Logical Relations for a Manifest Contract Calculus

Taro Sekiyama  Atsushi Igarashi

Kyoto University
A typed lambda calculus with (higher-order) software contracts

**hybrid** checking of software contracts

- Static type system: refinement type
  \[
x : T \mid e
  \]
  e.g. \(x : \text{int} \mid 0 < x\)

- Dynamic checking: cast \(\langle T_1 \Rightarrow T_2\rangle^\ell\)
  e.g. \(\langle \text{int} \Rightarrow \{x : \text{int} \mid x < 0\}\rangle^\ell\)

Programming in Manifest Contract Calculus

\[
div : \text{int} \rightarrow \{ x : \text{int} \mid 0 \neq x \} \rightarrow \text{int}
\]

\[
div \ "abc" \ 2 \quad (\ast \text{ Compiler error } \ast)
\]

\[
div \ 6 \ 0 \quad (\ast \text{ Compiler error } \ast)
\]

\[
(\ast \text{ Compiler doesn’t know that } y \text{ is non-zero } \ast)
\]

\[
(\text{fun } y : \text{int}. \; \text{div} \ 6 \ y)
\]
Programming in Manifest Contract Calculus

\text{div} : \text{int} \rightarrow \{x:\text{int} \mid 0 \neq x\} \rightarrow \text{int}

\text{div} \ "\text{abc}\" \ 2 \quad (\ast \ \text{Compiler error} \ \ast)

\text{div} \ 6 \ 0 \quad (\ast \ \text{Compiler error} \ \ast)

(\ast \ \text{Compiler inserts a cast} \ \ast)

(\text{fun} \ y : \text{int}. \ \text{div} \ 6 \ ((\langle \text{int} \Rightarrow \{x:\text{int} \mid 0 \neq x\}\rangle^\ell \ y)))
Previous Work: Upcast Elimination

Upcast Elimination [1,2]

An upcast and an identity function are contextually equivalent

An upcast is a cast from a type to its supertype

- $\langle \{x: \text{int} \mid 0 < x\} \Rightarrow \text{int} \rangle^\ell$
- $\langle \{x: \text{int} \mid \text{is\_square\_} x\} \Rightarrow \{x: \text{int} \mid 0 < x\} \rangle^\ell$

Upcast elimination is useful for optimization

[2] Belo et al., 2011
Previous work

- tried to prove upcast elimination by using *logical relations*
- didn’t really prove soundness of the logical relations w.r.t contextual equivalence

<table>
<thead>
<tr>
<th>$\langle T_1 \Rightarrow T_2 \rangle^\ell \bowtie \approx \text{fun } x.x$</th>
<th>$\lambda_H^{[1]}$</th>
<th>$F_H^{[2]}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>proved</td>
<td>proved</td>
<td></td>
</tr>
<tr>
<td>flawed</td>
<td>not proved</td>
<td></td>
</tr>
<tr>
<td>not proved</td>
<td>not proved</td>
<td></td>
</tr>
</tbody>
</table>

$\bowtie$: contextual equivalence  
$\bowtie$: logical relation

[2] Belo et al., 2011
Logical Relations for a Manifest Contract Calculus, Fixed

Taro Sekiyama Atsushi Igarashi
Kyoto University
This work

- fixes the flaws of previous work
- introduces $F^\text{fix}_H$
- a polymorphic manifest contract calculus with fixed-point operator
- non-termination is only effect in $F^\text{fix}_H$

<table>
<thead>
<tr>
<th></th>
<th>$\lambda_H$</th>
<th>$F_H$</th>
<th>$F^\text{fix}_H$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subsumption rule</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Polymorphic types</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Fixed-point operator</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>
Contribution

- **Semi-typed** contextual equivalence
- A sound logical relation w.r.t **semi-typed** contextual equivalence
- Proof of upcast elimination by using the logical relation above
  - We believe correctness of our proof :-)

<table>
<thead>
<tr>
<th>\langle T_1 \Rightarrow T_2 \rangle^\ell \sim \text{fun x.x}</th>
<th>\lambda_H</th>
<th>F_H</th>
<th>F_H^{\text{fix}}</th>
</tr>
</thead>
<tbody>
<tr>
<td>proved</td>
<td>proved</td>
<td>proved</td>
<td></td>
</tr>
<tr>
<td>\sim \subseteq \sim</td>
<td>flawed</td>
<td>not proved</td>
<td>proved</td>
</tr>
<tr>
<td>\langle T_1 \Rightarrow T_2 \rangle^\ell \sim \text{fun x.x}</td>
<td>not proved</td>
<td>not proved</td>
<td>proved</td>
</tr>
</tbody>
</table>
1. A Manifest Contract Calculus: $F_{H}^{\text{fix}}$

2. Semi-Typed Contextual Equivalence

3. Logical Relation

4. Upcast Elimination

5. Discussion
Contents

1. A Manifest Contract Calculus: $F^\text{fix}_H$

2. Semi-Typed Contextual Equivalence

3. Logical Relation

4. Upcast Elimination

5. Discussion
Overview of $\mathcal{F}^\text{fix}_\text{H}$

$\mathcal{F}^\text{fix}_\text{H}$ is a typed lambda calculus with

- polymorphic types,
- refinement types $\{x:T \mid e\}$,
- dependent function types $x:T_1 \rightarrow T_2$,
- casts $\langle T_1 \Rightarrow T_2 \rangle^\ell$, and
- fixed-point operator (recursive functions)

<table>
<thead>
<tr>
<th></th>
<th>$\lambda_\text{H}$</th>
<th>$\mathcal{F}_\text{H}$</th>
<th>$\mathcal{F}^\text{fix}_\text{H}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subsumption rule</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Polymorphic types</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Recursive functions</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>
Refinement types: $\{x: T \mid e\}$
- denote a set of values which
  - are in $T$
  - satisfy the contract (boolean expression) $e$
- e.g. $\{x:\text{int} \mid 0 < x\} = \{1, 2, 3, \ldots\}$

Dependent function types: $x: T_1 \rightarrow T_2$
- denote a set of functions which
  - accept values $v$ of $T_1$
  - return values of $T_2[v/x]$
- e.g. $x:\text{int} \rightarrow \{y:\text{int} \mid x < y\}$
Dynamic Checking: Cast

Casts: $\langle T_1 \Rightarrow T_2 \rangle^\ell$

- accept values $v$ of $T_1$
- check whether $v$ can behave as $T_2$
  - If the checking fails, the cast is blamed with label $\ell$
- e.g. $\langle \text{int} \Rightarrow \{x:\text{int} \mid 0 < x\} \rangle^\ell$

\[
\langle \text{int} \Rightarrow \{x:\text{int} \mid 0 < x\} \rangle^\ell \ 0 \rightsquigarrow^* \uparrow \ell \\
\langle \text{int} \Rightarrow \{x:\text{int} \mid 0 < x\} \rangle^\ell \ 2 \rightsquigarrow^* 2
\]
At first, we gave A-normal form as syntax following [3] which uses A-normal form to simplify the definition and the proof:

\[ e ::= v_1 \ | \ v_2 \ | \ \text{let } x = e_1 \ \text{in } e_2 \ | \ \cdots \]

It is difficult to prove even *type soundness* to require substitution of *terms*.

A-normal form is *not* closed under substitution of terms:

\[
\Gamma \vdash e_1 : T_1 \quad \Gamma, x : T_1 \vdash e_2 : T_2 \\
\Gamma \vdash \text{let } x = e_1 \ \text{in } e_2 : T_2[e_1/x]
\]

[3] Pitts, 2005
1 A Manifest Contract Calculus: $\mathcal{F}_{H}^{\text{fix}}$

2 Semi-Typed Contextual Equivalence

3 Logical Relation

4 Upcast Elimination

5 Discussion
$$e_1 \approx_{\text{typed}} e_2 : T$$

- $e_1$ and $e_2$ have the same observable result under any contexts
  - which are well-typed and accept any terms of $T$
- $e_1$ and $e_2$ are typed at the same type $T$

\[(\text{fun } x : \text{int. } 0) \approx_{\text{typed}} (\text{fun } x : \text{int. } x \ast 0) : \text{int} \rightarrow \text{int}\]

\[(\text{fun } x : \text{int. } 0) \not\approx_{\text{typed}} (\text{fun } x : \text{int. } x + 2) : \text{int} \rightarrow \text{int}\]

\[(\text{fun } x : \text{int. } 0) \not\approx_{\text{typed}} (\text{fun } x : \text{bool. } 0) : \text{int} \rightarrow \text{int}\]
Problem

- Upcast elimination doesn’t hold in typed contextual equivalence
  - An upcast and an identity function may have different types
  - Note lack of a subsumption rule

\[
\begin{array}{c|c|c}
\langle T_1 \Rightarrow T_2 \rangle^\ell & \text{fun} \ x : T_1. \ x & \text{fun} \ x : T_2. \ x \\
T_1 \rightarrow T_2 & T_1 \rightarrow T_1 & T_2 \rightarrow T_2
\end{array}
\]

- We must relax typed contextual equivalence
$e_1 \simeq e_2 : T$

- $e_1$ and $e_2$ have the same observable result under any well-typed contexts
- Only $e_1$ is typed at $T$
  - $e_2$ can even be ill-typed

\[(\text{fun } x : \text{int. } 0) \simeq (\text{fun } x : \text{int. } x \ast 0) : \text{int } \rightarrow \text{int}\]

\[(\text{fun } x : \text{int. } 0) \not\simeq (\text{fun } x : \text{int. } x + 2) : \text{int } \rightarrow \text{int}\]

\[(\text{fun } x : \text{int. } 0) \simeq (\text{fun } x : \text{bool. } 0) : \text{int } \rightarrow \text{int}\]
Formal Definition

Definition

Semi-typed contextual equivalence $\approx$ is the largest set satisfying the following:

1. If $\Gamma \vdash e_1 \approx e_2 : T$, then $\Gamma \vdash e_1 : T$
2. If $\emptyset \vdash e_1 \approx e_2 : T$, then $e_1$ and $e_2$ have the same observable result
3. Reflexivity, Transitivity, (Typed) Symmetry
4. Compatibility
5. Substitutivity
Compatibility and Substitutivity Rules

Choose *typed* terms for substitution on types

- so that the type after the substitution is well-formed

E.g.

**Compatibility: term application**

\[
\Gamma \vdash e_{11} \approx e_{21} : (x : T_1 \to T_2) \quad \Gamma \vdash e_{12} \approx e_{22} : T_1
\]

\[
\Gamma \vdash e_{11} \ e_{12} \approx e_{21} \ e_{22} : T_2 [e_{12}/x]
\]

**Substitutivity: value substitution**

\[
\Gamma, x : T_1, \Gamma' \vdash e_1 \approx e_2 : T_2 \quad \Gamma \vdash v_1 \approx v_2 : T_1
\]

\[
\Gamma, \Gamma'[v_1/x] \vdash e_1 [v_1/x] \approx e_2 [v_2/x] : T_2 [v_1/x]
\]
1. A Manifest Contract Calculus: $F_{H}^{\text{fix}}$

2. Semi-Typed Contextual Equivalence

3. Logical Relation

4. Upcast Elimination

5. Discussion
Overview of Logical Relation

\[ e_1 \bis e_2 : T \]

- \( \bis \) is defined by using
  - basic ideas of the logical relation for \( F_H \)[2]
  - \( \top \top \)-closure[3]
    - A method to give a logical relation to a lambda calculus with recursive functions

- Only \( e_1 \) is typed
  - similarly to semi-typed contextual equivalence

[2] Belo et al., 2011
[3] Pitts, 2005
Define value relations for base types

bool: \{(true, true), (false, false)\}

int: \{..., (-1, -1), (0, 0), (1, 1), ...\}
How to Define Logical Relation by $\top \top$

1. Define value relations for base types
2. Define term relations for base types by operation $\top \top$
   - $\top \top$ expands value relations to term relations

bool: $\{(true, \text{not false}), (true \&\& true, true) \ldots\}$

int: $\{(1+1, 2), (0*3, 0+0), \ldots\}$
How to Define Logical Relation by $\top\top$

1. Define value relations for base types
2. Define term relations for base types by operation $\top\top$
3. Define value relations for complex types

\[
\text{int} \to \text{int} : \{(\text{succ}, \text{fun } x. x + 1), \ldots\}
\]

Value relation $\top\top$ Term relation
How to Define Logical Relation by $\top\top$

1. Define value relations for base types
2. Define term relations for base types by operation $\top\top$
3. Define value relations for complex types
4. Define term relations for complex types by operation $\top\top$

Value relation $\xrightarrow{\top\top}$ Term relation
How to Define Logical Relation by $\top\top$

1. Define value relations for base types
2. Define term relations for base types by operation $\top\top$
3. Define value relations for complex types
4. Define term relations for complex types by operation $\top\top$
Relations for Closed Terms

- Value relation: \( T(\theta, \delta)^{val} \)
- Term relation: \( T(\theta, \delta)^{tm} \)

Here,
- \( \theta \) is a valuation for type variables in \( T \)
  \[ \theta = \{ \alpha \mapsto (r, T_1, T_2), \ldots \} \]
  - \( r \) is a term relation and an interpretation of \( \alpha \)
  - Notation: \( \theta_i = \{ (\alpha \mapsto T_i), \ldots \} \)
- \( \delta \) is a valuation for variables in \( T \)
  \[ \delta = \{ x \mapsto (v_1, v_2), \ldots \} \]
  - Notation: \( \delta_i = \{ (x \mapsto v_i), \ldots \} \)
Base type: $B$

**Value Relation**

$(v_1, v_2) \in B(\theta, \delta)^{val}$ iff

$v_1 = v_2$ and $v_1$ is a constant of $B$

**Term Relation**

$B(\theta, \delta)^{tm} = (B(\theta, \delta)^{val})^{TT}$
### Value Relation

\[(v_1, v_2) \in (x: T_1 \rightarrow T_2)(\theta, \delta)^{val} \text{ iff for any } (v_1', v_2') \in T_1(\theta, \delta)^{tm}, (v_1, v_1', v_2, v_2') \in T_2(\theta, \delta \{x \mapsto v_1', v_2'\})^{tm}\]

### Term Relation

\[(x: T_1 \rightarrow T_2)(\theta, \delta)^{tm} = ((x: T_1 \rightarrow T_2)(\theta, \delta)^{val})^{TT}\]
Value Relation

$$(v_1, v_2) \in \{ x: T \mid e \} (\theta, \delta)^{val}$$ if

- $$(v_1, v_2) \in T(\theta, \delta)^{tm}$$
- $$\theta_1(\delta_1(e[v_1/x])) \leadsto^* true$$
- $$\theta_2(\delta_2(e[v_2/x])) \leadsto^* true$$

Term Relation

$${x: T \mid e}(\theta, \delta)^{tm} = ({x: T \mid e}(\theta, \delta)^{val})^{TT}$$
Definition (Logical Relation for Open Terms)

\[ \Gamma \vdash e_1 \simeq e_2 : T \iff \]

1. \[ \Gamma \vdash e_1 : T \]

2. \[ (\theta_1(\delta_1(e_1)), \theta_2(\delta_2(e_2))) \in T(\theta, \delta)^{tm} \]

where \( \Gamma \vdash \theta; \delta \)

- \( e_1 \) and \( e_2 \) are related for well-formed substitution \( \theta \) and \( \delta \)
### Theorem (Soundness)

If $\Gamma \vdash e_1 \simeq e_2 : T$, then $\Gamma \vdash e_1 \approx e_2 : T$

- Prove that $\simeq$ satisfies the properties defining $\approx$

### Theorem (Completeness w.r.t Typed Terms)

If $\Gamma \vdash e_1 \approx e_2 : T$ and $\Gamma \vdash e_2 : T$, then $\Gamma \vdash e_1 \simeq e_2 : T$

- An orthodox method doesn’t go through
We must prove that for soundness the logical relation satisfies:
- reflexivity, transitivity, typed symmetry
- compatibility
- substitutivity

Note that:
- it suffices to prove only compatibility and substitutivity in [3]
- all the properties are proved in this work

[3] Pitts, 2005
1. A Manifest Contract Calculus: $F^\text{fix}_H$

2. Semi-Typed Contextual Equivalence

3. Logical Relation

4. Upcast Elimination

5. Discussion
**Upcast Elimination**

An upcast and an identity function are contextually equivalent

**Lemma**

If $\Gamma \vdash T_1 <: T_2$, then

$\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^\ell \simeq (\text{fun } x : T_1 \cdot x) : T_1 \rightarrow T_2$

**Corollary**

If $\Gamma \vdash T_1 <: T_2$, then

$\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^\ell \simeq (\text{fun } x : T_1 \cdot x) : T_1 \rightarrow T_2$
A Manifest Contract Calculus: $F^\text{fix}_H$

Semi-Typed Contextual Equivalence

Logical Relation

Upcast Elimination

Discussion
Conclusion

- A sound logical relation w.r.t semi-typed contextual equivalence
- Proof of upcast elimination

Technically,
- $\top\top$-closure works in manifest contract calculus with non-termination
  - The proofs of the properties are troublesome
- “Semi-typedness” doesn’t complicate the proof of soundness
  - affects the proof of completeness
Future Work

- Unrestricted completeness
- removal of “typedness” assumption
- Correctness of other optimizations
- Effects other than non-termination