States and Actions: An Automata-theoretic Model of Objects

Uday S. Reddy[1] Brian P. Dunphy[2]

[1]University of Birmingham

[2]University of Illinois at Urbana-Champaign

Portland, Oct 2011

Outline

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?

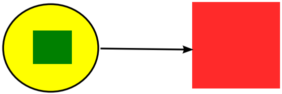
Reynolds [1981] anticipates many of these ideas.

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, ingegrity),
  - reasoning principles based on **simulation relations** (useful for program equivalence and data refinement).

Information hiding leads to Invariants

- Consider:

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

- $x \geq 0$ is invariant in $x := x + 1$.
- Hence, $x \geq 0$ is invariant in the entire whle-loop.

$$\frac{\{P\}\ C\ \{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ C\ \{P\}}$$

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

$$\frac{\{P\}\ C_1\ \{P\} \quad \{P\}\ C_2\ \{P\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \{P\}}$$

Information hiding leads to Invariants - 2

- The same principle also applies to user-defined higher-order procedure constants (with no free identifiers) $F : \mathbf{comm} \to \mathbf{comm}$:

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

  We don't even have to know what F does to establish the invariant principle!
- Why do these principles work?
- Answer: Information hiding.
- **while** is a constant (no free identifiers) of type:

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

  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 Invariants - 2

- The same principle also applies to user-defined higher-order procedure constants (with no free identifiers) $F : \mathbf{comm} \to \mathbf{comm}$:

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

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

- Why do these principles work?
- Answer: Information hiding.
- **while** is a constant (no free identifiers) of type:

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

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 Invariants - 2

- The same principle also applies to user-defined higher-order procedure constants (with no free identifiers) $F : \mathbf{comm} \to \mathbf{comm}$:

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

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

- Why do these principles work?
- Answer: Information hiding.
- **while** is a constant (no free identifiers) of type:

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

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 Invariants - 2

- The same principle also applies to user-defined higher-order procedure constants (with no free identifiers) $F : \mathbf{comm} \to \mathbf{comm}$:

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

  We don't even have to know what F does to establish the invariant principle!
- Why do these principles work?
- Answer: Information hiding.
- **while** is a constant (no free identifiers) of type:

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

  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:

$$\begin{array}{l} x \leq 100 \; [R_{\mathbf{exp[bool]}}] \; y \geq -100 \\ (x := x + 1) \; [R_{\mathbf{comm}}] \; (y := y - 1) \end{array}$$

- Infer

$$\begin{array}{l} \mathbf{while}\ x \leq 100\ \mathbf{do}\ x := x + 1 \\ \qquad [R_{\mathbf{comm}}] \\ \mathbf{while}\ y \geq -100\ \mathbf{do}\ y := y - 1 \end{array}$$

- Again, it is not necessary to know what **while** does, in order to infer this fact.
- Evey constant combinator will preserve the binary simulation relation in the same way (including user-defined combinators).

Information hiding in OOP

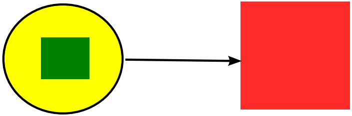
- The same ideas also work for O-O classes.

$$\begin{array}{l} \textbf{class} \\ \quad \textbf{local}\ \mathrm{Var}[\textbf{int}]\ x; \\ \quad \textbf{init}\ x := 0; \\ \quad \textbf{meth} \\ \quad\quad \{val() = x, \\ \quad\quad\ \ inc() = (x := x + 1)\} \end{array}$$

- 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.
- Agian, information hiding is what is at play: The variable x is hidden from the clients.

Relational parametricity

- 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:

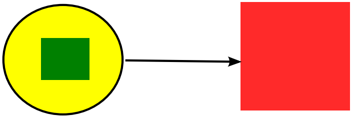


$$\text{client} : \forall_X F(X) \to K(X)$$

- 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.

Relational parametricity

- 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:



$$\text{client} : \forall_X F(X) \to K(X) \qquad \text{client} : (\exists_X F(X)) \to K$$

- 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

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:

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

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

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

  also satisfies the history invariant.
- Similar discussion as before applies: It does not matter what **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:

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

  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:

$$[\![x := x + 1; x := x + 1]\!] \stackrel{?}{=} [\![x := x + 2]\!]$$

- Example in favour of the Event-based model:

$$[\![\textbf{class}\ \textbf{local}\ \mathrm{Var}[\textbf{int}]\ x;\ \textbf{init}\ x := 0;\ \textbf{meth}\ \{val() = x,\ inc() = (x := x + 1)\}]\!] \stackrel{?}{=} [\![\textbf{class}\ \textbf{local}\ \mathrm{Var}[\textbf{int}]\ x;\ \textbf{init}\ x := 0;\ \textbf{meth}\ \{val() = -x,\ inc() = (x := x - 1)\}]\!]$$

O'Hearn-Tennent vs. Event-based

- Example in favour of the O'Hearn-Tennent model:

$$[\![x := x + 1; x := x + 1]\!] \quad \stackrel{?}{=} \quad [\![x := x + 2]\!]$$

- Example in favour of the Event-based model:

$$[\![\textbf{class}\ \ \textbf{local}\ \mathrm{Var}[\textbf{int}]\ x;\ \ \textbf{init}\ x := 0;\ \ \textbf{meth}\ \{val() = x,\ inc() = (x := x + 1)\}]\!] \quad \stackrel{?}{=} \quad [\![\textbf{class}\ \ \textbf{local}\ \mathrm{Var}[\textbf{int}]\ x;\ \ \textbf{init}\ x := 0;\ \ \textbf{meth}\ \{val() = -x,\ inc() = (x := x - 1)\}]\!]$$

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?

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?

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?

Semiautomata

$$\langle Q, \Sigma, \alpha : \Sigma \to [Q \rightharpoonup Q] \rangle$$

- Q is a set of states.
- Σ is a set of events.
- α interprets events as state transformations.
- The O'Hearn-Tennent model focuses on Q.
- The Event-based model focuses on Σ.
- Automata provide a framework to combine nthe two.

Semiautomata

$$\langle Q, \Sigma, \alpha : \Sigma \to [Q \rightharpoonup Q] \rangle$$

- Q is a set of states.
- Σ is a set of events.
- α interprets events as state transformations.
- The O'Hearn-Tennent model focuses on Q.
- The Event-based model focuses on Σ.
- Automata provide a framework to combine nthe two.

Transformation monoids

- Transformation monoids represent an "algebraic" version of semiautomata:

$$\langle Q,\ T \subseteq [Q \rightharpoonup Q] \rangle$$

- Q is a set of states.
- T is a submonoid of state transformations.
  - $[Q \rightharpoonup Q]$ 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:

$$\langle Q = Int,\ q_0 = 0, \{ val : Q \rightharpoonup Int = \lambda n.\, n,\\ inc : Q \rightharpoonup Q = \lambda n.\, n+1 \} \rangle$$

- An alternative model of counter objects:

$$\langle Q' = Int,\ q'_0 = 0, \{ val' : Q' \rightharpoonup Int = \lambda n.\, -n,\\ inc' : Q' \rightharpoonup Q' = \lambda n.\, n-1 \} \rangle$$

- Their equivalence can be shown using a simulation relation:

$$Q \overset{R}{\updownarrow} Q' \qquad n\ [R]\ n' \iff n \geq 0 \wedge n' = -n$$

- The verification conditions are:

$$val\ [R \rightharpoonup \Delta_{Int}]\ val' \qquad inc\ [R \rightharpoonup R]\ inc'$$

Example 1: Counters (state-based)

- A state-based model of counter objects:

$$\langle Q = Int,\ q_0 = 0, \{ val : Q \rightharpoonup Int = \lambda n.\, n, \\ inc : Q \rightharpoonup Q = \lambda n.\, n + 1 \} \rangle$$

- An alternative model of counter objects:

$$\langle Q' = Int,\ q_0' = 0, \{ val' : Q' \rightharpoonup Int = \lambda n.\, -n, \\ inc' : Q' \rightharpoonup Q' = \lambda n.\, n - 1 \} \rangle$$

- Their equivalence can be shown using a simulation relation:

$$Q \overset{R}{\updownarrow} Q' \qquad n\, [R]\, n' \iff n \geq 0 \wedge n' = -n$$

- The verification conditions are:

$$val\, [R \rightharpoonup \Delta_{Int}]\, val' \qquad inc\, [R \rightharpoonup R]\, inc'$$

Example 1: Counters (state-based)

- A state-based model of counter objects:

$$\langle Q = Int,\ q_0 = 0, \{ val : Q \rightharpoonup Int = \lambda n.\, n,\ \ inc : Q \rightharpoonup Q = \lambda n.\, n+1 \} \rangle$$

- An alternative model of counter objects:

$$\langle Q' = Int,\ q'_0 = 0, \{ val' : Q' \rightharpoonup Int = \lambda n.\, -n,\ \ inc' : Q' \rightharpoonup Q' = \lambda n.\, n-1 \} \rangle$$

- Their equivalence can be shown using a simulation relation:

$$Q \overset{R}{\longleftrightarrow} Q' \qquad n\ [R]\ n' \iff n \geq 0 \wedge n' = -n$$

- The verification conditions are:

$$val\ [R \rightharpoonup \Delta_{Int}]\ val' \qquad inc\ [R \rightharpoonup R]\ inc'$$

Example 1: Counters (automata-based)

- An automata-based model of counter objects:

$$\langle Q = Int,\ T = Int^+,\ q_0 = 0, \{ val : Q \rightharpoonup Int = \lambda n.\, n,\ \ inc : T = \lambda n.\, n + 1 \} \rangle$$

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

- The alternative model of counter objects:

$$\langle Q' = Int,\ T' = Int^-,\ q'_0 = 0, \{ val' : Q' \rightharpoonup Int = \lambda n.\, -n,\ \ inc' : T' = \lambda n.\, n - 1 \} \rangle$$

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

- Their equivalence is shown using *two* relations:

$$Q \overset{R_Q}{\updownarrow} Q' \qquad T \overset{R_T}{\updownarrow} T'$$

$$n\ [R_Q]\ n' \iff n \geq 0 \wedge n' = -n$$
$$a\ [R_T]\ a' \iff \forall n, n'.\ a(n) - n \simeq -(a(n') - n')$$

- The verification conditions are:

$$val\ [R_Q \rightharpoonup \Delta_{Int}]\ val' \qquad inc\ [R_T]\ inc'$$

Example 1: Counters (automata-based)

- An automata-based model of counter objects:

$$\langle Q = Int,\ T = Int^{+},\ q_0 = 0, \{ val : Q \rightharpoonup Int = \lambda n.\, n, \\ inc : T = \lambda n.\, n+1 \} \rangle$$

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

- The alternative model of counter objects:

$$\langle Q' = Int,\ T' = Int^{-},\ q'_0 = 0, \{ val' : Q' \rightharpoonup Int = \lambda n.\, -n, \\ inc' : T' = \lambda n.\, n-1 \} \rangle$$

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

- Their equivalence is shown using *two* relations:

$$Q \overset{R_Q}{\updownarrow} Q' \qquad T \overset{R_T}{\updownarrow} T'$$

$$n\ [R_Q]\ n' \iff n \geq 0 \wedge n' = -n$$
$$a\ [R_T]\ a' \iff \forall n, n'.\, a(n) - n \simeq -(a(n') - n')$$

- The verification conditions are:

$$val\ [R_Q \rightharpoonup \Delta_{Int}]\ val' \qquad inc\ [R_T]\ inc'$$

Example 1: Counters (automata-based)

- An automata-based model of counter objects:

$$\langle Q = Int,\ T = Int^{+},\ q_0 = 0, \{ val : Q \rightharpoonup Int = \lambda n.\, n,\ inc : T = \lambda n.\, n + 1 \} \rangle$$

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

- The alternative model of counter objects:

$$\langle Q' = Int,\ T' = Int^{-},\ q'_0 = 0, \{ val' : Q' \rightharpoonup Int = \lambda n.\, -n,\ inc' : T' = \lambda n.\, n - 1 \} \rangle$$

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

- Their equivalence is shown using *two* relations:

$$Q \overset{R_Q}{\updownarrow} Q' \qquad T \overset{R_T}{\updownarrow} T'$$

$$n \,[R_Q]\, n' \iff n \geq 0 \wedge n' = -n$$

$$a \,[R_T]\, a' \iff \forall n, n'.\, a(n) - n \simeq -(a(n') - n')$$

- The verification conditions are:

$$val \,[R_Q \rightharpoonup \Delta_{Int}]\, val' \qquad inc \,[R_T]\, inc'$$

Example 1: Counters (automata-based)

- 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 $Q \rightharpoonup Q$. 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.

Interlude: Idealized Algol

- Reynolds's Idealized Algol is a simply typed lambda calculus (CBN) with base types for state manipulation:

$$\textbf{comm} \qquad \textbf{exp}[\delta] \qquad \textbf{val}[\delta]$$

  where δ ranges over "data" types.
- Sample constants:

$$\begin{array}{lcl} 0 & : & \textbf{exp}[\textbf{int}] \\ + & : & \textbf{exp}[\textbf{int}] \times \textbf{exp}[\textbf{int}] \to \textbf{exp}[\textbf{int}] \\ \textbf{skip} & : & \textbf{comm} \\ -;- & : & \textbf{comm} \times \textbf{comm} \to \textbf{comm} \\ \textbf{diverge} & : & \textbf{comm} \\ \textbf{if} & : & \textbf{exp}[\textbf{bool}] \times \textbf{comm} \times \textbf{comm} \to \textbf{comm} \end{array}$$

Interlude: IA+

- IA+ extends Idealized Algol with classes.
- **cls** θ - the *type* of classes that have instances of type θ.
- Primitive class (constant):

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

- Class definition (and its equivalent ML fragment):

class : θ	$\lambda().\ \textbf{let}\ x : \theta = newC()$
local $C\ x$;	**in** $A; M$
init A;	
meth M	

- Class instantiation:

$$\textbf{new}\ C\ o.\ P(o)$$

- Additional constants:

$$\begin{array}{lcl} := & : & \textbf{var}[\delta] \times \textbf{exp}[\delta] \to \textbf{comm} \\ \textbf{deref} & : & \textbf{var}[\delta] \to \textbf{exp}[\delta] \end{array}$$

Interlude: IA+

- IA+ extends Idealized Algol with classes.
- **cls** θ - the *type* of classes that have instances of type θ.
- Primitive class (constant):

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

- Class definition (and its equivalent ML fragment):

$$\begin{array}{ll} \textbf{class} : \theta & \lambda().\ \textbf{let}\ x : \theta = newC() \\ \quad \textbf{local}\ C\ x; & \quad \textbf{in}\ A; M \\ \quad \textbf{init}\ A; & \\ \quad \textbf{meth}\ M & \end{array}$$

- Class instantiation:

$$\textbf{new}\ C\ o.\ P(o)$$

- Additional constants:

$$\begin{array}{lll} := & : & \textbf{var}[\delta] \times \textbf{exp}[\delta] \to \textbf{comm} \\ \textbf{deref} & : & \textbf{var}[\delta] \to \textbf{exp}[\delta] \end{array}$$

Interlude: IA+

- IA+ extends Idealized Algol with classes.
- **cls** θ - the *type* of classes that have instances of type θ.
- Primitive class (constant):

$$\mathrm{Var}[\delta] : \mathbf{cls}\ \{\mathrm{get} : \mathbf{exp}[\delta],\ \mathrm{put} : \mathbf{val}[\delta] \to \mathbf{comm}\}$$

- Class definition (and its equivalent ML fragment):

$$\begin{array}{ll} \mathbf{class} : \theta & \qquad \lambda().\,\mathbf{let}\ x : \theta = newC() \\ \quad \mathbf{local}\ C\ x; & \qquad\quad \mathbf{in}\ A; M \\ \quad \mathbf{init}\ A; & \\ \quad \mathbf{meth}\ M & \end{array}$$

- Class instantiation:

$$\mathbf{new}\ C\ o.\ P(o)$$

- Additional constants:

$$\begin{array}{lll} := & : & \mathbf{var}[\delta] \times \mathbf{exp}[\delta] \to \mathbf{comm} \\ \mathbf{deref} & : & \mathbf{var}[\delta] \to \mathbf{exp}[\delta] \end{array}$$

Interlude: IA+

- IA+ extends Idealized Algol with classes.
- **cls** θ - the *type* of classes that have instances of type θ.
- Primitive class (constant):

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

- Class definition (and its equivalent ML fragment):

class : θ
  local $C\ x$;
  init A;
  meth M
	$\lambda().\ \textbf{let}\ x : \theta = newC()$
	  in A; M

- Class instantiation:

$$\textbf{new}\ C\ o.\ P(o)$$

- Additional constants:

$$:= \quad : \quad \textbf{var}[\delta] \times \textbf{exp}[\delta] \to \textbf{comm}$$
$$\textbf{deref} \quad : \quad \textbf{var}[\delta] \to \textbf{exp}[\delta]$$

Counter class in IA+

$$\begin{array}{l} \textbf{class} : \{ \mathit{val} : \textbf{exp}[\textbf{int}],\ \mathit{inc} : \textbf{comm} \} \\ \quad \textbf{local}\ \mathrm{Var}[\textbf{int}]\ x; \\ \quad \textbf{init}\ x := 0; \\ \quad \textbf{meth}\ \{ \mathit{val} = \textbf{deref}\ x,\ \mathit{inc} = (x := (\textbf{deref}\ x) + 1) \} \end{array}$$

"Awkward" Example [Pitts & Stark, 1998]

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

$$C = \textbf{class} : \{m : \textbf{comm} \to \textbf{comm}\}$$
$$\textbf{local}\ \mathrm{Var[int]}\ x;$$
$$\textbf{init}\ x := 0;$$
$$\textbf{meth}\ \{m = \lambda c.\, x := 1; c; \mathit{test}(x = 1)\}$$

$$\mathit{test}(b) \triangleq \textbf{if}\ b\ \textbf{then skip else diverge}$$

- 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):

$$
\begin{aligned}
C = \textbf{class} : \{m : \textbf{comm} \to \textbf{comm}\} \\
&\textbf{local}\ \mathrm{Var[int]}\ x; \\
&\textbf{init}\ x := 0; \\
&\textbf{meth}\ \{m = \lambda c.\, x := 1; c; \mathit{test}(x = 1)\}
\end{aligned}
$$

$$\mathit{test}(b) \triangleq \textbf{if}\ b\ \textbf{then skip else diverge}$$

- 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]

$$
\begin{aligned}
C = \textbf{class} : \{m : \textbf{comm} \to \textbf{comm}\} \\
&\textbf{local}\ \mathrm{Var[int]}\ x; \\
&\textbf{init}\ x := 0; \\
&\textbf{meth}\ \{m = \lambda c.\, x := 1; c; \mathit{test}(x = 1)\}
\end{aligned}
$$

- It is not sound to say that c does not have "access" to x.
- Consider the following client:

$$
\begin{aligned}
&\textbf{new}\ C\ o. \quad \text{// create an instance of } C \text{ and call it } o \\
&\quad o.m\ (o.m\ \textbf{skip})
\end{aligned}
$$

- 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.

"Awkward" Example [Pitts & Stark, 1998]

$$
\begin{array}{l}
C = \textbf{class} : \{m : \textbf{comm} \to \textbf{comm}\} \\
\qquad \textbf{local}\ \mathrm{Var[int]}\ x; \\
\qquad \textbf{init}\ x := 0; \\
\qquad \textbf{meth}\ \{m = \lambda c.\, x := 1; c; \mathit{test}(x = 1)\}
\end{array}
$$

- It is not sound to say that c does not have "access" to x.
- Consider the following client:

$$
\begin{array}{l}
\textbf{new}\ C\ o. \quad \text{// create an instance of } C \text{ and call it } o \\
\qquad o.m\ (o.m\ \textbf{skip})
\end{array}
$$

- 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.

"Awkward" Example [Pitts & Stark, 1998]

$$
\begin{array}{l}
C = \textbf{class} : \{m : \textbf{comm} \to \textbf{comm}\} \\
\quad \textbf{local}\ \mathrm{Var[int]}\ x; \\
\quad \textbf{init}\ x := 0; \\
\quad \textbf{meth}\ \{m = \lambda c.\, x := 1; c; \mathit{test}(x = 1)\}
\end{array}
$$

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

$$
\begin{array}{lcl}
P_Q(x) & \iff & x = 0 \lor x = 1 \\
P_T(a) & \iff & a \sqsubseteq (\lambda n.\, n) \lor a \sqsubseteq (\lambda n.\, 1)
\end{array}
$$

- We must show that the body of *m* preserves the two-part invariant, while *assuming* that the argument *c* preserves the two-part invariant.

"Very awkward" Example: Dreyer, Neis, Birkedal, 2010

$$
\begin{aligned}
C = \textbf{class} : \{m : \textbf{comm} \to \textbf{comm}\} \\
\quad \textbf{local}\ \mathrm{Var[int]}\ x; \\
\quad \textbf{init}\ x := 0; \\
\quad \textbf{meth}\ \{m = \lambda c.\, x := 0; c; x := 1; c; \mathit{test}(x = 1)\}
\end{aligned}
$$

- 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.

$$
\begin{array}{lcl}
P_Q(x) & \iff & x = 0 \lor x = 1 \\
P_T(a) & \iff & a \sqsubseteq (\lambda n.\, n) \lor a \sqsubseteq (\lambda n.\, 1)
\end{array}
$$

- 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.

“Very awkward” Example: Dreyer, Neis, Birkedal, 2010

$$\begin{array}{l} C = \textbf{class} : \{m : \textbf{comm} \to \textbf{comm}\} \\ \qquad \textbf{local}\ \mathrm{Var[int]}\ x; \\ \qquad \textbf{init}\ x := 0; \\ \qquad \textbf{meth}\ \{m = \lambda c.\, x := 0; c; x := 1; c; \mathit{test}(x = 1)\} \end{array}$$

- 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

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 θ are interpreted as **functors** (with some technicalities):

$$[\![\theta]\!] : \mathbf{W}^{\mathrm{op}} \to \mathbf{CPO}$$

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

$$x_1 : \theta_1, \ldots, x_n : \theta_n \vdash M : \theta' \qquad [\![M]\!] : [\![\vec{\theta}]\!] \to [\![\theta']\!]$$

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 θ are interpreted as **functors** (with some technicalities):

$$[\![\theta]\!] : \mathbf{W}^{\mathrm{op}} \to \mathbf{CPO}$$

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

$$x_1 : \theta_1, \ldots, x_n : \theta_n \vdash M : \theta' \qquad [\![M]\!] : [\![\vec{\theta}]\!] \to [\![\theta']\!]$$

Possible world semantics - contd

- Denote $[\![\theta]\!]$ by F.
- For each world W, $F(W)$ is the set of meanings of type θ for store W.
- Morphisms, relations and squares are mapped as well:

$$\begin{array}{c} X \\ \Big\downarrow f \\ W \end{array} \quad \mapsto \quad \begin{array}{c} F(X) \\ \Big\uparrow F(f) \\ F(W) \end{array} \qquad\qquad \begin{array}{c} X \\ \Big\updownarrow R \\ X' \end{array} \quad \mapsto \quad \begin{array}{c} F(X) \\ \Big\updownarrow F(R) \\ F(X') \end{array}$$

$$\begin{array}{ccc} X & \xrightarrow{f} & W \\ S \Big\updownarrow & & \Big\updownarrow R \\ X' & \xrightarrow{f'} & W' \end{array} \quad \mapsto \quad \begin{array}{ccc} F(X) & \xleftarrow{F(f)} & F(W) \\ F(S) \Big\updownarrow & & \Big\updownarrow F(R) \\ F(X') & \xleftarrow{F(f')} & F(W') \end{array}$$

Possible world semantics - contd

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

$$
\begin{array}{ccccc}
X & & [\![\vec{\theta}]\!](X) & \xrightarrow{[\![M]\!]_X} & [\![\theta']\!](X) \\
\updownarrow R & [\![\vec{\theta}]\!](R) & \updownarrow & & \updownarrow [\![\theta']\!](R) \\
X' & & [\![\vec{\theta}]\!](X') & \xrightarrow{[\![M]\!]_{X'}} & [\![\theta']\!](X')
\end{array}
$$

- 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:

$$
\begin{aligned}
[\![\mathbf{comm}]\!](W) &= \ldots \\
[\![\mathbf{exp}[\delta]]\!](W) &= \ldots \\
[\![\mathbf{val}[\delta]]\!](W) &= [\![\delta]\!] \\
[\![\theta_1 \times \theta_2]\!](W) &= [\![\theta_1]\!](W) \times [\![\theta_2]\!](W) \\
[\![\theta \to \theta']\!](W) &= \forall_{h:X \to W}\, [\![\theta]\!](X) \to [\![\theta']\!](X) \\
[\![\mathbf{cls}\ \theta]\!](W) &= \exists_Z\, (\mathcal{Q}_Z)_\perp \times [\![\theta]\!](Z)
\end{aligned}
$$

- Note the correspondence with the counter classes seen earlier:

$$
\begin{aligned}
\langle Q = Int,\ T = Int^+,\ q_0 = 0, \{ val : Q \rightharpoonup Int = \lambda n.\, n, \\
inc : T = \lambda n.\, n + 1 \} \rangle \\
Int^+ = \text{down closure of } \{\, \lambda n.\, n + k \mid k \geq 0 \,\}
\end{aligned}
$$

- We use automata-theoretic ideas to define the possible worlds **W** and the interpretation of the base types **comm** and **exp**.

Possible world semantics - contd

- The meanings of types have this form:

$$
\begin{aligned}
[\![\mathbf{comm}]\!](W) &= \ldots \\
[\![\mathbf{exp}[\delta]]\!](W) &= \ldots \\
[\![\mathbf{val}[\delta]]\!](W) &= [\![\delta]\!] \\
[\![\theta_1 \times \theta_2]\!](W) &= [\![\theta_1]\!](W) \times [\![\theta_2]\!](W) \\
[\![\theta \to \theta']\!](W) &= \forall_{h:X \to W}\, [\![\theta]\!](X) \to [\![\theta']\!](X) \\
[\![\mathbf{cls}\ \theta]\!](W) &= \exists_Z\, (\mathcal{Q}_Z)_\perp \times [\![\theta]\!](Z)
\end{aligned}
$$

- Note the correspondence with the counter classes seen earlier:

$$
\begin{aligned}
\langle Q = Int,\ T = Int^+,\ q_0 = 0, \{ val : Q \rightharpoonup Int = \lambda n.\, n, \\
inc : T = \lambda n.\, n + 1 \} \rangle
\end{aligned}
$$

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

- We use automata-theoretic ideas to define the possible worlds **W** and the interpretation of the base types **comm** and **exp**.

Semantics of types

- The meanings of types have this form:

$$
\begin{array}{rcl}
[\![\textbf{comm}]\!](W) & = & \mathcal{T}_W \\
[\![\textbf{exp}[\delta]]\!](W) & = & [\mathcal{Q}_W \rightharpoonup [\![\delta]\!]] \\
[\![\textbf{val}[\delta]]\!](W) & = & [\![\delta]\!] \\
[\![\theta_1 \times \theta_2]\!](W) & = & [\![\theta_1]\!](W) \to [\![\theta_2]\!](W) \\
[\![\theta \to \theta']\!](W) & = & \forall_{h:X \to W} [\![\theta]\!](X) \to [\![\theta']\!](X) \\
[\![\textbf{cls}\ \theta]\!](W) & = & \exists_Z (\mathcal{Q}_Z)_\perp \times [\![\theta]\!](Z)
\end{array}
$$

- Note that the meanings of commands are the *allowed* transformations of the store (an automaton).
- The corresponding relational actions are as expected. $[\![\textbf{comm}]\!](R) = R_T$. $[\![\textbf{exp}[\delta]]\!](R) = [R_Q \rightharpoonup \Delta_{[\![\delta]\!]}]$.

Semantics of primitives

$$\text{cond}^E : \textsc{Exp}_{Bool} \times \textsc{Exp}_\delta \times \textsc{Exp}_\delta \to \textsc{Exp}_\delta$$
$$\text{cond}^E_W(e, e_1, e_2) = \lambda s.\, (\lambda v.\, v \to e_1(s);\ e_2(s))^*(e(s))$$
$$\text{cond}^C : \textsc{Exp}_{Bool} \times \textsc{Comm} \times \textsc{Comm} \to \textsc{Comm}$$
$$\text{cond}^C_W(e, a, b) = \text{read}_W\, \lambda s.\, (\lambda v.\, v \to a;\ b)^*(e(s))$$
$$\text{deref} : \textsc{Var}_\delta \to \textsc{Exp}_\delta$$
$$\text{deref}_W(e, a) = e$$
$$\text{assign} : \textsc{Var}_\delta \times \textsc{Exp}_\delta \to \textsc{Comm}$$
$$\text{assign}_W((d, a), e) = \text{read}_W\, \lambda s.\, a^*(e(s))$$
$$\text{Var}[\delta] : 1 \to \textsc{cls}\ \textsc{Var}_\delta$$
$$\text{Var}[\delta]_W(*) = \langle\!|\, V, init_\delta, \text{mkvar} \,|\!\rangle$$
$$\text{where } V = (\delta, T(\delta)) \quad \text{mkvar} = (\lambda n.\, n,\ \lambda k.\, \lambda n.\, k)$$
$$\text{newvar} : (\textsc{Var}_\delta \Rightarrow \textsc{Comm}) \to \textsc{Comm}$$
$$\text{newvar}_W(p) = (\lambda s.\, (s, init_\delta)) \cdot p[\pi_1](\text{mkvar}\!\uparrow_V^{W\star V}) \cdot (\lambda(s, n).\, s)$$

Results

- **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.,

$$[\![\mathbf{comm} \to \mathbf{comm}]\!](\mathbf{1}) \cong \mathit{Nat}$$

- Some representation results with divergence, especially for passive types:

$$[\![\mathbf{comm} \to \mathbf{exp}[\delta]]\!](W) \cong [\![\mathbf{exp}[\delta]]\!](W)$$

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