Lecture 23 – Refinement Types

AAA551: Programming Language Theory

Jihyeok Park

PLRG

2026 Spring

Recall

- Dependent Types
  - Barendregt's λ-Cube: a **type that depends on a term**
  - Dependent Function Types (Π) and Pair Types (Σ)
  - The Calculus of Constructions (λC) and CIC
  - Curry-Howard, Reloaded: Π as $\forall$, Σ as $\exists$
  - Propositional Equality ($a =_A b$)
  - Dependent Types in Practice (Coq/Rocq, Agda, Idris, Lean, F*)
- Power vs. automation: type checking is **undecidable**
- Idea: restrict propositions to a **decidable** logic

Contents

Contents

From Proofs to Solvers

Recall the **dependent** type from Lecture 22 – a number **bundled with a proof** of a proposition about it:

$$\texttt{Even} \triangleq \Sigma(n : \texttt{Nat}).\ (n \bmod 2 =_{\texttt{Nat}} 0)$$

From Proofs to Solvers

Recall the **dependent** type from Lecture 22 – a number **bundled with a proof** of a proposition about it:

$$\texttt{Even} \triangleq \Sigma(n : \texttt{Nat}).\ (n \bmod 2 =_{\texttt{Nat}} 0)$$

Dependent types let that proposition be **arbitrary**, but the price is steep:

- the programmer must **write the proof term** by hand, and
- term equality is **undecidable**, so checking needs a termination checker.

From Proofs to Solvers

Recall the **dependent** type from Lecture 22 – a number **bundled with a proof** of a proposition about it:

$$\texttt{Even} \triangleq \Sigma(n : \texttt{Nat}).\ (n \bmod 2 =_{\texttt{Nat}} 0)$$

Dependent types let that proposition be **arbitrary**, but the price is steep:

- the programmer must **write the proof term** by hand, and
- term equality is **undecidable**, so checking needs a termination checker.

Refinement types keep the idea – **a type bundled with a proposition** – but restrict the proposition to a **decidable logic**, so that an **SMT solver** discharges every proof obligation **automatically**.

The Expressiveness–Automation Spectrum

Every type system trades **expressiveness** against **automation**:

system	**expressiveness**	**checking**
simple types (STLC)	low	decidable, fully automatic
refinement types	medium	decidable, SMT-automated
dependent types	high	undecidable, manual proofs

The Expressiveness–Automation Spectrum

PLRG

Every type system trades **expressiveness** against **automation**:

system	**expressiveness**	**checking**
simple types (STLC)	low	decidable, fully automatic
refinement types	medium	decidable, SMT-automated
dependent types	high	undecidable, manual proofs

Refinement types sit in the **sweet spot**: far more precise than simple types – they can express "a positive integer", "a non-empty list", "an in-bounds index" – yet type checking stays **fully automatic**.

The Expressiveness–Automation Spectrum

Every type system trades **expressiveness** against **automation**:

system	**expressiveness**	**checking**
simple types (STLC)	low	decidable, fully automatic
refinement types	medium	decidable, SMT-automated
dependent types	high	undecidable, manual proofs

Refinement types sit in the **sweet spot**: far more precise than simple types – they can express "a positive integer", "a non-empty list", "an in-bounds index" – yet type checking stays **fully automatic**.

The bargain: give up arbitrary propositions, gain a **push-button** checker.

Contents

Refinement Types

A **refinement type** refines a base type B with a **predicate** p – a boolean-valued expression that may mention the bound variable x:

$$\{\, x : B \mid p \,\}$$

read as "the values x of base type B **such that** p holds".

Refinement Types

A **refinement type** refines a base type B with a **predicate** p – a boolean-valued expression that may mention the bound variable x:

$$\{\, x : B \mid p \,\}$$

read as "the values x of base type B **such that** p holds".

Its inhabitants are exactly the values whose substitution makes p true:

$$[\![\{\, x : B \mid p \,\}]\!] \;=\; \{\, v \in [\![B]\!] \mid p[x \mapsto v] = \mathtt{true} \,\}$$

Refinement Types

A **refinement type** refines a base type B with a **predicate** p – a boolean-valued expression that may mention the bound variable x:

$$\{\, x : B \mid p \,\}$$

read as “the values x of base type B **such that** p holds”.

Its inhabitants are exactly the values whose substitution makes p true:

$$[\![\{\, x : B \mid p \,\}]\!] \;=\; \{\, v \in [\![B]\!] \mid p[x \mapsto v] = \texttt{true} \,\}$$

A plain base type is the **trivial** refinement, and $\bot$ is the **empty** one:

$$B \;\triangleq\; \{\, x : B \mid \texttt{true} \,\} \qquad\qquad \bot \;\triangleq\; \{\, x : B \mid \texttt{false} \,\}$$

Examples of Refinement Types

$\texttt{Nat}$	$\triangleq$	$\{\, x : \texttt{num} \mid x \geq 0 \,\}$	the non-negative integers
$\texttt{Pos}$	$\triangleq$	$\{\, x : \texttt{num} \mid x > 0 \,\}$	the positive integers
$\texttt{Even}$	$\triangleq$	$\{\, x : \texttt{num} \mid x \bmod 2 = 0 \,\}$	the even integers
$\texttt{Rng}\, n$	$\triangleq$	$\{\, x : \texttt{num} \mid 0 \leq x \land x < n \,\}$	an in-bounds index
$\{\, x : \texttt{num} \mid x = 42 \,\}$			a singleton type

Examples of Refinement Types

$$
\begin{array}{rcll}
\texttt{Nat} & \triangleq & \{\, x : \texttt{num} \mid x \geq 0 \,\} & \text{the non-negative integers} \\
\texttt{Pos} & \triangleq & \{\, x : \texttt{num} \mid x > 0 \,\} & \text{the positive integers} \\
\texttt{Even} & \triangleq & \{\, x : \texttt{num} \mid x \bmod 2 = 0 \,\} & \text{the even integers} \\
\texttt{Rng}\, n & \triangleq & \{\, x : \texttt{num} \mid 0 \leq x \wedge x < n \,\} & \text{an in-bounds index} \\
\{\, x : \texttt{num} \mid x = 42 \,\} & & & \text{a singleton type}
\end{array}
$$

In **LiquidHaskell**, such types are written as ordinary signatures whose base types are decorated with predicates (here shown without the surrounding annotation syntax):

```
type Nat   = {v:Int | v >= 0}
type Pos   = {v:Int | v >  0}
type Rng N = {v:Int | 0 <= v && v < N}
```

Contents

Subtyping is Logical Implication

Recall the **subsumption** rule (Lecture 18): if $e : \tau$ and $\tau <: \tau'$, then $e : \tau'$.

Subtyping is Logical Implication

Recall the **subsumption** rule (Lecture 18): if $e : \tau$ and $\tau <: \tau'$, then $e : \tau'$.

For refinements, $\{x : B \mid p\}$ is a subtype of $\{x : B \mid q\}$ exactly when **every value satisfying** p **also satisfies** q:

$$\frac{\models \forall x.\ (p \implies q)}{\{\, x : B \mid p \,\} <: \{\, x : B \mid q \,\}}\ \textsc{Sub-Base}$$

Subtyping is Logical Implication

Recall the **subsumption** rule (Lecture 18): if $e : \tau$ and $\tau <: \tau'$, then $e : \tau'$.

For refinements, $\{x : B \mid p\}$ is a subtype of $\{x : B \mid q\}$ exactly when **every value satisfying** p **also satisfies** q:

$$\frac{\models \forall x.\ (p \implies q)}{\{\, x : B \mid p \,\} <: \{\, x : B \mid q \,\}}\ \textsc{Sub-Base}$$

So subtyping reduces to checking the **validity of an implication** – a job for an **SMT solver**. For instance,

$$\texttt{Pos} <: \texttt{Nat} \quad \text{because} \quad \models \forall x.\ (x > 0 \implies x \geq 0).$$

Verification Conditions and SMT

The implications collected during type checking are called **verification conditions** (VCs). They live in **decidable theories**:

- linear arithmetic over integers and reals,
- equality with uninterpreted functions (EUF),
- arrays, bit-vectors, . . .

Verification Conditions and SMT

The implications collected during type checking are called **verification conditions** (VCs). They live in **decidable theories**:

- linear arithmetic over integers and reals,
- equality with uninterpreted functions (EUF),
- arrays, bit-vectors, ...

An **SMT solver** (Z3, cvc5, ...) decides each VC. To check $\{x \mid x = 2 + 3\} <: \{x \mid x > 4\}$, it asks whether the **negation** of the VC is satisfiable:

```
(declare-const x Int)
(assert (not (=> (= x (+ 2 3)) (> x 4))))
(check-sat)   ; unsat  =>  the implication is valid
```

Verification Conditions and SMT

The implications collected during type checking are called **verification conditions** (VCs). They live in **decidable theories**:

- linear arithmetic over integers and reals,
- equality with uninterpreted functions (EUF),
- arrays, bit-vectors, ...

An **SMT solver** (Z3, cvc5, ...) decides each VC. To check $\{x \mid x = 2 + 3\} <: \{x \mid x > 4\}$, it asks whether the **negation** of the VC is satisfiable:

```
(declare-const x Int)
(assert (not (=> (= x (+ 2 3)) (> x 4))))
(check-sat)   ; unsat  =>  the implication is valid
```

An `unsat` result means **no counterexample exists**, so the VC is **valid** and the subtyping holds.

Contents

Refining Function Types

Just like Π-types, a function argument can be **named** so the result type refers to it:

$$(x : \{\, y : B \mid p \,\}) \to \{\, z : B' \mid q \,\} \qquad \text{where } q \text{ may mention } x.$$

Refining Function Types

PLRG

Just like Π-types, a function argument can be **named** so the result type refers to it:

$$(x : \{\, y : B \mid p \,\}) \to \{\, z : B' \mid q \,\} \qquad \text{where } q \text{ may mention } x.$$

Such refinements act as **pre**- and **postconditions** on functions:

```
abs :: Int -> {v:Int | v >= 0}  -- return value is non-negative
div :: Int -> {v:Int | v /= 0} -> Int  -- divisor is non-zero
(!) :: a:[b] -> {i:Int | 0 <= i && i < len a} -> b
```

Refining Function Types

Just like Π-types, a function argument can be **named** so the result type refers to it:

$$(x : \{\, y : B \mid p \,\}) \to \{\, z : B' \mid q \,\} \qquad \text{where } q \text{ may mention } x.$$

Such refinements act as **pre-** and **postconditions** on functions:

```
abs :: Int -> {v:Int | v >= 0}  -- return value is non-negative
div :: Int -> {v:Int | v /= 0} -> Int  -- divisor is non-zero
(!) :: a:[b] -> {i:Int | 0 <= i && i < len a} -> b
```

- `div`'s precondition rules out **division by zero**;
- `(!)`'s precondition rules out **out-of-bounds** indexing.

Each is checked by the solver at **every call site**.

Typing with a Logical Context

The type environment Γ now carries **logical facts**, not just bindings. Subtyping flows them into the VC:

$$\frac{\models\ [\![\Gamma]\!] \land p \implies q}{\Gamma \vdash \{\, x : B \mid p \,\} <: \{\, x : B \mid q \,\}}\ \textsc{Sub}$$

where $[\![\Gamma]\!]$ is the **conjunction of the refinement predicates** of all variables in scope.

Typing with a Logical Context

The type environment Γ now carries **logical facts**, not just bindings. Subtyping flows them into the VC:

$$\frac{\models\ [\![\Gamma]\!] \land p \implies q}{\Gamma \vdash \{\, x : B \mid p \,\} <: \{\, x : B \mid q \,\}}\ \textsc{Sub}$$

where $[\![\Gamma]\!]$ is the **conjunction of the refinement predicates** of all variables in scope.

A conditional **strengthens** the context with the branch condition – this is **path-sensitivity**:

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma\,,\, e_1 \vdash e_2 : \tau \qquad \Gamma\,,\, \neg e_1 \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}\ \textsc{If}$$

Inside the THEN-branch we may **assume** e_1 holds; inside the ELSE-branch, that it does not.

Example: Checking abs

We claim abs always returns a Nat, i.e. a value ≥ 0:

```
abs :: Int -> Nat   -- Nat = {v:Int | v >= 0}
abs x = if x >= 0 then x else 0 - x
```

Example: Checking abs

We claim abs always returns a Nat, i.e. a value ≥ 0:

```
abs :: Int -> Nat    -- Nat = {v:Int | v >= 0}
abs x = if x >= 0 then x else 0 - x
```

Two branches give **two verification conditions**, each demanding the result satisfy $v \geq 0$:

branch	**context**	**VC**
then : x	$x \geq 0$	$x \geq 0 \implies x \geq 0$ ✓
else : $0 - x$	$\neg(x \geq 0)$	$\neg(x \geq 0) \implies (0 - x) \geq 0$ ✓

Example: Checking abs

We claim abs always returns a Nat, i.e. a value ≥ 0:

```
abs :: Int -> Nat    -- Nat = {v:Int | v >= 0}
abs x = if x >= 0 then x else 0 - x
```

Two branches give **two verification conditions**, each demanding the result satisfy $v \geq 0$:

branch	context	VC
then : x	$x \geq 0$	$x \geq 0 \implies x \geq 0$ ✓
else : $0 - x$	$\neg(x \geq 0)$	$\neg(x \geq 0) \implies (0 - x) \geq 0$ ✓

Both VCs are **valid** in linear arithmetic, so abs type-checks – with **no manual proof**. Drop the else guard and the second VC becomes true $\implies (0 - x) \geq 0$, which the solver **refutes** with the counterexample $x = 1$.

Contents

Liquid Types: Inferring Refinements

Annotating **every** type by hand is tedious. **Liquid types** (Logically Qualified Data types) **infer** the refinements automatically:

1. Fix a finite set of **qualifiers** – predicate templates such as $\{0 \leq \star,\ \ \star < \star,\ \ \star = \star\}$.
2. Attach an **unknown** refinement variable κ to each type to be inferred.
3. Type checking emits **Horn constraints** (implications) over the κ's.
4. Solve them by **predicate abstraction** and **least-fixed-point** iteration: pick the strongest conjunction of qualifiers satisfying every constraint.

Liquid Types: Inferring Refinements

PLRG

Annotating **every** type by hand is tedious. **Liquid types** (Logically Qualified Data types) **infer** the refinements automatically:

1. Fix a finite set of **qualifiers** – predicate templates such as $\{\, 0 \leq \star, \;\; \star < \star, \;\; \star = \star \,\}$.
2. Attach an **unknown** refinement variable κ to each type to be inferred.
3. Type checking emits **Horn constraints** (implications) over the κ's.
4. Solve them by **predicate abstraction** and **least-fixed-point** iteration: pick the strongest conjunction of qualifiers satisfying every constraint.

The refinements are as expressive as the qualifiers allow, yet found **automatically** – the very **fixed-point** machinery we used for static analysis, now over a logical lattice.

Liquid Types: Inferring abs

Take `abs` again, but **drop** the result annotation: put an **unknown** refinement κ in its place, and fix a qualifier set Q.

$$\texttt{abs} : \texttt{num} \to \{\, v : \texttt{num} \mid \kappa \,\} \qquad Q = \{\, 0 \le v,\;\; v \le 0,\;\; v = 0 \,\}$$

Liquid Types: Inferring abs

Take `abs` again, but **drop** the result annotation: put an **unknown** refinement κ in its place, and fix a qualifier set Q.

$$\texttt{abs} : \texttt{num} \to \{\, v : \texttt{num} \mid \kappa \,\} \qquad\qquad Q = \{\, 0 \le v,\ \ v \le 0,\ \ v = 0 \,\}$$

Checking the body $\texttt{if}\ x \ge 0\ \texttt{then}\ x\ \texttt{else}\ 0 - x$ emits two **Horn constraints** on κ – one per branch, with its path condition:

$$\begin{aligned} \text{then } (x) : &\quad x \ge 0 \ \wedge\ v = x \quad\Longrightarrow\quad \kappa \\ \text{else } (0 - x) : &\quad x < 0 \ \wedge\ v = 0 - x \quad\Longrightarrow\quad \kappa \end{aligned}$$

Liquid Types: Inferring abs

Take `abs` again, but **drop** the result annotation: put an **unknown** refinement κ in its place, and fix a qualifier set Q.

$$\texttt{abs} : \texttt{num} \to \{\, v : \texttt{num} \mid \kappa \,\} \qquad\qquad Q = \{\, 0 \le v,\;\; v \le 0,\;\; v = 0 \,\}$$

Checking the body $\texttt{if}\ x \ge 0\ \texttt{then}\ x\ \texttt{else}\ 0 - x$ emits two **Horn constraints** on κ – one per branch, with its path condition:

$$\begin{aligned} \text{then } (x): &\quad x \ge 0 \;\wedge\; v = x \;\;\Longrightarrow\;\; \kappa \\ \text{else } (0 - x): &\quad x < 0 \;\wedge\; v = 0 - x \;\;\Longrightarrow\;\; \kappa \end{aligned}$$

Solve for the **strongest** conjunction of qualifiers in Q entailed by **both** constraints. Only $0 \le v$ survives ($v \le 0$ and $v = 0$ each fail a branch):

$$\kappa \;=\; (0 \le v) \qquad \Longrightarrow \qquad \texttt{abs} : \texttt{num} \to \{\, v : \texttt{num} \mid 0 \le v \,\} = \texttt{num} \to \texttt{Nat}$$

Liquid Types: Inferring abs

PLRG

Take `abs` again, but **drop** the result annotation: put an **unknown** refinement κ in its place, and fix a qualifier set Q.

$$\texttt{abs} : \texttt{num} \to \{\, v : \texttt{num} \mid \kappa \,\} \qquad\qquad Q = \{\, 0 \le v,\;\; v \le 0,\;\; v = 0 \,\}$$

Checking the body $\texttt{if}\ x \ge 0\ \texttt{then}\ x\ \texttt{else}\ 0 - x$ emits two **Horn constraints** on κ – one per branch, with its path condition:

$$\begin{aligned} \text{then } (x) : &\quad x \ge 0 \;\wedge\; v = x \quad\Longrightarrow\quad \kappa \\ \text{else } (0 - x) : &\quad x < 0 \;\wedge\; v = 0 - x \quad\Longrightarrow\quad \kappa \end{aligned}$$

Solve for the **strongest** conjunction of qualifiers in Q entailed by **both** constraints. Only $0 \le v$ survives ($v \le 0$ and $v = 0$ each fail a branch):

$$\kappa \;=\; (0 \le v) \qquad\Longrightarrow\qquad \texttt{abs} : \texttt{num} \to \{\, v : \texttt{num} \mid 0 \le v \,\} = \texttt{num} \to \texttt{Nat}$$

The annotation `Nat` from the previous section was **inferred** – not written.

Contents

Refinement Types in Practice

Many modern verifiers reduce types to **SMT queries**:

Tool	**Host**	**Notes**
LiquidHaskell	Haskell	liquid-type inference over Haskell
$F^\star$	ML-like	refinements **and** full dependent types
Flux	Rust	refinements layered on ownership
Stainless	Scala	contract verification
Dafny	imperative	SMT-backed pre/postconditions

Refinement Types in Practice

PLRG

Many modern verifiers reduce types to **SMT queries**:

Tool	**Host**	**Notes**
LiquidHaskell	Haskell	liquid-type inference over Haskell
$F^\star$	ML-like	refinements **and** full dependent types
Flux	Rust	refinements layered on ownership
Stainless	Scala	contract verification
Dafny	imperative	SMT-backed pre/postconditions

All share one engine: **elaborate types into verification conditions and hand them to a solver**. The solver is the trusted, automated proof-finder.

Refinement vs. Dependent Types

Two points on the same Curry-Howard landscape:

	Refinement Types	**Dependent Types**
proposition	decidable logic	arbitrary
proof	SMT, automatic	manual proof term
checking	decidable	undecidable
expressiveness	medium	maximal

Refinement vs. Dependent Types

Two points on the same Curry-Howard landscape:

	Refinement Types	**Dependent Types**
proposition	decidable logic	arbitrary
proof	SMT, automatic	manual proof term
checking	decidable	undecidable
expressiveness	medium	maximal

They are **complementary**, not rivals. **F*** combines both: it uses refinements for everything an SMT solver can settle, and falls back to full dependent-type proofs only where automation gives up.

Refinement vs. Dependent Types

Two points on the same Curry-Howard landscape:

	Refinement Types	**Dependent Types**
proposition	decidable logic	arbitrary
proof	SMT, automatic	manual proof term
checking	decidable	undecidable
expressiveness	medium	maximal

They are **complementary**, not rivals. **F*** combines both: it uses refinements for everything an SMT solver can settle, and falls back to full dependent-type proofs only where automation gives up.

Next, we follow the **path-sensitive** idea from the IF rule further, to types that **change as control flows**.

Next Lecture

- Flow-Sensitive Types

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr