

Refinement Types

- Extend verification for functions (contracts, pre/post) to a functional language, where functions can be partially applied and passed as arguments
- Idea: Base types like Int and Bool get an extra logical predicate called refinement.
- The refinement restricts the values in habitats of a type may take.
- Example: $\text{type nat} \triangleq \text{int} \setminus \{v : v \geq 0\}$

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

- Example:
 - size: $x : \text{array}(\alpha) \rightarrow \text{nat} \setminus \{v : v = \text{length}(x)\}$
 - get: $x : \text{array}(\alpha) \rightarrow \text{nat} \setminus \{v < \text{length}(x)\} \rightarrow \alpha$
- Refinement refers
to argument.
- \uparrow
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 of array elements.
-

Running Example:

$(\lambda x. \text{add } x 1) :: [x : \text{nat} \rightarrow \text{int} \{ v \mid x < v \}]$

- increments input x
 - We'll check that the return value is larger than the input.
 - Our language is the lambda calculus.
-

Terms:

$e ::= c$ -- constant
| x -- variable
| let $x = e$ in e -- let binding
| $\lambda x. e$ -- function

| e.x -- application

| e:t -- type annotation

Types

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

base

b::= int -- integers

formula in first-order
theory

refinements

r::= v:p -- v refined with p

Types

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

| x:t → t -- dependent function type

Typing Environment

Γ ::= φ -- empty

| Γ,x:t -- binding

Entailment:

- o We define a judgement $\Gamma \vdash C$

saying that under typing environment Γ ,

Logical predicate c is valid:

Example:

(1) $x:\text{int} \vdash \sum 0 \leq x \vdash \forall y:\text{int}. y = x + 1 \Rightarrow 0 \leq y \quad c$

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

Rules Entailment:

$$\boxed{\Delta \vdash c}$$

$\frac{\text{SMT solver says } c \text{ is valid}}{\text{SMT Valid } (c)}$ ENT-EMP

$\frac{\Gamma \vdash \#x:b. \rho \Rightarrow c}{\Gamma; x:b \models \rho \vdash c}$ 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-EXT which yields
$$\emptyset \vdash \forall x : \text{int}. \ 0 \leq x \Rightarrow \forall y : \text{int}. \ y = x + 1 \Rightarrow y > 0$$
- Applying rule ENT-EMP, we can query an SMT solver, which returns that the formula is valid.

Subtyping:

- Next, we define a judgement $\Gamma \vdash t_1 \leq t_2$ 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) $x : \text{int} \{ 0 \leq x \} \vdash \text{int} \{ y : y = x + 1 \} \leq \text{int} \{ v : v > 0 \}$

Rules:

$$\boxed{\Gamma \vdash t_1 \leq t_2}$$

$$\frac{\Gamma \vdash t : v_1 : b. \rho_1 \Rightarrow \rho_2[v_1/v_2]}{\Gamma \vdash b \{ v_1 : \rho_1 \} \leq b \{ v_2 : \rho_2 \}}$$

SUB-
base

$$\frac{\Gamma \vdash s_2 \leq s_1 \quad \begin{array}{c} \text{not a typo} \\ \Gamma ; x_2 : s_2 \vdash t_1[x_2/s_2] \leq t_2 \end{array}}{\Gamma \vdash x_1 : s_1 \rightarrow t_1 \leq x_2 : s_2 \rightarrow t_2}$$

SUB-
FUN

- The requirement on input types in **SUB-FUN** says it's okay to use a function $f_1 : s_1 \rightarrow t_1$ instead of a function $t_2 : s_2 \rightarrow t_2$, 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

$$\emptyset \vdash x : \text{int} \rightarrow \text{int} \{ y : y = x + 1 \} \Leftarrow \{ x \geq 0 \leq x \} \rightarrow \text{int} \{ v : 0 < v \}$$

gives us two constraints

$$a) \emptyset \vdash x : \text{int} \{ 0 \leq x \} \Leftarrow x : \text{int} \quad \text{and}$$

$$b) x : \text{int} \{ 0 \leq x \} \vdash \text{int} \{ y : y \geq x + 1 \} \Leftarrow \text{int} \{ v : 0 < v \}$$

Judgement a) reduces to the constraint

$$\forall x : \text{int} \quad x \geq 0 \Rightarrow T, \text{ which is trivially valid.}$$

We derived judgement b) in (2).

Type Checking:

- We define two judgements for type-checking

$\Gamma \vdash e \Rightarrow t$, which says we can
synthesize type t for e , under environment Γ .

$\Gamma \vdash e : t$, which says we can
check e to have type t .

- Called bidirectional type checking.

Synthesis:

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

Rules :

$\Gamma \vdash e \Rightarrow t$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x \Rightarrow t} \quad \boxed{\begin{matrix} \text{SYN} \\ \text{VAR} \end{matrix}}$$

$$\frac{\text{prim}(c) = t}{\Gamma \vdash c \Rightarrow t} \quad \boxed{\begin{matrix} \text{SYN} \\ \text{CON} \end{matrix}}$$

$$\frac{\Gamma \vdash e \in t}{\Gamma \vdash e : t \Rightarrow t} \quad \boxed{\begin{matrix} \text{SYN-} \\ \text{ANN} \end{matrix}}$$

$$\frac{\Gamma \vdash e \Rightarrow x : s \rightarrow t \quad \Gamma \vdash e \in s}{\Gamma \vdash e y \Rightarrow t [y/x]} \quad \boxed{\begin{matrix} \text{SYN-} \\ \text{APP} \end{matrix}}$$

- **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 $\Gamma_0 \hat{=} x:\text{nat}; \text{one}:\{\text{one}=1\}$ then

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

$$\Gamma_0 \vdash \text{add} \Rightarrow x:\text{int} \rightarrow y:\text{int} \rightarrow \Sigma v: v=x+y$$

using SUN-VAR and SUN-CON.

We can show the judgement

$$\Gamma_0 \vdash \text{add } x \Rightarrow \text{int} \rightarrow \text{int} \Sigma v: v=x+1$$

using rule SUN-APP, if we can show

$$\Gamma \vdash x \in \text{int},$$

Similarly, we can show the judgement

(3) $\Gamma_0 \vdash (\text{add } x) \text{ one} \Rightarrow \text{int} \Sigma v: v=x+1$

by using rule SYN-APP; if we

can show $\Gamma_0 \vdash \text{One} \Leftarrow \text{int}$.

Checking:

$\Gamma \vdash e \Leftarrow t$

$$\frac{\Gamma \vdash e \Rightarrow s \quad \Gamma \vdash s \leq t}{\Gamma \vdash e \Leftarrow t}$$

CHK
SYM

$$\frac{\Gamma ; x : t_1 \vdash e \Leftarrow t_2}{\Gamma \vdash \lambda x. e \Leftarrow x : t_1 \rightarrow t_2}$$

CHK-LAM

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

CHK-LET

• Rule CHK-SUM lets us check e

against type t , if we can synthesize a types for e , such that s is a subtype

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

- Rule CHX-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 $\Gamma_0 \vdash x : \text{int}$

we need to check $\Gamma_0 \vdash x : \{x \geq 0\} \vdash \text{int} \wedge \top$
which holds trivially, and we can check

$\Gamma_0 \vdash \text{one} : \text{int}$ in the same way.

Example: To check

$$\emptyset \vdash (\lambda x. \text{add } x \cdot 1) \in (\forall x: \text{nat} \rightarrow \text{int} \Sigma v \mid x < v)$$

we need to check

(4) $x: \text{nat} \vdash \text{add } x \cdot 1 \in \text{int} \Sigma v: x < v$

using rule CH X-SUM, and our

derivation in (3), we need to show

$$x: \text{nat} \vdash \text{int} \Sigma v: v = x + 1 \vdash \text{int} \Sigma v: x < v$$

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