Negations in Refinement Type Systems

T. Tsukada (U. Tokyo)

18th Nov, 2015
Oxford, UK
This Talk

About refinement intersection type systems that refute judgements of other type systems.

\[ \not\vdash M : \tau \]

\[ \iff \vdash M : \neg \tau \]
Background

Refinement intersection type systems are the basis for

- model checkers for higher-order model checking (cf. [Kobayashi 09] [Broadbent&Kobayashi 11] [Ramsay+ 14]),
- software model-checker for higher-order programs (cf. MoCHi [Kobayashi+ 11]).

In those type systems,

- a derivation gives a witness of derivability,
- but nothing witnesses that a given derivation is not derivable.
Motivation

A witness of underivability would be useful for

• a compact representation of an error trace

• an efficient model-checker in collaboration with the affirmative system
  • Cf. [Ramsay+ 14] [Godefroid+ 10]

• development of a type system proving safety
  • In some cases (e.g. [T&Kobayashi 14]), a type system proving failure is easier to be developed.
Contribution

Development of type systems refuting derivability in some type systems such as

- a basic type system for the $\lambda$-calculus
- a type system for call-by-value reachability

Theoretical study of the development
Outline

- Negations in type systems for
  - the call-by-name $\lambda\rightarrow$-calculus
    - Target language
    - Affirmative System
    - Negative System
  - the call-by-name $\lambda\rightarrow$-calculus + recursion
  - a call-by-value language + nondeterminism
- Semantic analysis
- Discussions
CbN $\lambda \rightarrow$-calculus

A simply typed calculus equipped with $\beta\eta$-equivalence.

Kinds (i.e. simple types):

$A, B ::= o \mid A \rightarrow A$

Terms:

$M, N ::= x \mid \lambda x^A. M \mid M \; M$
CbN $\lambda \rightarrow$-calculus

A simply typed calculus equipped with $\beta \eta$-equivalence.

Typing rules:

\[
\frac{(x :: A) \in \Delta}{\Delta \vdash x :: A}
\]

\[
\frac{\Delta, x :: A \vdash M :: B}{\Delta \vdash \lambda x^A.M :: A \rightarrow B}
\]

\[
\frac{\Delta \vdash M :: A \rightarrow B \quad \Delta \vdash N :: A}{\Delta \vdash MN :: B}
\]
CbN $\rightarrow$-calculus

A simply typed calculus equipped with $\beta\eta$-equivalence.

Equational theory:

$$(\lambda x.M)\;N = M[N/x]$$

$$\lambda x.M \;x = M \quad \text{(if } x \notin \text{fv}(M)\text{)}$$
Outline

• Negations in type systems for
  • the call-by-name $\lambda\rightarrow$-calculus
    • Target language
    • Affirmative System
    • Negative System
  • the call-by-name $\lambda\rightarrow$-calculus + recursion
  • a call-by-value language + nondeterminism

• Semantic analysis

• Discussions
Affirmative system for CbN $\lambda \rightarrow$

The type system for higher-order model checking (without the rule for recursion).

Types are parameterised by kinds and ground type sets:

$$\text{Ty}_Q(o) := Q$$
$$\text{Ty}_Q(A \rightarrow B) := \mathcal{P}(\text{Ty}_Q(A)) \times \text{Ty}_Q(B)$$

We use the following syntax for types:

$$\tau, \sigma ::= q \mid \bigwedge X \rightarrow \tau$$

$$X, Y \in \mathcal{P}(\text{Ty}_Q(A))$$
Sets of Types via Refinement Relation

Let $A$ be a kind.

The set $\text{Ty}_Q(A)$ of types that refines $A$ is given by

$$\text{Ty}_Q(A) = \{ \tau \mid \tau :: A \}$$

where $\tau :: A$ is the refinement relation:

$$q \in Q \quad \frac{\forall \sigma \in X. \sigma :: A}{\left( \bigwedge X \rightarrow \tau \right) :: A \rightarrow B} \quad \tau :: B$$
Subtyping

The subtyping relation is defined by induction on kinds.

\[ q \preceq_o q \]

\[ X \preceq_A Y \quad \tau \preceq_B \sigma \]

\[ (\land X \rightarrow \tau) \preceq_{A\rightarrow B} (\land Y \rightarrow \sigma) \]

\[ \forall \sigma \in Y. \exists \tau \in X. \tau \preceq_A \sigma \]

\[ X \preceq_A Y \]
Type Environments

A (finite) map from variables to sets of types (or intersection types).

\[ \Gamma ::= x_1 : X_1, \ldots, x_n : X_n \quad (n \geq 0) \]
Typing rules

\[(x : X) \in \Gamma \quad \tau \in X \quad \tau \preceq \sigma\]

\[\Gamma \vdash x : \sigma\]

\[\Gamma, x : X \vdash M : \tau\]

\[\Gamma \vdash \lambda x. M : \bigwedge X \rightarrow \tau\]

\[\Gamma \vdash M : \bigwedge X \rightarrow \tau \quad \Gamma \vdash N : \bigwedge X\]

\[\Gamma \vdash MN : \tau\]

\[\forall \tau \in X. \Gamma \vdash M : \tau\]

\[\Gamma \vdash M : \bigwedge X\]
Fact: Invariance under $\beta\eta$-equivalence

Suppose that $M =_{\beta\eta} N$. Then

$$\Gamma \vdash M : \tau \iff \Gamma \vdash N : \tau$$

- This fact will not be used in the sequel.
Convention: Subtyping closure

In what follows, sets of types are assumed to be closed under the subtyping relation.

\[ \tau \geq \sigma \in X \Rightarrow \tau \in X \]

Now posets of types are simply defined by:

\[ Ty_Q(o) := (Q, =) \]
\[ Ty_Q(A \rightarrow B) := u(Ty_Q(A))^{op} \times Ty_Q(B) \]

where \( u(P, \leq) := (\{X \subseteq P \mid x \geq y \in X \Rightarrow x \in X\}, \supseteq) \)

(cf. \( X \subseteq Y \) implies \( \forall x \in X \Rightarrow \forall y \in Y \))
Convention: Subtyping closure

In what follows, sets of types are assumed to be closed under the subtyping relation.

\[ \tau \succeq \sigma \in X \Rightarrow \tau \in X \]

The rule for variables becomes simpler.

\[
\begin{align*}
(x : X) \in \Gamma & \quad \tau \in X \\
\hline
\Gamma \vdash x : \tau
\end{align*}
\]
Outline

• Negations in type systems for
  • the call-by-name $\lambda\rightarrow$-calculus
    • Target language
    • Affirmative System
    • Negative System
  • the call-by-name $\lambda\rightarrow$-calculus + recursion
  • a call-by-value language + nondeterminism

• Semantic analysis

• Discussions
Negative Type System

Negative types are those constructed from the negative ground types \( \overline{Q} := \{ \overline{q} \mid q \in Q \} \):

\[
\overline{\text{Ty}_Q(A)} := \text{Ty}_{\overline{Q}}(A)
\]

\[
\overline{\tau}, \overline{\sigma} ::= \overline{q} \mid \bigwedge \overline{X} \rightarrow \overline{\tau}
\]

\( \overline{X}, \overline{Y} \in u(\text{Ty}_{\overline{Q}}(A)) \)

Typing rules are the same as the affirmative system.
Negation of a type

We define the two anti-monotone bijections on types as follows:

\[ \neg_A : \text{Ty}_Q(A) \longrightarrow \overline{\text{Ty}_Q(A)} \]

\[ \downarrow_A : u(\text{Ty}_Q(A)) \longrightarrow u(\overline{\text{Ty}_Q(A)}) \]

as follows:

\[ \neg_0q := \overline{q} \]

\[ \neg_{A \rightarrow B}(\bigwedge X \rightarrow \tau) := \bigwedge(\downarrow_A X) \rightarrow (\neg_B \tau) \]

\[ \downarrow_A X := \{ \neg_A \tau \mid \tau \notin X \} \]
Negation

\[ \neg_A : \text{Ty}_Q(A) \rightarrow \overline{\text{Ty}_Q(A)} \]

\[ \text{Ty}_Q(o) = \text{Ty}_{\overline{Q}}(o) \]
Natural \[ \triangleright_A : u(Ty_Q(A)) \rightarrow u(\overline{Ty_Q(A)}) \]

\[ Ty_Q(o) = Ty_{\overline{Q}}(o) \]

- \( q_1 \)
- \( q_2 \)
- \( q_3 \)

- \( \overline{q}_1 \)
- \( \overline{q}_2 \)
- \( \overline{q}_3 \)
Natural \[ \mathfrak{A} : u(Ty_Q(A)) \rightarrow u(Ty_{\overline{Q}}(A)) \]

\[ Ty_Q(o) = Ty_{\overline{Q}}(o) \]
Natural

\[ \lambda_A: u(Ty_Q(A)) \rightarrow u(\overline{Ty_Q(A)}) \]

\[ Ty_Q(A) = Ty_{\overline{Q}}(A) \]
Negation of a type

We define the two anti-monotone bijections on types as follows:

\[ \neg_A : \text{Ty}_Q(A) \rightarrow \text{Ty}_Q(A) \]
\[ \mathcal{A}_A : u(\text{Ty}_Q(A)) \rightarrow u(\text{Ty}_Q(A)) \]

as follows:

\[ \neg_o q := \overline{q} \]
\[ \neg_{A \rightarrow B}(\bigwedge X \rightarrow \tau) := \bigwedge (\mathcal{A}_A X) \rightarrow (\neg_B \tau) \]
\[ \mathcal{A}_A X := \{ \neg_A \tau \mid \tau \notin X \} \]
We define the two anti-monotone bijections on types as follows:

\[
x : \bigwedge X \vdash x : \neg \tau \iff x : \bigwedge X \nvdash x : \tau \iff \tau \notin X
\]

\[
\iff \neg \tau \in \llbracket X \rrbracket \iff x : \bigwedge (\llbracket X \rrbracket) \vdash x : \neg \tau
\]

\[
M : \neg (\bigwedge X \rightarrow \tau) \iff x : \bigwedge X \vdash M x : \neg \tau
\]

\[
\iff x : \bigwedge (\llbracket X \rrbracket) \vdash M x : \neg \tau
\]

\[
\neg_{A \rightarrow B} (\bigwedge X \rightarrow \tau) := \bigwedge (\llbracket A X \rrbracket) \rightarrow (\neg_B \tau)
\]

\[
\llbracket A X \rrbracket := \{ \neg_A \tau \mid \tau \notin X \}
\]
Theorem

• $\Gamma \not
\vdash M : \tau$ if and only if $\forall \Gamma \vdash M : \neg \tau$,

where $\forall (x_1 : X_1, \ldots, x_n : X_n) := x_1 : (\forall X_1), \ldots, x_n : (\forall X_n)$

• Let $X = \{ \tau | \Gamma \vdash M : \tau \}$. Then

$\forall \Gamma \vdash M : \bigwedge (\forall X)$

Proof) By mutual induction on the structure of the term.
Main Theorem

Theorem

- \( \Gamma 
  \not \models M : \tau \) if and only if \( \models \Gamma \vdash M : \neg \tau \),
  where \( \models (x_1 : X_1, \ldots, x_n : X_n) := x_1 : (\models X_1), \ldots, x_n : (\models X_n) \)

- Let \( X = \{ \tau \mid \Gamma \vdash M : \tau \} \). Then
  \[
  \models \Gamma \vdash M : \bigwedge (\models X)
  \]

Proof:

- \( \Gamma \vdash M : \bigwedge X \) iff \( \models \Gamma \vdash M : \bigwedge (\models X) \)
- under a certain condition
Outline

• Negations in type systems for
  • the call-by-name $\lambda\rightarrow$-calculus
  • the call-by-name $\lambda\rightarrow$-calculus + recursion
  • a call-by-value language + nondeterminism

• Semantic analysis

• Discussions
\(\lambda\rightarrow + \text{ Recursion}\)

**Term:**

\[
M, N ::= x \mid \lambda x^A.M \mid M \; M \mid Y \; M
\]

**Equational theory:**

\[
(\lambda x.M) \; N = M[N/x]
\]

\[
\lambda x.M \; x = M \quad \text{(if } x \notin \text{fv}(M)\text{)}
\]

\[
Y \; M = M \; (Y \; M)
\]
Recursion Rule in Affirmative System

The rule for recursion is given by:

\[
\begin{align*}
\Gamma \vdash M : \bigwedge X \rightarrow \tau & \quad \Gamma \vdash Y M : \bigwedge X \\
\hline
\Gamma \vdash Y M : \tau
\end{align*}
\]

This is a co-inductive rule: a derivation can be infinite.
Recursion Rule in Negative System

The rule for recursion is given by:

\[
\frac{\Gamma \vdash M : \bigwedge X \rightarrow \tau \quad \Gamma \vdash Y \ M : \bigwedge X}{\Gamma \vdash Y \ M : \tau}
\]

This is an inductive rule: a derivation must be finite.
Main Theorem

Lemma

\( \forall f. Y f : \tau \iff \vdash \lambda f. Y f : \neg \tau \)

Theorem

• \( \Gamma \not\vdash M : \tau \) if and only if \( \models \Gamma \vdash M : \neg \tau \).

• Let \( X = \{ \tau \mid \Gamma \vdash M : \tau \} \). Then
  \[ \models \Gamma \vdash M : \bigwedge (\models X) \]
Outline

• Negations in type systems for
  • the call-by-name $\lambda\rightarrow$-calculus
  • the call-by-name $\lambda\rightarrow$-calculus + recursion
  • a call-by-value language + nondeterminism

• Semantic analysis

• Discussions
Target Language

Kinds (or simple types):

\[ A, B ::= o \mid A \rightarrow TA \]
\[ U ::= A \mid TA \]

Terms:

\[ M ::= v \mid vv \mid \text{let } x = M \text{ in } M \mid M \oplus M \]
\[ \quad \mid \text{if } v \text{ then } M \text{ else } M \]
\[ v ::= t \mid f \mid x \mid \lambda x. M \]
Simple Type System

Value judgements ($\Delta \vdash M : A$):

- $\frac{(x :: A) \in \Delta}{\Delta \vdash x :: A}$
- $\frac{v \in \{t, f\}}{\Delta \vdash v :: o}$
- $\frac{\Delta, x :: A \vdash M :: TB}{\Delta \vdash M :: A \rightarrow TB}$

Computation judgements ($\Delta \vdash M : TA$):

- $\frac{\Delta \vdash v :: A}{\Delta \vdash v :: TA}$
- $\frac{\Delta \vdash v_1 :: A \rightarrow TB}{\Delta \vdash v_1 v_2 :: TB}$
- $\frac{\Delta \vdash M_i :: TA \ (i = 1, 2)}{\Delta \vdash M_1 \oplus M_2 :: TA}$
- $\frac{\Delta \vdash M :: TA}{\Delta \vdash \text{let } x = M \text{ in } N :: TB}$
- $\frac{\Delta, x :: A \vdash N :: TB}{\Delta \vdash \text{let } x = M \text{ in } N :: TB}$
- $\frac{\Delta \vdash v :: o}{\Delta \vdash \text{if } v \text{ then } M_1 \text{ else } M_2 :: TA}$
Reduction Semantics

Base cases:

\[(\lambda x. M) v \longrightarrow M[v/x]\]

\[\text{let } x = v \text{ in } M \longrightarrow M[v/x]\]

\[\text{if } t \text{ then } M_1 \text{ else } M_2 \longrightarrow M_1\]

\[\text{if } f \text{ then } M_1 \text{ else } M_2 \longrightarrow M_2\]

\[M_1 \oplus M_2 \longrightarrow M_1\]

\[M_1 \oplus M_2 \longrightarrow M_2\]

Evaluation context: \[E ::= [] \mid \text{let } x = E \text{ in } M\]
Affirmative System

Types (formally defined by induction on types):

\[ \tau, \sigma ::= t \mid f \mid \bigwedge X \mid \tau \rightarrow \tau \]

\( X, Y \in (\text{sets of types}) \)

Refinement relation:

\[ \tau \in \{t, f\} \quad \quad \forall \tau \in X.\tau :: A \]

\[ \bigwedge X :: TA \]

\[ \forall \sigma \in X.\sigma :: A \quad \tau :: TB \]

\[ (\bigwedge X \rightarrow \tau) :: A \rightarrow TB \]
Example of a type

Let

\[ \tau = \bigwedge \left\{ \bigwedge \{ t \} \rightarrow \bigwedge \{ f \}, \bigwedge \{ f \} \rightarrow \bigwedge \{ t \} \right\} \rightarrow \bigwedge \{ t \} \]

Then \( \tau :: (o \rightarrow To) \rightarrow To \).

Examples of derivable/underivable judgements

\[ \vdash \lambda f.(f \oplus \text{let } x = f t \text{ in } f y) : \tau \]

\[ \not\vdash \lambda f.(f \oplus f t) : \tau \]
Typing Rules

Value judgements ($\Gamma \vdash M : \tau$ with $\tau :: A$):

\[
(x : X) \in \Gamma \quad \tau \in X \quad \frac{}{\Gamma \vdash x : \tau}
\]

\[
\frac{}{\Gamma \vdash t : t}
\]

\[
\frac{}{\Gamma \vdash f : f}
\]

\[
\frac{}{\Gamma \vdash \lambda x. M : \land X \rightarrow \tau}
\]

Computation judgements ($\Gamma \vdash M : \tau$ with $\tau :: TA$):

\[
\forall \tau \in X. \frac{}{\Gamma \vdash v : \tau}
\]

\[
\frac{}{\Gamma \vdash v_1 : \land X \rightarrow \tau}
\]

\[
\frac{}{\Gamma \vdash v_2 : \land X}
\]

\[
\frac{}{\Gamma \vdash v_1 v_2 : \tau}
\]

\[
\exists i \in \{1, 2\}. \frac{}{\Gamma \vdash M_i : \tau}
\]

\[
\frac{}{\Gamma \vdash M_1 \oplus M_2 : \tau}
\]

\[
\frac{}{\Gamma \vdash v : t}
\]

\[
\frac{}{\Gamma \vdash M_1 : \tau}
\]

\[
\frac{}{\Gamma \vdash \text{if } v \text{ then } M_1 \text{ else } M_2 : \tau}
\]

\[
\frac{}{\Gamma \vdash \text{let } x = M \text{ in } N : \tau}
\]

\[
\frac{}{\Gamma \vdash \text{if } v \text{ then } M_1 \text{ else } M_2 : \tau}
\]
Soundness and Completeness

**Theorem**

\[ \vdash M : \bigwedge X \iff \exists v \in \langle M \rangle. \vdash v : \bigwedge X \]

(where \( \langle M \rangle := \{ v \mid M \rightarrow^* v \} \))

In particular,

\[ \vdash M : \text{t} \iff M \rightarrow^* \text{t} \]

\[ \vdash M : \text{f} \iff M \rightarrow^* \text{f} \]
Negative System

Types (formally defined by induction on types):

\[ \bar{\tau}, \bar{\sigma} ::= \bar{t} \mid \bar{f} \mid \bigwedge \bar{X} \mid \bar{\tau} \rightarrow \bar{\tau} \mid \bigvee \bar{X} \]

\( \bar{X}, \bar{Y} \in \text{(sets of types)} \)

Refinement relation:

\[ \bar{\tau} \in \{\bar{t}, \bar{f}\} \quad \forall \bar{\tau} \in \bar{X} \cdot \bar{\tau} :: A \quad \bigvee \bar{X} :: TA \]

\[ \forall \bar{\sigma} \in \bar{X} \cdot \bar{\sigma} :: A \quad \bar{\tau} :: TB \]

\[ (\bigwedge \bar{X} \rightarrow \bar{\tau}) :: A \rightarrow TB \]
Typing Rules

Value judgements (Γ ⊢ M : τ with τ :: A):

\(\langle x : \bar{X} \rangle \in \bar{\Gamma} \quad \bar{\tau} \in \bar{\bar{X}}\)

\[\bar{\Gamma} \vdash x : \bar{\tau}\]

Γ ⊢ t : f

Γ ⊢ f : t

Γ ⊢ λx. M : ∨ \bar{X} → \bar{\tau}

Computation judgements (Γ ⊢ M : τ with τ :: TA):

\[\forall \bar{\tau} \in \bar{\bar{X}}. \bar{\Gamma} \vdash v : \bar{\tau}\]

\[\forall \bar{\tau} \in \bar{\bar{X}}. \bar{\Gamma} \vdash v : \bar{\tau}\]

\[\exists \bar{\tau} \in \bar{\bar{X}}. \bar{\Gamma} \vdash v : \bar{\tau}\]

\[\bar{\Gamma} \vdash M_1 \oplus M_2 : \bar{\tau}\]

\[\bar{\Gamma} \vdash v_1 : \bar{\tau} \quad \bar{\Gamma} \vdash v_2 : \bar{\tau}\]

\[\bar{\Gamma} \vdash v_1 \cdot v_2 : \bar{\tau}\]
Typing rules (cont.)

Rules for conditional branch:

\[
\begin{align*}
\Gamma \vdash v : \text{f} & \quad \Gamma \vdash M_1 : \bar{\tau} \\
\Gamma \vdash \text{if } v \text{ then } M_1 \text{ else } M_2 : \bar{\tau} \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash v : \text{t} & \quad \Gamma \vdash M_2 : \bar{\tau} \\
\Gamma \vdash \text{if } v \text{ then } M_1 \text{ else } M_2 : \bar{\tau} \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash v : \text{t} \land \text{f} & \\
\Gamma \vdash \text{if } v \text{ then } M_1 \text{ else } M_2 : \bar{\tau} \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash M_1 : \bar{\tau} & \quad \Gamma \vdash M_2 : \bar{\tau} \\
\Gamma \vdash \text{if } v \text{ then } M_1 \text{ else } M_2 : \bar{\tau} \\
\end{align*}
\]
Typing rules

Rule for let-expression:

\[ \forall i \in I. \quad \Gamma \vdash M : \bigvee_{j \in J_i} \bar{\tau}_{i,j} \]

\[ \bigwedge_{i \in I} \bigvee_{j \in J_i} \bar{\tau}_{i,j} \xrightarrow{\text{dist. law}} \bigvee_{k \in K} \bigwedge_{l \in L_k} \bar{\sigma}_{k,l} \]

\[ \forall k \in K. \quad \Gamma, x : \{ \bar{\sigma}_{k,l} \mid l \in L_k \} \vdash N : \bar{\gamma} \]

\[ \frac{\Gamma \vdash \text{let } x = M \text{ in } N : \bar{\gamma}}{\Gamma \vdash \text{let } x = M \text{ in } N : \bar{\gamma}} \]
Soundness and Completeness

**Theorem**

\[ \vdash M : \bigvee X \iff \forall v \in \langle M \rangle. \ \vdash v : \bigvee X \]

(where \( \langle M \rangle := \{ v \mid M \rightarrow^* v \} \))

In particular,

\[ \vdash M : \bar{t} \iff M \not\rightarrow^* t \]

\[ \vdash M : \bar{f} \iff M \not\rightarrow^* f \]
Negation of a type

Given a kind $A$, let

$$\text{Ty}(A) := \{ \tau \mid \tau :: A \}$$

$$\overline{\text{Ty}(A)} := \{ \overline{\tau} \mid \overline{\tau} :: U \}$$

We define two operations:

$$\neg_A : \text{Ty}_Q(A) \longrightarrow \overline{\text{Ty}_Q(A)}$$

$$\mathcal{A}_A : u(\text{Ty}_Q(A)) \longrightarrow u(\overline{\text{Ty}_Q(A)})$$
Definition of the Negation

\[ \neg_o v := \bar{v} \quad (v \in \{ t, f \}) \]

\[ \neg_{A \rightarrow TB}(\bigwedge X \rightarrow \tau) := \bigwedge (\models_A X) \rightarrow (\neg_{TB} \tau) \]

\[ \neg_{TA}(\bigwedge X) := \bigvee \{ \neg_A \tau \mid \tau \in X \} \]

\[ \models_A X := \{ \neg_A \tau \mid \tau \notin X \} \]
Examples

\neg t = \bar{t}
\neg f = \bar{f}

\models (\wedge \{ \} ) = \bar{t} \wedge \bar{f}
\neg (\wedge \{ \} ) = \lor \{ \} 

\models (\wedge \{ t \} ) = \wedge \{ \bar{f} \}
\neg (\wedge \{ t \} ) = \lor \{ \bar{t} \}

\models (\wedge \{ f \} ) = \wedge \{ \bar{t} \}
\neg (\wedge \{ f \} ) = \lor \{ \bar{f} \}

\models (\wedge \{ t, f \} ) = \wedge \{ \} 
\neg (\wedge \{ t, f \} ) = \lor \{ \bar{t}, \bar{f} \}
Examples

\[ \neg (\text{\texttt{\textbackslash{\text{\{}t\text\{\}}} \rightarrow \text{\texttt{\textbackslash{\text{\{}t\text\{\}}}}) = \text{\texttt{\textbackslash{l}}}(\text{\texttt{\textbackslash{\text{\{}t\text\{\}}} \rightarrow \neg (\text{\texttt{\textbackslash{\text{\{}t\text\{\}}} })
\]

\[ = \text{\texttt{\textbackslash{\text{\{\texttt{\textbackslash{\text{\{}f\text\{\}}} \rightarrow \text{\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}t\text\{\}}} \}}}}}}}})
\]

\[ \neg (\text{\texttt{\textbackslash{\text{\{}\}} \rightarrow \text{\texttt{\textbackslash{\text{\{}\}}}}) = \text{\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}t\text\{\}}} \}, \texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}f\text\{\}}} \}}}}}}}}}}}) \rightarrow \text{\texttt{\textbackslash{\text{\{}\}}})
\]

\[ \neg (\text{\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}t\text\{\}}} \}, \texttt{\textbackslash{\text{\{}f\text\{\}}} \rightarrow \text{\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}t\text\{\}}} \}, \texttt{\textbackslash{\text{\{}f\text\{\}}} \}}}})) = \text{\texttt{\textbackslash{\text{\{}\}}} \rightarrow \text{\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}t\text\{\}}} \}}}}}}}} \rightarrow \text{\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}\texttt{\textbackslash{\text{\{}f\text\{\}}} \}}}}}}}}}})
\]
Examples

Let

\[ \tau = \bigwedge \left\{ \begin{array}{l}
\bigwedge \{t\} \rightarrow \bigwedge \{f\} \\
\bigwedge \{f\} \rightarrow \bigwedge \{t\}
\end{array} \right\} \rightarrow \bigwedge \{t\} \]

Then

\[ \neg \tau = \bigwedge \left\{ \begin{array}{l}
\bigwedge \{\bar{f}\} \rightarrow \bigvee \{\bar{t}\} \\
\bigwedge \{\bar{t}\} \rightarrow \bigvee \{\bar{f}\} \\
\bigwedge \{\bar{t}, \bar{f}\} \rightarrow \bigvee \{} \\
\bigwedge \{} \rightarrow \bigvee \{\bar{t}, \bar{f}\}
\end{array} \right\} \rightarrow \bigvee \{\bar{t}\} \]

\[ \not\vdash \lambda f. (f \oplus f \cdot t) : \tau \]

\[ \vdash \lambda f. (f \oplus f \cdot t) : \neg \tau \]
Main Theorem

Theorem

$\Gamma \not\models M : \tau$ if and only if $\models \Gamma \models M : \neg \tau$.

Let $X = \{ \tau \mid \Gamma \vdash \nu : \tau \}$ . Then

$\models \Gamma \models \nu : \bigwedge (\models X)$
Some (Possible) Extensions

1. CbV calculus with integers
   - Straightforward.
   - One needs infinite intersection and union.

2. CbV calculus with recursion
   - I believe that it is straightforward, though I have not yet checked.
Outline

- Negations in type systems for
  - the call-by-name $\lambda \rightarrow$-calculus
  - the call-by-name $\lambda \rightarrow$-calculus + recursion
  - a call-by-value language + nondeterminism
- Semantic analysis
- Discussions
How to Develop the Negative Systems

1. The CbN system has a categorical description.

\[ \Gamma \vdash M : \tau \iff (\Gamma, \tau) \in \llbracket M \rrbracket_{\text{ScottL}_u} \]

2. The negation induces an automorphism.

\[ \varphi : \text{ScottL}_u \xrightarrow{\cong} \text{ScottL}_u \]

3. A CbV system is given by a monad on \( \text{ScottL}_u \).

4. The negative system is given by the monad

\[
\begin{align*}
\text{ScottL}_u & \xrightarrow{\varphi^{-1}} \text{ScottL}_u \\
& \xrightarrow{T} \text{ScottL}_u \\
& \xrightarrow{\varphi} \text{ScottL}_u
\end{align*}
\]
Category $\text{ScottL}_u$

**Definition** The category $\text{ScottL}$ is given by:

**Object**  Poset $(A, \leq_A)$.

**Morphism** An upward-closed relation

\[ R \subseteq u(A)^{\text{op}} \times B \]

**Composition** Let \[ R \subseteq u(A)^{\text{op}} \times B \]
\[ S \subseteq u(B)^{\text{op}} \times C \]. Then

\[ \exists Y \in u(B). \left( \forall b \in Y. (X, b) \in R \text{ and } (Y, c) \in S \right) \]

\[ (X, c) \in (S \circ R) \]
Interpretation of CbN $\lambda \rightarrow_{in} \text{ScottL}_u$

**Fact**  \( \text{ScottL}_u \) is a cartesian closed category.

Interpretation of kinds is given by:

\[
[o]_Q := (Q, =) \\
[A \rightarrow B]_Q := u([A]_Q)^{op} \times [B]_Q
\]

Hence \( [A]_Q \cong Ty_Q(A) \).

**Fact**  \( \Gamma \vdash M : \tau \iff (\Gamma, \tau) \in [M] \)
Negation Functor on $\text{ScottL}_u$

The functor $\varphi : \text{ScottL}_u \rightarrow \text{ScottL}_u$ is defined by:

$$\varphi(A) := A^{op}$$
$$\varphi(R) := \{ (A \setminus X, b) \in u(A)^{op} \times B \mid (X, b) \notin R \}$$

**Lemma** $\varphi$ is an isomorphism on $\text{ScottL}_u$.

If $R \in u(A)^{op} \times B$ and $A = \emptyset$, then

$$\varphi(R) = \{ (\emptyset, b) \mid (\emptyset, b) \notin R \}$$

which is essentially the complement of $R$. 
A monad on a category $C$ is a functor $C \to C$ with some additional structures.

A (strong) monad on a CCC gives rise to a model of a call-by-value calculus [Moggi 91].

A monad on $\text{ScottL}_u$ can be seen as a refinement type system for a call-by-value calculus.
Negated Monad and Negative System

Let $T : \text{ScottL}_u \rightarrow \text{ScottL}_u$ be a strong monad. Then

$$\text{ScottL}_u \xrightarrow{\varphi^{-1}} \text{ScottL}_u \xrightarrow{T} \text{ScottL}_u \xrightarrow{\varphi} \text{ScottL}_u$$

has the canonical monad structure. Furthermore the respective Kliesli categories are isomorphic

$$(\text{ScottL}_u)_T \cong (\text{ScottL}_u)_\varphi T \varphi^{-1}$$

and the refinement type system corresponding to the right-hand-side is the negation of the left-hand-side.
Example

The previous type system for CbV calculus is given by the following monad.

\[
T(A) := u(A) \\
T(R) := \{ (\exists Y, X) \in u(u(A))^\text{op} \times u(B) \mid \exists X \in \exists. \forall b \in Y. (X, b) \in R \}
\]
Outline

• Negations in type systems for
  • the call-by-name $\lambda\rightarrow$-calculus
  • the call-by-name $\lambda\rightarrow$-calculus + recursion
  • a call-by-value language + nondeterminism

• Semantic analysis

• Discussions
Automata complementation

Corresponds to negation of a 2nd-order judgement.
Boolean Closedness of Types

Let $A$ be a kind and $B_A$ be the set of all Böhm trees of type $A$. A language is a subset of $B_A$.

**Definition** A language $L \subseteq B_A$ is type-definable if there exists a type $\tau$ such that

$$L = \{ M \in B_A \mid \vdash M : \tau \}$$

in the type system for higher-order model checking [Kobayashi&Ong 09] [T&Ong 14].

**Corollary** The class of type-definable languages are closed under Boolean operations on sets.
Further Applications

The technique presented in this talk is applicable to:

• the type system for the full higher-order model-checking [Kobayashi&Ong 09]

• a type system witnessing call-by-value reachability [T&Kobayashi 14]

• a dependent intersection type system in [Kobayashi+ 11], via the translation of dependent types to intersection and union types
Consistency and Inconsistency

The negation of a "small" type can be very large. So the negation may not be efficiently computable.

The notion of consistency and inconsistency may be useful in the practical use:

**Definition** Let $\tau \in \text{Ty}_Q(A)$ and $\bar{\sigma} \in \overline{\text{Ty}_Q(A)}$. They are consistent if $\neg \tau \leq \bar{\sigma}$ and inconsistent otherwise.

**Proposition** If $\tau$ and $\bar{\sigma}$ are inconsistent, then

$$\models M : \bar{\sigma} \implies \nexists M : \tau$$
Inductive Definition of Consistency

\[
q \neq p \\
\Rightarrow q \triangleleft_o \bar{p}
\]

\[
\forall \tau \in X. \forall \bar{\sigma} \in \bar{Y}. \tau \triangleleft_A \bar{\sigma} \\
\Rightarrow \bigwedge X \triangleleft_A \bigwedge \bar{Y}
\]

\[
\tau_1 \triangleleft_A \bar{\sigma}_1 \Rightarrow \tau_2 \triangleleft_B \bar{\sigma}_2 \\
(\tau_1 \rightarrow \tau_2) \triangleleft_{A \rightarrow B} (\bar{\sigma}_1 \rightarrow \bar{\sigma}_2)
\]

Inductive definition of inconsistency is now trivial.
Related Work

"Krivine machines and higher-order schemes"
[Salvati&Walkiewicz 12]

- The notion of consistency and inconsistency can be found in their work (called complementarity for the former and the latter has no name).
- This talk is partially inspired by their work.
Conclusion

Negation is a definable operation in the refinement intersection type system for the call-by-name $\lambda \rightarrow$.

This observation leads to the construction of negative type systems for other refinement type systems, e.g.,

- call-by-name $\lambda \rightarrow$ + recursion
- the type system for HOMC
- a type system for a call-by-value language

Application to verification needs some work.