States and Actions: An Automata-theoretic Model of Objects

Uday S. Reddy\textsuperscript{1}  Brian P. Dunphy\textsuperscript{2}

\textsuperscript{1}University of Birmingham
\textsuperscript{2}University of Illinois at Urbana-Champaign

Portland, Oct 2011
What it is about

Study the semantics of imperative object-oriented programming, using Idealized Algol as the foundational language.

- Can we bridge the gap between the state-based paradigm (Scott-Strachey approach) of semantics and the event-based paradigm (Milner-Hoare approach)?
- What is the right notion of *relational parametricity* (capturing data abstraction) for programs manipulating store?
- Can we push the full abstraction results *beyond the second-order active types* of Idealized Algol?

Reynolds [1981] anticipates many of these ideas.
Section 1

Information Hiding
Information hiding

- Imperative programs have information hiding pretty much everywhere.
- Object-oriented programming exploits this information hiding.

Information hiding arises much more fundamentally:
  - when local variables are declared (hidden outside their declaring blocks),
  - when procedures are called (data in the calling context hidden from the procedure)

As a result of information hiding, we get:
  - reasoning principles based on **invariants** (useful for proving properties, safety, consistency, integrity),
  - reasoning principles based on **simulation relations** (useful for program equivalence and data refinement).
Consider:

\[
\text{while } x \leq 100 \text{ do } \quad x := x + 1
\]

\(x \geq 0\) is invariant in \(x := x + 1\).

Hence, \(x \geq 0\) is invariant in the entire while-loop.

\[
\{P\} \ C \ \{P\} \\
\{P\} \text{ while } B \text{ do } C \ \{P\}
\]

This invariant principle has nothing to do with while-loops as such. It also applies to \textit{all} primitive control structures (repeat-until, for-loops, if-then-else etc.)

\[
\{P\} \ C_1 \ \{P\} \quad \{P\} \ C_2 \ \{P\} \\
\{P\} \text{ if } B \text{ then } C_1 \text{ else } C_2 \ \{P\}
\]
The same principle also applies to user-defined higher-order procedure constants (with no free identifiers) $F : \text{comm} \rightarrow \text{comm}$:

$\{P\} C \{P\}$

$\{P\} F(C) \{P\}$

We don’t even have to know what $F$ does to establish the invariant principle!

- Why do these principles work?
- Answer: Information hiding.

$\textbf{while}$ is a constant (no free identifiers) of type:

$\text{while} : \text{exp[bool]} \times \text{comm} \rightarrow \text{comm}$

The action of $\textbf{while}$ has no direct access to any storage, other than what is provided by its arguments.

- Hence, any property left invariant by the arguments is also left invariant by the $\textbf{while}$ loop.
- The storage of the arguments is $\textit{hidden}$ from the $\textbf{while}$
The same principle also applies to user-defined higher-order procedure constants (with no free identifiers) \( F : \text{comm} \rightarrow \text{comm} \):

\[
\{ P \} \ C \ {\{ P \} \over \{ P \} \ F(C) \ {\{ P \} \}}
\]

We don’t even have to know what \( F \) does to establish the invariant principle!

Why do these principles work?

Answer: Information hiding.

while is a constant (no free identifiers) of type:

\[
\text{while} : \text{exp}[\text{bool}] \times \text{comm} \rightarrow \text{comm}
\]

The action of while has no direct access to any storage, other than what is provided by its arguments.

Hence, any property left invariant by the arguments is also left invariant by the while loop.

The storage of the arguments is hidden from the while.
The same principle also applies to user-defined higher-order procedure constants (with no free identifiers) $F : \text{comm} \rightarrow \text{comm}$:

$$\{ P \} \ C \ \{ P \} \ \frac{}{\{ P \} \ F(C) \ \{ P \}}$$

We don’t even have to know what $F$ does to establish the invariant principle!

Why do these principles work?

Answer: Information hiding.

while is a constant (no free identifiers) of type:

$$\text{while} : \exp[\text{bool}] \times \text{comm} \rightarrow \text{comm}$$

The action of while has no direct access to any storage, other than what is provided by its arguments.

Hence, any property left invariant by the arguments is also left invariant by the while loop.

The storage of the arguments is hidden from the while.
The same principle also applies to user-defined higher-order procedure constants (with no free identifiers) $F : \text{comm} \to \text{comm}$:

$$
\begin{align*}
\{ P \} C \{ P \} \\
\{ P \} F(C) \{ P \}
\end{align*}
$$

We don’t even have to know what $F$ does to establish the invariant principle!

Why do these principles work?
Answer: Information hiding.

**while** is a constant (no free identifiers) of type:

$$
\text{while} : \text{exp[bool]} \times \text{comm} \to \text{comm}
$$

The action of **while** has no direct access to any storage, other than what is provided by its arguments.

Hence, any property left invariant by the arguments is also left invariant by the **while** loop.

The storage of the arguments is *hidden* from the **while**
Information hiding leads to binary simulation relations

- Consider a relation:
  \[ x \ [R] y \iff y = -x \]

- Notice:
  \[ x \leq 100 \ [R_{\text{exp,bool}}] \ y \geq -100 \]
  \[ (x := x + 1) \ [R_{\text{comm}}] (y := y - 1) \]

- Infer
  
  \[
  \begin{align*}
  &\text{while } x \leq 100 \text{ do } x := x + 1 \\
  &\quad [R_{\text{comm}}] \\
  &\text{while } y \geq -100 \text{ do } y := y - 1
  \end{align*}
  \]

- Again, it is not necessary to know what `while` does, in order to infer this fact.

- Every constant combinator will preserve the binary simulation relation in the same way (including user-defined combinators).
The same ideas also work for O-O classes.

```plaintext
class
    local Var[int] x;
    init x := 0;
    meth
        { val() = x,
          inc() = (x := x + 1) }
```

The class has an invariant $x \geq 0$, i.e., all the methods preserve it.

Hence, when the class is used in the context of any client, the entire program will preserve the invariant.

Again, information hiding is what is at play: The variable $x$ is hidden from the clients.
Invariants and binary simulation relations are both instances of the same concept: *Relational parametricity*. Formulated by John Reynolds in 1983 for polymorphic lambda calculus.

In the OOP context:

\[
\text{client} : \forall X F(X) \rightarrow K(X)
\]

The mathematical meaning of $\forall$ says that all possible relations between potential representation types $X$ will be preserved by client.

[Dunphy and Reddy, 2004] give a general category-theoretic axiomatization of this concept.
Invariants and binary simulation relations are both instances of the same concept: *Relational parametricity*.
Formulated by John Reynolds in 1983 for polymorphic lambda calculus.
In the OOP context:

\[
\text{client} : \forall X \mathcal{F}(X) \rightarrow \mathcal{K}(X) \quad \text{client} : (\exists X \mathcal{F}(X)) \rightarrow \mathcal{K}
\]

The mathematical meaning of \(\forall\) says that all possible relations between potential representation types \(X\) will be preserved by client.

[Dunphy and Reddy, 2004] give a general category-theoretic axiomatization of this concept.
Section 2

States and actions
In 1998, I discovered that there were two orthogonal dimensions to modeling mutable storage:

- states
- actions

State-invariants and state simulation relations may not be enough.

[Reynolds, 1981] used a similar modeling too. Simpler state-only models later invented by Oles, Tennent and O’Hearn. Reynolds’s model was essentially “forgotten”.

In the Formal Methods community, similar orthogonality was discovered in terms of *history Invariants*. Originally from Ina Jo [Scheid and Hostsberg, 1980-1992] and popularized by [Liskov and Wing, 1994] and used in Spec#. 
History invariants

- $x := x + 1$ satisfies a history invariant:

$$x \geq \text{old}(x)$$

No matter how many times the command $x := x + 1$ is run, the initial state and the final state will satisfy this property.

- It follows that

$$\text{while } x \leq 100 \text{ do } x := x + 1$$

also satisfies the history invariant.

- Similar discussion as before applies: It does not matter what \textbf{while} does for the preservation of history invariants. User-defined combinators will also preserve it, if they are constant.
Action invariants

- A slightly more general concept than history invariants.

\[ P(a) \iff \forall n. a(n) \geq n \]

\( P \) is a property of “actions,” i.e., state transformations.

- Another example:

\[ Q(a) \iff \exists k. \forall n. a(n) = n + k \]

or

\[ Q(a) \iff \forall n. \exists k. a(n) = n + k \]

- Binary action relations are similar:

\[ a [R] b \iff \forall m, n. \exists k. a(n) = n + k \land b(m) = m - k \]

It is hard to see how to generalize traditional history invariants to binary relations.
Where do action invariants come from?

- [O’Hearn and Tennent, 1993]: *Relational parametricity and local variables.*
  - Showed that the information hiding aspects of local variables can be modelled using relational parametricity (state-based relations).
  - [O’Hearn and Reynolds, 2000] used a variant using strict functions (linear functions) to get rid of some snapback effects. Proved it fully abstract for up to second-order function types.
- [Reddy, 1993]: *Global state considered unnecessary: Introduction to object-based semantics.*
  - Produced an event-based description of objects and classes, so that information hiding is directly represented.
  - Only observable behavior of classes is captured in the semantics. *No data representations.*
  - [O’Hearn and Reddy, 1995] proved it fully abstract for up to second-order function types.
- 1993-98: I thought how to combine the best features of both the models.
  - Algebraic automata theory was the inspiration.
Where do action invariants come from?

- [O’Hearn and Tennent, 1993]: *Relational parametricity and local variables.*
  - Showed that the information hiding aspects of local variables can be modelled using relational parametricity (state-based relations).
  - [O’Hearn and Reynolds, 2000] used a variant using strict functions (linear functions) to get rid of some snapback effects. Proved it fully abstract for up to second-order function types.
- [Reddy, 1993]: *Global state considered unnecessary: Introduction to object-based semantics.*
  - Produced an event-based description of objects and classes, so that information hiding is directly represented.
  - Only observable behavior of classes is captured in the semantics. *No data representations.*
  - [O’Hearn and Reddy, 1995] proved it fully abstract for up to second-order function types.

1993-98: I thought how to combine the best features of both the models.

Algebraic automata theory was the inspiration.
Where do action invariants come from?

- [O’Hearn and Tennent, 1993]: *Relational parametricity and local variables*.
  - Showed that the information hiding aspects of local variables can be modelled using relational parametricity (state-based relations).
  - [O’Hearn and Reynolds, 2000] used a variant using strict functions (linear functions) to get rid of some snapback effects. Proved it fully abstract for up to second-order function types.
- [Reddy, 1993]: *Global state considered unnecessary: Introduction to object-based semantics*.
  - Produced an event-based description of objects and classes, so that information hiding is directly represented.
  - Only observable behavior of classes is captured in the semantics.
  - *No data representations*.
  - [O’Hearn and Reddy, 1995] proved it fully abstract for up to second-order function types.
- 1993-98: I thought how to combine the best features of both the models.
  - Algebraic automata theory was the inspiration.
O’Hearn-Tennent vs. Event-based

- **Example in favour of the O’Hearn-Tennent model:**

  \[
  [x := x + 1; x := x + 1] \overset{?}{=} [x := x + 2]
  \]

- **Example in favour of the Event-based model:**

  \[
  \text{\texttt{class}} \quad \text{local Var[int] x; } \\
  \text{init } x := 0; \\
  \text{meth} \\
  \{ val() = x, \\
  inc() = (x := x + 1) \} \]

  \[
  \overset{?}{=} \text{\texttt{class}} \quad \text{local Var[int] x; } \\
  \text{init } x := 0; \\
  \text{meth} \\
  \{ val() = -x, \\
  inc() = (x := x - 1) \}
  \]
Example in favour of the O’Hearn-Tennent model:

\[
\begin{align*}
[x := x + 1; x := x + 1] & \equiv [x := x + 2] \\
\end{align*}
\]

Example in favour of the Event-based model:

```
{class
  local Var[int] x;
  init x := 0;
  meth
  {val() = x,
   inc() = (x := x + 1)}
} ?= {class
  local Var[int] x;
  init x := 0;
  meth
  {val() = -x,
   inc() = (x := x - 1)}
}
```
The O’Hearn-Tennent approach is good for computation, bad for data.

The Event-based approach is good for data, bad for computation.

Is it possible to have the best of both worlds?

Can we have external behavior of agents described in terms of events/traces, but the internal behavior as *extensional* state transformation?
O’Hearn-Tennent vs. Event-based

- The O’Hearn-Tennent approach is good for computation, bad for data.
- The Event-based approach is good for data, bad for computation.
- Is it possible to have the best of both worlds?
- Can we have external behavior of agents described in terms of events/traces, but the internal behavior as extensional state transformation?
The O’Hearn-Tennent approach is good for computation, bad for data.
The Event-based approach is good for data, bad for computation.
Is it possible to have the best of both worlds?
Can we have external behavior of agents described in terms of events/traces, but the internal behavior as extensional state transformation?
Semiautomata

\[ \langle Q, \Sigma, \alpha : \Sigma \rightarrow [Q \rightarrow Q] \rangle \]

- \( Q \) is a set of states.
- \( \Sigma \) is a set of events.
- \( \alpha \) interprets events as state transformations.
- The O’Hearn-Tennent model focuses on \( Q \).
- The Event-based model focuses on \( \Sigma \).
- Automata provide a framework to combine the two.
Semiautomata

\[ \langle Q, \Sigma, \alpha : \Sigma \rightarrow [Q \rightarrow Q] \rangle \]

- \( Q \) is a set of states.
- \( \Sigma \) is a set of events.
- \( \alpha \) interprets events as state transformations.
- The O’Hearn-Tennent model focuses on \( Q \).
- The Event-based model focuses on \( \Sigma \).
- Automata provide a framework to combine nthe two.
Transformation monoids represent an “algebraic” version of semiautomata:

\[ \langle Q, \ T \subseteq [Q \rightarrow Q] \rangle \]

- \( Q \) is a set of states.
- \( T \) is a submonoid of state transformations.
  - \( [Q \rightarrow Q] \) is the set of state transformations.
  - Sequential composition is “multiplication”.
  - The identity transformation is the “unit”.
- The elements of \( T \) are thought of as “actions”.
  - More abstract variants of events.
  - Composable
Section 3

Examples
Example 1: Counters (state-based)

- A state-based model of counter objects:

\[
\langle Q = \text{Int}, \ q_0 = 0, \{\text{val} : Q \rightarrow \text{Int} = \lambda n. \ n, \\
\text{inc} : Q \rightarrow Q = \lambda n. \ n + 1\}\rangle
\]

- An alternative model of counter objects:

\[
\langle Q' = \text{Int}, \ q'_0 = 0, \{\text{val}' : Q' \rightarrow \text{Int} = \lambda n. \ -n, \\
\text{inc}' : Q' \rightarrow Q' = \lambda n. \ n - 1\}\rangle
\]

- Their equivalence can be shown using a simulation relation:

\[
Q^R n [\sim] n' \iff n \geq 0 \land n' = -n
\]

- The verification conditions are:

\[
\text{val} [R \rightarrow \Delta_{\text{Int}}] \text{val}' \quad \text{inc} [R \rightarrow R] \text{inc}'
\]
Example 1: Counters (state-based)

- A state-based model of counter objects:
  \[
  \langle Q = \text{Int}, \; q_0 = 0, \{ \text{val} : Q \rightarrow \text{Int} = \lambda n. \; n, \\
  \text{inc} : Q \rightarrow Q = \lambda n. \; n + 1 \} \rangle
  \]

- An alternative model of counter objects:
  \[
  \langle Q' = \text{Int}, \; q'_0 = 0, \{ \text{val}' : Q' \rightarrow \text{Int} = \lambda n. \; -n, \\
  \text{inc}' : Q' \rightarrow Q' = \lambda n. \; n - 1 \} \rangle
  \]

- Their equivalence can be shown using a simulation relation:
  \[
  Q \xrightarrow{R} n \xleftarrow{R} n' \iff n \geq 0 \land n' = -n
  \]

- The verification conditions are:
  \[
  \text{val} [R \rightarrow \Delta_{\text{Int}}] \; \text{val}' \quad \text{inc} [R \rightarrow R] \; \text{inc}'
  \]
Example 1: Counters (state-based)

- A state-based model of counter objects:

  \[\langle Q = Int, \ q_0 = 0, \ \{\text{val} : Q \rightarrow Int = \lambda n. n, \ 
  \text{inc} : Q \rightarrow Q = \lambda n. n + 1\} \rangle\]

- An alternative model of counter objects:

  \[\langle Q' = Int, \ q'_0 = 0, \ \{\text{val}' : Q' \rightarrow Int = \lambda n. -n, \ 
  \text{inc}' : Q' \rightarrow Q' = \lambda n. n - 1\} \rangle\]

- Their equivalence can be shown using a simulation relation:

  \[R \sqsubseteq n [R] n' \iff n \geq 0 \land n' = -n\]

- The verification conditions are:

  \[\text{val} \ [R \rightarrow \Delta_{Int}] \ \text{val}' \ \ \text{inc} \ [R \rightarrow R] \ \text{inc}'\]
Example 1: Counters (automata-based)

- An automata-based model of counter objects:

\[
\langle Q = \text{Int}, T = \text{Int}^+, q_0 = 0, \{ \text{val} : Q \rightarrow \text{Int} = \lambda n. n, \text{inc} : T = \lambda n. n + 1 \} \rangle
\]

\[
\text{Int}^+ = \text{down closure of } \{ \lambda n. n + k \mid k \geq 0 \}
\]

- The alternative model of counter objects:

\[
\langle Q' = \text{Int}, T' = \text{Int}^-, q'_0 = 0, \{ \text{val}' : Q' \rightarrow \text{Int} = \lambda n. -n, \text{inc}' : T' = \lambda n. n - 1 \} \rangle
\]

\[
\text{Int}^- = \text{down closure of } \{ \lambda n. n - k \mid k \geq 0 \}
\]

- Their equivalence is shown using two relations:

\[
\begin{align*}
Q & \quad T \\
R_Q & \quad R_T \\
Q' & \quad T'
\end{align*}
\]

\[
\begin{align*}
R_Q &
\begin{cases}
\text{n} & \text{[R}_Q\text{]} \quad \text{n'} & \iff \quad \text{n} \geq 0 \land \text{n'} = -\text{n} \\
\text{a} & \text{[R}_T\text{]} \quad \text{a'} & \iff \quad \forall \text{n}, \text{n'}. \text{a(n)} - \text{n} \simeq -(\text{a(n')} - \text{n'})
\end{cases}
\end{align*}
\]

- The verification conditions are:

\[
\begin{align*}
\text{val} \left[ R_Q \rightarrow \Delta_{\text{Int}} \right] & \quad \text{val}' \\
\text{inc} \left[ R_T \right] & \quad \text{inc}'
\end{align*}
\]
Example 1: Counters (automata-based)

- An automata-based model of counter objects:

\[ \langle Q = \text{Int}, \; T = \text{Int}^+, \; q_0 = 0, \{ \text{val} : Q \rightarrow \text{Int} = \lambda n. \; n, \]
\[ \text{inc} : T = \lambda n. \; n + 1 \rangle \]

\[ \text{Int}^+ = \text{down closure of } \{ \lambda n. \; n + k \mid k \geq 0 \} \]

- The alternative model of counter objects:

\[ \langle Q' = \text{Int}, \; T' = \text{Int}^-, \; q'_0 = 0, \{ \text{val}' : Q' \rightarrow \text{Int} = \lambda n. \; -n, \]
\[ \text{inc}' : T' = \lambda n. \; n - 1 \rangle \]

\[ \text{Int}^- = \text{down closure of } \{ \lambda n. \; n - k \mid k \geq 0 \} \]

Their equivalence is shown using two relations:

\[ Q \overset{R_Q}{\leftrightarrow} Q', \quad T \overset{R_T}{\leftrightarrow} T' \]

\[ n \left[ R_Q \right] n' \iff n \geq 0 \land n' = -n \]

\[ a \left[ R_T \right] a' \iff \forall n, n'. \; a(n) - n \simeq -(a(n') - n') \]

- The verification conditions are:

\[ \text{val} \left[ R_Q \rightarrow \Delta_{\text{Int}} \right] \text{val}' \quad \text{inc} \left[ R_T \right] \text{inc}' \]
Example 1: Counters (automata-based)

- An automata-based model of counter objects:
  \[
  \langle Q = \text{Int}, T = \text{Int}^+, q_0 = 0, \{\text{val} : Q \rightarrow \text{Int} = \lambda n. n, \\
  \text{inc} : T = \lambda n. n + 1\} \rangle
  \]
  \[
  \text{Int}^+ = \text{down closure of } \{\lambda n. n + k | k \geq 0\}
  \]

- The alternative model of counter objects:
  \[
  \langle Q' = \text{Int}, T' = \text{Int}^-, q'_0 = 0, \{\text{val}' : Q' \rightarrow \text{Int} = \lambda n. -n, \\
  \text{inc}' : T' = \lambda n. n - 1\} \rangle
  \]
  \[
  \text{Int}^- = \text{down closure of } \{\lambda n. n - k | k \geq 0\}
  \]

- Their equivalence is shown using two relations:
  \[
  Q \xleftarrow{R_Q} T \quad n \leftarrow \Delta_{\text{Int}} \quad n' \quad n \geq 0 \land n' = -n
  \]
  \[
  Q' \xleftarrow{R_Q'} T' \quad a \leftarrow \Delta_{\text{Int}} \quad a' \quad \forall n, n'. a(n) - n \simeq -(a(n') - n')
  \]

- The verification conditions are:
  \[
  \text{val} [R_Q] \Delta_{\text{Int}} \quad \text{val}' \quad \text{inc} [R_T] \quad \text{inc}'
  \]
In addition to the state sets \((Q)\), we also represent the allowed state transformations \((T)\), with some natural coherence conditions. The state change operations of the objects are of type \(T\), not \(Q \rightarrow Q\). So, one can only perform allowed state transformations.

The coherence conditions ensure that we cannot “cheat.” New transformations can be made only by composing the allowed transformations.
Reynolds’s Idealized Algol is a simply typed lambda calculus (CBN) with base types for state manipulation:

\[
\text{comm} \quad \text{exp}[\delta] \quad \text{val}[\delta]
\]

where \(\delta\) ranges over “data” types.

Sample constants:

- \(0\) : \(\text{exp[\text{int}]\)}
- \(+\) : \(\text{exp[\text{int}] \times \text{exp[\text{int}]} \rightarrow \text{exp[\text{int}]}}
- \text{skip}\) : \(\text{comm}\)
- \(-; -\) : \(\text{comm} \times \text{comm} \rightarrow \text{comm}\)
- \text{diverge}\) : \(\text{comm}\)
- \text{if}\) : \(\text{exp[\text{bool}] \times \text{comm} \times \text{comm} \rightarrow \text{comm}}

Interlude: IA+

- IA+ extends Idealized Algol with classes.
- \textbf{cls }\theta \text{ - the type of classes that have instances of type } \theta.
- Primitive class (constant):
  \begin{equation*}
  \text{Var}[\delta]: \text{cls} \{\text{get : exp}[\delta], \text{put : val}[\delta] \rightarrow \text{comm}\}
  \end{equation*}
- Class definition (and its equivalent ML fragment):
  \begin{equation*}
  \text{class }\theta \lambda(). \text{let } x : \theta = \text{newC()} \text{ in } A; M
  \end{equation*}
  \begin{align*}
  \text{local } C x; \\
  \text{init } A; \\
  \text{meth } M
  \end{align*}
- Class instantiation:
  \begin{equation*}
  \text{new } C o. P(o)
  \end{equation*}
- Additional constants:
  \begin{align*}
  := & : \text{var}[\delta] \times \text{exp}[\delta] \rightarrow \text{comm} \\
  \text{deref} & : \text{var}[\delta] \rightarrow \text{exp}[\delta]
  \end{align*}
IA+ extends Idealized Algol with classes.

**cls** $\theta$ - the *type* of classes that have instances of type $\theta$.

**Primitive class (constant):**

$$\text{Var}[\delta] : \textbf{cls} \{ \text{get} : \exp[\delta], \text{put} : \textbf{val}[\delta] \rightarrow \textbf{comm} \}$$

**Class definition (and its equivalent ML fragment):**

```ml
class : $\theta$

let $x : \theta = \text{newC}()$

local $C$ $x$;
init $A$;
meth $M$
```

**Class instantiation:**

```ml
\text{new} \ C \ o. \ P(o)
```

**Additional constants:**

```plaintext
:= \quad : \ var[\delta] \times \ exp[\delta] \rightarrow \textbf{comm}

deref \quad : \ var[\delta] \rightarrow \exp[\delta]
```
IA+ extends Idealized Algol with classes.

- **cls** \(\theta\) - the type of classes that have instances of type \(\theta\).
- Primitive class (constant):
  
  \[
  \text{Var}[\delta] : \text{cls} \{ \text{get} : \text{exp}[\delta], \text{put} : \text{val}[\delta] \rightarrow \text{comm} \}
  \]

- Class definition (and its equivalent ML fragment):
  
  \[
  \text{class} : \theta \\
  \text{local} C \ x; \\
  \text{init} A; \\
  \text{meth} M
  \]

- Class instantiation:
  
  \[
  \text{new} \ C \ o. \ P(o)
  \]

- Additional constants:
  
  \[
  := : \text{var}[\delta] \times \text{exp}[\delta] \rightarrow \text{comm} \\
  \text{deref} : \text{var}[\delta] \rightarrow \text{exp}[\delta]
  \]
IA+ extends Idealized Algol with classes.

- **cls** $\theta$ - the *type* of classes that have instances of type $\theta$.

- **Primitive class (constant):**

  \[
  \text{Var}[\delta] : \text{cls} \{ \text{get} : \text{exp}[\delta], \text{put} : \text{val}[\delta] \rightarrow \text{comm} \}
  \]

- **Class definition (and its equivalent ML fragment):**

  ```ml
  class : $\theta$ 
  \lambda(). let $x : \theta = \text{newC}()$
  local $C \ x$; 
  init $A$; 
  meth $M$
  ```

- **Class instantiation:**

  \[
  \text{new} \ C \ o. \ P(o)
  \]

- **Additional constants:**

  \[
  := : \text{var}[\delta] \times \text{exp}[\delta] \rightarrow \text{comm} \\
  \text{deref} : \text{var}[\delta] \rightarrow \text{exp}[\delta]
  \]
class : {val : exp[int], inc : comm}
local Var[int] x;
init x := 0;
meth {val = deref x, inc = (x := (deref x) + 1)}
"Awkward" Example [Pitts & Stark, 1998]

Example written in IA+ (Idealized Algol extended with classes):

\[
C = \text{class} : \{m : \text{comm} \rightarrow \text{comm}\}
\]
\[
\quad \text{local \ Var[\text{int}] \ x;}
\]
\[
\quad \text{init} \ x := 0;
\]
\[
\quad \text{meth} \ \{m = \lambda c. \ x := 1; \ c; \ test(x = 1)\}
\]

\[
test(b) \triangleq \text{if } b \text{ then skip else diverge}
\]

- Does \( m \) terminate (assuming \( c \) terminates)? Equivalently, do we believe that \( c \) does not change \( x \)?
  - Tommy Hacker says “yes”. \( x \) is a local variable of the class. So, \( c \) can’t have access to it.
  - What say you?
“Awkward” Example [Pitts & Stark, 1998]

Example written in IA+ (Idealized Algol extended with classes):

\[
C = \text{class} : \{ m : \text{comm} \rightarrow \text{comm} \}
\]

local Var[int] x;
init x := 0;
meth \{ m = \lambda c. x := 1; c; test(x = 1) \}

\[
test(b) \triangleq \text{if } b \text{ then skip else diverge}
\]

- Does \( m \) terminate (assuming \( c \) terminates)? Equivalently, do we believe that \( c \) does not change \( x \)?
- Tommy Hacker says “yes”. \( x \) is a local variable of the class. So, \( c \) can’t have access to it.
- What say you?
“Awkward” Example [Pitts & Stark, 1998]

\[
C = \text{class} : \{ m : \text{comm} \rightarrow \text{comm} \}
\]

\[
\begin{align*}
\text{local} & \ \text{Var}[\text{int}] \ x; \\
\text{init} & \ x := 0; \\
\text{meth} & \ \{ m = \lambda c. \ x := 1; c; \text{test}(x = 1) \}
\end{align*}
\]

- It is not sound to say that \( c \) does not have “access” to \( x \).
- Consider the following client:

\[
\text{new } C \ o. \ // \ create \ an \ instance \ of \ C \ and \ call \ it \ o \\
\ o.m (o.m \ \text{skip})
\]

- When \( o.m \) is called, the argument passed involves another call to \( o.m \). So the argument \( c \) can change \( x \).
- The correct argument says that the \textit{the only change} \( c \) can make to \( x \) is to set it to 1. If it does that change, the test will still succeed.
“Awkward” Example [Pitts & Stark, 1998]

\[ C = \text{class} : \{ m : \text{comm} \rightarrow \text{comm} \} \]

\quad \text{local} \ Var[\text{int}] \ x; \\
\quad \text{init} \ x := 0; \\
\quad \text{meth} \ \{ m = \lambda c. \ x := 1; c; \text{test}(x = 1) \} \]

- It is not sound to say that \( c \) does not have “access” to \( x \).
- Consider the following client:

  \begin{verbatim}
  new C o.  // create an instance of C and call it o
  o.m (o.m skip)
  \end{verbatim}

- When \( o.m \) is called, the argument passed involves another call to \( o.m \). So the argument \( c \) can change \( x \).
- The **correct argument** says that the *the only change* \( c \) can make to \( x \) is to set it to 1. If it does that change, the test will still succeed.
A “Awkward” Example [Pitts & Stark, 1998]

\[ C = \text{class} : \{ m : \text{comm} \rightarrow \text{comm} \} \]

\begin{align*}
\text{local} & \quad \text{Var}[\text{int}] \ x; \\
\text{init} & \quad x := 0; \\
\text{meth} & \quad \{ m = \lambda c. x := 1; c; \text{test}(x = 1) \} \\
\end{align*}

- We can formalize the correct argument by formulating a two part invariant:

\[ \begin{align*}
P_Q(x) & \iff x = 0 \lor x = 1 \\
P_T(a) & \iff a \subseteq (\lambda n. \ n) \lor a \subseteq (\lambda n. \ 1) \\
\end{align*} \]

- We must show that the body of \( m \) preserves the two-part invariant, while \textit{assuming} that the argument \( c \) preserves the two-part invariant.
“Very awkward” Example: Dreyer, Neis, Birkedal, 2010

\[
C = \text{class : } \{ m : \text{comm} \rightarrow \text{comm}\}
\]

\[
\begin{align*}
\text{local } & \text{Var[\text{int}]} \ x; \\
\text{init } & x := 0; \\
\text{meth } & \{ m = \lambda c. \ x := 0; c; x := 1; c; \text{test}(\ x = 1) \}
\end{align*}
\]

- This is a twist on the “awkward” example, by introducing an additional assignment \( x := 0 \) in \( m \).
- This seems to suggest that we should enlarge the action invariant to include \( \lambda n. 0 \).
- No need. The old invariant still works.

\[
\begin{align*}
P_Q(x) & \iff x = 0 \lor x = 1 \\
P_T(a) & \iff a \subseteq (\lambda n. n) \lor a \subseteq (\lambda n. 1)
\end{align*}
\]

- The first call to \( c \) will either leave \( x \) unchanged or set it to 1. But, we don’t care either way. \( m \) will immediately overwrite \( x \) with 1.
- The second call to \( c \) is the same as before.
C = class : \{ m : \text{comm} \rightarrow \text{comm} \}
local Var[\text{int}] x;
init x := 0;
meth \{ m = \lambda c. x := 0; c; x := 1; c; test(x = 1) \}

Dreyer et al. prove that \( m \) terminates (assuming \( c \) terminates), by using two separate kinds of transitions:

- **Private transitions**, such as \( x := 0 \), which represent internal state transitions inside methods.
- **Public transitions**, such as \( x := 1 \), which are visible to the callers.

We don’t find a need for any special treatment of “private transitions.” They are handled automatically by the normal parametricity reasoning.
Section 4

Semantic Overview
The semantics of Idealized Algol is given as a possible world semantics.

- \( \mathbf{W} \) - a category of worlds (with relations), formally a *parametricity graph*.
  - The objects of \( \mathbf{W} \) represent *store shapes*.
  - Morphisms \( f : X \rightarrow \mathbf{W} \) in \( \mathbf{W} \) represent the idea that \( X \) is a possible *future world* of \( \mathbf{W} \), typically a larger store than \( \mathbf{W} \).

Types of the programming language \( \theta \) are interpreted as *functors* (with some technicalities):

\[
\llbracket \theta \rrbracket : \mathbf{W}^{\text{op}} \rightarrow \text{CPO}
\]

Terms of the programming language are interpreted as *parametric transformations*:

\[
\begin{align*}
\Gamma : \theta_1, \ldots, \theta_n & \vdash M : \theta' \\
\llbracket M \rrbracket : \llbracket \overline{\theta} \rrbracket & \rightarrow \llbracket \overline{\theta'} \rrbracket
\end{align*}
\]
Possible world semantics

- The semantics of Idealized Algol is given as a possible world semantics.
- \( W \) - a category of worlds (with relations), formally a *parametricity graph*.
  - The objects of \( W \) represent *store shapes*.
  - Morphisms \( f : X \to W \) in \( W \) represent the idea that \( X \) is a possible *future world* of \( W \), typically a larger store than \( W \).
- Types of the programming language \( \theta \) are interpreted as *functors* (with some technicalities):
  \[
  \llbracket \theta \rrbracket : \mathcal{W}^{\text{op}} \to \text{CPO}
  \]
- Terms of the programming language are interpreted as *parametric transformations*:
  \[
  x_1 : \theta_1, \ldots, x_n : \theta_n \vdash M : \theta' \quad \llbracket M \rrbracket : \llbracket \theta \rrbracket \to \llbracket \theta' \rrbracket
  \]
Denote \([\theta]\) by \(F\).

For each world \(W\), \(F(W)\) is the set of meanings of type \(\theta\) for store \(W\).

Morphisms, relations and squares are mapped as well:

\[
\begin{array}{c}
X \\
\downarrow f \\
W
\end{array} \quad \Rightarrow \quad
\begin{array}{c}
F(X) \\
\downarrow F(f) \\
F(W)
\end{array}
\]

\[
\begin{array}{c}
X' \\
\uparrow R \\
X
\end{array} \quad \Rightarrow \quad
\begin{array}{c}
F(X) \\
\downarrow F(R) \\
F(X')
\end{array}
\]

\[
\begin{array}{c}
X \\
\downarrow S \\
X'
\end{array} \quad \Box \quad
\begin{array}{c}
W \\
\downarrow R \\
W'
\end{array} \quad \Rightarrow \quad
\begin{array}{c}
F(X) \\
\downarrow F(f) \\
F(W)
\end{array} \quad \Box \quad
\begin{array}{c}
F(X') \\
\downarrow F(f') \\
F(W')
\end{array}
\]

\[
\begin{array}{c}
F(S) \\
\downarrow F(R)
\end{array} \quad \Rightarrow \quad
\begin{array}{c}
F(S) \\
\downarrow F(R)
\end{array}
\]

\[
\begin{array}{c}
F(X') \\
\downarrow F(f') \\
F(W')
\end{array} \quad \Box \quad
\begin{array}{c}
F(X') \\
\downarrow F(f') \\
F(W')
\end{array}
\]
Possible world semantics - contd

- The meaning of a term is a uniform family of functions, preserving all possible relations between store shapes:

  \[ \begin{align*}
  &X \\
  \downarrow & R \\
  X' & \end{align*} \]

  \[ \begin{align*}
  &\lbrack \theta \rbrack (X) \xrightarrow{\lbrack M \rbrack_X} \lbrack \theta' \rbrack (X) \\
  \downarrow & \\
  &\lbrack \theta \rbrack (R) \xrightarrow{\lbrack \theta' \rbrack (R) \\
  \downarrow & \\
  &\lbrack \theta \rbrack (X') \xrightarrow{\lbrack M \rbrack_{X'}} \lbrack \theta' \rbrack (X')
  \end{align*} \]

- The uniformity property says that the meanings of terms act the same way for all store shapes.
Possible world semantics - contd

- The meanings of types have this form:

\[ [\text{comm}] (W) = \ldots \]
\[ [\text{exp}[\delta]] (W) = \ldots \]
\[ [\text{val}[\delta]] (W) = [\delta] \]
\[ [\theta_1 \times \theta_2] (W) = [\theta_1] (W) \times [\theta_2] (W) \]
\[ [\theta \rightarrow \theta'] (W) = \forall h : \text{X} \rightarrow \text{W} [\theta](\text{X}) \rightarrow [\theta'] (\text{X}) \]
\[ [\text{cls } \theta] (W) = \exists Z (Q_Z \bot \times [\theta](Z)) \]

Note the correspondence with the counter classes seen earlier:

\[ \langle Q = \text{Int}, T = \text{Int}^+, q_0 = 0, \{ \text{val} : Q \rightarrow \text{Int} = \lambda n. n, \text{inc} : T = \lambda n. n + 1 \rangle \]
\[ \text{Int}^+ = \text{down closure of } \{ \lambda n. n + k \mid k \geq 0 \} \]

- We use automata-theoretic ideas to define the possible worlds \( W \) and the interpretation of the base types \text{comm} and \text{exp}.
Possible world semantics - contd

- The meanings of types have this form:

\[
\begin{align*}
\llbracket \text{comm} \rrbracket(W) &= \ldots \\
\llbracket \exp[\delta] \rrbracket(W) &= \ldots \\
\llbracket \text{val}[\delta] \rrbracket(W) &= \llbracket \delta \rrbracket \\
\llbracket \theta_1 \times \theta_2 \rrbracket(W) &= \llbracket \theta_1 \rrbracket(W) \times \llbracket \theta_2 \rrbracket(W) \\
\llbracket \theta \to \theta' \rrbracket(W) &= \forall h : X \to W \ [\llbracket \theta \rrbracket(X) \to \llbracket \theta' \rrbracket(X)] \\
\llbracket \text{cls} \theta \rrbracket(W) &= \exists Z (QZ_\bot \times \llbracket \theta \rrbracket(Z))
\end{align*}
\]

- Note the correspondence with the counter classes seen earlier:

\[
\langle Q = \text{Int}, \ T = \text{Int}^+, \ q_0 = 0, \{ \text{val} : Q \to \text{Int} = \lambda n. n, \text{inc} : T = \lambda n. n + 1 \} \rangle
\]

\[
\text{Int}^+ = \text{down closure of } \{ \lambda n. n + k \mid k \geq 0 \}
\]

- We use automata-theoretic ideas to define the possible worlds \( W \) and the interpretation of the base types \text{comm} and \exp.
Semantics of types

The meanings of types have this form:

\[
\begin{align*}
\llbracket \text{comm} \rrbracket(W) & = \mathcal{T}_W \\
\llbracket \text{exp}[\delta] \rrbracket(W) & = [Q_W \rightarrow \llbracket \delta \rrbracket] \\
\llbracket \text{val}[\delta] \rrbracket(W) & = \llbracket \delta \rrbracket \\
\llbracket \theta_1 \times \theta_2 \rrbracket(W) & = \llbracket \theta_1 \rrbracket(W) \rightarrow \llbracket \theta_2 \rrbracket(W) \\
\llbracket \theta \rightarrow \theta' \rrbracket(W) & = \forall h : X \rightarrow W \llbracket \theta \rrbracket(X) \rightarrow \llbracket \theta' \rrbracket(X) \\
\llbracket \text{cls} \; \theta \rrbracket(W) & = \exists Z (Q_Z) \perp \times \llbracket \theta \rrbracket(Z)
\end{align*}
\]

Note that the meanings of commands are the allowed transformations of the store (an automaton).

The corresponding relational actions are as expected.

\[
\begin{align*}
\llbracket \text{comm} \rrbracket(R) & = R_T. \quad \llbracket \text{exp}[\delta] \rrbracket(R) = [R_Q \rightarrow \Delta[\delta]].
\end{align*}
\]
Semantics of primitives

\[ \text{cond}^E : \text{EXP}_{\text{Bool}} \times \text{EXP}_\delta \times \text{EXP}_\delta \rightarrow \text{EXP}_\delta \]
\[ \text{cond}_W^E(e, e_1, e_2) = \lambda s. (\lambda v. v \rightarrow e_1(s); e_2(s))^*(e(s)) \]
\[ \text{cond}^C : \text{EXP}_{\text{Bool}} \times \text{COMM} \times \text{COMM} \rightarrow \text{COMM} \]
\[ \text{cond}_W^C(e, a, b) = \text{read}_W \lambda s. (\lambda v. v \rightarrow a; b)^*(e(s)) \]
\[ \text{deref} : \text{VAR}_\delta \rightarrow \text{EXP}_\delta \]
\[ \text{deref}_W(e, a) = e \]
\[ \text{assign} : \text{VAR}_\delta \times \text{EXP}_\delta \rightarrow \text{COMM} \]
\[ \text{assign}_W((d, a), e) = \text{read}_W \lambda s. a^*(e(s)) \]
\[ \text{Var}[\delta] : 1 \rightarrow \text{CLS} \text{ VAR}_\delta \]
\[ \text{Var}[\delta]_W(\ast) = \langle \mid V, \text{init}_\delta, \text{mkvar} \rangle \]
\[ \text{where } V = (\delta, T(\delta)) \quad \text{mkvar} = (\lambda n. n, \lambda k. \lambda n. k) \]
\[ \text{newvar} : (\text{VAR}_\delta \Rightarrow \text{COMM}) \rightarrow \text{COMM} \]
\[ \text{newvar}_W(p) = (\lambda s. (s, \text{init}_\delta)) \cdot p[\pi_1](\text{mkvar}^{W\ast V}_V) \cdot (\lambda (s, n). s) \]
Theorem: The semantics is parametric.

This implies the soundness of the reasoning principles with two-part simulation relations and two-part invariants.

Several representation results without divergence, e.g.,

\[
\lbrack \text{comm} \rightarrow \text{comm} \rbrack(1) \cong \text{Nat}
\]

Some representation results with divergence, especially for passive types:

\[
\lbrack \text{comm} \rightarrow \text{exp}[\delta] \rbrack(\mathcal{W}) \cong \lbrack \text{exp}[\delta] \rbrack(\mathcal{W})
\]
Section 5

Conclusion
Summary

- We made a small beginning to bridge the gap between state-based (Scott-Strachey) and event-based (Milner-Hoare) paradigms in semantics.

- Automata seem to provide the right structure to capture the intuitions about “agents” and “objects” that have internal structure and external behaviour.

- This seems to be quite worthwhile exercise as it gives simple reasoning principles to prove equivalences that were heretofore difficult to prove using denotational methods.
Further work - Theory

- Partial functions vs. strict functions.
- Weaken the Reynolds diagonal.
  Treat reading as a separate action.
- Call by value.
- Concurrency.
- Higher-order state.
Further work - Applications

- Heap storage.
- Programming logics (Hoare logic, specification logic, separation logic).
- Rely-guarantee and deny-guarantee reasoning.