hiJaC

Higher-Order UTP for Theories of Object-Orientation

Frank Zeyda and Ana Cavalcanti

University of York (UK)

28 August 2012

Context of the work

Higher-order programming is a paradigm that admits programs as values.

For instance,

$$\textbf{var}\ m \bullet m := \{\!|x := x + 1|\!\}\ ;\ \textbf{call}\ m$$

is equivalent to the program $x := x + 1$.

1. x is an integer variable; however,
2. the local variable m holds a program value;

Generally, we have that **call** $\{\!|p|\!\}$ is equivalent to p.

Motivations for higher-order programming

- Many useful application in (modern) programming languages.
- Supported by the UTP (inspired by David Naumann's work).
- In particular useful in theories of object-orientation.

Context of the work

Higher-order programming is a paradigm that admits programs as values.

For instance,

$$\textbf{var}\ m \bullet m := \{\!|x := x + 1|\!\}\ ;\ \textbf{call}\ m$$

is equivalent to the program $x := x + 1$.

1. x is an integer variable; however,
2. the local variable m holds a program value;

Generally, we have that **call** $\{\!|p|\!\}$ is equivalent to p.

Motivations for higher-order programming

- Many useful application in (modern) programming languages.
- Supported by the UTP (inspired by David Naumann's work).
- In particular useful in theories of object-orientation.

Context of the work

Higher-order programming is a paradigm that admits programs as values.

For instance,

$$\textbf{var}\ m \bullet m := \{\!|x := x + 1|\!\}\ ;\ \textbf{call}\ m$$

is equivalent to the program $x := x + 1$.

1. x is an integer variable; however,
2. the local variable m holds a program value;

Generally, we have that **call** $\{\!|p|\!\}$ is equivalent to p.

Motivations for higher-order programming

- Many useful application in (modern) programming languages.
- Supported by the UTP (inspired by David Naumann's work).
- In particular useful in theories of object-orientation.

Java example program

```
class A {
   public int x, y;

   void m1() {
      if (x > 0) { x := x - 1; m2(); }
   }

   void m2() {
      if (y > 0) { y := y - 1; m1(); }
   }

   void m3() { m1(); }
}
```

Encoding in higher-order UTP

$$m_1, m_2 := \{\!|\mu\, X, Y \bullet \left\langle \begin{array}{l} (x := x - 1\,;\; Y) \triangleleft x > 0 \triangleright \mathbf{II}, \\ (y := y - 1\,;\; X) \triangleleft y > 0 \triangleright \mathbf{II} \end{array} \right\rangle |\!\}\; ;\; m_3 := \{\!|\mathbf{call}\, m_1|\!\}$$

Challenges we address

hiJaC

But what about...

$$m_1 := \{\!|(x := x - 1 \,;\; Y) \lhd x > 0 \rhd \mathbf{II}|\!\};$$

$$m_2 := \{\!|(y := y - 1 \,;\; X) \lhd y > 0 \rhd \mathbf{II}|\!\};$$

$$m_3 := \{\!|\mathbf{call}\, m_1|\!\}$$

$\hookrightarrow$ Not valid due to restrictions on HO variable types.

Four main challenges

1. Consistency of the program model;
2. Redefinition of methods in subclasses;
3. Recursion and mutual recursion;
4. Simplicity.

UTP book: HO values "range over predicates, or rather some subset of predicates (programs)." HO value constructor has to be injective.

Challenges we address

hiJaC

But what about...

$$m_1 := \{\!|(x := x - 1\,;\ \ Y) \triangleleft x > 0 \triangleright \mathbf{II}|\!\};$$

$$m_2 := \{\!|(y := y - 1\,;\ \ X) \triangleleft y > 0 \triangleright \mathbf{II}|\!\};$$

$$m_3 := \{\!|\mathbf{call}\, m_1|\!\}$$

$\hookrightarrow$ Not valid due to restrictions on HO variable types.

Four main challenges

1. Consistency of the program model;
2. Redefinition of methods in subclasses;
3. Recursion and mutual recursion;
4. Simplicity.

UTP book: HO values "range over predicates, or rather some subset of predicates (programs)." HO value constructor has to be injective.

Challenges we address

hiJaC

But what about...

$$m_1 := \{\!|(x := x - 1 \,;\; Y) \lhd x > 0 \rhd \mathbf{II}|\!\};$$

$$m_2 := \{\!|(y := y - 1 \,;\; X) \lhd y > 0 \rhd \mathbf{II}|\!\};$$

$$m_3 := \{\!|\mathbf{call}\, m_1|\!\}$$

$\hookrightarrow$ Not valid due to restrictions on HO variable types.

Four main challenges

1. Consistency of the program model;
2. Redefinition of methods in subclasses;
3. Recursion and mutual recursion;
4. Simplicity.

UTP book: HO values "range over predicates, or rather some subset of predicates (programs)." HO value constructor has to be injective.

Overview

hiJaC

1. Higher-order UTP
2. Santos' theory of object-orientation
3. A program model
4. A theory of methods
5. Conclusions and Future Work

hiJaC

Higher-order UTP

Higher-order UTP (1)

hiJaC

Procedure variables

- Primal extension in higher-order UTP;
- However, their types are alphabets.
- Declared and used just like standard variables.
- Directly identified with predicates of some theory of designs or programs (?).

The purpose of $\{\!|\_|\!\}$ is merely "to distinguish what is to be stored from what is to be executed". Otherwise, the brackets have no semantic significance and are simply omitted in a procedure call.

We think of $\{\!|\_|\!\}$ as a type constructor for values.

- But is it sound?
- Predicates over a single variable $x : T$ is equipotent to $\mathbb{P}\, T$.

Higher-order UTP (1)

hiJaC

Procedure variables

- Primal extension in higher-order UTP;
- However, their types are alphabets.
- Declared and used just like standard variables.
- Directly identified with predicates of some theory of designs or programs (?).

The purpose of $\{\!|\_|\!\}$ is merely "to distinguish what is to be stored from what is to be executed". Otherwise, the brackets have no semantic significance and are simply omitted in a procedure call.

We think of $\{\!|\_|\!\}$ as a type constructor for values.

- But is it sound?
- Predicates over a single variable $x : T$ is equipotent to $\mathbb{P}\, T$.

Higher-order UTP (1)

hiJaC

Procedure variables

- Primal extension in higher-order UTP;
- However, their types are alphabets.
- Declared and used just like standard variables.
- Directly identified with predicates of some theory of designs or programs (?).

The purpose of $\{\!|\_|\!\}$ is merely "to distinguish what is to be stored from what is to be executed". Otherwise, the brackets have no semantic significance and are simply omitted in a procedure call.

We think of $\{\!|\_|\!\}$ as a type constructor for values.

- But is it sound?
- Predicates over a single variable $x : T$ is equipotent to $\mathbb{P}\, T$.

Higher-order UTP (1)

hiJaC

Procedure variables

- Primal extension in higher-order UTP;
- However, their types are alphabets.
- Declared and used just like standard variables.
- Directly identified with predicates of some theory of designs or programs (?).

The purpose of $\{\!|\_|\!\}$ is merely "to distinguish what is to be stored from what is to be executed". Otherwise, the brackets have no semantic significance and are simply omitted in a procedure call.

We think of $\{\!|\_|\!\}$ as a type constructor for values.

- But is it sound?
- Predicates over a single variable $x : T$ is equipotent to $\mathbb{P}\, T$.

Higher-order UTP (2)

Fundamental Law

$$(p := \{\!|Q|\!\} ;\ \mathbf{call}\, p) \ =\ (p := \{\!|Q|\!\} ;\ Q)$$

We also like to have the following refinement law.

Monotonicity of assignment

$$P \sqsubseteq Q \ \Rightarrow\ (p := \{\!|P|\!\}) \sqsubseteq (p := \{\!|Q|\!\})$$

Requires a new definition for assignment.

Assignment in higher-order UTP

$$p := \{\!|Q|\!\} \ \widehat{=}\ (\mathbf{true} \vdash (Q \sqsubseteq p')) \wedge (v \sqsubseteq v')$$

where $\alpha(p := \{\!|Q|\!\}) = \{p, p', v, v'\}$

Higher-order UTP (2)

hiJaC

Fundamental Law

$$(p := \{\!|Q|\!\} ;\ \mathbf{call}\, p) \ =\ (p := \{\!|Q|\!\} ;\ Q)$$

We also like to have the following refinement law.

Monotonicity of assignment

$$P \sqsubseteq Q \ \Rightarrow\ (p := \{\!|P|\!\}) \sqsubseteq (p := \{\!|Q|\!\})$$

Requires a new definition for assignment.

Assignment in higher-order UTP

$$p := \{\!|Q|\!\} \ \widehat{=}\ (\mathbf{true} \vdash (Q \sqsubseteq p')) \wedge (v \sqsubseteq v')$$

where $\alpha(p := \{\!|Q|\!\}) = \{p, p', v, v'\}$

Higher-order UTP (2)

hiJaC

Fundamental Law

$$(p := \{\!|Q|\!\} \,;\ \mathbf{call}\, p) \ =\ (p := \{\!|Q|\!\} \,;\ Q)$$

We also like to have the following refinement law.

Monotonicity of assignment

$$P \sqsubseteq Q \ \Rightarrow\ (p := \{\!|P|\!\}) \sqsubseteq (p := \{\!|Q|\!\})$$

Requires a new definition for assignment.

Assignment in higher-order UTP

$$p := \{\!|Q|\!\} \ \widehat{=}\ (\mathbf{true} \vdash (Q \sqsubseteq p')) \wedge (v \sqsubseteq v')$$

where $\alpha(p := \{\!|Q|\!\}) = \{p, p', v, v'\}$

Santos' theory of object-orientation

The UTP theory of object-orientation

hiJaC

Paradigm variables

1. cls of type $\mathbb{P}\, CName$ to record class names;
2. sc of type $CName \leftrightarrow CName$ to record the subclass relation;
3. $attr$ of type $CName \pfun (AttrName \pfun Type)$ to record class attributes.

Healthiness conditions

- Ensure the consistency of the value of those variables;
- For instance, sc must not be cyclic.

Theory Operators

- Declare new classes and methods.
- Redeclare (override) methods in subclasses.

Method redefinition in Santos' work

hiJaC

Particular elegant solution.

- **One** method variable to represents all behaviours of a method as it is redefined in the subclass hierarchy.
- Dynamic binding is resolved by the method itself upon a call to it.

Method specification have (in essence) the following form:

$$m := (p_1 \lhd \mathit{self} \textbf{ is } C_1 \rhd (p_2 \lhd \mathit{self} \textbf{ is } C_2 \rhd (\ldots (p_n \lhd \mathit{self} \textbf{ is } C_n \rhd \bot_{oo}) \ldots)))$$

1. *self* is an auxiliary variable that determines the target of the call;
2. The p_i are basically specifications of the same method, albeit defined in different subclasses C_1, C_2, ..., C_n.
3. The cascade of tests is used to resolve dynamic binding when the method is called on an object.

Method redefinition has to inject a new test $(p \lhd \mathit{self} \textbf{ is } C \rhd \ldots)$ at the right place into this cascade.

Method redefinition in Santos' work

hiJaC

Particular elegant solution.

- **One** method variable to represents all behaviours of a method as it is redefined in the subclass hierarchy.
- Dynamic binding is resolved by the method itself upon a call to it.

Method specification have (in essence) the following form:

$$m := (p_1 \lhd \mathit{self} \textbf{ is } C_1 \rhd (p_2 \lhd \mathit{self} \textbf{ is } C_2 \rhd (\ldots (p_n \lhd \mathit{self} \textbf{ is } C_n \rhd \bot_{oo}) \ldots)))$$

1. *self* is an auxiliary variable that determines the target of the call;
2. The p_i are basically specifications of the same method, albeit defined in different subclasses C_1, C_2, ..., C_n.
3. The cascade of tests is used to resolve dynamic binding when the method is called on an object.

Method redefinition has to inject a new test $(p \lhd \mathit{self} \textbf{ is } C \rhd \ldots)$ at the right place into this cascade.

hiJaC

A program model

Types in Higher-Order UTP

In a sound program model, program values cannot range over arbitrary predicates.

- Predicates have to be well-typed.

Type notion in HO predicates

<**type**> ::= <**program type**> | <**base type**>

<**program type**> ::= *ProcType*(<**alphabet**>)

<**alphabet**> ::= list of (<**variable**> : <**type**>)

<**base type**> ::= *BaseType*(*int*) | *BaseType*(*bool*) | . . .

- Types are non-circular by construction.
- We may have further type constructors for base values, for instance, for composite values like pairs or (finite) sets.
- A precise account of typing is crucial in higher-order UTP.

Types in Higher-Order UTP

In a sound program model, program values cannot range over arbitrary predicates.

- Predicates have to be well-typed.

Type notion in HO predicates

<**type**> ::= <**program type**> | <**base type**>

<**program type**> ::= *ProcType*(<**alphabet**>)

<**alphabet**> ::= list of (<**variable**> : <**type**>)

<**base type**> ::= *BaseType*(*int*) | *BaseType*(*bool*) | . . .

- Types are non-circular by construction.
- We may have further type constructors for base values, for instance, for composite values like pairs or (finite) sets.
- A precise account of typing is crucial in higher-order UTP.

Types in Higher-Order UTP

In a sound program model, program values cannot range over arbitrary predicates.

- Predicates have to be well-typed.

Type notion in HO predicates

$\langle$**type**$\rangle ::= \langle$**program type**$\rangle \mid \langle$**base type**$\rangle$

$\langle$**program type**$\rangle ::= \mathit{ProcType}(\langle$**alphabet**$\rangle)$

$\langle$**alphabet**$\rangle ::=$ list of $(\langle$**variable**$\rangle : \langle$**type**$\rangle)$

$\langle$**base type**$\rangle ::= \mathit{BaseType}(\mathit{int}) \mid \mathit{BaseType}(\mathit{bool}) \mid \ldots$

- Types are non-circular by construction.
- We may have further type constructors for base values, for instance, for composite values like pairs or (finite) sets.
- A precise account of typing is crucial in higher-order UTP.

Types in Higher-Order UTP

In a sound program model, program values cannot range over arbitrary predicates.

- Predicates have to be well-typed.

Type notion in HO predicates

$\langle$**type**$\rangle ::= \langle$**program type**$\rangle \mid \langle$**base type**$\rangle$

$\langle$**program type**$\rangle ::= \mathit{ProcType}(\langle$**alphabet**$\rangle)$

$\langle$**alphabet**$\rangle ::=$ list of $(\langle$**variable**$\rangle : \langle$**type**$\rangle)$

$\langle$**base type**$\rangle ::= \mathit{BaseType}(\mathit{int}) \mid \mathit{BaseType}(\mathit{bool}) \mid \ldots$

- Types are non-circular by construction.
- We may have further type constructors for base values, for instance, for composite values like pairs or (finite) sets.
- A precise account of typing is crucial in higher-order UTP.

Types in Higher-Order UTP

In a sound program model, program values cannot range over arbitrary predicates.

- Predicates have to be well-typed.

Type notion in HO predicates

$\langle$**type**$\rangle ::= \langle$**program type**$\rangle \mid \langle$**base type**$\rangle$

$\langle$**program type**$\rangle ::= \mathit{ProcType}(\langle$**alphabet**$\rangle)$

$\langle$**alphabet**$\rangle ::=$ list of $(\langle$**variable**$\rangle : \langle$**type**$\rangle)$

$\langle$**base type**$\rangle ::= \mathit{BaseType}(\mathit{int}) \mid \mathit{BaseType}(\mathit{bool}) \mid \ldots$

- Types are non-circular by construction.
- We may have further type constructors for base values, for instance, for composite values like pairs or (finite) sets.
- A precise account of typing is crucial in higher-order UTP.

Types in Higher-Order UTP

In a sound program model, program values cannot range over arbitrary predicates.

- Predicates have to be well-typed.

Type notion in HO predicates

$\langle$**type**$\rangle ::= \langle$**program type**$\rangle \mid \langle$**base type**$\rangle$

$\langle$**program type**$\rangle ::= \mathit{ProcType}(\langle$**alphabet**$\rangle)$

$\langle$**alphabet**$\rangle ::=$ list of $(\langle$**variable**$\rangle : \langle$**type**$\rangle)$

$\langle$**base type**$\rangle ::= \mathit{BaseType}(\mathit{int}) \mid \mathit{BaseType}(\mathit{bool}) \mid \ldots$

- Types are non-circular by construction.
- We may have further type constructors for base values, for instance, for composite values like pairs or (finite) sets.
- A precise account of typing is crucial in higher-order UTP.

Example of an ill-typed predicate

We consider the predicate

$$p = \{\!|x := x + 1 \,;\ \textbf{call}\, p|\!\}$$

Is it well-typed?

$$type(p)$$
$$= \text{“type of lhs and rhs of equality have to agree”}$$
$$type(\{\!|x := x + 1 \,;\ \textbf{call}\, p|\!\})$$
$$= \text{“derivation of procedure type”}$$
$$ProcType(\langle x : BaseType(int), p : , \ldots \rangle)$$

No, since $type(p)$ occurs recursively in its definition.

Recursion (or mutual recursion) is prohibited.

- But we can still use standard fixed-point constructions.

Example of an ill-typed predicate

We consider the predicate

$$p = \{\!|x := x + 1 \,;\ \mathbf{call}\, p|\!\}$$

Is it well-typed?

$type(p)$

$=$ "type of lhs and rhs of equality have to agree"

$type(\{\!|x := x + 1 \,;\ \mathbf{call}\, p|\!\})$

$=$ "derivation of procedure type"

$ProcType(\langle x : BaseType(int), p : , \ldots \rangle)$

No, since $type(p)$ occurs recursively in its definition.

Recursion (or mutual recursion) is prohibited.

- But we can still use standard fixed-point constructions.

Example of an ill-typed predicate

We consider the predicate

$$p = \{\!|x := x + 1 \,;\ \mathbf{call}\, p|\!\}$$

Is it well-typed?

$type(p)$

$=$ "type of lhs and rhs of equality have to agree"

$type(\{\!|x := x + 1 \,;\ \mathbf{call}\, p|\!\})$

$=$ "derivation of procedure type"

$ProcType(\langle x : BaseType(int), p : type(p), \ldots \rangle)$

No, since $type(p)$ occurs recursively in its definition.

Recursion (or mutual recursion) is prohibited.

- But we can still use standard fixed-point constructions.

Example of an ill-typed predicate

We consider the predicate

$$p = \{\!|x := x + 1 \,;\ \textbf{call}\, p|\!\}$$

Is it well-typed?

$type(p)$

$=$ "type of lhs and rhs of equality have to agree"

$type(\{\!|x := x + 1 \,;\ \textbf{call}\, p|\!\})$

$=$ "derivation of procedure type"

$ProcType(\langle x : BaseType(int), p : type(p), \ldots \rangle)$

No, since $type(p)$ occurs recursively in its definition.

Recursion (or mutual recursion) is prohibited.

- But we can still use standard fixed-point constructions.

Example of an ill-typed predicate

We consider the predicate

$$p = \{\!|x := x + 1 \,;\ \mathbf{call}\, p|\!\}$$

Is it well-typed?

$type(p)$

$=$ "type of lhs and rhs of equality have to agree"

$type(\{\!|x := x + 1 \,;\ \mathbf{call}\, p|\!\})$

$=$ "derivation of procedure type"

$ProcType(\langle x : BaseType(int), p : , \ldots \rangle)$

No, since $type(p)$ occurs recursively in its definition.

Recursion (or mutual recursion) is prohibited.

- But we can still use standard fixed-point constructions.

Example of an ill-typed predicate

We consider the predicate

$$p = \{\!|x := x + 1 \,;\ \mathbf{call}\, p|\!\}$$

Is it well-typed?

$type(p)$

$=$ "type of lhs and rhs of equality have to agree"

$type(\{\!|x := x + 1 \,;\ \mathbf{call}\, p|\!\})$

$=$ "derivation of procedure type"

$ProcType(\langle x : BaseType(int), p : , \ldots \rangle)$

No, since $type(p)$ occurs recursively in its definition.

Recursion (or mutual recursion) is prohibited.

- But we can still use standard fixed-point constructions.

Consistency of the program model

Main Question

- Is the finitary natures of types sufficient to ensure consistency?
  ↪ A result left implicit in the UTP book.
  ↪ Proved here by inductive construction of a program model.

Motivation

1. Justifies the use of arbitrary theories as program values.
2. Supports sound generalisations of program value notions.
   ↪ Dealing with procedures through functional abstraction.
   ↪ Mixing syntactic and semantic notions.
   ↪ Elaborating the notion of predicate types.
3. Paves the way for mechanisation in a theorem prover.

Consistency of the program model

hiJaC

Main Question

- Is the finitary natures of types sufficient to ensure consistency?
  ↪ A result left implicit in the UTP book.
  ↪ Proved here by inductive construction of a program model.

Motivation

1. Justifies the use of arbitrary theories as program values.
2. Supports sound generalisations of program value notions.
   ↪ Dealing with procedures through functional abstraction.
   ↪ Mixing syntactic and semantic notions.
   ↪ Elaborating the notion of predicate types.
3. Paves the way for mechanisation in a theorem prover.

Consistency: proof sketch

We first define the notion of the rank of a type.

Inductively definition over the type structure.

$$rank(BaseType(t)) = 0$$

$$rank(ProcType(\text{list of } [v_1 : t_1, v_2 : t_2, \dots])) = max\{rank(t_1), rank(t_2), \dots\} + 1$$

Intuitively, the rank determines the maximal nesting level of program abstractions in a predicate.

Motivation

1. Partitions the set of HO predicates.
2. Subsequently used in defining a theory of methods.

$\hookrightarrow$ Every higher-order predicate must have a finite rank.

Consistency: proof sketch

We first define the notion of the rank of a type.

Inductively definition over the type structure.

$$rank(BaseType(t)) = 0$$

$$rank(ProcType(\text{list of } [v_1 : t_1, v_2 : t_2, \dots])) = max\{rank(t_1), rank(t_2), \dots\} + 1$$

Intuitively, the rank determines the maximal nesting level of program abstractions in a predicate.

Motivation

1. Partitions the set of HO predicates.
2. Subsequently used in defining a theory of methods.

$\hookrightarrow$ Every higher-order predicate must have a finite rank.

Consistency: proof sketch

We first define the notion of the rank of a type.

Inductively definition over the type structure.

$$rank(BaseType(t)) = 0$$

$$rank(ProcType(\text{list of } [v_1 : t_1, v_2 : t_2, \dots])) = max\{rank(t_1), rank(t_2), \dots\} + 1$$

Intuitively, the rank determines the maximal nesting level of program abstractions in a predicate.

Motivation

1. Partitions the set of HO predicates.
2. Subsequently used in defining a theory of methods.

$\hookrightarrow$ Every higher-order predicate must have a finite rank.

More about ranks

Ranks of other entities

1. Rank of a variable is the rank of its type.
   $rank(v : t) = rank(t)$
2. Rank of an alphabet is the maximum rank of its variables.
   $rank\{v_1, v_2, \dots\} = max\{v_1, v_2, \dots\}$
3. Rank of a predicate is just the rank of its alphabet.
   $rank(p) = rank(\alpha\, p)$

Examples

1. $x := 1$ is a rank 0 predicate;
2. $m_1 := (\!|x := 1|\!)$ is a rank 1 predicate;
3. $m_1 := (\!|x := 1|\!) \land m_2 := (\!|\mathbf{call}\, m_1|\!)$ is a rank 2 predicate.

More about ranks

Ranks of other entities

1. Rank of a variable is the rank of its type.
   $rank(v : t) = rank(t)$
2. Rank of an alphabet is the maximum rank of its variables.
   $rank\,\{v_1, v_2, \dots\} = max\,\{v_1, v_2, \dots\}$
3. Rank of a predicate is just the rank of its alphabet.
   $rank(p) = rank(\alpha\, p)$

Examples

1. $x := 1$ is a rank 0 predicate;
2. $m_1 := \{\!|x := 1|\!\}$ is a rank 1 predicate;
3. $m_1 := \{\!|x := 1|\!\} \land m_2 := \{\!|\mathbf{call}\, m_1|\!\}$ is a rank 2 predicate.

More about ranks

Ranks of other entities

1. Rank of a variable is the rank of its type.
   $rank(v : t) = rank(t)$
2. Rank of an alphabet is the maximum rank of its variables.
   $rank\,\{v_1, v_2, \dots\} = max\,\{v_1, v_2, \dots\}$
3. Rank of a predicate is just the rank of its alphabet.
   $rank(p) = rank(\alpha\, p)$

Examples

1. $x := 1$ is a rank 0 predicate;
2. $m_1 := \{\!|x := 1|\!\}$ is a rank 1 predicate;
3. $m_1 := \{\!|x := 1|\!\} \land m_2 := \{\!|\mathbf{call}\, m_1|\!\}$ is a rank 2 predicate.

More about ranks

Ranks of other entities

1. Rank of a variable is the rank of its type.
   $rank(v : t) = rank(t)$
2. Rank of an alphabet is the maximum rank of its variables.
   $rank\,\{v_1, v_2, \dots\} = max\,\{v_1, v_2, \dots\}$
3. Rank of a predicate is just the rank of its alphabet.
   $rank(p) = rank(\alpha\, p)$

Examples

1. $x := 1$ is a rank 0 predicate;
2. $m_1 := \{\!|x := 1|\!\}$ is a rank 1 predicate;
3. $m_1 := \{\!|x := 1|\!\} \land m_2 := \{\!|\mathbf{call}\, m_1|\!\}$ is a rank 2 predicate.

Constructive definitions of HO predicates

We first introduce a function $pred(n)$ for the predicates whose rank is less or equal than a given rank n.

Inductive definition of *pred*

$pred(0) = StdPred$

$pred(n+1) = lift(pred(n), pred(n))$

where $StdPred$ are the standard (non higher-order) predicates.

It rests on the existence of a lifting function $lift(ps, vs)$.

Informal definition of HO lifting

1. $lift(\mathbf{ps}, vs)$ takes a set of predicates **ps**;
2. lifts them into a set of predicates that introduce program variables that range over the values in vs.
   ↪ Values in **vs** are predicates themselves.

Constructive definitions of HO predicates

We first introduce a function $pred(n)$ for the predicates whose rank is less or equal than a given rank n.

Inductive definition of *pred*

$pred(0) = StdPred$

$pred(n+1) = lift(pred(n), pred(n))$

where *StdPred* are the standard (non higher-order) predicates.

It rests on the existence of a lifting function $lift(ps, vs)$.

Informal definition of HO lifting

1. $lift(ps, \mathbf{vs})$ takes a set of predicates ps;
2. lifts them into a set of predicates that introduce program variables that range over the values in **vs**.
   ↪ Values in **vs** are predicates themselves.

Constructive definitions of HO predicates

We first introduce a function $pred(n)$ for the predicates whose rank is less or equal than a given rank n.

Inductive definition of *pred*

$pred(0) = StdPred$

$pred(n+1) = lift(pred(n), pred(n))$

where *StdPred* are the standard (non higher-order) predicates.

It rests on the existence of a lifting function $lift(ps, vs)$.

Informal definition of HO lifting

1. $lift(ps, \mathbf{vs})$ takes a set of predicates ps;
2. lifts them into a set of predicates that introduce program variables that range over the values in **vs**.
   $\hookrightarrow$ Values in **vs** are predicates themselves.

Lifting example

We have that

$$pred(1) = lift(pred(0), pred(0)) = lift(StdPred, StdPred) \quad ,$$

the standard predicates augmented with variables whose values can range over basic values and standard predicates.

For example, $m = \{\!|x := y|\!\} \land y = 1 \in pred(1)$.

Property of lifting

- $lift\,(ps, ps)$ admits predicates one rank higher than those in ps.
- Admits new program values into the model.
- Constructs the entire set of HO predicates.

$\hookrightarrow$ Can we say more about the lifting function?

Lifting example

We have that

$$pred(1) = lift(pred(0), pred(0)) = lift(StdPred, StdPred) \quad ,$$

the standard predicates augmented with variables whose values can range over basic values and standard predicates.

For example, $m = \{\!|x := y|\!\} \land y = 1 \in pred(1)$.

Property of lifting

- $lift\,(ps, ps)$ admits predicates one rank higher than those in ps.
- Admits new program values into the model.
- Constructs the entire set of HO predicates.

$\hookrightarrow$ Can we say more about the lifting function?

Lifting example

We have that

$$pred(1) = lift(pred(0), pred(0)) = lift(StdPred, StdPred) \quad ,$$

the standard predicates augmented with variables whose values can range over basic values and standard predicates.

For example, $m = \{\!|x := y|\!\} \land y = 1 \in pred(1)$.

Property of lifting

- $lift\,(ps, ps)$ admits predicates one rank higher than those in ps.
- Admits new program values into the model.
- Constructs the entire set of HO predicates.

$\hookrightarrow$ Can we say more about the lifting function?

Axiomatic characterisation of lifting

A precise definition of *lift* can only be given with respect to a semantic model / encoding of predicates.

$\hookrightarrow$ We may, nevertheless, describe its properties axiomatically.

Axiomatic characterisation: $hps = lift(ps, vs)$

A1 $ps \subseteq hps$

A2 $\forall m : ProcType(I) \bullet \forall v : vs \mid SetOf(I) = [\alpha]\, v \bullet m\, [=]\, v \in hps$

A3 $\forall ps \subseteq hps \bullet [\sqcap]\, ps \in hps$

A4 $\forall ps \subseteq hps \bullet [\sqcup]\, ps \in hps$

A5 $[\sqcap]$ and $[\sqcup]$ are the meet and join of a complete lattice $[\sqsubseteq]$

1. Boxed operators provided by the core predicate model.
2. Axioms capture completeness and correctness properties
   $\hookrightarrow$ We want to retain the property of a complete lattice.

Axiomatic characterisation of lifting

A precise definition of *lift* can only be given with respect to a semantic model / encoding of predicates.

$\hookrightarrow$ We may, nevertheless, describe its properties axiomatically.

Axiomatic characterisation: $hps = lift(ps, vs)$

A1 $ps \subseteq hps$

A2 $\forall m : ProcType(l) \bullet \forall v : vs \mid SetOf(l) = \boxed{\alpha} v \bullet m \boxed{=} v \in hps$

A3 $\forall ps \subseteq hps \bullet \boxed{\sqcap} ps \in hps$

A4 $\forall ps \subseteq hps \bullet \boxed{\sqcup} ps \in hps$

A5 $\boxed{\sqcap}$ and $\boxed{\sqcup}$ are the meet and join of a complete lattice $\boxed{\sqsubseteq}$

1. Boxed operators provided by the core predicate model.
2. Axioms capture completeness and correctness properties
   $\hookrightarrow$ We want to retain the property of a complete lattice.

Axiomatic characterisation of lifting

A precise definition of *lift* can only be given with respect to a semantic model / encoding of predicates.

$\hookrightarrow$ We may, nevertheless, describe its properties axiomatically.

Axiomatic characterisation: $hps = lift(ps, vs)$

A1 $ps \subseteq hps$

A2 $\forall m : ProcType(l) \bullet \forall v : vs \mid SetOf(l) = \boxed{\alpha} v \bullet m \boxed{=} v \in hps$

A3 $\forall ps \subseteq hps \bullet \boxed{\sqcap} ps \in hps$

A4 $\forall ps \subseteq hps \bullet \boxed{\sqcup} ps \in hps$

A5 $\boxed{\sqcap}$ and $\boxed{\sqcup}$ are the meet and join of a complete lattice $\boxed{\sqsubseteq}$

1. Boxed operators provided by the core predicate model.
2. Axioms capture completeness and correctness properties
   $\hookrightarrow$ We want to retain the property of a complete lattice.

Conclusion of the proof

Define the cumulative set of HO predicates

$$pred = \bigcup \{ n \in \mathbb{N} \bullet pred(n) \}$$

1. Contains all predicates of any rank.
2. Precisely the HO predicates in the UTP book.

Now $\{\!|\_|\!\}$ only has to be injective on the predicates that are well-formed.

We trivially define...

$$\{\!|\_|\!\} =_{\mathrm{df}} (\lambda\, p : pred \bullet p) \quad \text{where} \quad \mathrm{dom}\, \{\!|\_|\!\} = pred$$

- Simply the identity on *pred*, thus injective.
- Serves as a sound **type constructor** for program values.

We have shown that it is safe to treat program values as semantics, that is the predicates of some arbitrary UTP theory.

Conclusion of the proof

Define the cumulative set of HO predicates

$$pred = \bigcup \{ n \in \mathbb{N} \bullet pred(n) \}$$

1. Contains all predicates of any rank.
2. Precisely the HO predicates in the UTP book.

Now $\{\!|\_|\!\}$ only has to be injective on the predicates that are well-formed.

We trivially define...

$$\{\!|\_|\!\} =_{\mathrm{df}} (\lambda\, p : pred \bullet p) \quad \text{where} \quad \mathrm{dom}\, \{\!|\_|\!\} = pred$$

- Simply the identity on *pred*, thus injective.
- Serves as a sound **type constructor** for program values.

We have shown that it is safe to treat program values as semantics, that is the predicates of some arbitrary UTP theory.

Mixing syntax and semantics

Required in Santos' approach to method redefinition.

Supported by altering the iterative definition of $pred(n)$

$$pred(n+1) = lift(pred(n), embed(pred(n)))$$

where *embed* realises the syntactic embedding of the semantic entities.

1. The definition of lift remains fundamentally the same.
2. The $\boxed{\alpha}$ function (**A2**) now has to extract the alphabet of a predicate embedded in a segment of syntax. $\hookrightarrow$ Not a problem.

Implications

1. Result of the lifting is still a predicate set.
2. Does not affect the validity of the previous proof.
3. Semantics of the **call** operation might need to be adjusted.

Example of embedded syntax

Let us recall the shape of method definitions in Santos' theory.

$$(p_1 \lhd self \textbf{ is } C_1 \rhd (p_2 \lhd self \textbf{ is } C_2 \rhd (p_3 \lhd self \textbf{ is } C_n \rhd \bot_{oo})))$$

- The p_i are semantic entities while the rest is syntax.
- Redefinition requires transforming this piece of syntax.

Solution: embed this syntax using a (generic) free type

$$METH[PRED] ::= CondSytx \langle\langle METH \times CVALUE \times METH \rangle\rangle \mid BotSytx \mid Body \langle\langle PRED \rangle\rangle$$

1. The constructor *Body* injects the semantic object (a predicate encoding the method body) into the syntactic program value.
2. The semantic predicate domain is provided by the type *PRED*.
3. In particular, we have $embed(ps) = METH[ps]$.

hiJaC

A theory of methods

Method definition revisited

Consider the following higher-order predicate.

$$S_1 \mathrel{\widehat{=}} m_1 := \{\!|x := x + 1|\!\} \; ; \; m_2 := \{\!|x := x + 2 \; ; \; \textbf{call}\, m_1|\!\}$$

We observe that

- m_1 is a rank 1 variable;
- m_2 is a rank 2 variable;
- hence, the predicate S_1 is a rank 2 predicate.

Question

Is rank 2 sufficient to encode **all** possible object-oriented programs?

The answer is "no". Consider the predicate S_2 below.

Counterexample

$S_2 \mathrel{\widehat{=}} S_1 \; ; \; m_3 := \{\!|x := x + 3 \; ; \; \textbf{call}\, m_2|\!\}$ where m_2 is of rank 3.

Method definition revisited

hiJaC

Consider the following higher-order predicate.

$$S_1 \mathrel{\widehat{=}} m_1 := \{\!|x := x + 1|\!\} \,;\; m_2 := \{\!|x := x + 2 \,;\; \textbf{call}\, m_1|\!\}$$

We observe that

- m_1 is a rank 1 variable;
- m_2 is a rank 2 variable;
- hence, the predicate S_1 is a rank 2 predicate.

Question

Is rank 2 sufficient to encode **all** possible object-oriented programs?

The answer is "no". Consider the predicate S_2 below.

Counterexample

$S_2 \mathrel{\widehat{=}} S_1 \,;\; m_3 := \{\!|x := x + 3 \,;\; \textbf{call}\, m_2|\!\}$ where m_2 is of rank 3.

Method definition complications

$$S_2 \mathrel{\widehat{=}} S_1 \,;\; m_3 := \{\!|x := x + 3 \,;\; \textbf{call}\, m_2|\!\}$$

Observations

1. The type of m_3 depends on the type of m_2.
2. Types of the called methods need to be known in advance.
3. Method redefinition in this form is not compositional.

To recover the situation...

- Composed predicate needs to be a function applied to a type.
- Mechanisms required to instantiate such type parameters when encoding real object-oriented programs.
- Not unsound with our result on ranks but considerably complicates the theory of object-orientation and its application.

Method redefinition complications

Let us introduce another method m_4 and then redefine m_1.

$$S_3 \mathrel{\widehat{=}} S_2 \,;\; m_4 := \{\!|x := x + 4|\!\} \,;\; m_1 := \{\!|\mathbf{call}\, m_4|\!\}$$

Variable m_1 above cannot be the same m_1 as in the definition of S_1.

- In $S1$, it was introduced as a rank 1 variable;
- Here, its rank has to be (at least) 2;
- Clearly, variables are different if they have different ranks (!)

As a consequence...

1. We have to know in advance that m_1 is going to be redefined by a program with a higher rank.
2. Such knowledge is doubtful in practice.
3. Again shows non-compositionality of a direct / naïve approach.

Method redefinition complications

Only solution: redefine m_1 *together with all previous method definitions* that depend on its value.

Redefinition of m_1 with all methods calling it:

$$S_3' \mathrel{\widehat{=}} \left(\begin{array}{l} m_1 := \{\!|\, \textbf{call}\, m_4 \,|\!\}\ ;\ \ m_2 := \{\!|\, x := x + 2\ ;\ \ \textbf{call}\, m_1 \,|\!\}; \\ m_3 := \{\!|\, x := x + 3\ ;\ \ \textbf{call}\, m_2 \,|\!\}\ ;\ \ m_4 := \{\!|\, x := x + 4 \,|\!\} \end{array} \right)$$

We note that m_2 and m_3 are redefined alongside m_1.

But there is a problem even when redefinition is at the same rank.

Let us redefine m_2 instead (whose rank is 2)

$$S_4 \mathrel{\widehat{=}} S_2\ ;\ \ m_4 := \{\!|\, x := x + 4 \,|\!\}\ ;\ \ m_2 := \{\!|\, \textbf{call}\, m_4 \,|\!\}$$

1. Originally m_4 was not in the alphabet of m_2;
2. Introducing it during redefinition changes the type of m_2;
3. Same problem as before. ☹

Solution: Method alphabets include all other methods? This is tricky!

A theory of methods

Main features

1. Observational variables are program variables representing methods.
2. We only include program variables at rank 1 and rank 2.
3. Method definitions constrain rank 2 variables.
4. Calls to methods are **always** to rank 1 variables.

To facilitate the presentation of our theory, we introduce...

A little bit of notation

Overbars are used to highlight the rank of a method variable.

- $\overline{m}$ is a rank 1 method variable; and
- $\overline{\overline{m}}$ is a rank 2 method variable.

$\hookrightarrow$ In quantifications, overbars implicitly constrain the rank of the bound variable. For instance, $\forall \overline{m} \bullet P(\overline{m}) \equiv \forall m \mid rank(m) = 1 \bullet P(m)$.

A theory of methods

hiJaC

Main features

1. Observational variables are program variables representing methods.
2. We only include program variables at rank 1 and rank 2.
3. Method definitions constrain rank 2 variables.
4. Calls to methods are **always** to rank 1 variables.

To facilitate the presentation of our theory, we introduce...

A little bit of notation

Overbars are used to highlight the rank of a method variable.

- $\overline{m}$ is a rank 1 method variable; and
- $\overline{\overline{m}}$ is a rank 2 method variable.

$\hookrightarrow$ In quantifications, overbars implicitly constrain the rank of the bound variable. For instance, $\forall\, \overline{m} \bullet P(\overline{m}) \equiv \forall\, m \mid rank(m) = 1 \bullet P(m)$.

Examples of encoding methods

Encoding of $S1$

$$T_1 \mathrel{\widehat{=}} \overline{\overline{m}}_1 := \{\!|x := x + 1|\!\} \,;\; \overline{\overline{m}}_2 := \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}_1|\!\}$$

We note that $\overline{m_1}$ and $\overline{\overline{m_1}}$ are not the same variable.

Encoding of $S2$

$$T_2 \mathrel{\widehat{=}} T_1 \,;\; \overline{\overline{m}}_3 := \{\!|x := x + 3 \,;\; \mathbf{call}\, \overline{m}_2|\!\}$$

We note that m_3 was a rank 3 variable in $S2$; here it is a rank 2 variable.

Observations

- Approach establishes uniformity of method definitions.
- Rank of method variables does not increase with
  1. subsequent definitions of methods;
  2. upon method **re**definition.
- However, defined and called method variable are not the same (!)

Healthiness condition

We have a single healthiness condition **HM**:

$$\mathbf{HM}(P) \;=\; P \wedge (\forall\, \overline{m}\; \overline{\overline{m}} \mid \{\overline{m}, \overline{\overline{m}}\} \subseteq \alpha\, P \bullet [\mathbf{call}\, \overline{m} \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}]_0)$$

where $[\_]_0$ is closure over standard (program) variables.

Rationale

- Two method variables of the same name, but **at different ranks**, have to be consistent in terms of the constraints they impose on method variables.
- **HM** defines a family of equations; interpretation of a method as a standard predicate falls out when quantifying over rank 2 variables.

Naturally supports recursion:

$\hookrightarrow$ The defined and called method variables are always different (!)

Fundamental equivalence

Conjecture

Method definitions in our theory correspond to an encoding of methods in terms of **weakest fixed points** of a recursive equation...

- ...using recursion parameters X, Y, ... for method variables.

For instance, the predicate

$$\forall \overline{\overline{m_1}}\ \overline{\overline{m_2}} \bullet \mathbf{HM}(\overline{\overline{m}}_1 := \{\!|x := x + 1|\!\}\ ;\ \overline{\overline{m}}_2 := \{\!|x := x + 2\ ;\ \mathbf{call}\, \overline{\overline{m}}_1|\!\})$$

is equivalent to the concurrent assignment

$$\overline{\overline{m}}_1, \overline{\overline{m}}_2 := \{\!|\, \mu\, X, Y \bullet \langle x := x + 1, (x := x + 2\ ;\ X)\rangle \,|\!\}.$$

A formal proof of this conjecture is still a current work.

General Note

↪ Santos' work used weakest fixed point constructs in standard predicates to support recursion. The theory of methods offers both options.

Properties of the theory

The healthiness condition of our theory is 'almost' conjunctive.

Defining $\gamma_{\mathrm{HM}}(a)$ as

$$\gamma_{\mathrm{HM}}\,(a) \;=\; (\forall\, \overline{m}\; \overline{\overline{m}} \mid \{\overline{m}, \overline{\overline{m}}\} \subseteq a \bullet [\mathbf{call}\,\overline{m} \Leftrightarrow \mathbf{call}\,\overline{\overline{m}}]_0)$$

we have that

$$\mathbf{HM}(P) = P \wedge \gamma_{\mathrm{HM}}\,(\alpha\, P).$$

Though $\gamma_{\mathrm{HM}}(a)$ is not constant it only depends on the alphabet of P.

Conjunctive healthiness conditions

1. Can be expressed as $\mathbf{CH}(P) = P \wedge \gamma$.
2. Give rise to theories with many desirable algebraic properties.
   $\hookrightarrow$ For instance closure under most operators.

Compatible alphabets

We can recover essential theory properties of conjunctive healthiness conditions. $\hookrightarrow$ They are, however, subject to additional caveats.

Compatibility of alphabets

Two alphabets a_1 and a_2 are compatible if, and only if,

$$\forall\, \overline{m}\ \overline{\overline{m}} \mid \{\overline{m}, \overline{\overline{m}}\} \subseteq a_1 \cup a_2 \bullet (\overline{m} \in a_1 \land \overline{\overline{m}} \in a_2) \Leftrightarrow (\overline{\overline{m}} \in a_1 \land \overline{m} \in a_2)$$

If alphabets share a method variable with the same name but at different ranks, each alphabet has to include both instances of that variable.

Examples

- $\{\overline{m}_1, \overline{\overline{m}}_1\}$ and $\{\overline{m}_1, \overline{\overline{m}}_1\}$ are compatible;
- $\{\overline{m}_1, \overline{\overline{m}}_1\}$ and $\{\overline{m}_2, \overline{\overline{m}}_2\}$ are compatible;
- $\{\overline{m}_1\}$ and $\{\overline{\overline{m}}_2\}$ are compatible; but
- $\{\overline{m}_1\}$ and $\{\overline{\overline{m}}_1\}$ are not compatible.

Closure of operators

An important property of γ_{HM} is formulated by the following law.

Distribution law for γ_{HM}

Let a_1 and a_2 be compatible alphabets. Then we have

$$\gamma_{\mathrm{HM}}(a_1 \cup a_2) = \gamma_{\mathrm{HM}}(a_1) \wedge \gamma_{\mathrm{HM}}(a_2)$$

With this we can prove the following two closure laws.

Closure law for conjunction

Let P and Q be **HM**-healthy predicates with compatible alphabets.

$P \wedge Q$ is a **HM**-healthy predicate.

Closure law for disjunction

Let P and Q be **HM**-healthy predicates with equal alphabets.

$P \vee Q$ is a **HM**-healthy predicate.

Closure of operators

If we restrict ourselves to predicates over the same alphabet, all theorems for conjunctive healthiness conditions continue to hold.

- We can treat $\gamma_{\mathrm{HM}}(\alpha\, P)$ as a constant predicate then.

Properties assuming fixed alphabets

- Closure under sequential composition.
- Theory predicates form a complete lattice.
  $\hookrightarrow$ $\mathbf{HM}(p)$ is idempotent and monotonic.
- Additional properties related to designs.
  $\hookrightarrow$ Proved by Harwood in: "A Theory of Pointers for the UTP".

Discovery of further laws is on-going research.

Closure of operators

If we restrict ourselves to predicates over the same alphabet, all theorems for conjunctive healthiness conditions continue to hold.

- We can treat $\gamma_{\mathrm{HM}}(\alpha\, P)$ as a constant predicate then.

Properties assuming fixed alphabets

- Closure under sequential composition.
- Theory predicates form a complete lattice.
  $\hookrightarrow$ $\mathbf{HM}(p)$ is idempotent and monotonic.
- Additional properties related to designs.
  $\hookrightarrow$ Proved by Harwood in: "A Theory of Pointers for the UTP".

Discovery of further laws is on-going research.

Proof example

Let us consider the method definitions by $T1$.

$$T1 \equiv \mathbf{HM}(\overline{m}_1 := \{\!|x := x + 1|\!\} \,;\; \overline{\overline{m}}_2 := \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}_1|\!\})$$

$\equiv$ "unfolding definition of **HM**, let $\gamma^*_{\mathrm{HM}} \mathrel{\widehat{=}} \gamma_{\mathrm{HM}}(\{\overline{m}_1, \overline{\overline{m}}_2, \overline{m}_1, \overline{m}_2\})$"

$$(\overline{m}_1 := \{\!|x := x + 1|\!\} \,;\; \overline{\overline{m}}_2 := \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}_1|\!\}) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "unfolding sequential compositions and assignments, one-point rule"

$$(\overline{m}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\}) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "method variable consistency law, predicate is **HM**-healthy"

$$\left(\begin{array}{l} \overline{m}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\, \overline{m}'_1 \Leftrightarrow \mathbf{call}\, \overline{m}'_1) \wedge (\mathbf{call}\, \overline{\overline{m}}'_2 \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}'_2) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "one-point rule using $\overline{m}'_1 = \{\!|x := x + 1|\!\}$ and $\overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\}$"

$$\left(\begin{array}{l} \overline{m}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\, \overline{m}'_1 \Leftrightarrow \mathbf{call}\{\!|x := x + 1|\!\}) \wedge \\ (\mathbf{call}\, \overline{\overline{m}}'_2 \Leftrightarrow \mathbf{call}\{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\}) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "..."

Proof example

Let us consider the method definitions by $T1$.

$$T1 \equiv \mathbf{HM}(\overline{\overline{m}}_1 := \{\!|x := x + 1|\!\} \,;\; \overline{\overline{m}}_2 := \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}_1|\!\})$$

$\equiv$ "unfolding definition of **HM**, let $\gamma^*_{\mathrm{HM}} \mathrel{\widehat{=}} \gamma_{\mathrm{HM}}(\{\overline{\overline{m}}_1, \overline{\overline{m}}_2, \overline{m}_1, \overline{m}_2\})$"

$$(\overline{\overline{m}}_1 := \{\!|x := x + 1|\!\} \,;\; \overline{\overline{m}}_2 := \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}_1|\!\}) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "unfolding sequential compositions and assignments, one-point rule"

$$(\overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\}) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "method variable consistency law, predicate is **HM**-healthy"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\, \overline{m}'_1 \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}'_1) \wedge (\mathbf{call}\, \overline{m}'_2 \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}'_2) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "one-point rule using $\overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\}$ and $\overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\}$"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\, \overline{m}'_1 \Leftrightarrow \mathbf{call}\{\!|x := x + 1|\!\}) \wedge \\ (\mathbf{call}\, \overline{m}'_2 \Leftrightarrow \mathbf{call}\{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\}) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "..."

Proof example

Let us consider the method definitions by $T1$.

$$T1 \equiv \mathbf{HM}(\overline{\overline{m}}_1 := \{\!|x := x + 1|\!\} \,;\; \overline{\overline{m}}_2 := \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}_1|\!\})$$

$\equiv$ "unfolding definition of **HM**, let $\gamma^*_{\mathrm{HM}} \mathrel{\widehat{=}} \gamma_{\mathrm{HM}}(\{\overline{\overline{m}}_1, \overline{\overline{m}}_2, \overline{m}_1, \overline{m}_2\})$"

$$(\overline{\overline{m}}_1 := \{\!|x := x + 1|\!\} \,;\; \overline{\overline{m}}_2 := \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}_1|\!\}) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "unfolding sequential compositions and assignments, one-point rule"

$$(\overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\}) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "method variable consistency law, predicate is **HM**-healthy"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\, \overline{m}'_1 \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}'_1) \wedge (\mathbf{call}\, \overline{m}'_2 \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}'_2) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "one-point rule using $\overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\}$ and $\overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\}$"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\, \overline{m}'_1 \Leftrightarrow \mathbf{call}\{\!|x := x + 1|\!\}) \wedge \\ (\mathbf{call}\, \overline{m}'_2 \Leftrightarrow \mathbf{call}\{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\}) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "..."

Proof example

Let us consider the method definitions by $T1$.

$$T1 \equiv \mathbf{HM}(\overline{\overline{m}}_1 := \{\!|x := x + 1|\!\} \,;\; \overline{\overline{m}}_2 := \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}_1|\!\})$$

$\equiv$ "unfolding definition of **HM**, let $\gamma^*_{\mathrm{HM}} \mathrel{\widehat{=}} \gamma_{\mathrm{HM}}(\{\overline{\overline{m}}_1, \overline{\overline{m}}_2, \overline{m}_1, \overline{m}_2\})$"

$$(\overline{\overline{m}}_1 := \{\!|x := x + 1|\!\} \,;\; \overline{\overline{m}}_2 := \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}_1|\!\}) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "unfolding sequential compositions and assignments, one-point rule"

$$(\overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\}) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "method variable consistency law, predicate is **HM**-healthy"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\, \overline{m}'_1 \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}'_1) \wedge (\mathbf{call}\, \overline{m}'_2 \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}'_2) \end{array} \right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "one-point rule using $\overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\}$ and $\overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\}$"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\, \overline{m}'_1 \Leftrightarrow \mathbf{call}\, \{\!|x := x + 1|\!\}) \wedge \\ (\mathbf{call}\, \overline{m}'_2 \Leftrightarrow \mathbf{call}\, \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\}) \end{array} \right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "..."

Proof example

Let us consider the method definitions by $T1$.

$$T1 \equiv \mathbf{HM}(\overline{\overline{m}}_1 := \{\!|x := x + 1|\!\} \,;\; \overline{\overline{m}}_2 := \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}_1|\!\})$$

$\equiv$ "unfolding definition of **HM**, let $\gamma^*_{\mathrm{HM}} \mathrel{\widehat{=}} \gamma_{\mathrm{HM}}(\{\overline{\overline{m}}_1, \overline{\overline{m}}_2, \overline{m}_1, \overline{m}_2\})$"

$$(\overline{\overline{m}}_1 := \{\!|x := x + 1|\!\} \,;\; \overline{\overline{m}}_2 := \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}_1|\!\}) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "unfolding sequential compositions and assignments, one-point rule"

$$(\overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\}) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "method variable consistency law, predicate is **HM**-healthy"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\, \overline{m}'_1 \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}'_1) \wedge (\mathbf{call}\, \overline{m}'_2 \Leftrightarrow \mathbf{call}\, \overline{\overline{m}}'_2) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "one-point rule using $\overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\}$ and $\overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\}$"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x + 1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\, \overline{m}'_1 \Leftrightarrow \mathbf{call}\{\!|x := x + 1|\!\}) \wedge \\ (\mathbf{call}\, \overline{m}'_2 \Leftrightarrow \mathbf{call}\{\!|x := x + 2 \,;\; \mathbf{call}\, \overline{m}'_1|\!\}) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "..."

Proof example continued

$\equiv$ "cancellation law: $\mathbf{call}\{\!|p|\!\} = p$"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x+1|\!\} \land \overline{\overline{m}}'_2 = \{\!|x := x+2 \,;\ \mathbf{call}\,\overline{m}'_1|\!\} \land \\ (\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1) \land \\ (\mathbf{call}\,\overline{m}'_2 \Leftrightarrow (x := x+2 \,;\ \mathbf{call}\,\overline{m}'_1)) \end{array}\right) \land \gamma^*_{\mathrm{HM}}$$

$\equiv$ "predicative one-point rule using $\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1$ "

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x+1|\!\} \land \overline{\overline{m}}'_2 = \{\!|x := x+2 \,;\ \mathbf{call}\,\overline{m}'_1|\!\} \land \\ (\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1) \land \\ (\mathbf{call}\,\overline{m}'_2 \Leftrightarrow (x := x+2 \,;\ x := x+1)) \end{array}\right) \land \gamma^*_{\mathrm{HM}}$$

$\equiv$ "simplification of sequence: $(x := x+2 \,;\ x := x+1) = x := x+3$"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x+1|\!\} \land \overline{\overline{m}}'_2 = \{\!|x := x+2 \,;\ \mathbf{call}\,\overline{m}'_1|\!\} \land \\ (\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1) \land \\ (\mathbf{call}\,\overline{m}'_2 \Leftrightarrow x := x+3) \end{array}\right) \land \gamma^*_{\mathrm{HM}}$$

Derivation made explicit the value of the rank 1 method variables.

↪ The essence of the proof is an extraction law that exploits consistency between rank 1 and rank 2 method variables in **HM**-healthy predicates.

Proof example continued

$\equiv$ "cancellation law: $\mathbf{call}\{\!|p|\!\} = p$"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x+1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x+2 \,;\ \mathbf{call}\,\overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1) \wedge \\ (\mathbf{call}\,\overline{m}'_2 \Leftrightarrow (x := x+2 \,;\ \mathbf{call}\,\overline{m}'_1)) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "predicative one-point rule using $\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1$ "

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x+1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x+2 \,;\ \mathbf{call}\,\overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1) \wedge \\ (\mathbf{call}\,\overline{m}'_2 \Leftrightarrow (x := x+2 \,;\ x := x+1)) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "simplification of sequence: $(x := x+2 \,;\ x := x+1) = x := x+3$"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x+1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x+2 \,;\ \mathbf{call}\,\overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1) \wedge \\ (\mathbf{call}\,\overline{m}'_2 \Leftrightarrow x := x+3) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

Derivation made explicit the value of the rank 1 method variables.

$\hookrightarrow$ The essence of the proof is an extraction law that exploits consistency between rank 1 and rank 2 method variables in **HM**-healthy predicates.

Proof example continued

$\equiv$ "cancellation law: $\mathbf{call}\{\!|p|\!\} = p$"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x+1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x+2 \,;\ \mathbf{call}\,\overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1) \wedge \\ (\mathbf{call}\,\overline{m}'_2 \Leftrightarrow (x := x+2 \,;\ \mathbf{call}\,\overline{m}'_1)) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "predicative one-point rule using $\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1$ "

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x+1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x+2 \,;\ \mathbf{call}\,\overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1) \wedge \\ (\mathbf{call}\,\overline{m}'_2 \Leftrightarrow (x := x+2 \,;\ x := x+1)) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "simplification of sequence: $(x := x+2 \,;\ x := x+1) = x := x+3$"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x+1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x+2 \,;\ \mathbf{call}\,\overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1) \wedge \\ (\mathbf{call}\,\overline{m}'_2 \Leftrightarrow x := x+3) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

Derivation made explicit the value of the rank 1 method variables.

$\hookrightarrow$ The essence of the proof is an extraction law that exploits consistency between rank 1 and rank 2 method variables in **HM**-healthy predicates.

Proof example continued

$\equiv$ "cancellation law: $\mathbf{call}\{\!|p|\!\} = p$"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x+1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x+2 \,;\ \mathbf{call}\,\overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1) \wedge \\ (\mathbf{call}\,\overline{m}'_2 \Leftrightarrow (x := x+2 \,;\ \mathbf{call}\,\overline{m}'_1)) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "predicative one-point rule using $\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1$ "

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x+1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x+2 \,;\ \mathbf{call}\,\overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1) \wedge \\ (\mathbf{call}\,\overline{m}'_2 \Leftrightarrow (x := x+2 \,;\ x := x+1)) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

$\equiv$ "simplification of sequence: $(x := x+2 \,;\ x := x+1) = x := x+3$"

$$\left(\begin{array}{l} \overline{\overline{m}}'_1 = \{\!|x := x+1|\!\} \wedge \overline{\overline{m}}'_2 = \{\!|x := x+2 \,;\ \mathbf{call}\,\overline{m}'_1|\!\} \wedge \\ (\mathbf{call}\,\overline{m}'_1 \Leftrightarrow x := x+1) \wedge \\ (\mathbf{call}\,\overline{m}'_2 \Leftrightarrow x := x+3) \end{array}\right) \wedge \gamma^*_{\mathrm{HM}}$$

Derivation made explicit the value of the rank 1 method variables.

$\hookrightarrow$ The essence of the proof is an extraction law that exploits consistency between rank 1 and rank 2 method variables in **HM**-healthy predicates.

Procedures with parameters

A standard approach to support procedures is to encode them as functions:

- Domains correspond to the kind of objects being passed;
- Can be either a variable or value.

Examples

$$p_1 := \{\!|\lambda\, v : \mathit{var}(\mathbb{Z}) \bullet v := v + x|\!\}$$

$$p_2 := \{\!|\lambda\, x : \mathit{val}(\mathbb{Z}) \bullet y := y + x|\!\}$$

But p_1 is 'polymorphic'

- We do not know the alphabet of p_1 prior to evaluating a call.
- What ought its type to be?

Requires to elaborate the notion of type. Hence, we excluded it so far!

↪ But this is not a restriction in the context of a theory of pointers.

Procedures with parameters

A standard approach to support procedures is to encode them as functions:

- Domains correspond to the kind of objects being passed;
- Can be either a variable or value.

Examples

$$p_1 := \{\!|\lambda\, v : \mathit{var}(\mathbb{Z}) \bullet v := v + x|\!\}$$

$$p_2 := \{\!|\lambda\, x : \mathit{val}(\mathbb{Z}) \bullet y := y + x|\!\}$$

But p_1 is 'polymorphic'

- We do not know the alphabet of p_1 prior to evaluating a call.
- What ought its type to be?

Requires to elaborate the notion of type. Hence, we excluded it so far!

$\hookrightarrow$ But this is not a restriction in the context of a theory of pointers.

Procedures with parameters

A standard approach to support procedures is to encode them as functions:

- Domains correspond to the kind of objects being passed;
- Can be either a variable or value.

Examples

$$p_1 := \{\!|\lambda\, v : \mathit{var}(\mathbb{Z}) \bullet v := v + x|\!\}$$

$$p_2 := \{\!|\lambda\, x : \mathit{val}(\mathbb{Z}) \bullet y := y + x|\!\}$$

But p_1 is 'polymorphic'

- We do not know the alphabet of p_1 prior to evaluating a call.
- What ought its type to be?

Requires to elaborate the notion of type. Hence, we excluded it so far!

$\hookrightarrow$ But this is not a restriction in the context of a theory of pointers.

Encoding of Procedures

We use the same technique as in embedding syntax into program values.

Datatype definition

$$PROC[BODY] ::= ValArg \langle\!\langle TYPE \times PROC[BODY] \rangle\!\rangle \mid Body \langle\!\langle BODY \rangle\!\rangle$$

- *TYPE* encodes the type of a parameter;
- Type recursion supports arbitrary numbers of parameters;
- *BODY* provides the semantic model of the procedure body.
  ↪ Can be itself syntax — this is not a problem.

New definitions of...

1. **call**,
2. refinement and
3. at least assignment (to support refinement of procedure values)

have to be provided for $PROC[BODY]$.

Encoding of Procedures

We use the same technique as in embedding syntax into program values.

Datatype definition

$$PROC[BODY] ::= ValArg \langle\!\langle TYPE \times PROC[BODY] \rangle\!\rangle \mid Body \langle\!\langle BODY \rangle\!\rangle$$

- *TYPE* encodes the type of a parameter;
- Type recursion supports arbitrary numbers of parameters;
- *BODY* provides the semantic model of the procedure body.
  $\hookrightarrow$ Can be itself syntax — this is not a problem.

New definitions of...

1. **call**,
2. refinement and
3. at least assignment (to support refinement of procedure values)

have to be provided for $PROC[BODY]$.

Mutual Recursion

We return now to the initial problem of encoding

$$m_1, m_2 := \{\!|\, \mu\, X, Y \bullet \left\langle \begin{array}{l} (x := x - 1\,;\ Y) \lhd x > 0 \rhd \mathbf{II}, \\ (y := y - 1\,;\ X) \lhd y > 0 \rhd \mathbf{II} \end{array} \right\rangle \,|\!\}$$

In our theory of methods, this can now be written as

$$\begin{array}{lcll} T_1 & \widehat{=} & \mathbf{HM}(\overline{m}_1 := \{\!|(x := x - 1\,;\ \mathbf{call}\, \overline{m}_2) \lhd x > 0 \rhd \mathbf{II}|\!\}) & \text{and} \\ T_2 & \widehat{=} & \mathbf{HM}(\overline{m}_2 := \{\!|(y := y - 1\,;\ \mathbf{call}\, \overline{m}_1) \lhd y > 0 \rhd \mathbf{II}|\!\}) & \text{and} \\ T & \widehat{=} & T_1\,;\ T_2 & \end{array}$$

Improvement

1. No rewriting into weakest fixed points required.
2. Compositional translation from code to semantic models.

$\hookrightarrow$ Alphabets still require some care, but these are tractable issues.

Mutual Recursion

We return now to the initial problem of encoding

$$m_1, m_2 := \{\!|\, \mu\, X, Y \bullet \left\langle \begin{array}{l} (x := x - 1\,;\ Y) \lhd x > 0 \rhd \mathbf{II}, \\ (y := y - 1\,;\ X) \lhd y > 0 \rhd \mathbf{II} \end{array} \right\rangle \,|\!\}$$

In our theory of methods, this can now be written as

$$\begin{array}{lcll} T_1 & \widehat{=} & \mathbf{HM}(\overline{\overline{m}}_1 := \{\!|(x := x - 1\,;\ \mathbf{call}\, \overline{m}_2) \lhd x > 0 \rhd \mathbf{II}|\!\}) & \text{and} \\ T_2 & \widehat{=} & \mathbf{HM}(\overline{\overline{m}}_2 := \{\!|(y := y - 1\,;\ \mathbf{call}\, \overline{m}_1) \lhd y > 0 \rhd \mathbf{II}|\!\}) & \text{and} \\ T & \widehat{=} & T_1\,;\ T_2 & \end{array}$$

Improvement

1. No rewriting into weakest fixed points required.
2. Compositional translation from code to semantic models.

$\hookrightarrow$ Alphabets still require some care, but these are tractable issues.

hiJaC

Conclusions and Future Work

Conclusions

- We have examined ramifications of UTP-based theories of object-orientation.
- We have proposed a new UTP theory of methods.
- We presented solutions to four major challenges:
  1. "Consistency of the program model"
  2. "Redefinition of methods in subclasses"
  3. "Recursion and mutual recursion"
  4. "Simplicity"
- Our results, for instance, on ranks are relevant to other uses of higher-order UTP too.
- We laid the foundations for mechanisations, which is underway.

Future Work

- Further examine the theory of methods and its properties (laws).
- Prove the equivalence of the two different characterisations of recursions.
- Complete mechanisation efforts in Isabelle-HOL.
  ↪ Delicate issue here that requires a custom type axiom.
- Discover more laws in the theory of object-orientation.
- How do we automate reasoning about particular programs?
- Combine with other theories such as *Circus* Time.
  ↪ Work towards a UTP theory for SCJ*Circus*.

hiJaC

Thank you for your attention.

Any Questions?