Oracle-based Differential Operational Semantics
(Or explaining program differences with programs)

Thibaut Girka $^{1,2}$  David Mentré $^1$  Yann Régis-Gianas $^2$

$^1$Mitsubishi Electric R&D Centre Europe, F-35708 Rennes, France
$^2$Univ Paris Diderot, Sorbonne Paris Cité, IRIF, INRIA PiR2

November 8, 2016
What is the meaning of a program evolution?

\[ P_0 \]

```plaintext
sum = 0;
x = -x;
y = 0;
while (sum < x) {
    y = y + 1;
    sum += 1 + 2 * y;
}
sum = 0;
```

\[ P_2 \]

```plaintext
x = -x;
sum = 0;
count = 0;
while (sum < x) {
    count = count + 1;
    sum += 1 + 2 * count;
}
sum = 0;
```
How to express changes between two close programs?

From: Thibaut Girka <thibaut.girka@irif.fr>
Subject: [PATCH] Change stuff

```c
−sum = 0;
  x = −x;
−y = 0;
+sum = 0;
+count = 0;
  while (sum < x) {
−    y = y + 1;
−    sum += 1 + 2 * y;
+    count = count + 1;
+    sum += 1 + 2 * count;
  }
sum = 0;
```
How to express changes between two close programs?

From: Thibaut Girka <thibaut.girka@irif.fr>
Subject: [PATCH] Rename y to count;
    Change variable initialization order

```plaintext
- sum = 0;
  x = -x;
- y = 0;
+ sum = 0;
+ count = 0;
  while (sum < x) {
    - y = y + 1;
    - sum += 1 + 2 * y;
    + count = count + 1;
    + sum += 1 + 2 * count;
  }
sum = 0;
```
What could be a formal **difference language**?
A formal difference

From: Thibaut Girka <thibaut.girka@irif.fr>
Subject: [PATCH] Rename y → count ;
    SwapAssign@0

```sqlite
--sum = 0;
  x = −x;
−y = 0;
+sum = 0;
+count = 0;
  while (sum < x) {
       y = y + 1;
−    sum += 1 + 2 * y;
+    count = count + 1;
+    sum += 1 + 2 * count;
  }
sum = 0;
```
Difference languages
(as wanted by programmers)

Syntax

- Readable change descriptions to express intent
- Sufficiently declarative to be understood by programmers

Semantics

- Mechanically verifiable
- Reason about program evolution
- Ease code review
- Formal ground to build incremental development tools
Difference languages
(as wanted by semanticists)

A general framework to compare program behaviors

- Plenty of frameworks to reason about program equivalence
- Not that much to compare (inequivalent) programs
- To justify **differential static analysis**
- To specify transformations that **do not preserve semantics**
Difference languages
(as wanted by semanticists)

A general framework to compare program behaviors

- Plenty of frameworks to reason about program equivalence
- Not that much to compare (inequivalent) programs
- To justify differential static analysis
- To specify transformations that do not preserve semantics

How should we relate the reduction of two close programs?
Difference languages
(as wanted by semanticists)

A general framework to compare program behaviors

- Plenty of frameworks to reason about program equivalence
- Not that much to compare (inequivalent) programs
- To justify differential static analysis
- To specify transformations that do not preserve semantics

How should we relate the reduction of two close programs?

Using a program!
Oracle-based Differential Operational Semantics
(The idea)

\[ P_1 \xrightarrow{c_1^1} P_2 \]
Oracle-based Differential Operational Semantics
(The idea)

\[ P_1 \xrightarrow{c_1^1} \xrightarrow{\sim} \xrightarrow{O(\delta)} P_2 \xrightarrow{c_2^1} \]
Oracle-based Differential Operational Semantics
(The idea)

\[ P_1 \xrightarrow{c_1^1} c_1^2 \xrightarrow{\sim} O(\delta) \xrightarrow{\sim} c_1^1 \]

\[ P_2 \xrightarrow{c_2^1} c_2^2 \xrightarrow{\sim} O(\delta) \xrightarrow{\sim} c_2^1 \]
Oracle-based Differential Operational Semantics
(The idea)

\[
P_1 \quad c_1^1 \quad \longrightarrow \quad c_1^2
\]

\[
P_2 \quad c_2^1 \quad \dashrightarrow \quad c_2^2
\]
Oracle-based Differential Operational Semantics
(The idea)
Oracle-based Differential Operational Semantics
(The idea)

\[ P_1 \quad \begin{array}{c}
    c_1^1 \\ \sim \\
    \downarrow \\
    \delta \\
\end{array} \rightarrow \begin{array}{c}
    c_1^2 \\
    \sim \\
    \uparrow \\
\end{array} \rightarrow \begin{array}{c}
    c_1^3 \\
    \sim \\
\end{array} \cdots \]

\[ P_2 \quad \begin{array}{c}
    c_2^1 \\
\end{array} \rightarrow \begin{array}{c}
    c_2^2 \\
\end{array} \rightarrow \begin{array}{c}
    c_2^3 \\
\end{array} \cdots \]
Oracle-based Differential Operational Semantics
(by analogy)

An interpreter maps a program to its reduction trace.
Oracle-based Differential Operational Semantics (by analogy)

An *interpreter* maps a *program* to its *reduction trace*.

An *oracle* maps a *difference* between $P_1$ and $P_2$ to a *relation* between their reduction traces.
Oracle-based Differential Operational Semantics

Our contributions

- A formal language-agnostic framework for difference languages
- Instantiated on the Imp toy language
- Examples of difference languages implemented in Coq \(^1\)
- What makes a difference sound?
- How to compose two differences?

This talk

- A tour through several examples of difference languages
- To highlight the expressivity of this framework

\(^1\)Check it out! : https://www.irif.fr/~thib/oracles
Oracles through examples

\[ P_0 \]
\[
\text{sum} = 0;
\]
\[
x = -x;
\]
\[
y = 0;
\]
\[
\textbf{while} \ (\text{sum} < \ x) \ \{ \\
\quad y = y + 1; \\
\quad \text{sum} += 1 + 2 \times y;
\}
\]
\[
\text{sum} = 0;
\]

\[ P_2 \]
\[
x = -x;
\]
\[
\text{sum} = 0;
\]
\[
\text{count} = 0;
\]
\[
\textbf{while} \ (\text{sum} < \ x) \ \{ \\
\quad \text{count} = \text{count} + 1; \\
\quad \text{sum} += 1 + 2 \times \text{count};
\}
\]
\[
\text{sum} = 0;
\]
Renaming

\[ y \rightarrow \text{count} \]

\[ P_0 \]
sum = 0;
x = −x;
y = 0;
while (sum < x) {
    y = y + 1;
    sum += 1 + 2 * y;
}
sum = 0;

\[ P_1 \]
sum = 0;
x = −x;
count = 0;
while (sum < x) {
    count = count + 1;
    sum += 1 + 2 * count;
}
sum = 0;
Renaming: semantics (simpl.)

\[ c_8 \xrightarrow{(\delta \kappa_8, [y:=1])} c_9 \xrightarrow{\sim} c_9' \xrightarrow{c_8} \]
Renaming: semantics (simpl.)

\[ (\delta \kappa_8, [y := 1]) \sim (\delta \kappa_9, [\text{sum} := 3]) \sim (\delta \kappa_8, [\text{count} := 1]) \]

\[ O(\delta) \]
Renaming: semantics (simpl.)

\[ c_8 \xrightarrow{([y := 1])} c_9 \xrightarrow{([\text{sum} := 3])} c_{10} \]

\[ c'_8 \xrightarrow{([\text{count} := 1])} c'_9 \]
Renaming: semantics (simpl.)

\[
\begin{align*}
&c_8 \xrightarrow{(\delta k_8, [y:=1])} c_9 \\
&\sim \quad \sim \\
&\quad \downarrow \quad \downarrow \\
&c'_8 \xrightarrow{(\delta k_8, [count:=1])} c'_9 \\
&c_9 \xrightarrow{(\delta k_9, [sum:=3])} c_{10} \\
&c'_9 \xrightarrow{(\delta k_9, [sum:=3])} c'_{10}
\end{align*}
\]
Renaming: semantics (simpl.)

\[ O(y \mapsto \text{count})(\delta \kappa, \delta M) = (\delta \kappa, \delta M[y \mapsto \text{count}]) \]
Renaming: semantics (simpl.)

\[
\begin{array}{cccc}
  c_0 & \xrightarrow{\delta_0} & c_1 & \xrightarrow{\delta_1} & c_2 & \xrightarrow{\delta_2} & c_3 \\
  \sim & & \sim & & \sim & & \sim \\
  c'_0 & \xrightarrow{\delta'_0} & c'_1 & \xrightarrow{\delta'_1} & c'_2 & \xrightarrow{\delta'_2} & c'_3 \\
\end{array}
\]
Type of an oracle’s prediction function
(As a first approximation)

\[ d \times \delta c \rightarrow \delta c \]

- \( d = \{\downarrow, \uparrow\} \)
  (direction of the prediction)
- \( \delta c \): reified step
Assignment commutation

\textit{SwapAssign@0}

\begin{align*}
P_1 \quad \text{sum} &= 0; \\
\text{x} &= -x; \\
\text{count} &= 0; \\
\text{while (sum < x) \{ \\ &\quad \text{count} = \text{count} + 1; \\
&\quad \text{sum} += 1 + 2 \times \text{count}; \\
\}} \\
\text{sum} &= 0;
\end{align*}

\begin{align*}
P_2 \quad \text{x} &= -x; \\
\text{sum} &= 0; \\
\text{count} &= 0; \\
\text{while (sum < x) \{ \\ &\quad \text{count} = \text{count} + 1; \\
&\quad \text{sum} += 1 + 2 \times \text{count}; \\
\}} \\
\text{sum} &= 0;
\end{align*}
Assignment commutation

\textit{SwapAssign}@0

\[
\begin{align*}
\text{sum} &:= 0 \\
x &:= -x
\end{align*}
\]
Assignment commutation

Oracle features highlight

- Cannot always be called in both directions
- Need to maintain an internal state
- May predict 0, 1 or 2 steps
- Relate syntactic path to dynamic evaluation points
Assignment commutation: semantics
A peek into its internal state

This oracle maintains an internal state with:

- A list of *continuation modifiers*
  to relate the syntactic path to dynamic execution points
- A delta on stores during commutations
- The direction it was called on during commutations
Assignment commutation: semantics

\[
\begin{array}{cccccc}
& & \delta_1 & & & \\
\vdash & \downarrow & \Downarrow & \downarrow & \Downarrow & \\
\vdash & \downarrow & \Downarrow & \downarrow & \Downarrow & \\
\end{array}
\]

\[
\begin{array}{ccccccc}
\cdots & \delta_1 = (\text{[unfold-seq]}, []) & \cdots & \delta_1' = (\text{[unfold-seq]}, [])
\end{array}
\]
**Assignment commutation: semantics**

\[
\begin{align*}
C_1 & \xrightarrow{\delta_1} C_2 \xrightarrow{\delta_2} C_3 \\
C'_1 & \xrightarrow{\delta'_1} C'_2 \xrightarrow{\delta'_2} C'_3
\end{align*}
\]

\[
\delta_1 = ([\text{unfold-seq}], []) \\
\delta_2 = ([\text{pop}], [\text{sum} := 0])
\]

\[
\delta'_1 = ([\text{unfold-seq}], [])
\]
Assignment commutation: semantics

\[ \begin{align*}
    c_1 & \xrightarrow{\delta_1} c_2 & & \xrightarrow{\delta_2} c_3 & & \xrightarrow{\delta_3} c_4 \\
    c_1' & \equiv c_2' & & c_2' & & \equiv c_4'
\end{align*} \]

\[ \begin{align*}
    \delta_1 &= ([\text{unfold-seq}], []) \\
    \delta_2 &= ([\text{pop}], [\text{sum} := 0]) \\
    \delta_3 &= ([\text{pop}], [x := 42]) \\
    \delta_1' &= ([\text{unfold-seq}], []) \\
    \delta_2' &= ([\text{pop}], [x := 42]) \\
    \delta_3' &= ([\text{pop}], [\text{sum} := 0])
\end{align*} \]
Assignment commutation: semantics

\[
\begin{align*}
C_1 \xrightarrow{\delta_1} C_2 \xrightarrow{\delta_2} C_3 \xrightarrow{\delta_3} C_4 \xrightarrow{\delta_4} C_5 \\
C'_1 \xrightarrow{\delta_1'} C'_2 \xrightarrow{\equiv} C'_2 \xrightarrow{\delta_3 \delta_2} C'_4 \xrightarrow{\delta_4'} C'_5
\end{align*}
\]

\[
\begin{align*}
\delta_1 &= ([\text{unfold-seq}], []) \\
\delta_2 &= ([\text{pop}], [\text{sum} := 0]) \\
\delta_3 &= ([\text{pop}], [x := 42]) \\
\delta_4 &= ([\text{unfold-seq}], [])
\end{align*}
\]

\[
\begin{align*}
\delta_1' &= ([\text{unfold-seq}], []) \\
\delta_2' &= ([\text{pop}], [x := 42]) \\
\delta_3' &= ([\text{pop}], [\text{sum} := 0]) \\
\delta_4' &= ([\text{unfold-seq}], [])
\end{align*}
\]
Type of an oracle’s prediction function

\[ s \times d \times \delta c \rightarrow \]

\[ s \times \dot{d} \times ((\mathbb{N} \setminus \{0\} \times \delta c) + \text{wait}) \]

- \( \dot{d} = d \cup \{\downarrow \uparrow\} \)
  (allowed directions for the next prediction)
- \( s \): internal oracle state
Abstraction of equivalent sub-programs

\[ P_2 \]
\[
\begin{align*}
x &= -x; \\
sum &= 0; \\
count &= 0; \\
while \ (sum < x) \{ \\
    &\quad count = count + 1; \\
    &\quad sum += 1 + 2 * count; \\
\}
\]
sum = 0;

\[ P_3 \]
\[
\begin{align*}
x &= -x; \\
sum &= 0; \\
if \ (x < 4) \{ \\
    &\quad count = 1; \\
\}
else \{ \\
    &\quad count = x; \\
    &\quad sum = (x + 1) / 2; \\
    &\quad while \ (sum < count) \{ \\
        &\quad count = sum; \\
        &\quad sum = x / sum + sum; \\
        &\quad sum /= 2; \\
    \}
\}
sum = 0;
Abstraction of equivalent sub-programs

\[
P_2
\]
\[
x = -x;
sum = 0;
count = 0;
while (sum < x) {
    count = count + 1;
    sum += 1 + 2 * count;
}
sum = 0;
\]

\[
P_3
\]
\[
x = -x;
sum = 0;
if (x < 4) {
    count = 1;
} else {
    count = x;
    sum = (x + 1) / 2;
    while (sum < count) {
        count = sum;
        sum = x / sum + sum;
        sum /= 2;
    }
}
sum = 0;
Abstraction of equivalent sub-programs

Oracle features highlight

- May predict 0, 1, or a dynamic number of steps
- Abstracts away from the small-step presentation
- Defer work to proof obligations
Abstraction of equivalent sub-programs

AbstractEquiv\@1, \ldots

- Path of equivalent subprograms (\@1)
- The sub-programs themselves
- Proof of big-step equivalence
- Proof termination for each sub-program
Abstraction of equivalent sub-programs: semantics

A peek into its internal state

- List of *continuation modifiers* the same length as the continuation

  or

- The current *prediction direction* (⇃ or ⇁)
- The number of remaining continuation elements from the sub-program being evaluated
- A snapshot of the store before executing the sub-program (used to compute the bound on execution steps)
- The accumulated delta on the store so far
Abstraction of equivalent sub-programs: semantics
Abstraction of equivalent sub-programs: semantics

\[
\begin{align*}
  c_2 & \xrightarrow{\delta_2} c_3 & \cdots & \xrightarrow{} c_n & \xrightarrow{\delta_n} c_{n+1} \\
  \sim & \quad \quad \quad \quad \quad \sim & \cdots & \quad \quad \sim & \quad \quad \sim \\
  c'_2 & \xrightarrow{\equiv} c'_2 & \cdots & \xrightarrow{\equiv} c'_2 & \xrightarrow{\equiv} c'_2
\end{align*}
\]
Abstraction of equivalent sub-programs: semantics

\[c_2 \xrightarrow{\delta_2} c_3 \quad \cdots \quad \xrightarrow{\delta_n} c_{n+1} \xrightarrow{\delta_{n+1}} c_{n+2}\]

\[c'_2 \quad \xrightarrow{\equiv} c'_2 \quad \cdots \quad \xrightarrow{\equiv} c'_2 \quad \xrightarrow{([\text{pop}], \ldots)} c'_m\]
Control-flow-preserving value changes

ValueChange \([pow, a] (42 \rightarrow 10)@0\)

<table>
<thead>
<tr>
<th>(F_0)</th>
<th>(F_1)</th>
</tr>
</thead>
</table>
| \(a = 42;\)  
\(n = 5; \ pow = 1;\)  
\(\text{while} (0 < n) \{\)  
  \(\quad n = n - 1;\)  
  \(\quad \text{pow} = \text{pow} \ast a;\)  
\}; | \(a = 10;\)  
\(n = 5; \ pow = 1;\)  
\(\text{while} (0 < n) \{\)  
  \(\quad n = n - 1;\)  
  \(\quad \text{pow} = \text{pow} \ast a;\)  
\}; |
Control-flow-preserving value changes

Oracle features highlight

- Always predict exactly one step
- Always bidirectional
- Relate programs that are not equivalent
- Soundness follows from (overly restrictive) syntactic criteria
Control-flow-preserving value changes

\textbf{ValueChange} \([\text{pow, a}]\) \((42 \rightarrow 10)@0\)

- Path of an assignment to change (@0)
- Expression to substitute in the assignment’s right-hand side (42 → 10)
- Variables possibly affected ([pow, a])
Control-flow-preserving value changes
A peek into its internal state

This oracle maintains an internal state with:

- A list of *continuation modifiers* to relate the syntactic path to dynamic execution points
- A store for each program
Control-flow-preserving value changes: semantics (simpl.)

\[
\begin{align*}
&c_8 \xrightarrow{(\delta \kappa_8, [\text{n:=4}])} c_9 \\
&c_8' \xrightarrow{(\delta \kappa_8, [\text{n:=4}])} c_9' \\
&c_9 \xrightarrow{(\delta \kappa_9, [\text{pow:=42}])} c_{10} \\
&c_9' \xrightarrow{(\delta \kappa_9, [\text{pow:=10}])} c_{10}'
\end{align*}
\]
Oracle languages on Imp: recap

We presented:

- Renaming
- Assignment commutation
- Abstraction of equivalent terminating sub-programs
- Control-flow-preserving value changes

We have also implemented:

- Sequence associativity
- Condition negation / Branch commutation
- Abstraction of equivalent unbounded sub-programs
- Abstraction of inequivalent sub-programs
- Crash avoidance
Composition: $P_0 \sim P_2$
Composition: \( P_0 \sim P_2 \)
Composition: $P_0 \sim P_2$
Composition: $P_0 \sim P_2$

\[
\begin{align*}
\delta_0^0 : c_0^0 & \rightarrow c_1^0 \\
\delta_1^0 & \sim \\
\delta_2^0 & \sim \\
\end{align*}
\]
Composition: \( P_0 \sim P_2 \)
Composition: $P_0 \sim P_2$

\[
\begin{align*}
  c_0^0 & \xrightarrow{\delta_0^0} c_1^0 & \xrightarrow{\delta_1^0} c_2^0 \\
  c_0^1 & \xrightarrow{\delta_0^1} c_1^1 & \xrightarrow{\delta_1^1} c_2^1
\end{align*}
\]
Composition: $P_0 \sim P_2$
Composition: $P_0 \sim P_2$
Composition: $P_0 \sim P_2$
Another composition example

\[ c_0 \longrightarrow c_1 \]

\[ \sim \quad \quad \quad \quad \sim \]

\[ c'_0 \quad \quad \longrightarrow \quad \quad c'_1 \]

\[ c'_0 \quad \quad \longrightarrow \quad \quad c'_1 \]

\[ \sim \quad \quad \quad \quad \sim \]

\[ \sim \quad \quad \quad \quad \sim \]

\[ c''_0 \quad \quad \longrightarrow \quad \quad c''_1 \]
Another composition example

\[
\begin{array}{ccc}
  c_0 & \rightarrow & c_1 \\
  \sim & \downarrow & \sim \\
  c_0' & \rightarrow & c_1' \\
  \sim & \downarrow & \sim \\
  c_0'' & \rightarrow & c_1'' \\
\end{array}
\]

\[
\begin{array}{ccc}
  \sim & \downarrow & \sim \\
\end{array}
\]
Another composition example

\[ c_0 \rightarrow c_1 \rightarrow c_2 \]
\[ \sim \downarrow \sim \downarrow \sim \]
\[ c_0' \rightarrow c_1' \rightarrow c_3' \]

\[ c_0' \rightarrow c_1' \rightarrow c_2' \]
\[ \sim \downarrow \sim \downarrow \sim \]
\[ c_0'' \rightarrow c_1'' \rightarrow c_2'' \]
Another composition example

\[
\begin{align*}
  c_0 &\longrightarrow c_1 &\longrightarrow c_2 \\
  \sim &\quad \uparrow &\quad \sim \\
  c_0' &\longrightarrow c_1' &\longrightarrow c_3' \\
  \sim &\quad \uparrow &\quad \sim
\end{align*}
\]

\[
\begin{align*}
  c_0' &\longrightarrow c_1' &\longrightarrow c_2 &\longrightarrow c_3' \\
  \sim &\quad \uparrow &\quad \sim &\quad \sim \\
  c_0'' &\longrightarrow c_1'' &\longrightarrow c_2'' &\longrightarrow c_3'' \\
\end{align*}
\]
Another composition example

\[
\begin{array}{c}
c_0 \quad \longrightarrow \quad c_1 \quad \longrightarrow \quad c_2 \\
\sim \quad \uparrow \quad \sim \quad \uparrow \quad \sim \\
\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
\end{array}
\]

\[
\begin{array}{c}
c_0'' \quad \longrightarrow \quad c_1'' \quad \longrightarrow \quad c_3'' \\
\end{array}
\]
Composition: incompatible directions

\[ c_0 \rightarrow c_1 \rightarrow c_2 \]
\[ \sim \quad \sim \quad \sim \]
\[ c'_0 \rightarrow c'_1 \rightarrow c'_2 \]

\[ c'_0 \rightarrow c'_1 \rightarrow c'_2 \]
\[ \sim \quad \uparrow \quad \sim \quad \uparrow \quad \sim \]
\[ c''_0 \rightarrow c''_1 \rightarrow c''_2 \]
Composition: incompatible directions

\[
c_0 \xrightarrow{\sim} c_1 \xrightarrow{\sim} c_2 \xrightarrow{\sim} c_3 \\
\vdash r \xrightarrow{\sim} \vdash r \xrightarrow{\sim} \vdash r \\
c'_0 \xrightarrow{\vdash r} c'_1 \xrightarrow{\vdash r} c'_2 \xrightarrow{\vdash r} c'_3
\]

\[
c'_0 \xrightarrow{\vdash r} c'_1 \xrightarrow{\vdash r} c'_2 \\
\vdash r \xrightarrow{\vdash r} \vdash r \xrightarrow{\sim} \vdash r \\
c''_0 \xrightarrow{\vdash r} c''_1 \xrightarrow{\vdash r} c''_2
\]
Composition: incompatible directions

\[
\begin{array}{cccc}
  c_0 & \rightarrow & c_1 & \rightarrow & c_2 & \rightarrow & c_3 \\
  \sim & \downarrow & \sim & \downarrow & \sim & \uparrow & \sim \\
  c'_0 & \leftrightarrow & c'_1 & \leftrightarrow & c'_2 & \leftrightarrow & c'_3 \\
  \sim & \leftrightarrow & \sim & \leftrightarrow & \sim & \leftrightarrow & \sim \\
  \sim & \leftrightarrow & \sim & \leftrightarrow & \sim & \leftrightarrow & \sim \\
  c''_0 & \leftrightarrow & c''_1 & \leftrightarrow & c''_2 & \leftrightarrow & c''_3 \\
\end{array}
\]
Composition: incompatible directions

\[
\begin{align*}
\mathcal{C}_0 & \xrightarrow{\sim} \mathcal{C}_1 & \mathcal{C}_1 & \xrightarrow{\sim} \mathcal{C}_2 & \mathcal{C}_2 & \xrightarrow{\sim} \mathcal{C}_3 \\
\mathcal{C}_0'' & \xrightarrow{\sim} \mathcal{C}_1'' & \mathcal{C}_1'' & \xrightarrow{?} \mathcal{C}_2'' & \mathcal{C}_2'' & \xrightarrow{\sim} \mathcal{C}_3''
\end{align*}
\]
Composition: incompatible directions (2)

\[ c_0 \rightarrow c_1 \rightarrow c_2 \]
\[ c_0' \rightarrow c_1' \rightarrow c_3' \]
\[ c_0'' \rightarrow c_1'' \]
Composition: incompatible directions (2)

\[
\begin{align*}
&c_0 \quad \longrightarrow \quad c_1 \quad \longrightarrow \quad c_2 \\
&\quad \sim \quad \downarrow \quad \sim \quad \downarrow \quad \sim \\
&c'_0 \quad \longrightarrow \quad c'_1 \quad \longrightarrow \quad c'_3 \\
\end{align*}
\]

\[
\begin{align*}
&c'_0 \quad \longrightarrow \quad c'_1 \quad \longrightarrow \quad c'_2 \\
&\quad \sim \quad \downarrow \quad \sim \quad \uparrow \quad \sim \\
&c''_0 \quad \longrightarrow \quad c''_1 \quad \longrightarrow \quad c''_2 \\
\end{align*}
\]
Composition: incompatible directions (2)

\[
\begin{array}{ccc}
  c_0 & \longrightarrow & c_1 & \longrightarrow & c_2 \\
  \sim & \uparrow & \sim & \uparrow & \sim \\
  c'_0 & \longrightarrow & c'_1 & \longrightarrow & c'_3 \\
\end{array}
\]

\[
\begin{array}{ccc}
  c'_0 & \longrightarrow & c'_1 & \longrightarrow & c'_2 & \longrightarrow & c'_3 \\
  \sim & \uparrow & \sim & \uparrow & \sim & \uparrow & \sim \\
  c''_0 & \longrightarrow & c''_1 & \longrightarrow & c''_2 & \longrightarrow & c''_3 \\
\end{array}
\]
Composition: incompatible directions (2)

\[ c_0 \quad \rightarrow \quad c_1 \quad \rightarrow \quad c_2 \]

\[ c_0'' \quad \Rightarrow \quad c_1'' \quad \rightarrow \quad c_2'' \]
Composition: stuck intermediate program

\[
\begin{array}{cccc}
  c_0 & \longrightarrow & c_1 & \longrightarrow & c_2 & \longrightarrow & c_3 \\
  \sim & \downarrow & \sim & \downarrow & \sim & \downarrow & \\
  c'_0 & \longrightarrow & c'_1 & \longrightarrow & c'_2 & \longrightarrow & \\
\end{array}
\]

\[
\begin{array}{cccc}
  c'_0 & \longrightarrow & c'_1 & \longrightarrow & c'_2 & \longrightarrow & \triangleright \\
  \sim & \downarrow & \sim & \downarrow & \sim & & \\
  c''_0 & \longrightarrow & c''_1 & \longrightarrow & c''_2 & & \\
\end{array}
\]
Composition: stuck intermediate program
Issues with composition

Composition is ill-defined if . . .

- The two underlying oracles require incompatible directions
- One oracle predicts a crash of the intermediate program
Sufficient criteria for well-defined composition

- Neither oracle predicts a crash of the intermediate program and

- Either one of the oracles is always lock-step and bidirectional
- Or neither oracle forces a change of direction
Coq Library

In numbers

- Oracle languages meta-definitions and properties, including identity and composition (about 1400 lines of spec and 3200 lines of proofs)
- Imp syntax and semantics (about 500 lines of spec)
- Oracle languages on Imp (about 3900 lines of spec and 14500 lines of proofs)
Coq Library
Language-agnostic aspects

Generic definitions

- Oracle languages meta-definitions and properties
- Identity and universal oracles
- Oracle composition
Proof-Of-Concept oracle inference tool

```
difftool --search--depth 3 P0.imp P2.imp
```

Renaming \([y \rightarrow \text{count}]\)
- `SeqAssocOracle.Mod Fold`
- `AssignSwapOracle.Mod 0`
- `SeqAssocOracle.Mod Unfold`

- Written in OCaml, with lots of heuristics
- Not complete, neither proved correct in Coq.
- **But**, it generates a difference.
- An extracted checker verifies its validity.
Proof-Of-Concept oracle inference tool

Implemented oracles

- Renaming
- Assignment commutation
- Condition negation / Branch commutation
- Sequence associativity
- Control-flow preserving value changes
Conclusion

• Differences between close programs can be formally defined
• Changes can usually be composed, but there are some limits

Future work

• Explore weaker conditions implying composability
• Need for a formal criteria characterizing “useful” oracles
• Instantiate the framework on more programming languages
# Properties of oracle languages

<table>
<thead>
<tr>
<th>Properties</th>
<th>Renaming</th>
<th>SeqAssoc</th>
<th>SwapAssign</th>
<th>SwapBranch</th>
<th>AbstractEquiv</th>
<th>AbsEqNoBound</th>
<th>CrashFix</th>
<th>ValueChange</th>
<th>AbstractInequiv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decidable Checking</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Applicative</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Cooperative</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>One-step &amp; Bidirectional</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
</tbody>
</table>

¹ Only decidable given underlying proofs