Relatively Complete Refinement Type System for Verification of Higher-Order Non-deterministic Programs

Hiroshi Unno (University of Tsukuba)
Yuki Satake (University of Tsukuba)
Tachio Terauchi (Waseda University)
Background

• Recent advances in (semi-)automated methods for verifying higher-order functional programs
  • safety [Rondon+ '08; U. & Kobayashi '08,'09; Terauchi '10; Ong & Ramsay '11; Jhala+ '11; Kobayashi+ '11; U.+ '13; ...]
  • termination [Sereni & Jones '05; Giesl+ '11; Kuwahara+ '14; Vazou+ '14]
  • non-termination [Kuwahara+ '15; Hashimoto & U. '15]
  • temporal properties [Koskinen & Terauchi '14; Murase+ '16]

• Different techniques are used to verify the different classes of properties, and are hard to combine in a unified framework
  • dependent refinement types,
  • predicate abstraction for higher-order model checking,
  • program transformation for (binary) reachability analysis,...
Our Contributions

• Novel dependent refinement type system that can:
  • uniformly express and verify universal and existential branching properties of call-by-value, higher-order, and non-deterministic programs:
    • (cond.) safety, non-safety, termination, and non-termination
  • seamlessly combine universal and existential reasoning
    • e.g., Prove non-safety via termination
    • e.g., Prove non-termination via safety
    • e.g., Prove termination and non-termination simultaneously

• Meta-theoretic properties of the type system:
  • Closure of types under complement
  • Soundness
  • Relative completeness
Our Contributions

• Novel dependent refinement type system that can:
  • uniformly express and verify universal and existential branching properties of call-by-value, higher-order, and non-deterministic programs:
    • (cond.) safety, non-safety, termination, and non-termination
  • seamlessly combine universal and existential reasoning
    • e.g., Prove non-safety via termination
    • e.g., Prove non-termination via safety
    • e.g., Prove termination and non-termination simultaneously

• Meta-theoretic properties of the type system:
  • Closure of types under complement
  • Soundness
  • Relative completeness
Dependent Refinement Types $\tau$

- $\{x : \text{int} \mid x \geq 0\}$
  Non-negative integers

- $(x : \text{int}) \rightarrow \{r : \text{int} \mid r \geq x\}$
  Functions that take an integer $x$ and (if terminated) return $r$ not less than $x$

พอใจ A type system ensures that a type-checked expression behaves according to the type

🐱 Only universal branching properties can be expressed
Overview: Our Type System

• Extends dependent refinement types with:
  • Qualified types $\tau^{Q_1 Q_2}$
    to express universal/existential branching behaviors and partial/total correctness
  • Qualified bindings $x:Q\tau$
    to cope with non-determinism from program inputs
  • Gödel encoding of function-type values & guarded intersection types
    to achieve relative completeness
Overview: Our Type System

• Extends dependent refinement types with:
  • Qualified types $\tau^{Q_1Q_2}$ to express universal/existential branching behaviors and partial/total correctness
  • Qualified bindings $x:Q_\tau$ to cope with non-determinism from program inputs
  • Gödel encoding of function-type values & guarded intersection types to achieve relative completeness
Qualified Types $\tau^{Q_1 Q_2}$

- $Q_1 \in \{\forall, \exists\}$ (universal/existential non-det.) specifies whether the expression being typed behaves according to the type:
  - for any non-det. evaluation ($Q_1 = \forall$), or
  - for some non-det. evaluation ($Q_1 = \exists$)

- $Q_2 \in \{\forall, \exists\}$ (partial/total correctness) specifies whether:
  - $\tau$ holds for all value obtained ($Q_2 = \forall$), or
  - there exists a final value for which $\tau$ holds ($Q_2 = \exists$)
Qualified Types $\tau^{Q_1 Q_2}$

- $Q_1 \in \{\forall, \exists\}$ (universal/existential non-det.) specifies whether the expression being typed behaves according to the type:
  - for any non-det. evaluation ($Q_1 = \forall$), or
  - for some non-det. evaluation ($Q_1 = \exists$)

- $Q_2 \in \{\forall, \exists\}$ (partial/total correctness) specifies whether:
  - the evaluation diverges or $\tau$ is satisfied ($Q_2 = \forall$), or
  - the evaluation terminates and $\tau$ is satisfied ($Q_2 = \exists$)
Examples: Qualified Types $\tau_{Q_1Q_2}$

- $\vdash e : \{u : \text{int} \mid u > 0\}^{\forall\forall}$
  for any non-deterministic evaluation of $e$, if any integer $u$ is obtained, then $u$ is positive

- $\vdash e : \{u : \text{int} \mid u > 0\}^{\exists\exists}$
  for some non-deterministic evaluation of $e$, some integer $u$ is obtained, and $u$ is positive
Typing Integer Constants

Rule:

\[ \Delta \vdash n : \{x : \text{int} \mid x = n\}^{\forall \exists} \]

- Integer constant
- Type environment
- Universal and Total
Converting Qualified Types

Subtyping Rule:

\[ \Delta \vdash \tau_1 <: \tau_2 \quad Q_1 Q_1' \sqsubseteq Q_2 Q_2' \]

\[ \Delta \vdash \tau_1^{Q_1 Q_1'} <: \tau_2^{Q_2 Q_2'} \]

Universal and Partial

Existential and Total

Universal and Total

Existential and Partial
Rule:

\[
\Delta \vdash e_1 : \tau_1^{Q_1Q_2} \\
\Delta, x : \tau_1 \vdash e_2 : \tau_2^{Q_1Q_2} \quad x \notin fvs(\tau_2) \\
\Delta \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2^{Q_1Q_2}
\]
Typing Recursive Functions for Partial Correctness

Rule:

\[
\Delta, x : \tau_1, f : (x : \tau_1) \rightarrow \tau_2^Q \vdash e : \tau_2^Q
\]

\[
\Delta \vdash \text{rec}(f, x, e) : (x : \tau_1) \rightarrow \tau_2^Q
\]

(recursive) function

let rec f x = e
Typing Recursive Functions for Total Correctness (cf. [Xi '01])

Rule:
\[ \tau_{rec} = (x':\tau_1') \rightarrow \phi \triangleright (\tau_2')^Q^\exists =_{\alpha} (x:\tau_1) \rightarrow \tau_2^Q^\exists \]
\[ \Delta \models WF(\lambda(x,x').\phi) \quad \Delta, x:\tau_1, f:\tau_{rec} \vdash e:\tau_2^Q^\exists \]
\[ \Delta \vdash \text{rec}(f, x, e): (x:\tau_1) \rightarrow \tau_2^Q^\exists \]

Example:
\[ (x':\{x' \mid x' \geq 0\}) \rightarrow x > x' \geq 0 \triangleright \{y' \mid y' \geq x'\}^\forall^\exists \]
\[ x:\{x \mid x \geq 0\}, \text{sum}:\tau_{rec} \vdash \text{if } x = 0 \text{ then 0 else } \ldots : \text{int}^\forall^\exists \]
\[ \models WF(\lambda(x,x').x > x' \geq 0) \]
\[ \vdash \text{rec}(\text{sum}, x, \text{if } x = 0 \text{ then 0 else } x + \text{sum } (x - 1)) : \tau \]
\[ (x:\{x \mid x \geq 0\}) \rightarrow \text{int}^\forall^\exists \]
Overview: Our Type System

• Extends dependent refinement types with:
  • Qualified types $\tau^{Q_1Q_2}$
    to express universal/existential branching behaviors and partial/total correctness

• Qualified bindings $x:Q\tau$
  to cope with non-determinism from program inputs

• Gödel encoding of function-type values & guarded intersection types
  to achieve relative completeness
Qualified Bindings $x: Q\tau$

- Occur in type environments and the argument of dependent function types
- $Q \in \{\forall, \exists\}$ specifies whether a certain fact must hold for:
  - any input $x$ that satisfies $\tau$ ($Q = \forall$), or
  - some input $x$ that satisfies $\tau$ ($Q = \exists$)
Examples: Qualified Bindings $x:Q\tau$

- $(x:\forall\text{int}) \rightarrow \{u: \text{int} | u > x\}\forall\exists$
  functions that, for any integer $x$ and for any run with the argument $x$, return some integer $u$, which is greater than $x$

- $(x:\exists\text{int}) \rightarrow \{u: \text{int} | u > x\}\forall\exists$
  functions that, there exists an integer $x$, for any run with the argument $x$, return some integer $u$, which is greater than $x$
Skolemizing Existential Bindings

Rule:

\[
\Delta, x : \exists \tau \models \phi \\
\Delta, x : \forall \tau, \phi, \Gamma \vdash e : \sigma \\
\Delta, x : \exists \tau, \Gamma \vdash e : \sigma
\]

Example:

\[
x : \forall \text{int}, y : \exists \text{int} \models y = -x \\
x : \forall \text{int}, y : \forall \text{int}, y = -x \vdash x + y : \{z \mid z = 0\}^{\forall \exists} \\
x : \forall \text{int}, y : \exists \text{int} \vdash x + y : \{z \mid z = 0\}^{\forall \exists}
\]
Typing Non-deterministic Choice

Rule:

\[ \Delta, x: Q_1 \text{int} \vdash e : \tau^{Q_1Q_2} \quad x \notin f\text{vs}(\tau) \]
\[ \Delta \vdash \text{let } x = * \text{ in } e : \tau^{Q_1Q_2} \]

Example:

\[ x: \forall \text{int}, y: \exists \text{int} \vdash x + y : \{ z \mid z = 0 \}^{\exists\exists} \]
\[ x: \forall \text{int} \vdash \text{let } y = * \text{ in } x + y : \{ z \mid z = 0 \}^{\exists\exists} \]
Typing Function Applications (Universal Bindings)

Rule:

\[ \Delta \vdash v_1 : (x : \forall \tau) \rightarrow \sigma \quad \Delta \vdash v_2 : \tau \]

\[ \Delta \vdash v_1 \, v_2 : [v_2/x]\sigma \]
Typing Function Applications (Existential Bindings)

Rule:

\[
\Delta \vdash v_1 : (x : \exists \tau) \to \sigma \quad \Delta, x : \forall \tau, \Gamma \models x \sim v_2
\]

\[
\Delta, \Gamma \vdash v_1 v_2 : [v_2/x] \sigma
\]

Example:

\[
f : \forall \tau \vdash f : \tau \quad f : \forall \tau, x : \forall \text{int}, y : \exists \text{int} \vdash x = y
\]

\[
f : \forall \tau, y : \exists \text{int} \vdash f \ y : \{z \mid \bot\}^\exists
\]

\[
f : \forall \tau \vdash \text{let } y = \ast \text{ in } f \ y : \{z \mid \bot\}^\exists
\]

\[
(x : \exists \text{int}) \to \{z \mid \bot\}^\forall
\]
Converting Function Types (1/4)

**Subtyping Rule:**

\[
\begin{align*}
\Delta \vdash \tau_2 <: \tau_1 & \quad \Delta, x : \forall \tau_2 \vdash \sigma_1 <: \sigma_2 \\
\Delta \vdash (x: \forall \tau_1) \rightarrow \sigma_1 <: (x: \forall \tau_2) \rightarrow \sigma_2
\end{align*}
\]

**Example:**

\[
\begin{align*}
\vdash (x: \forall \text{int}) \rightarrow \{y \mid y = x\}^{\forall \exists} \\
<: (x: \forall \{x \mid x \geq 0\}) \rightarrow \{y \mid y \geq 0\}^{\forall \exists}
\end{align*}
\]
Converting Function Types (2/4)

Subtyping Rule:

\[
\Delta \vdash \tau_1 \ll : \tau_2 \quad \Delta, x : \forall \tau_1 \vdash \sigma_1 \ll : \sigma_2 \\
\Delta \vdash (x : \exists \tau_1) \rightarrow \sigma_1 \ll : (x : \exists \tau_2) \rightarrow \sigma_2
\]

Example:

\[
\vdash (x : \exists \{x | x \geq 0\}) \rightarrow \{y | y = x\}^{\forall \exists} \ll : (x : \exists \text{int}) \rightarrow \{y | y \geq 0\}^{\forall \exists}
\]
Converting Function Types (3/4)

Subtyping Rule:

\[
\Delta \vdash \tau <: \tau_1 \quad \Delta \vdash \tau <: \tau_2
\]
\[
\Delta, x : \exists \tau \vdash \sigma_1 <: \sigma_2
\]
\[
\Delta \vdash (x : \forall \tau_1) \rightarrow \sigma_1 <: (x : \exists \tau_2) \rightarrow \sigma_2
\]

Example:

\[
\vdash \{x \mid x \geq 0\} <: \text{int} \quad \vdash \{x \mid x \geq 0\} <: \text{int}
\]
\[
\vdash x : \exists \{x \mid x \geq 0\} \vdash \{y \mid y = x\} \forall \exists <: \{y \mid y = 0\} \forall \exists
\]
\[
\vdash (x : \forall \text{int}) \rightarrow \{y \mid y = x\} \forall \exists <: (x : \exists \text{int}) \rightarrow \{y \mid y = 0\} \forall \exists
\]
Converting Function Types (4/4)

**Subtyping Rule:**

\[ \Delta, x: \forall (\tau_1 \land \tau_2), y: \forall (\tau_1 \land \tau_2) \vdash x \sim y \]

\[ \Delta, x: \forall (\tau_1 \land \tau_2) \vdash \sigma_1 <: \sigma_2 \]

\[ \Delta, x: \forall (\tau_1 \setminus \tau_2) \vdash \sigma_1 <: \bot \]

\[ \Delta, x: \forall (\tau_2 \setminus \tau_1) \vdash \top <: \sigma_2 \]

\[ \Delta \vdash (x: \exists \tau_1) \to \sigma_1 <: (x: \forall \tau_2) \to \sigma_2 \]

**Example:**

\[ \vdash (x: \exists \{x \mid x = 0\}) \to \{y \mid y = 0\}^{\forall \exists} \]

\[ <: (x: \forall \{x \mid x = 0\}) \to \{y \mid y = x\}^{\forall \exists} \]
Overview: Our Type System

• Extends dependent refinement types with:
  • Qualified types $\tau^{Q_1Q_2}$
    to express universal/existential branching behaviors
    and partial/total correctness

  • Qualified bindings $x:Q\tau$
    to cope with non-determinism from program inputs

• Gödel encoding of function-type values
  & guarded intersection types
  to achieve relative completeness
Gödel Encoding of Function-Type Values

- Enables predicates of the underlying logic $T$ (e.g., second-order arithmetic) to depend on function-type arguments encoded as $T$-objects

- $(f: \forall \text{int} \to \text{int}^{\forall \exists}) \to (g: \forall \text{int} \to \text{int}^{\forall \exists}) \to \{u \mid f \sim g\}^{\forall \forall}$

  Functions that, given two terminating functions $f$ and $g$, always diverge if $f$ is not observationally equivalent to $g$
Guarded Intersection Types

\( \land_i \left( \phi_i \triangleright \tau_i^{Q_i Q_i'} \right) \)

- Collectively express different behaviors of functions depending on the arguments

- \((x : \forall \text{int}) \rightarrow (x > 0 \triangleright \text{int}^{\forall \exists}) \land (x < 0 \triangleright \{ y | \perp \}^{\forall \forall}) \land (x = 0 \triangleright \text{int}^{\exists \exists}) \land (x = 0 \triangleright \{ y | \perp \}^{\exists \forall})\)

Functions that, given the argument \(x:\)

- always terminate if \(x > 0\),
- always diverge if \(x < 0\), and otherwise,
- non-deterministically terminate or diverge
Our Contributions

• Novel dependent refinement type system that can:
  • uniformly express and verify universal and existential branching properties of call-by-value, higher-order, and non-deterministic programs:
    • (cond.) safety, non-safety, termination, and non-termination
  • seamlessly combine universal and existential reasoning
    • e.g., Prove non-safety via termination
    • e.g., Prove non-termination via safety
    • e.g., Prove termination and non-termination simultaneously

• Meta-theoretic properties of the type system:
  • Closure of types under complement
  • Soundness
  • Relative completeness
Complement Types \( \neg \sigma \)

- Thanks to having both modes of non-determinism, the type complement operator \( \neg \) can be defined:

\[
\neg (\tau Q_1 Q_2) \equiv (\neg \tau)(\neg Q_1)(\neg Q_2)
\]

\[
\neg \{x : \text{int} \mid \phi\} \equiv \{x : \text{int} \mid \neg \phi\}
\]

\[
\neg \left( (x : Q \tau) \rightarrow \sigma \right) \equiv (x : \neg Q \tau) \rightarrow \neg \sigma
\]

\[
\neg \forall \equiv \exists \quad \neg \exists \equiv \forall
\]
Example: Complement Types \( \neg \sigma \)

- \( \sigma \triangleq (x : \forall \text{int}) \to \{ u : \text{int} | u = x \}^{\forall \forall} \)
  - functions that, for any integer \( x \) and for any run with the argument \( x \), diverge or return an integer \( u = x \)

- \( \neg \sigma = (x : \exists \text{int}) \to \{ u : \text{int} | u \neq x \}^{\exists \exists} \)
  - functions that, for some integer \( x \) and for some run with the argument \( x \), terminate and return an integer \( u \neq x \)
Example: Combined Reasoning (Non-safety via Termination)

Goal: prove that

\[
\text{let rec } f \ x \ y = \\
\quad \text{if } x = y \text{ then } 0 \\
\quad \text{else } f \ (x - 1) \ y \\
\text{let } r = f \ 10^9 \ 0 \text{ in} \\
\text{let } z = \star \text{ in } z + r
\]

violates \[
\{ u \mid u = 0 \}^\forall
\]
Example: Combined Reasoning (Non-safety via Termination)

Goal: prove that

```plaintext
let rec f x y =
  if x = y then 0
  else f (x - 1) y

let r = f 10^9 0 in
let z = * in z + r

satisfies
¬(\{u | u = 0\}^\forall^\forall)
```
Example: Combined Reasoning
(Non-safety via Termination)

Goal: prove that
let rec \( f \) \( x \ y = \)
  \text{if } x = y \text{ then } 0
  \text{ else } f (x - 1) y
let \( r = f \ 10^9 \ 0 \) in
let \( z = * \) in \( z + r \)
satisfies
\text{\{ } u \mid u \neq 0 \text{\}}^{\exists \exists}

1. Show that \( f \) is conditionally terminating:
The well-founded relation \( \lambda((x, y), (x', y')). \)
  \( x > x' \land y = y' \land x \geq y \)
witnesses that \( f \) has the type:
  \( (x : \text{int}) \rightarrow (y : \{y \mid y \leq x\}) \rightarrow \text{int}^{\forall \exists} \)

2. Show that the actual call \( f \ 10^9 \ 0 \) always terminates
  by checking \( \models 0 \leq 10^9 \)

3. Show that, for any integer \( r \),
   we can choose an integer \( z \)
   such that \( z + r \neq 0 \)

Q.E.D.
Our Contributions

• Novel dependent refinement type system that can:
  • uniformly express and verify universal and existential branching properties of call-by-value, higher-order, and non-deterministic programs:
    • (cond.) safety, non-safety, termination, and non-termination
  • seamlessly combine universal and existential reasoning
    • e.g., Prove non-safety via termination
    • e.g., Prove non-termination via safety
    • e.g., Prove termination and non-termination simultaneously

• Meta-theoretic properties of the type system:
  • Closure of types under complement
  • Soundness
  • Relative completeness
Closure of Types under Complement

For any type $\sigma$ refining a simple type $S$,

- $[\sigma] \cap [\neg \sigma] = \emptyset$ and
- $[\sigma] \cup [\neg \sigma] = [S]$  

where

- $[\sigma]$: the set of expressions that behave according to $\sigma$
- $[S]$: the set of expressions of the type $S$
Soundness

\[ \Gamma \vdash e : \sigma \implies e \in \llbracket \Gamma \vdash \sigma \rrbracket \]

the set of expressions that behave according to \( \sigma \) under any valuation conforming to \( \Gamma \)
Relative Completeness

\[ e \in \llbracket \Gamma \vdash \sigma \rrbracket \text{ implies } \Gamma \vdash e : \sigma \]

under the assumption that the underlying logic is sufficiently expressible

- to Gödel encode arbitrary functions definable in the target programming language
- to represent well-founded relations witnessing the termination of the definable functions
Summary

- Novel dependent refinement type system that can:
  - uniformly express and verify universal and existential branching properties of call-by-value, higher-order, and non-deterministic programs:
    - (cond.) safety, non-safety, termination, and non-termination
  - seamlessly combine universal and existential reasoning
    - e.g., Prove non-safety via termination
    - e.g., Prove non-termination via safety
    - e.g., Prove termination and non-termination simultaneously

- Meta-theoretic properties of the type system:
  - Closure of types under complement
  - Soundness
  - Relative completeness
Future Work

• Extensions with temporal specifications:
  • Temporal trace specs. (e.g., LTL)
  • Branching temporal specs. (e.g., CTL, modal-$\mu$)

• Extensions with language features:
  • Recursive data structures
  • Linked data structures
  • Call-by-name evaluation
  • Probabilistic choice

• Automation of type checking and inference
Towards Automation (Ongoing)

• Type Checking
  • How to leverage off-the-shelf SMT solvers?
    ➢ Abstraction and counterexample guided refinement for the encoding of function-type values

• Type Inference
  • How to synthesize inductive invariants, well-founded relations, and Skolemization predicates?
    ➢ Reduction to existentially-quantified Horn clause and well-foundedness constraints
  • How to achieve scalable inference?
    ➢ Combination of universal and existential reasoning