States and Actions: An Automata-theoretic Model of Objects

(In honour of John Reynolds on the occasion of his 75th birthday)

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

[1]University of Birmingham

[2]University of Illinois at Urbana-Champaign

Swansea, Sep 2011

Outline

1. Background
2. Motivation
3. Semantic Overview
4. Automata and Worlds
5. Semantics
6. Conclusion

What it is about

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

- Can we bridge the gap between the Scott-Strachey paradigm of semantics and the “Milner-Hoare paradigm”?
- 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?

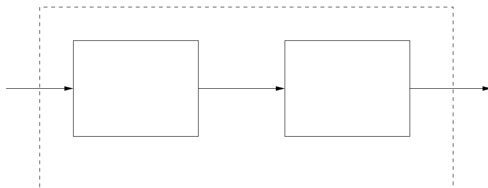
Reynolds [1981] anticipates many of these ideas.

Section 1

Background

Scott-Strachey paradigm of semantics

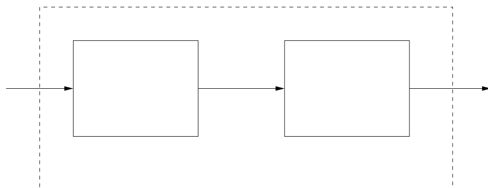
- Model computations as mathematical mappings from inputs to outputs, *a la* recursion function theory.
- Intermediate results are dissolved by *composition*.



- **However**: the inputs and outputs are *data representations*, which can be quite complex in real-life programs.

Scott-Strachey paradigm of semantics

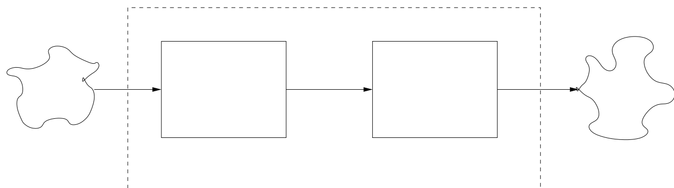
- Model computations as mathematical mappings from inputs to outputs, *a la* recursion function theory.
- Intermediate results are dissolved by *composition*.



- **However**: the inputs and outputs are *data representations*, which can be quite complex in real-life programs.

Scott-Strachey paradigm of semantics

- Model computations as mathematical mappings from inputs to outputs, *a la* recursion function theory.
- Intermediate results are dissolved by *composition*.



- **However**: the inputs and outputs are *data representations*, which can be quite complex in real-life programs.

Scott-Strachey paradigm of semantics

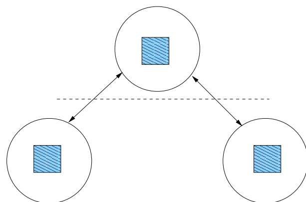
- Model computations as mathematical mappings from inputs to outputs, *a la* recursive function theory.
- Intermediate results are dissolved by *composition*.
- **However**: the inputs and outputs are *data representations*, which can be quite complex in real-life programs.
- **Solution**: Relational parametricity (simulation relations).

$$\begin{array}{ccccc} A & & F(A) & \xrightarrow{f_A} & G(A) \\ R \updownarrow & & F(R) \updownarrow & & \updownarrow G(R) \\ A' & & F(A') & \xrightarrow{f_{A'}} & G(A') \end{array}$$

Intuitively: all relational correspondences between data representations are preserved by computations.

Milner-Hoare paradigm of semantics

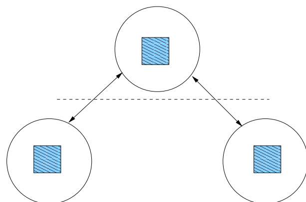
- Model computations as *agents*, which interact with other agents by exchanging simple tokens of information.



- External behavior represented as *traces*.
- Data representations are hidden inside agents.

Milner-Hoare paradigm of semantics

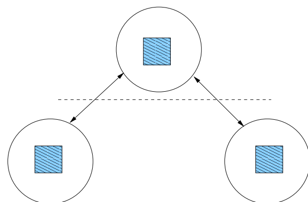
- Model computations as *agents*, which interact with other agents by exchanging simple tokens of information.



- External behavior represented as *traces*.
- Data representations are hidden inside agents.

Milner-Hoare paradigm of semantics

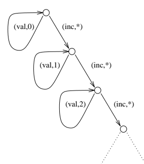
- Model computations as *agents*, which interact with other agents by exchanging simple tokens of information.



- External behavior represented as *traces*.
- Data representations are hidden inside agents.

Milner-Hoare paradigm of semantics

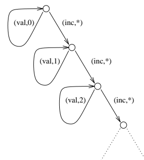
- Model computations as *agents*, which interact with other agents by exchanging simple tokens of information.
- External behavior represented as *traces*.
- Data representations are hidden inside agents.
- **However**: Traces represent *all* intermediate actions, not only the net effect ("intensional").



Milner-Hoare paradigm is quite pervasive:
Event structures - Nielsen, Plotkin, Winskel; *Concrete data structures* - Berry, Curien; *Coherent spaces* - Girard; *Geometry of interaction* - Girard, Abramsky; *Games semantics* - Abramsky, McCusker, Hyland, Ong

Milner-Hoare paradigm of semantics

- Model computations as *agents*, which interact with other agents by exchanging simple tokens of information.
- External behavior represented as *traces*.
- Data representations are hidden inside agents.
- **However**: Traces represent *all* intermediate actions, not only the net effect ("intensional").



Milner-Hoare paradigm is quite pervasive:
Event structures - Nielsen, Plotkin, Winskel; *Concrete data structures* - Berry, Curien; *Coherent spaces* - Girard; *Geometry of interaction* - Girard, Abramsky; *Games semantics* - Abramsky, McCusker, Hyland, Ong

Scott-Strachey vs. Milner-Hoare

- Example in favour of Scott-Strachey:

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

- Example in favour of Milner-Hoare:

$$\begin{array}{lcl} [\![\textbf{let}\ x = \mathrm{ref}\ 0 & \stackrel{?}{=} & [\![\textbf{let}\ x = \mathrm{ref}\ 0 \\ \ \textbf{in}\ (\lambda().\, !x, & & \ \textbf{in}\ (\lambda().\, -!x, \\ \qquad \lambda().\, x := !x + 1)]\!] & & \qquad \lambda().\, x := !x - 1)]\!] \end{array}$$

(We are using ML notation in this slide.)

Scott-Strachey vs. Milner-Hoare

- Example in favour of Scott-Strachey:

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

- Example in favour of Milner-Hoare:

$$\begin{array}{l}[\![\mathbf{let}\ x = \mathrm{ref}\ 0 \\ \ \mathbf{in}\ (\lambda().\, !x, \\ \qquad \lambda().\, x := !x + 1)]\!]\end{array} \quad \stackrel{?}{=} \quad \begin{array}{l}[\![\mathbf{let}\ x = \mathrm{ref}\ 0 \\ \ \mathbf{in}\ (\lambda().\, -!x, \\ \qquad \lambda().\, x := !x - 1)]\!]\end{array}$$

(We are using ML notation in this slide.)

Scott-Strachey vs. Milner-Hoare

- The Scott-Strachey approach is good for computation, bad for data.
- The Milner-Hoare 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?

Scott-Strachey vs. Milner-Hoare

- The Scott-Strachey approach is good for computation, bad for data.
- The Milner-Hoare 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?

Scott-Strachey vs. Milner-Hoare

- The Scott-Strachey approach is good for computation, bad for data.
- The Milner-Hoare 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?

Semantics of Algol-like languages

- Reynolds, 1981: *The Essence of Algol*:
  1. Idealized Algol as a *simply typed lambda calculus*.
  2. Semantics of state is a *possible world semantics*.
  3. Worlds as a form of *automata*.
- Oles, 1983: PhD thesis on *Category-theoretic approach to the semantics*.
  - Simpler state-based category of worlds.
- O'Hearn & Tennent, 1993: *Relational parametricity and local variables*.
  - Added relational parametricity to Oles's model.
  - O'Hearn & Reynolds, 2000 refined it with linear functions, and proved full abstraction for second-order active types.
- Reddy, 1993: *Global state considered unnecessary*.
  - Used a Milner-Hoare paradigm for modeling syntactic control of interference.
  - O'Hearn & Reddy, 1995 extended it to full Idealized Algol, and proved full abstraction for second-order active types.

Semantics of Algol-like languages

- Reynolds, 1981: *The Essence of Algol*:
  1. Idealized Algol as a *simply typed lambda calculus*.
  2. Semantics of state is a *possible world semantics*.
  3. Worlds as a form of *automata*.
- Oles, 1983: PhD thesis on *Category-theoretic approach to the semantics*.
  - Simpler state-based category of worlds.
- O'Hearn & Tennent, 1993: *Relational parametricity and local variables*.
  - Added relational parametricity to Oles's model.
  - O'Hearn & Reynolds, 2000 refined it with linear functions, and proved full abstraction for second-order active types.
- Reddy, 1993: *Global state considered unnecessary*.
  - Used a Milner-Hoare paradigm for modeling syntactic control of interference.
  - O'Hearn & Reddy, 1995 extended it to full Idealized Algol, and proved full abstraction for second-order active types.

Semantics of Algol-like languages

- Reynolds, 1981: *The Essence of Algol*:
  1. Idealized Algol as a *simply typed lambda calculus*.
  2. Semantics of state is a *possible world semantics*.
  3. Worlds as a form of *automata*.
- Oles, 1983: PhD thesis on *Category-theoretic approach to the semantics*.
  - Simpler state-based category of worlds.
- O'Hearn & Tennent, 1993: *Relational parametricity and local variables*.
  - Added relational parametricity to Oles's model.
  - O'Hearn & Reynolds, 2000 refined it with linear functions, and proved full abstraction for second-order active types.
- Reddy, 1993: *Global state considered unnecessary*.
  - Used a Milner-Hoare paradigm for modeling syntactic control of interference.
  - O'Hearn & Reddy, 1995 extended it to full Idealized Algol, and proved full abstraction for second-order active types.

Semantics of Algol-like languages

- Reynolds, 1981: *The Essence of Algol*:
  1. Idealized Algol as a *simply typed lambda calculus*.
  2. Semantics of state is a *possible world semantics*.
  3. Worlds as a form of *automata*.
- Oles, 1983: PhD thesis on *Category-theoretic approach to the semantics*.
  - Simpler state-based category of worlds.
- O'Hearn & Tennent, 1993: *Relational parametricity and local variables*.
  - Added relational parametricity to Oles's model.
  - O'Hearn & Reynolds, 2000 refined it with linear functions, and proved full abstraction for second-order active types.
- Reddy, 1993: *Global state considered unnecessary*.
  - Used a Milner-Hoare paradigm for modeling syntactic control of interference.
  - O'Hearn & Reddy, 1995 extended it to full Idealized Algol, and proved full abstraction for second-order active types.

Semantics of Algol-like languages

- Reynolds, 1981: *The Essence of Algol*:
  1. Idealized Algol as a *simply typed lambda calculus*.
  2. Semantics of state is a *possible world semantics*.
  3. Worlds as a form of *automata*.
- Oles, 1983: PhD thesis on *Category-theoretic approach to the semantics*.
  - Simpler state-based category of worlds.
- Reynolds's automata-like worlds were never used again. Essentially forgotten (!).
- In my effort to bridge the gap between the state-based and event-based approach, I considered automata-theoretic ideas in the 90's and ended up reinventing Reynolds's model in 1998.
- It remained in the background all these years, with interest rekindled by the work of Ahmed, Dreyer, Birkedal and colleagues, and the work on deny-guarantee reasoning.
- There was also the work of Pitts and Stark, 1998 (open problem).

Semantics of Algol-like languages

- Reynolds, 1981: *The Essence of Algol*:
  1. Idealized Algol as a *simply typed lambda calculus*.
  2. Semantics of state is a *possible world semantics*.
  3. Worlds as a form of *automata*.
- Oles, 1983: PhD thesis on *Category-theoretic approach to the semantics*.
  - Simpler state-based category of worlds.
- Reynolds's automata-like worlds were never used again. Essentially forgotten (!).
- In my effort to bridge the gap between the state-based and event-based approach, I considered automata-theoretic ideas in the 90's and ended up reinventing Reynolds's model in 1998.
- It remained in the background all these years, with interest rekindled by the work of Ahmed, Dreyer, Birkedal and colleagues, and the work on deny-guarantee reasoning.
- There was also the work of Pitts and Stark, 1998 (open problem).

Semantics of Algol-like languages

- Reynolds, 1981: *The Essence of Algol*:
  1. Idealized Algol as a *simply typed lambda calculus*.
  2. Semantics of state is a *possible world semantics*.
  3. Worlds as a form of *automata*.
- Oles, 1983: PhD thesis on *Category-theoretic approach to the semantics*.
  - Simpler state-based category of worlds.
- Reynolds's automata-like worlds were never used again. Essentially forgotten (!).
- In my effort to bridge the gap between the state-based and event-based approach, I considered automata-theoretic ideas in the 90's and ended up reinventing Reynolds's model in 1998.
- It remained in the background all these years, with interest rekindled by the work of Ahmed, Dreyer, Birkedal and colleagues, and the work on deny-guarantee reasoning.
- There was also the work of Pitts and Stark, 1998 (open problem).

Semantics of Algol-like languages

- Reynolds, 1981: *The Essence of Algol*:
  1. Idealized Algol as a *simply typed lambda calculus*.
  2. Semantics of state is a *possible world semantics*.
  3. Worlds as a form of *automata*.
- Oles, 1983: PhD thesis on *Category-theoretic approach to the semantics*.
  - Simpler state-based category of worlds.
- Reynolds's automata-like worlds were never used again. Essentially forgotten (!).
- In my effort to bridge the gap between the state-based and event-based approach, I considered automata-theoretic ideas in the 90's and ended up reinventing Reynolds's model in 1998.
- It remained in the background all these years, with interest rekindled by the work of Ahmed, Dreyer, Birkedal and colleagues, and the work on deny-guarantee reasoning.
- There was also the work of Pitts and Stark, 1998 (open problem).

Section 2

Motivation

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 (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)

- The state-based model is OK for proving equivalence of data representations.
- But it does not capture *irreversibility* of state transformation.
- The “snap-back” operator exists:

$$snap_Q(a) = \lambda q. \begin{cases} \bot, & \text{if } a(q) = \bot \\ q, & \text{otherwise} \end{cases}$$

and it is parametric:

$$snap_Q \; \big[[R \rightharpoonup R] \to [R \rightharpoonup R]\big] \; snap_{Q'}$$

- If we pass a counter as an argument to a procedure, we cannot prove that the procedure can at best increment the counter. (In the *model*, the procedure can change the state to some old value!)

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:

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

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

- 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.

Invariants

- Relational parametricity works for relations of any arity.
- Unary relational parametricity gives a notion of "invariants".
- Our theory suggests that "invariants" come in two parts: *state-invariants* and *action-invariants*. For example,

$$
\begin{array}{lcl}
P_Q(n) & \iff & n \geq 0 \\
P_T(a) & \iff & \exists k.\, a \sqsubseteq \lambda n.\, n + k
\end{array}
$$

- If an object is passed to an unknown procedure, we can be sure that
  - the final state of the object after return will satisfy the state invariant.
  - the only transformations that could be done by the procedure are those satisfying the action invariant.

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):

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

- Class definition (and its equivalent ML fragment):

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

- Class instantiation:

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

- Additional constants:

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

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:

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

- 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):

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

- Class instantiation:

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

- 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:

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

- Additional constants:

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

Counter class in IA+

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

"Awkward" Example [Pitts & Stark, 1998]

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

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

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

- 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):

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

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

- 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:

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

- 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]

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

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

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

- 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]

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

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

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

- 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

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

- 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.

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

- 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 3

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

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

$$\begin{array}{ccc} X & & [\![\theta]\!](X) \\ \downarrow f & \mapsto & \uparrow [\![\theta]\!](f) \\ W & & [\![\theta]\!](W) \end{array} \qquad \begin{array}{ccc} X & & [\![\theta]\!](X) \\ \updownarrow R & \mapsto & \updownarrow [\![\theta]\!](R) \\ X' & & [\![\theta]\!](X') \end{array}$$

$$\begin{array}{ccc} X & \xrightarrow{f} & W \\ S \updownarrow & & \updownarrow R \\ X' & \xrightarrow{f'} & W' \end{array} \quad \mapsto \quad \begin{array}{ccc} [\![\theta]\!](X) & \xleftarrow{[\![\theta]\!](f)} & [\![\theta]\!](W) \\ [\![\theta]\!](S) \updownarrow & & \updownarrow [\![\theta]\!](R) \\ [\![\theta]\!](X') & \xleftarrow{[\![\theta]\!](f')} & [\![\theta]\!](W') \end{array}$$

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:

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

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

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

Possible world semantics - contd

- The meanings of types have this form:

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

- Note the correspondence with the counter classes seen earlier:

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

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

Section 4

Automata and Worlds

Transformation monoids

These ideas are standard in *algebraic automata theory*.

- A *semiautomaton* is a triple (Q, Σ, α) where
  - Q is a set of states,
  - Σ is a set of events, and
  - $\alpha : \Sigma \to [Q \rightharpoonup Q]$ is a representation of the events as state transformations.
- The representation is immediately extended to sequences of events: $\Sigma^* \to [Q \rightharpoonup Q]$ as a monoid homomorphism. So, in reality, this is a monoid action (Q, Σ^*, α).
- A *transformation monoid* is a pair (Q, T), where
  - Q is a set of states,
  - $T \subseteq [Q \rightharpoonup Q]$ is a submonoid, and
  - $\alpha : T \hookrightarrow [Q \rightharpoonup Q]$ is an implicit representation.
- So, transformation monoids are abstract forms of automata, where the actions are represented not by symbols, but by state transformations.

Transformation monoids

These ideas are standard in *algebraic automata theory*.

- A *semiautomaton* is a triple (Q, Σ, α) where
  - Q is a set of states,
  - Σ is a set of events, and
  - $\alpha : \Sigma \to [Q \rightharpoonup Q]$ is a representation of the events as state transformations.
- The representation is immediately extended to sequences of events: $\Sigma^* \to [Q \rightharpoonup Q]$ as a monoid homomorphism. So, in reality, this is a monoid action (Q, Σ^*, α).
- A *transformation monoid* is a pair (Q, T), where
  - Q is a set of states,
  - $T \subseteq [Q \rightharpoonup Q]$ is a submonoid, and
  - $\alpha : T \hookrightarrow [Q \rightharpoonup Q]$ is an implicit representation.
- So, transformation monoids are abstract forms of automata, where the actions are represented not by symbols, but by state transformations.

Automata and semantic paradigms

- A *semiautomaton* is a triple (Q, Σ^*, α) where
  - Q is a set of states,
  - Σ^* is a free monoid of event traces, and
  - $\alpha : \Sigma^* \to [Q \rightharpoonup Q]$ is a representation of event traces as state transformation functions.
- Scott-Strachey semantics works with Q.
- Milner-Hoare semantics works with Σ^*.
- We work with both, to combine their advantages.
- The decision to replace Σ^* by a submonoid of $[Q \rightharpoonup Q]$ is a technical decision. In the long run, we expect to be able to work with monoids in general, including Σ^*.
- [Reynolds, 1981] used $[Q \rightharpoonup Q]$ directly as the transformation component, instead of a submonoid. There is no loss of generality in doing so, because action-invariants can still cut down the full monoid to the relevant part.

Automata and semantic paradigms

- A *semiautomaton* is a triple (Q, Σ^*, α) where
  - Q is a set of states,
  - Σ^* is a free monoid of event traces, and
  - $\alpha : \Sigma^* \to [Q \rightharpoonup Q]$ is a representation of event traces as state transformation functions.
- Scott-Strachey semantics works with Q.
- Milner-Hoare semantics works with Σ^*.
- We work with both, to combine their advantages.
- The decision to replace Σ^* by a submonoid of $[Q \rightharpoonup Q]$ is a technical decision. In the long run, we expect to be able to work with monoids in general, including Σ^*.
- [Reynolds, 1981] used $[Q \rightharpoonup Q]$ directly as the transformation component, instead of a submonoid. There is no loss of generality in doing so, because action-invariants can still cut down the full monoid to the relevant part.

Automata and semantic paradigms

- A *semiautomaton* is a triple (Q, Σ^*, α) where
  - Q is a set of states,
  - Σ^* is a free monoid of event traces, and
  - $\alpha : \Sigma^* \to [Q \rightharpoonup Q]$ is a representation of event traces as state transformation functions.
- Scott-Strachey semantics works with Q.
- Milner-Hoare semantics works with Σ^*.
- We work with both, to combine their advantages.
- The decision to replace Σ^* by a submonoid of $[Q \rightharpoonup Q]$ is a technical decision. In the long run, we expect to be able to work with monoids in general, including Σ^*.
- [Reynolds, 1981] used $[Q \rightharpoonup Q]$ directly as the transformation component, instead of a submonoid. There is no loss of generality in doing so, because action-invariants can still cut down the full monoid to the relevant part.

Automata and semantic paradigms

- A *semiautomaton* is a triple (Q, Σ^*, α) where
  - Q is a set of states,
  - Σ^* is a free monoid of event traces, and
  - $\alpha : \Sigma^* \to [Q \rightharpoonup Q]$ is a representation of event traces as state transformation functions.
- Scott-Strachey semantics works with Q.
- Milner-Hoare semantics works with Σ^*.
- We work with both, to combine their advantages.
- The decision to replace Σ^* by a submonoid of $[Q \rightharpoonup Q]$ is a technical decision. In the long run, we expect to be able to work with monoids in general, including Σ^*.
- [Reynolds, 1981] used $[Q \rightharpoonup Q]$ directly as the transformation component, instead of a submonoid. There is no loss of generality in doing so, because action-invariants can still cut down the full monoid to the relevant part.

Monoids - technical

- A *monoid* is a set M along with an associative binary operation "$\cdot$" which has a unit element.
- A *complete ordered monoid* is a monoid that is also a pointed CPO and "$\cdot$" is strict and continuous.
- $[Q \rightharpoonup Q]$ is a complete ordered monoid with sequential composition as multiplication, the injection $\mathrm{null}_Q : Q \rightharpoonup Q$ as the unit. It is evidently a pointed CPO.
- A *complete ordered submonoid* of M is a subset that is
  - a submonoid of M (closed under multiplication and including the unit), and
  - a subcpo of M (closed under sup's of directed sets and including the least element).
- A *morphism* of complete ordered monoids $f : M \to M'$ is a strict, continuous function that preserves the units and the multiplication.
- We abbreviate "complete ordered monoid" to "monoid" for brevity.

Relations for transformation monoids

- Our use of transformation monoids is to represent stores of locations. But it is possible for the "locations" to be abstract, i.e., they could stand for general objects instead of simple variables.
- A relation $R : (\mathcal{Q}_X, \mathcal{T}_X) \leftrightarrow (\mathcal{Q}_{X'}, \mathcal{T}_{X'})$ is a pair (R_Q, R_T) where

$$\begin{array}{c} X \\ \updownarrow R \\ X' \end{array} \quad = \quad \left(\begin{array}{c} \mathcal{Q}_X \\ \updownarrow R_Q \\ \mathcal{Q}_{X'} \end{array}, \quad \begin{array}{c} \mathcal{T}_X \\ \updownarrow R_T \\ \mathcal{T}_{X'} \end{array} \right)$$

  - $R_Q : Q_X \leftrightarrow Q_{X'}$ is a relation, and
  - $R_T : T_X \leftrightarrow T_{X'}$ is a (complete ordered) monoid relation,

  such that $\alpha \left[R_T \hookrightarrow [R_Q \rightharpoonup R_Q]\right] \alpha'$.
  (It basically means $R_T \subseteq [R_Q \rightharpoonup R_Q]$.)
- Note that $[R_Q \rightharpoonup R_Q] \subseteq R_T$ is not necessary. So, R_T can be more *constrained* than $[R_Q \rightharpoonup R_Q]$. This is where the power of transformation monoids comes from.

Relations for transformation monoids

- Our use of transformation monoids is to represent stores of locations. But it is possible for the “locations” to be abstract, i.e., they could stand for general objects instead of simple variables.
- A relation $R : (\mathcal{Q}_X, \mathcal{T}_X) \leftrightarrow (\mathcal{Q}_{X'}, \mathcal{T}_{X'})$ is a pair (R_Q, R_T) where

$$\begin{array}{c} X \\ \updownarrow R \\ X' \end{array} \quad = \quad \left(\begin{array}{c} \mathcal{Q}_X \\ \updownarrow R_Q \\ \mathcal{Q}_{X'} \end{array}, \quad \begin{array}{c} \mathcal{T}_X \\ \updownarrow R_T \\ \mathcal{T}_{X'} \end{array} \right)$$

  - $R_Q : Q_X \leftrightarrow Q_{X'}$ is a relation, and
  - $R_T : T_X \leftrightarrow T_{X'}$ is a (complete ordered) monoid relation,

  such that $\alpha \left[R_T \hookrightarrow [R_Q \rightharpoonup R_Q] \right] \alpha'$.
  (It basically means $R_T \subseteq [R_Q \rightharpoonup R_Q]$.)
- Note that $[R_Q \rightharpoonup R_Q] \subseteq R_T$ is not necessary. So, R_T can be more *constrained* than $[R_Q \rightharpoonup R_Q]$. This is where the power of transformation monoids comes from.

Example relation

- Recall the relation from the "awkward" example:

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

- Clearly, $P_T \subseteq [P_Q \rightharpoonup P_Q]$, but $[P_Q \rightharpoonup P_Q] \not\subseteq P_T$.
- To check for read closure, we consider all possible functions p satisfying $P_Q \to P_T$ and check that $\mathrm{read}\ p$ satisfies P_T.
  - Example: $p = \{0 \mapsto \lambda n.\, 1,\ 1 \mapsto \lambda n.\, n\}$. Then $\mathrm{read}\ p = \lambda n.\, 1$ which satisfies P_T.
- This is a bit cumbersome.
- It might also be an overkill. Reynolds transformation monoids assume that transformations may read *all* the information in the state. This is not always true.
  In the "awkward" example, they can't read any information of the state because there are no methods to do so.

Example relation

- Recall the relation from the "awkward" example:

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

- Clearly, $P_T \subseteq [P_Q \rightharpoonup P_Q]$, but $[P_Q \rightharpoonup P_Q] \not\subseteq P_T$.
- To check for read closure, we consider all possible functions p satisfying $P_Q \to P_T$ and check that $\mathrm{read}\ p$ satisfies P_T.
  - Example: $p = \{0 \mapsto \lambda n.\, 1,\ 1 \mapsto \lambda n.\, n\}$. Then $\mathrm{read}\ p = \lambda n.\, 1$ which satisfies P_T.
- This is a bit cumbersome.
- It might also be an overkill. Reynolds transformation monoids assume that transformations may read *all* the information in the state. This is not always true.
  In the "awkward" example, they can't read any information of the state because there are no methods to do so.

Example relation

- Recall the relation from the "awkward" example:

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

- Clearly, $P_T \subseteq [P_Q \rightharpoonup P_Q]$, but $[P_Q \rightharpoonup P_Q] \not\subseteq P_T$.
- To check for read closure, we consider all possible functions p satisfying $P_Q \to P_T$ and check that $\mathrm{read}\ p$ satisfies P_T.
  - Example: $p = \{0 \mapsto \lambda n.\, 1,\ 1 \mapsto \lambda n.\, n\}$. Then $\mathrm{read}\ p = \lambda n.\, 1$ which satisfies P_T.
- This is a bit cumbersome.
- It might also be an overkill. Reynolds transformation monoids assume that transformations may read *all* the information in the state. This is not always true.
  In the "awkward" example, they can't read any information of the state because there are no methods to do so.

Reynolds transformation monoids

- The structure of transformation monoids is not enough for semantics. An automaton can't observe its own state. But a program can.
- Reynolds diagonal operation

$$\mathrm{read}_X : (\mathcal{Q}_X \to \mathcal{T}_X) \to \mathcal{T}_X \qquad \mathrm{read}_X(p)(x) = p(x)(x)$$

- **Intuition**: p is a state-dependent action. $\mathrm{read}_X(p)$ is an action that reads the current state and uses it for the state-dependence of p. E.g.,

$$cond(b, a_1, a_2) = \mathrm{read}\ \lambda x.\, b(x) \to a_1 ; a_2$$

where $b : \mathcal{Q}_X \rightharpoonup Bool$ and $a_1, a_2 : \mathcal{T}_X$.
- A *Reynolds transformation monoid (rtm)* is a transformation monoid closed under the read operation.
- A *relation of rtm's* is a relation of tm's $R : X \leftrightarrow X'$ that is compatible with the read operations

$$\mathrm{read}_X \left[(R_Q \to R_T) \to R_T\right] \mathrm{read}_{X'}$$

Reynolds transformation monoids

- The structure of transformation monoids is not enough for semantics. An automaton can't observe its own state. But a program can.
- Reynolds diagonal operation

$$\mathrm{read}_X : (\mathcal{Q}_X \to \mathcal{T}_X) \to \mathcal{T}_X \qquad \mathrm{read}_X(p)(x) = p(x)(x)$$

- **Intuition**: p is a state-dependent action. $\mathrm{read}_X(p)$ is an action that reads the current state and uses it for the state-dependence of p. E.g.,

$$cond(b, a_1, a_2) = \mathrm{read}\ \lambda x.\, b(x) \to a_1 \,;\, a_2$$

where $b : \mathcal{Q}_X \rightharpoonup Bool$ and $a_1, a_2 : \mathcal{T}_X$.
- A *Reynolds transformation monoid (rtm)* is a transformation monoid closed under the read operation.
- A *relation of rtm's* is a relation of tm's $R : X \leftrightarrow X'$ that is compatible with the read operations

$$\mathrm{read}_X \ \left[(R_Q \to R_T) \to R_T\right] \ \mathrm{read}_{X'}$$

Morphisms for worlds

- We want a "category of possible worlds" based on Reynolds transformation monoids (together with relations).
- A morphism $f : X \to W$ means that X is a possible "future world" of W, and f represents the manner in which it is a future world.
- We use exactly the morphisms defined by Reynolds [1981].
- A morphism is a pair $f = (\phi_f, \tau_f)$

$$\begin{array}{lc} \text{(future world)} & X \\ & \Big\downarrow f \\ \text{(current world)} & W \end{array} = \left(\begin{array}{c} \mathcal{Q}_X \\ \Big\downarrow \phi_f \\ \mathcal{Q}_W \end{array} , \begin{array}{c} \mathcal{T}_X \\ \Big\uparrow \tau_f \\ \mathcal{T}_W \end{array} \right)$$

  such that $(\langle \phi_f \rangle, \langle \tau_f \rangle^{\smile})$ is a relation of rtm's.
  $\langle g \rangle$ is the function graph of g. $R^{\smile}$ is the converse of relation R.
- The state part ϕ_f projects a state of the current world from that of a future world (which is expected to be larger).
- The transformation part τ_f extends a state transformation of the current world to work at a future world.

Relation-preservation

- A relation-preservation square of rtm's

$$\begin{array}{ccc} (\mathcal{Q}_X, \mathcal{T}_X) & \xrightarrow{f = (\phi_f, \tau_f)} & (\mathcal{Q}_W, \mathcal{T}_W) \\ (S_Q, S_T) \updownarrow & & \updownarrow (R_Q, R_T) \\ (\mathcal{Q}_{X'}, \mathcal{T}_{X'}) & \xrightarrow{f' = (\phi_{f'}, \tau_{f'})} & (\mathcal{Q}_{W'}, \mathcal{T}_{W'}) \end{array}$$

  exists iff $\phi_f \; [S_Q \to R_Q] \; \phi_{f'}$ and $\tau_f \; [R_T \to S_T] \; \tau_{f'}$.
- **Theorem:** This data (rtm's, rtm-morphisms, rtm-relations and relation-preservation squares) forms an *op-parametricity graph* **RTM**, i.e., a reflexive graph of categories that is relational and op-fibred, satisfying the identity condition. [Dunphy & Reddy, 2004]

Structure of RTM

- **Tensor Product**: If $X = (\mathcal{Q}_X, \mathcal{T}_X)$ and $Y = (\mathcal{Q}_Y, \mathcal{T}_Y)$ are rtm's, we define $X \star Y = (\mathcal{Q}_X \times \mathcal{Q}_Y,\ \mathcal{T}_{X\star Y})$. Defining $\mathcal{T}_{X\star Y}$ requires some thinking.
- If $a \in \mathcal{T}_X$, $b \in \mathcal{T}_Y$, let $a \otimes b \in [Q_X \times Q_Y \rightharpoonup Q_X \times Q_Y]$ be:

$$(a \otimes b)(x, y) = [a(x), b(y)]_\perp$$

- Then

$$\mathcal{T}_X \otimes \mathcal{T}_Y = \{\, a \otimes b \mid a \in \mathcal{T}_X,\ b \in \mathcal{T}_Y \,\}$$

$$\mathcal{T}_{X\star Y} = \text{read-closure of } \mathcal{T}_X \otimes \mathcal{T}_Y$$

- An element of $\mathcal{T}_{X*Y}$ may not be of the form $a \otimes b$. But it can be written in the form:

$$\mathrm{read}_{X\star Y}\ \lambda(x, y).\ a_{x,y} \otimes b_{x,y}$$

Structure of RTM

- **Tensor Product**: If $X = (\mathcal{Q}_X, \mathcal{T}_X)$ and $Y = (\mathcal{Q}_Y, \mathcal{T}_Y)$ are rtm's, we define $X \star Y = (\mathcal{Q}_X \times \mathcal{Q}_Y,\ \mathcal{T}_{X\star Y})$. Defining $\mathcal{T}_{X\star Y}$ requires some thinking.
- If $a \in \mathcal{T}_X$, $b \in \mathcal{T}_Y$, let $a \otimes b \in [Q_X \times Q_Y \rightharpoonup Q_X \times Q_Y]$ be:

$$(a \otimes b)(x, y) = [a(x), b(y)]_\perp$$

- Then

$$\mathcal{T}_X \otimes \mathcal{T}_Y = \{\, a \otimes b \mid a \in \mathcal{T}_X,\ b \in \mathcal{T}_Y \,\}$$

$$T_{X\star Y} = \text{read-closure of } \mathcal{T}_X \otimes \mathcal{T}_Y$$

- An element of $\mathcal{T}_{X*Y}$ may not be of the form $a \otimes b$. But it can be written in the form:

$$\mathrm{read}_{X\star Y}\ \lambda(x, y).\ a_{x,y} \otimes b_{x,y}$$

Structure of RTM - continued

- The relational action of $\star$ is a bit involved:

$$
\begin{array}{l}
t\ \left[(R \star S)_T\right]\ t' \iff \\
\quad \forall x, x', y, y'.\ (x, y)\ \left[R \star S\right]\ (x', y') \implies \\
\quad\quad (\exists a \in \mathcal{Q}_X,\ a' \in \mathcal{Q}_{X'},\ b \in \mathcal{Q}_Y,\ b' \in \mathcal{Q}_{Y'}. \\
\quad\quad\quad a\ \left[R\right]\ a' \wedge b\ \left[S\right]\ b' \wedge \\
\quad\quad\quad t(x, y) = (a \otimes b)(x, y) \wedge \\
\quad\quad\quad t'(x', y') = (a' \otimes b')(x', y'))
\end{array}
$$

- The terminal object in **RTM** is $\mathbf{1} = (\mathbf{1}, \mathbf{0_1})$ where $\mathbf{0_1} = \{\overline{\bot}, \mathrm{null}_{\mathbf{1}}\}$. It represents the “empty store” (which has a single state).
- $\mathbf{1}$ is the unit for $\star$. There are projections:

$$
\begin{array}{ll}
\pi_1 : X \star Y \to X & \pi_1 = (\pi_1, \iota_1) \\
\pi_2 : X \star Y \to Y & \pi_1 = (\pi_2, \iota_2)
\end{array}
$$

where $\iota_1(a) = a \otimes \mathrm{null}_Y$ and $\iota_2(b) = \mathrm{null}_X \otimes b$.

- This gives an affine SMC structure to **RTM**.

Structure of RTM - continued

- The relational action of $\star$ is a bit involved:

$$
\begin{array}{l}
t\ \left[(R \star S)_T\right]\ t' \iff \\
\quad \forall x, x', y, y'.\ (x, y)\ \left[R \star S\right]\ (x', y') \implies \\
\quad\quad (\exists a \in \mathcal{Q}_X,\ a' \in \mathcal{Q}_{X'},\ b \in \mathcal{Q}_Y,\ b' \in \mathcal{Q}_{Y'}. \\
\quad\quad\quad a\ \left[R\right]\ a' \wedge b\ \left[S\right]\ b' \wedge \\
\quad\quad\quad t(x, y) = (a \otimes b)(x, y) \wedge \\
\quad\quad\quad t'(x', y') = (a' \otimes b')(x', y'))
\end{array}
$$

- The terminal object in **RTM** is $\mathbf{1} = (\mathbf{1}, \mathbf{0_1})$ where $\mathbf{0_1} = \{\overline{\bot}, \mathrm{null}_\mathbf{1}\}$. It represents the "empty store" (which has a single state).
- $\mathbf{1}$ is the unit for $\star$. There are projections:

$$
\begin{array}{ll}
\pi_1 : X \star Y \to X & \pi_1 = (\pi_1, \iota_1) \\
\pi_2 : X \star Y \to Y & \pi_1 = (\pi_2, \iota_2)
\end{array}
$$

  where $\iota_1(a) = a \otimes \mathrm{null}_Y$ and $\iota_2(b) = \mathrm{null}_X \otimes b$.
- This gives an affine SMC structure to **RTM**.

Section 5

Semantics

Semantics of types

- The meanings of types have this form:

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

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

Semantics of primitives

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

Example - counters again

$$\begin{aligned} counter_1 = {} & \textbf{class} : \textbf{exp}[\textbf{int}] \times \textbf{comm} \\ & \quad \textbf{local}\ \mathrm{Var}[\textbf{int}]\ x; \\ & \quad \textbf{init}\ x := 0; \\ & \quad \textbf{meth}\ (\mathrm{deref}\ x,\ x := x + 1) \end{aligned}$$

- Its meaning should be a semantic value of type:

$$\exists_Z\, (\mathcal{Q}_Z)_\perp \times (\mathrm{EXP}_{Int} \times \mathrm{COMM})(Z)$$

- The store Z for the object is given by

$$\begin{aligned} \mathcal{Q}_Z &= Int \\ \mathcal{T}_Z &= \text{read-closure of } \{\overline{\perp}\} \cup \{\, \mathrm{inc}(k) \mid k \geq 0 \,\} \end{aligned}$$

- The initial value is $0 \in (\mathcal{Q}_Z)_\perp$.
- The method suite in $(\mathrm{EXP}_{Int} \times \mathrm{COMM})(Z)$ is the pair:

$$meth_1 = (\lambda n.\, n,\ \mathrm{inc}(1))$$

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.,

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

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

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

Section 6

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