarrow \text{int}$

we need to check $\Gamma_0 \vdash x : \{x \mid x \geq 0\} <: \text{int}$ which holds trivially, and we can check $\Gamma_0 \vdash \text{one} \Leftarrow \text{int}$ in the same way.

Example: To check

$\emptyset \vdash (\lambda x.\ add\ x\ 1) \Leftarrow (x:nat \to int\{v \mid 0 < v\})$

we need to check

(4) $x:nat \vdash add\ x\ 1 \Leftarrow int\{v : 0 < v\}$

using rule CHK-SUB, and our derivation in (3), we need to show

$x:nat, x:int\{v : v = x+1\}, 2: int\{v : 0 < v\}$

We showed this in (2) and therefore our proof is complete.

Try it out !

https://liquid.kosmikus.org/01-intro.html