Game Semantics

Andrzej S. Murawski and Nikos Tzevelekos

Day 5: General references, operational games

Lift Beyond the Ground: RefML

Recall the **types** of GroundML:

$$\theta \;::=\; \zeta \mid \theta \times \theta \mid \theta \to \theta$$
$$\zeta \;::=\; \mathsf{unit} \mid \mathsf{int} \mid \mathsf{ref}\,\zeta$$

Restriction: all references are of *ground* type

- OK: $\mathsf{let}\, x = \mathsf{ref}(0)\, \mathsf{in} \cdots$
- OK: $\mathsf{let}\, x = \mathsf{ref}(\mathsf{ref}(0))\, \mathsf{in} \cdots$
- not OK: $\mathsf{let}\, x = \mathsf{ref}(\langle 0, 1 \rangle)\, \mathsf{in} \cdots$
- not OK: $\mathsf{let}\, f = \mathsf{ref}(\lambda x^{\mathsf{int}}.x)\, \mathsf{in} \cdots$

Lift Beyond the Ground: RefML

Recall the **types** of GroundML:

$$\theta \ ::= \ \zeta \mid \theta \times \theta \mid \theta \to \theta$$
$$\zeta \ ::= \ \mathsf{unit} \mid \mathsf{int} \mid \mathsf{ref}\,\zeta$$

Restriction: all references are of *ground* type

- OK: $\mathsf{let}\,x = \mathsf{ref}(0)\,\mathsf{in}\,\cdots$
- OK: $\mathsf{let}\,x = \mathsf{ref}(\mathsf{ref}(0))\,\mathsf{in}\,\cdots$
- not OK: $\mathsf{let}\,x = \mathsf{ref}(\langle 0, 1\rangle)\,\mathsf{in}\,\cdots$
- not OK: $\mathsf{let}\,f = \mathsf{ref}(\lambda x^{\mathsf{int}}.x)\,\mathsf{in}\,\cdots$

Does this matter?

- we can simulate $\mathsf{let}\,x = \mathsf{ref}(\langle 0, 1\rangle)\,\mathsf{in}\,M$ by:

$$\mathsf{let}\,x_l = \mathsf{ref}(0)\,\mathsf{in}\,\mathsf{let}\,x_r = \mathsf{ref}(1)\,\mathsf{in}\,\text{'}M\{\langle x_l, x_r\rangle / x\}\text{'}$$

Higher-order references, expressivity

But we cannot simulate let $f = \mathsf{ref}(\lambda x^{\mathsf{int}}.x)$ in M.

Does it matter – can we express fewer programs?

Higher-order references, expressivity

But we cannot simulate let $f = \mathsf{ref}(\lambda x^{\mathsf{int}}.x)$ in M.

Does it matter – can we express fewer programs?

- $\vdash M$: int

Higher-order references, expressivity

But we cannot simulate let $f = \mathsf{ref}(\lambda x^{\mathsf{int}}.x)$ in M.

Does it matter – can we express fewer programs?

- $\vdash M : \mathsf{int}$: same programs (i.e. same integers)
- $\vdash M : \mathsf{int} \to \mathsf{int}$

Higher-order references, expressivity

But we cannot simulate let $f = \mathsf{ref}(\lambda x^{\mathsf{int}}.x)$ in M.

Does it matter – can we express fewer programs?

- $\vdash M : \mathsf{int}$: same programs (i.e. same integers)
- $\vdash M : \mathsf{int} \to \mathsf{int}$: same programs (i.e. same terms up to equivalence)
- but this does not hold in general!

A delay buffer

Example. Consider a term

$$\vdash \textbf{dBuf} : (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$$

evaluating to some function f such that:

- the first time we call f, say by executing $f(f_1)$, it returns $\lambda x^{\mathsf{int}}.x$;
- the second time we call f, say by executing $f(f_2)$, it returns f_1;
- $\cdots$
- the i-th time we call f, say by $f(f_i)$, it returns f_{i-1}.

A delay buffer

Example. Consider a term

$$\vdash \textbf{dBuf} : (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$$

evaluating to some function f such that:

- the first time we call f, say by executing $f(f_1)$, it returns $\lambda x^{\mathsf{int}}.x$;
- the second time we call f, say by executing $f(f_2)$, it returns f_1;
- $\ldots$
- the i-th time we call f, say by $f(f_i)$, it returns f_{i-1}.

We can implement it using references of type int → int:

$$\begin{aligned}\textbf{dBuf} \;\equiv\; & \mathsf{let}\, r = \mathsf{ref}(\lambda x^{\mathsf{int}}.x)\,\mathsf{in} \\ & \lambda f^{\mathsf{int}\to\mathsf{int}}.\ \mathsf{let}\, f_{old} = !r\,\mathsf{in}\;\; r := f;\;\; f_{old}\end{aligned}$$

A delay buffer – not codable in GroundML

Example. Consider a term

$$\vdash \textbf{dBuf} : (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$$

evaluating to some function f such that:

- the first time we call f, say by executing $f(f_1)$, it returns $\lambda x^{\mathsf{int}}.x$;
- thereafter, the i-th time we call f, say by $f(f_i)$, it returns f_{i-1}.

Lemma. *We cannot implement* **dBuf** *in* GroundML*.*

A delay buffer – not codable in GroundML

Example. Consider a term

$$\vdash \textbf{dBuf} : (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$$

evaluating to some function f such that:

- the first time we call f, say by executing $f(f_1)$, it returns $\lambda x^{\mathsf{int}}.x$;
- thereafter, the i-th time we call f, say by $f(f_i)$, it returns f_{i-1}.

Lemma. *We cannot implement* **dBuf** *in* GroundML*.*

Proof (sketch). If we could, then $[\![\textbf{dBuf}]\!]$ would contain plays like:

$$\begin{array}{ccccccccc} \star & \dagger & \dagger_1 & \dagger_1' & \dagger_2 & \dagger_2' & 42 & 42 & \cdots \\ \mathit{OQ} & \mathit{PA} & \mathit{OQ} & \mathit{PA} & \mathit{OQ} & \mathit{PA} & \mathit{OQ} & \mathit{PQ} & \end{array}$$

(where there are no explicit pointers, assume the move points to its predecessor)

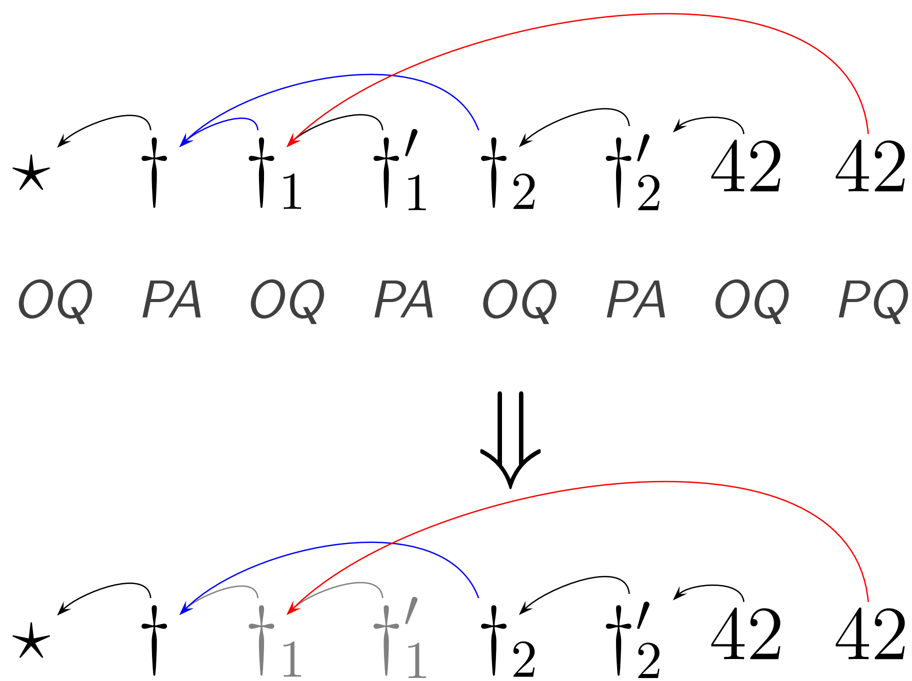
which would break visibility. □

Recall Visibility

In any play

$$s\ m$$

the move that m points to must be in the **view** of s



The view hides moves from suspended function calls

Higher-order references: more distinctions

Lemma. *The following terms are equivalent in* GroundML*:*

$$f : \mathsf{unit}\to\mathsf{unit} \vdash \ \mathsf{let}\, n = \mathsf{ref}(0)\,\mathsf{in}\, \lambda\_.\ \mathsf{if}\ !n\, \mathsf{then}\, ()\, \mathsf{else}\, n := 1; f() : \mathsf{unit}\to\mathsf{unit}$$

$$f : \mathsf{unit}\to\mathsf{unit} \vdash \ \mathsf{let}\, n = \mathsf{ref}(0)\,\mathsf{in}\, \lambda\_.\ \mathsf{if}\ !n\, \mathsf{then}\, ()\, \mathsf{else}\, f(); n := 1 : \mathsf{unit}\to\mathsf{unit}$$

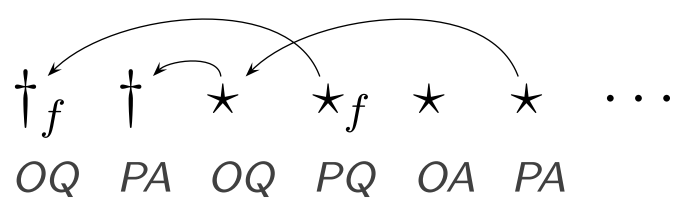
Higher-order references: more distinctions

Lemma. *The following terms are equivalent in* GroundML*:*

$$f : \mathsf{unit}{\to}\mathsf{unit} \vdash \ \mathsf{let}\, n = \mathsf{ref}(0)\, \mathsf{in}\, \lambda\_.\ \mathsf{if}\, !n\, \mathsf{then}\, ()\, \mathsf{else}\, n := 1; f() : \mathsf{unit}{\to}\mathsf{unit}$$

$$f : \mathsf{unit}{\to}\mathsf{unit} \vdash \ \mathsf{let}\, n = \mathsf{ref}(0)\, \mathsf{in}\, \lambda\_.\ \mathsf{if}\, !n\, \mathsf{then}\, ()\, \mathsf{else}\, f(); n := 1 : \mathsf{unit}{\to}\mathsf{unit}$$

Proof. Computing the game semantics in each case, we get that it must consist of plays of the form:



In particular, the question $\star_f$ must be immediately answered as, at that point in the play, O has no other move to play (by visibility). □

But the terms can be distinguished by a context that uses higher-order references.

The language RefML – Definition

$$\theta \ ::= \ \mathsf{unit} \mid \mathsf{int} \mid \mathsf{ref}\,\theta \mid \theta \times \theta \mid \theta \to \theta \qquad \mathbb{A} = \biguplus_\theta \mathbb{A}_\theta$$

$$\frac{}{\mathrm{U},\Gamma \vdash () : \mathsf{unit}} \qquad \frac{i \in \mathbb{Z}}{\mathrm{U},\Gamma \vdash i : \mathsf{int}} \qquad \frac{(x:\theta) \in \Gamma}{\mathrm{U},\Gamma \vdash x : \theta} \qquad \frac{a \in \mathrm{U} \cap \mathbb{A}_\theta}{\mathrm{U},\Gamma \vdash a : \mathsf{ref}\theta}$$

$$\frac{\mathrm{U},\Gamma \vdash M : \mathsf{int} \quad \mathrm{U},\Gamma \vdash N_0 : \theta \quad \mathrm{U},\Gamma \vdash N_1 : \theta}{\mathrm{U},\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_0 : \theta} \qquad \frac{\mathrm{U},\Gamma \vdash M : \mathsf{int}}{\mathrm{U},\Gamma \vdash \mathsf{while}(M) : \mathsf{unit}}$$

$$\frac{\mathrm{U},\Gamma \uplus \{\,x:\theta\,\} \vdash M : \theta'}{\mathrm{U},\Gamma \vdash \lambda x^\theta.M : \theta \to \theta'} \qquad \frac{\mathrm{U},\Gamma \vdash M : \theta \to \theta' \quad \mathrm{U},\Gamma \vdash N : \theta}{\mathrm{U},\Gamma \vdash MN : \theta'}$$

$$\frac{\mathrm{U},\Gamma \vdash M : \theta \quad \mathrm{U},\Gamma \vdash N : \theta'}{\mathrm{U},\Gamma \vdash \langle M,N \rangle : \theta \times \theta'} \qquad \frac{\mathrm{U},\Gamma \vdash M : \theta_1 \times \theta_2}{\mathrm{U},\Gamma \vdash \pi_i M : \theta_i}\ {\scriptstyle i\in\{1,2\}}$$

$$\frac{\mathrm{U},\Gamma \vdash M : \mathsf{int} \quad \mathrm{U},\Gamma \vdash N : \mathsf{int}}{\mathrm{U},\Gamma \vdash M \oplus N : \mathsf{int}} \qquad \frac{\mathrm{U},\Gamma \vdash M : \mathsf{ref}\theta \quad \mathrm{U},\Gamma \vdash N : \mathsf{ref}\theta}{\mathrm{U},\Gamma \vdash M = N : \mathsf{int}}$$

$$\frac{\mathrm{U},\Gamma \vdash M : \theta}{\mathrm{U},\Gamma \vdash \mathsf{ref}(M) : \mathsf{ref}\theta} \qquad \frac{\mathrm{U},\Gamma \vdash M : \mathsf{ref}\theta}{\mathrm{U},\Gamma \vdash\ !M : \theta} \qquad \frac{\mathrm{U},\Gamma \vdash M : \mathsf{ref}\theta \quad \mathrm{U},\Gamma \vdash N : \theta}{\mathrm{U},\Gamma \vdash M := N : \mathsf{unit}}$$

Operational semantics of RefML – remains the same

$$
\begin{array}{rcll}
(i \oplus j, S) & \longrightarrow & (k, S) & (k = i \oplus j) \\
((\lambda x.M)V, S) & \longrightarrow & (M[V/x], S) & \\
(\pi_1\langle V_1, V_2\rangle, S) & \longrightarrow & (V_1, S) & \\
(\pi_2\langle V_1, V_2\rangle, S) & \longrightarrow & (V_2, S) & \\
(\mathsf{if}\ 0\ \mathsf{then}\ M\ \mathsf{else}\ M', S) & \longrightarrow & (M', S) & \\
(\mathsf{if}\ i\ \mathsf{then}\ M\ \mathsf{else}\ M', S) & \longrightarrow & (M, S) & (i > 0) \\
(\mathsf{while}(M), S) & \longrightarrow & (\mathsf{if}\ M\ \mathsf{then}\ \mathsf{while}(M)\ \mathsf{else}\ (), S) & \\
(a = b, S) & \longrightarrow & (0, S) & (a \neq b) \\
(a = a, S) & \longrightarrow & (1, S) & \\
(!a, S) & \longrightarrow & (S(a), S) & \\
(a := V, S) & \longrightarrow & ((), S[a \mapsto V]) & \\
(\mathsf{ref}(V), S) & \longrightarrow & (a', S[a' \mapsto V]) & (a' \notin \mathsf{dom}(S))
\end{array}
$$

$$
\frac{(M, S) \longrightarrow (M', S')}{(E[M], S) \longrightarrow (E[M'], S')}
$$

Example RefML terms

$$\textbf{dBuf} \equiv \mathsf{let}\, r = \mathsf{ref}(\lambda x^{\mathsf{int}}.x)\,\mathsf{in}$$
$$\lambda f^{\mathsf{int}\to\mathsf{int}}.\ \mathsf{let}\, f_{old} = !r\,\mathsf{in}\ \ r := f;\ \ f_{old} : (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$$

Example RefML terms

$$\textbf{dBuf} \equiv \text{let}\, r = \text{ref}(\lambda x^{\text{int}}.x)\,\text{in}$$
$$\lambda f^{\text{int}\to\text{int}}.\ \text{let}\, f_{old} = !r\,\text{in}\ \ r := f;\ \ f_{old} : (\text{int} \to \text{int}) \to (\text{int} \to \text{int})$$

$$\textbf{lambdaMax} \equiv \text{let}\, r = \text{ref}(\lambda x^{\text{int}}.x), n = \text{ref}(1)\,\text{in}$$
$$\lambda f^{\text{int}\to\text{int}}.\ (\ \text{if}\ !n\,\text{then}\, n := 0; r := f$$
$$\text{else let}\, f_{old} = !r\,\text{in}\, r := \lambda x^{\text{int}}.\ \text{max}(f_{old}x, fx)\ );$$
$$!r$$
$$: (\text{int} \to \text{int}) \to (\text{int} \to \text{int})$$

Example RefML terms

$$\textbf{dBuf} \equiv \text{let}\, r = \text{ref}(\lambda x^{\text{int}}.x)\, \text{in}$$
$$\lambda f^{\text{int}\to\text{int}}.\ \text{let}\, f_{old} = !r\, \text{in}\ \ r := f;\ \ f_{old} : (\text{int} \to \text{int}) \to (\text{int} \to \text{int})$$

$$\begin{aligned}\textbf{lambdaMax} \equiv\ & \text{let}\, r = \text{ref}(\lambda x^{\text{int}}.x), n = \text{ref}(1)\, \text{in} \\ & \lambda f^{\text{int}\to\text{int}}.\ (\ \text{if}\, !n\, \text{then}\, n := 0; r := f \\ & \qquad \text{else let}\, f_{old} = !r\, \text{in}\, r := \lambda x^{\text{int}}.\ \text{max}(f_{old}x, fx)\ ); \\ & \qquad !r \\ & : (\text{int} \to \text{int}) \to (\text{int} \to \text{int})\end{aligned}$$

How to distinguish:

$$f : \text{unit}\to\text{unit} \vdash\ \text{let}\, n = \text{ref}(0)\, \text{in}\, \lambda\_.\ \text{if}\, !n\, \text{then}\, ()\, \text{else}\, n := 1; f() : \text{unit}\to\text{unit}$$

$$f : \text{unit}\to\text{unit} \vdash\ \text{let}\, n = \text{ref}(0)\, \text{in}\, \lambda\_.\ \text{if}\, !n\, \text{then}\, ()\, \text{else}\, f(); n := 1 : \text{unit}\to\text{unit}$$

Games for higher-order references

How can we adapt the game model of GroundML to model higher-order references? How should we change:

arenas?	
moves?	
plays?	
strategies?	

Games for higher-order references

How can we adapt the game model of GroundML to model higher-order references? How should we change:

arenas?	these remain the same
moves?	
plays?	
strategies?	

Games for higher-order references

How can we adapt the game model of GroundML to model higher-order references? How should we change:

arenas?	these remain the same
moves?	stores need to be higher-order
plays?	
strategies?	

Games for higher-order references

How can we adapt the game model of GroundML to model higher-order references? How should we change:

arenas?	these remain the same
moves?	stores need to be higher-order
plays?	can break visibility
strategies?	

Games for higher-order references

How can we adapt the game model of GroundML to model higher-order references? How should we change:

arenas?	these remain the same
moves?	stores need to be higher-order
plays?	can break visibility
strategies?	same conditions as before + additional conditions for composition

Stores with higher-order values

What is a higher-order value?

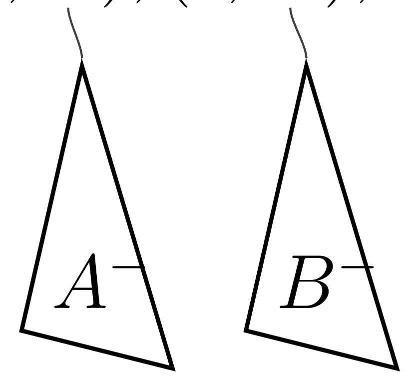
Stores with higher-order values

What is a higher-order value?

It is a move opening an arena, and should contain some †.

Stores storing ground and higher-order values:

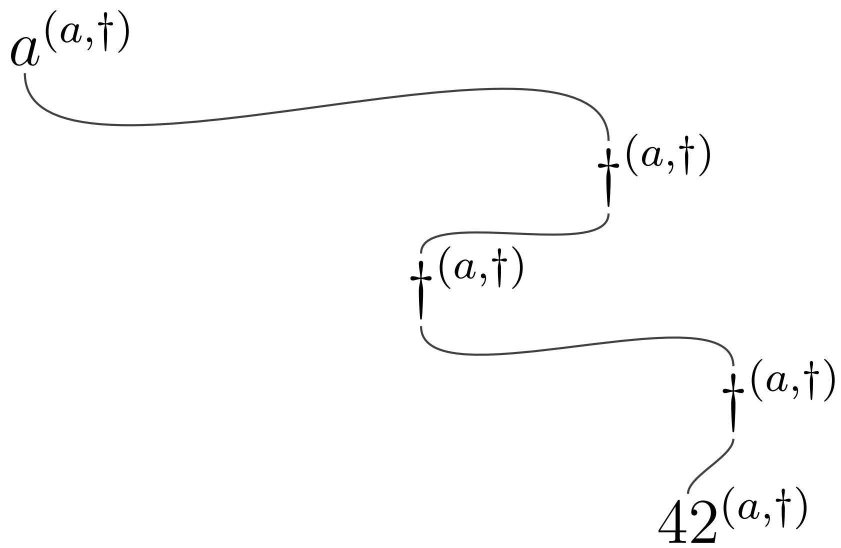
GroundML	RefML
$S = \{(a, 0), (b, a), \cdots\}$	$S = \{(a, \mathrm{i}_A), (b, \mathrm{i}_B), \cdots\}$



Pointers pointers everywhere

$r : \mathsf{ref}(\mathsf{int}{\to}\mathsf{int}) \vdash \lambda f^{\mathsf{int}\to\mathsf{int}}.\ \mathsf{let}\ f_{old} = !r\ \mathsf{in}\ \ r := f;\ \ f_{old} : (\mathsf{int}{\to}\mathsf{int}) \to (\mathsf{int}{\to}\mathsf{int})$

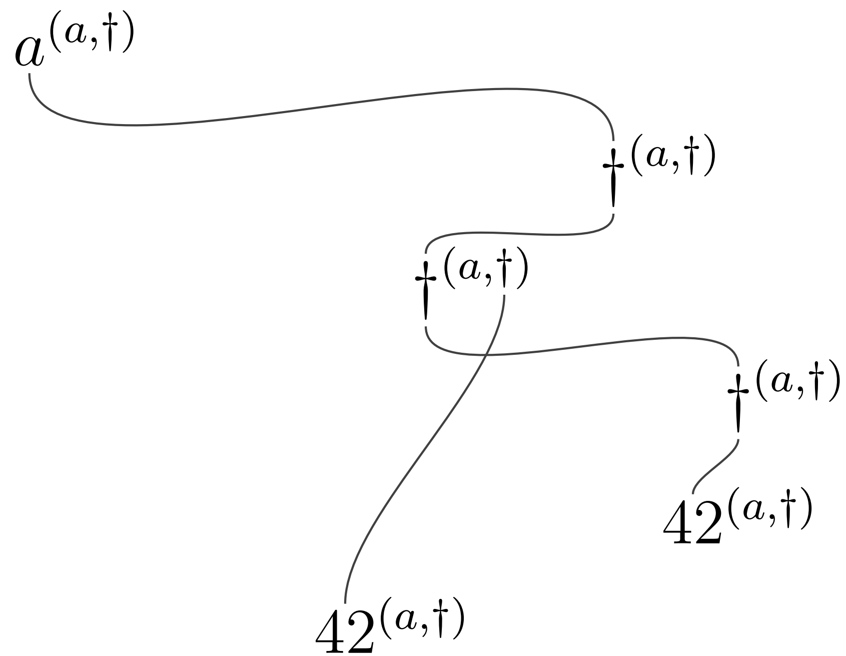
$$\mathbb{A}_{\mathsf{int}\to\mathsf{int}} \longrightarrow (\mathbb{Z} \Rightarrow \mathbb{Z}) \Rightarrow (\mathbb{Z} \Rightarrow \mathbb{Z})$$



Pointers pointers everywhere

$r : \mathsf{ref}(\mathsf{int}{\to}\mathsf{int}) \vdash \lambda f^{\mathsf{int}\to\mathsf{int}}.\ \mathsf{let}\ f_{old} = !r\ \mathsf{in}\ \ r := f;\ \ f_{old} : (\mathsf{int}{\to}\mathsf{int}) \to (\mathsf{int}{\to}\mathsf{int})$

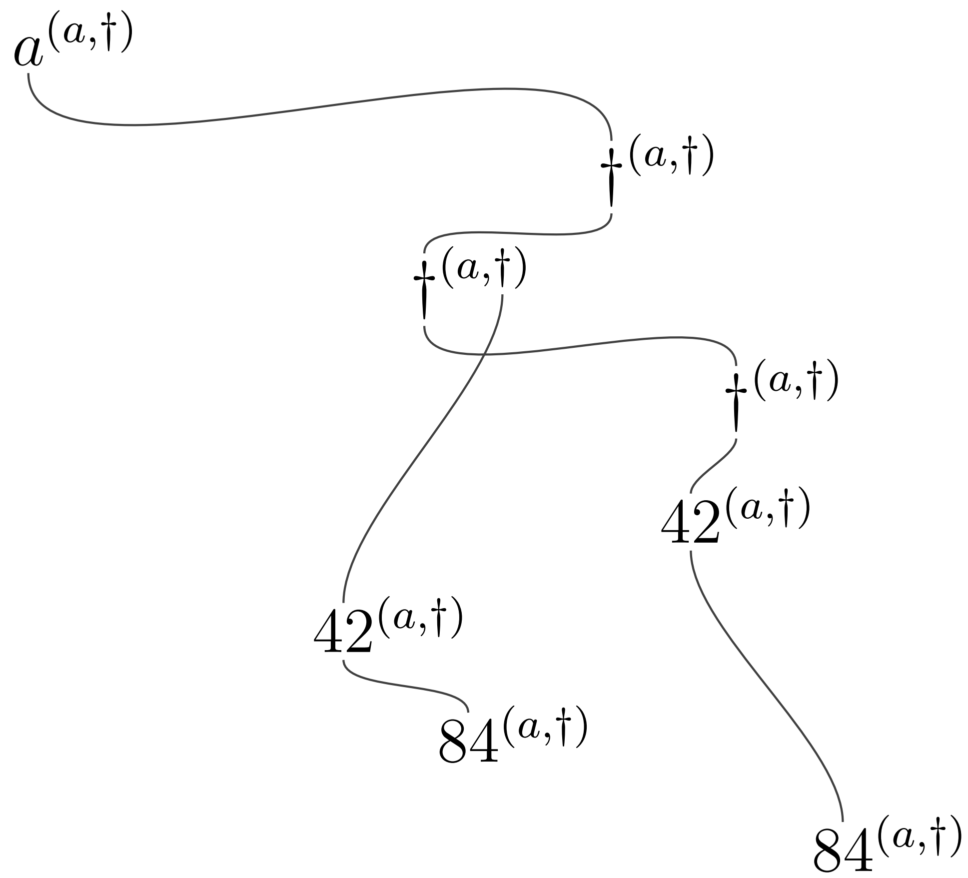
$$\mathbb{A}_{\mathsf{int}\to\mathsf{int}} \longrightarrow (\mathbb{Z} \Rightarrow \mathbb{Z}) \Rightarrow (\mathbb{Z} \Rightarrow \mathbb{Z})$$



Pointers pointers everywhere

$r : \mathsf{ref}(\mathsf{int}{\to}\mathsf{int}) \vdash \lambda f^{\mathsf{int}\to\mathsf{int}}.\ \mathsf{let}\ f_{old} = {!r}\ \mathsf{in}\ \ r := f;\ \ f_{old} : (\mathsf{int}{\to}\mathsf{int}) \to (\mathsf{int}{\to}\mathsf{int})$

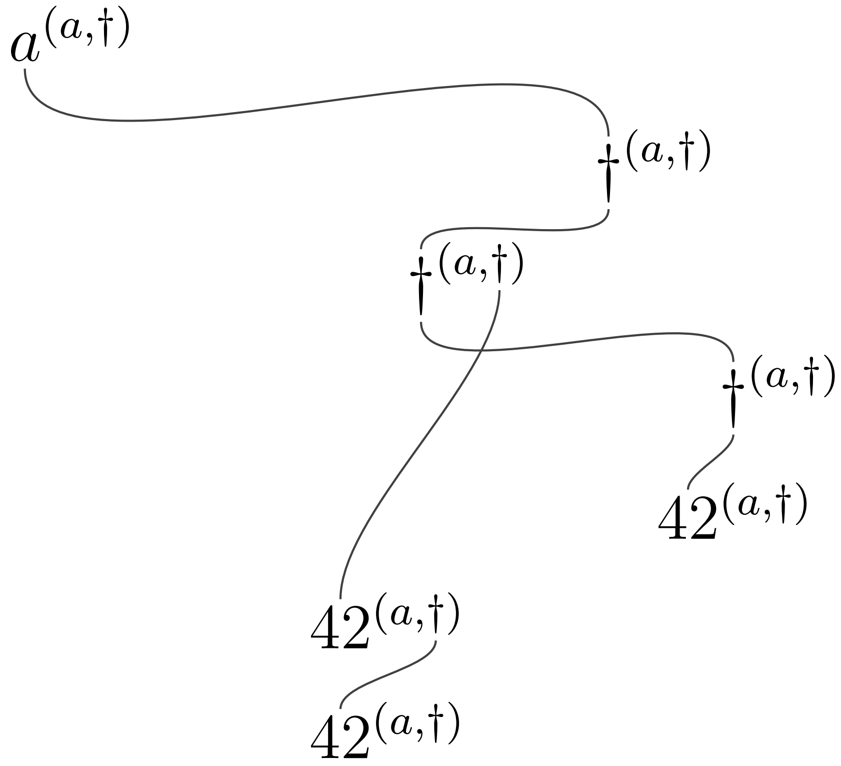
$$\mathbb{A}_{\mathsf{int}\to\mathsf{int}} \longrightarrow (\mathbb{Z} \Rightarrow \mathbb{Z}) \Rightarrow (\mathbb{Z} \Rightarrow \mathbb{Z})$$



Pointers pointers everywhere

$r : \mathsf{ref}(\mathsf{int}{\to}\mathsf{int}) \vdash \lambda f^{\mathsf{int}\to\mathsf{int}}.\ \mathsf{let}\, f_{old} = !r\ \mathsf{in}\ \ r := f;\ \ f_{old} : (\mathsf{int}{\to}\mathsf{int}) \to (\mathsf{int}{\to}\mathsf{int})$

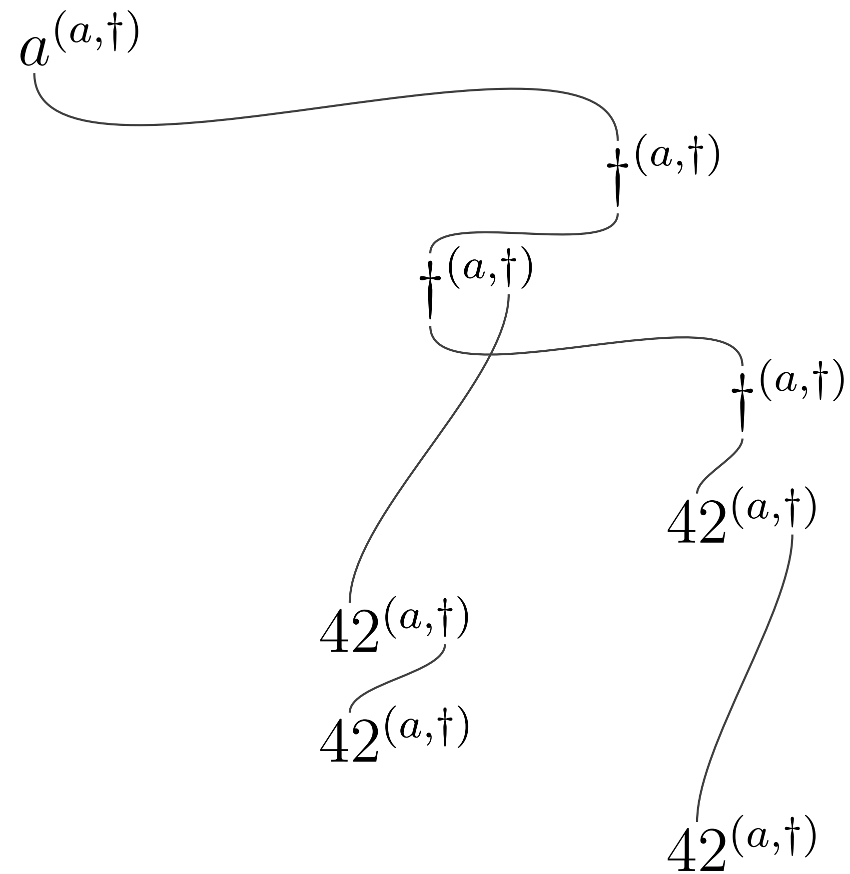
$$\mathbb{A}_{\mathsf{int}\to\mathsf{int}} \longrightarrow (\mathbb{Z} \Rightarrow \mathbb{Z}) \Rightarrow (\mathbb{Z} \Rightarrow \mathbb{Z})$$



Pointers pointers everywhere

$r : \mathsf{ref}(\mathsf{int}{\to}\mathsf{int}) \vdash \lambda f^{\mathsf{int}\to\mathsf{int}}.\ \mathsf{let}\ f_{old} = !r\ \mathsf{in}\ \ r := f;\ \ f_{old} : (\mathsf{int}{\to}\mathsf{int}) \to (\mathsf{int}{\to}\mathsf{int})$

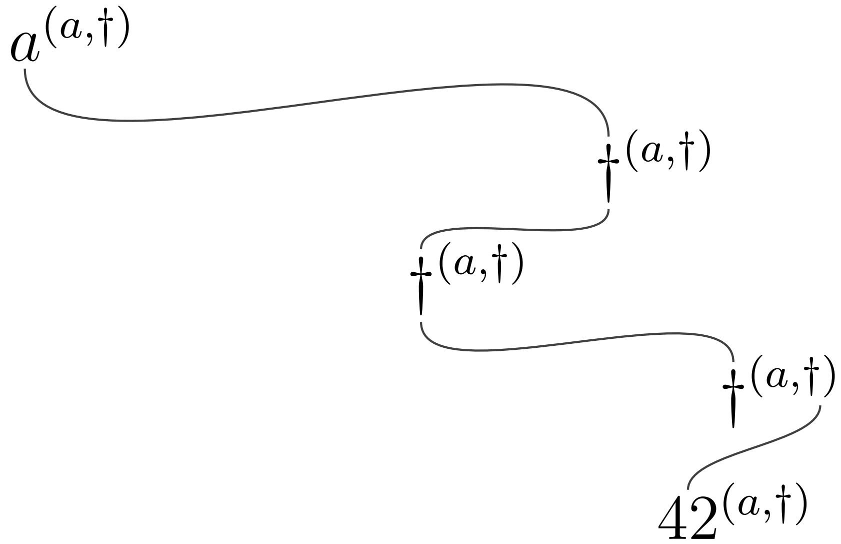
$$\mathbb{A}_{\mathsf{int}\to\mathsf{int}} \longrightarrow (\mathbb{Z} \Rightarrow \mathbb{Z}) \Rightarrow (\mathbb{Z} \Rightarrow \mathbb{Z})$$



Pointers pointers everywhere

$r : \mathsf{ref}(\mathsf{int}{\to}\mathsf{int}) \vdash \lambda f^{\mathsf{int}\to\mathsf{int}}.\ \mathsf{let}\ f_{old} = !r\ \mathsf{in}\ \ r := f;\ \ f_{old} : (\mathsf{int}{\to}\mathsf{int}) \to (\mathsf{int}{\to}\mathsf{int})$

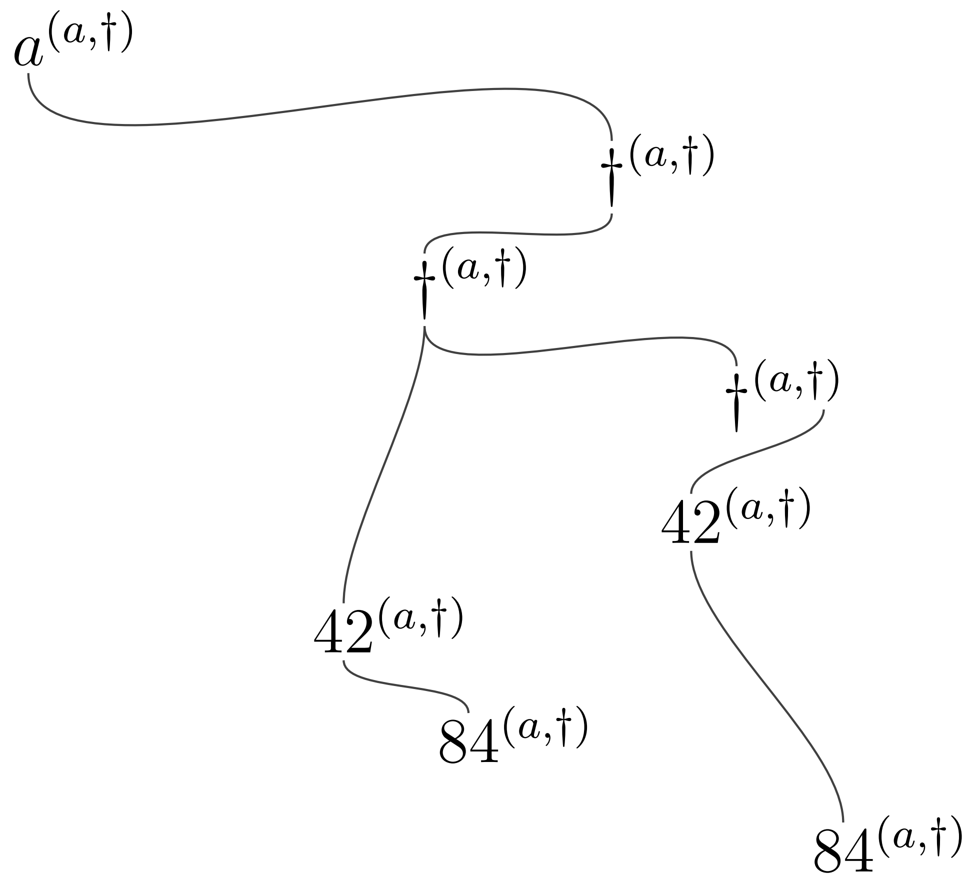
$$\mathbb{A}_{\mathsf{int}\to\mathsf{int}} \longrightarrow (\mathbb{Z} \Rightarrow \mathbb{Z}) \Rightarrow (\mathbb{Z} \Rightarrow \mathbb{Z})$$



Pointers pointers everywhere

$r : \mathsf{ref}(\mathsf{int}{\to}\mathsf{int}) \vdash \lambda f^{\mathsf{int}\to\mathsf{int}}.\ \mathsf{let}\ f_{old} = !r\ \mathsf{in}\ \ r := f;\ \ f_{old} : (\mathsf{int}{\to}\mathsf{int}) \to (\mathsf{int}{\to}\mathsf{int})$

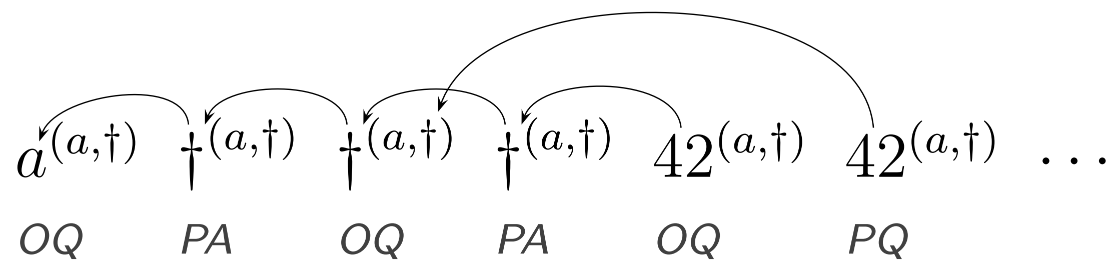
$$\mathbb{A}_{\mathsf{int}\to\mathsf{int}} \longrightarrow (\mathbb{Z} \Rightarrow \mathbb{Z}) \Rightarrow (\mathbb{Z} \Rightarrow \mathbb{Z})$$



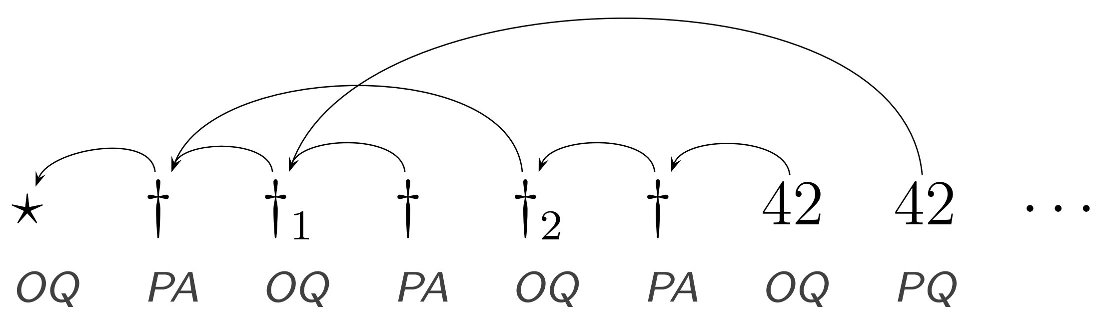
Plays

There are two kinds of modifications we make on plays:

- There are now two kinds of pointers: to-move and to-store pointers:



- Visibility is no longer imposed

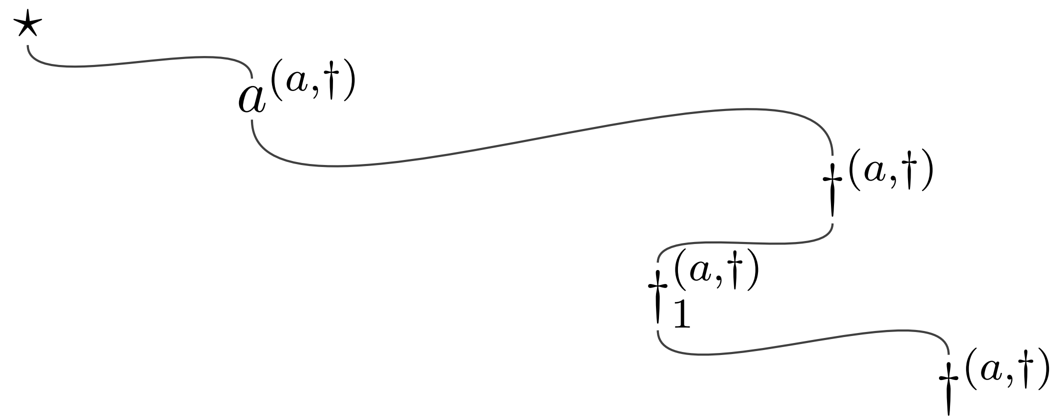


Composition: more copycat conditions

$\sigma = [\![\mathsf{ref}(\lambda x^{\mathsf{int}}.\, x : \mathsf{int} \to \mathsf{int})]\!]$

$\tau = [\![r : \mathsf{ref}(\mathsf{int}\to\mathsf{int}) \vdash \lambda f^{\mathsf{int}\to\mathsf{int}}.\, \mathsf{let}\, f_{old} = !r \,\mathsf{in}\;\; r := f;\;\; f_{old} : (\mathsf{int}\to\mathsf{int})\to(\mathsf{int} \to \mathsf{int})]\!]$

$$1 \xrightarrow{\ \sigma\ } \mathbb{A}_{\mathsf{int}\to\mathsf{int}} \xrightarrow{\ \tau\ } (\mathbb{Z} \Rightarrow \mathbb{Z}) \Rightarrow (\mathbb{Z} \Rightarrow \mathbb{Z})$$

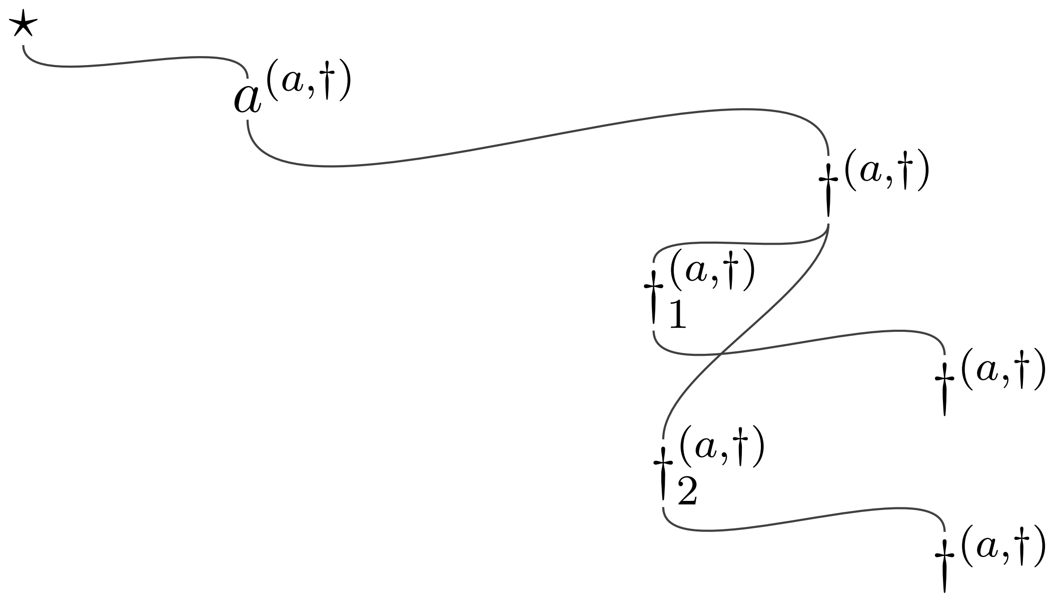


Composition: more copycat conditions

$\sigma = [\![\mathsf{ref}(\lambda x^{\mathsf{int}}.\, x : \mathsf{int} \to \mathsf{int})]\!]$

$\tau = [\![r : \mathsf{ref}(\mathsf{int}{\to}\mathsf{int}) \vdash \lambda f^{\mathsf{int}\to\mathsf{int}}.\, \mathsf{let}\, f_{old} = !r \,\mathsf{in}\ \ r := f;\ \ f_{old} : (\mathsf{int}{\to}\mathsf{int}){\to}(\mathsf{int} \to \mathsf{int})]\!]$

$$1 \xrightarrow{\ \sigma\ } \mathbb{A}_{\mathsf{int}\to\mathsf{int}} \xrightarrow{\ \tau\ } (\mathbb{Z} \Rightarrow \mathbb{Z}) \Rightarrow (\mathbb{Z} \Rightarrow \mathbb{Z})$$

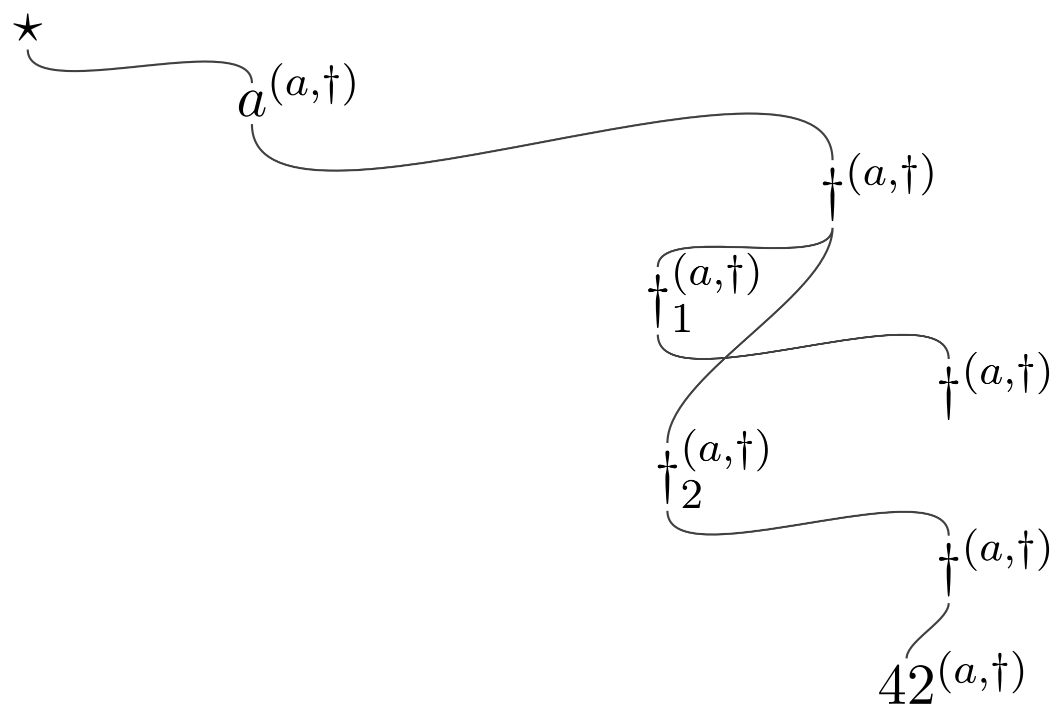


Composition: more copycat conditions

$\sigma = [\![\mathsf{ref}(\lambda x^{\mathsf{int}}.\, x : \mathsf{int} \to \mathsf{int})]\!]$

$\tau = [\![r : \mathsf{ref}(\mathsf{int}\to\mathsf{int}) \vdash \lambda f^{\mathsf{int}\to\mathsf{int}}.\, \mathsf{let}\, f_{old} = !r \,\mathsf{in}\;\; r := f;\;\; f_{old} : (\mathsf{int}\to\mathsf{int})\to(\mathsf{int} \to \mathsf{int})]\!]$

$$1 \xrightarrow{\;\sigma\;} \mathbb{A}_{\mathsf{int}\to\mathsf{int}} \xrightarrow{\;\tau\;} (\mathbb{Z} \Rightarrow \mathbb{Z}) \Rightarrow (\mathbb{Z} \Rightarrow \mathbb{Z})$$

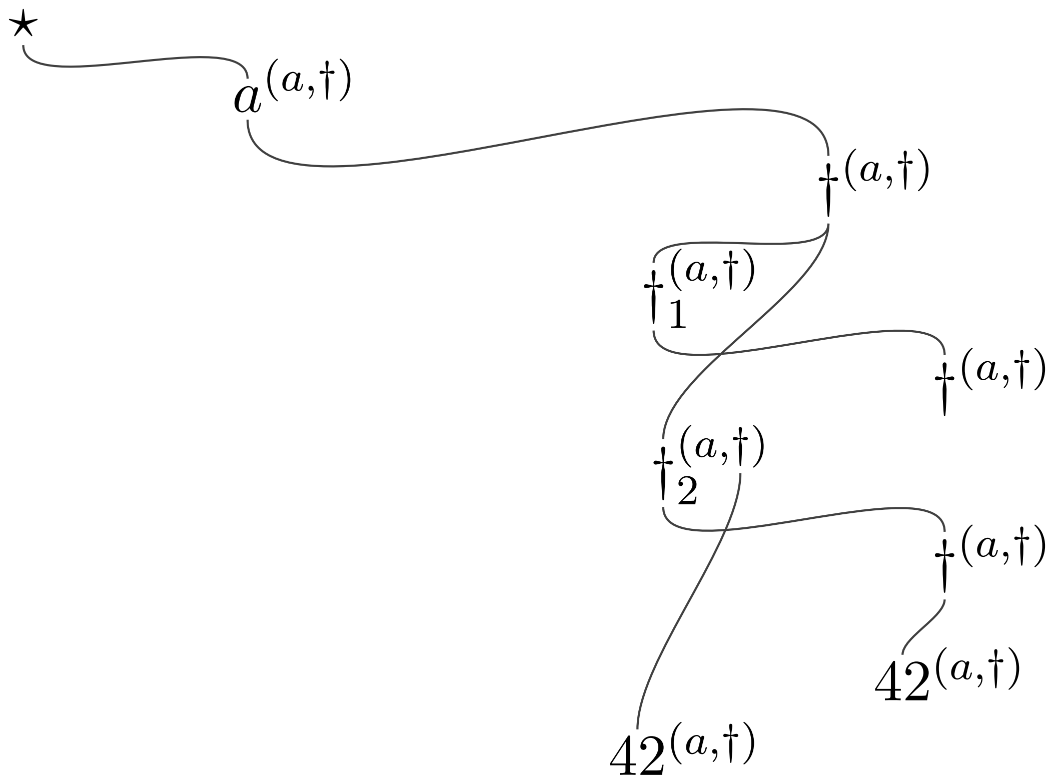


Composition: more copycat conditions

$\sigma = [\![\mathsf{ref}(\lambda x^{\mathsf{int}}.\, x : \mathsf{int} \to \mathsf{int})]\!]$

$\tau = [\![r : \mathsf{ref}(\mathsf{int}{\to}\mathsf{int}) \vdash \lambda f^{\mathsf{int}\to\mathsf{int}}.\, \mathsf{let}\, f_{old} = !r \;\mathsf{in}\;\; r := f;\;\, f_{old} : (\mathsf{int}{\to}\mathsf{int}){\to}(\mathsf{int} \to \mathsf{int})]\!]$

$$1 \xrightarrow{\;\sigma\;} \mathbb{A}_{\mathsf{int}\to\mathsf{int}} \xrightarrow{\;\tau\;} (\mathbb{Z} \Rightarrow \mathbb{Z}) \Rightarrow (\mathbb{Z} \Rightarrow \mathbb{Z})$$

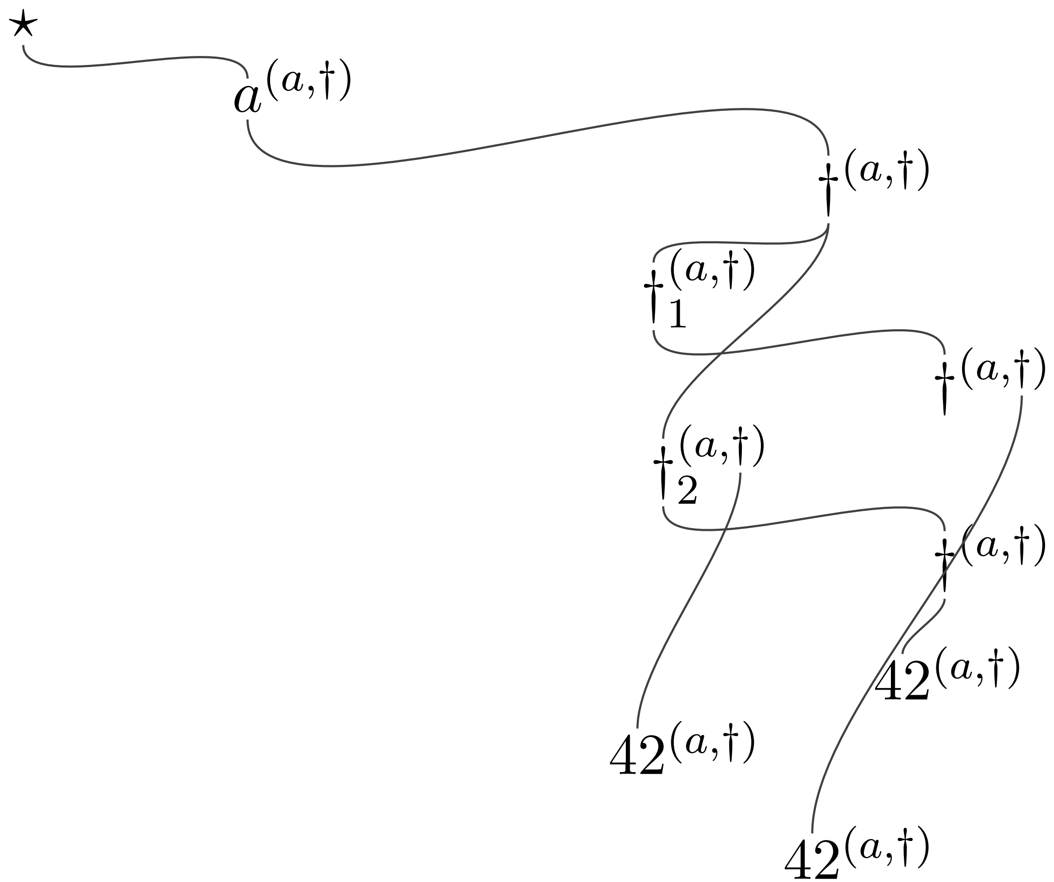


Composition: more copycat conditions

$\sigma = [\![\mathsf{ref}(\lambda x^{\mathsf{int}}.\, x : \mathsf{int} \to \mathsf{int})]\!]$

$\tau = [\![r : \mathsf{ref}(\mathsf{int}{\to}\mathsf{int}) \vdash \lambda f^{\mathsf{int}\to\mathsf{int}}.\, \mathsf{let}\ f_{old} = !r\ \mathsf{in}\ \ r := f;\ \ f_{old} : (\mathsf{int}{\to}\mathsf{int}){\to}(\mathsf{int} \to \mathsf{int})]\!]$

$$1 \xrightarrow{\ \sigma\ } \mathbb{A}_{\mathsf{int}\to\mathsf{int}} \xrightarrow{\ \tau\ } (\mathbb{Z} \Rightarrow \mathbb{Z}) \Rightarrow (\mathbb{Z} \Rightarrow \mathbb{Z})$$

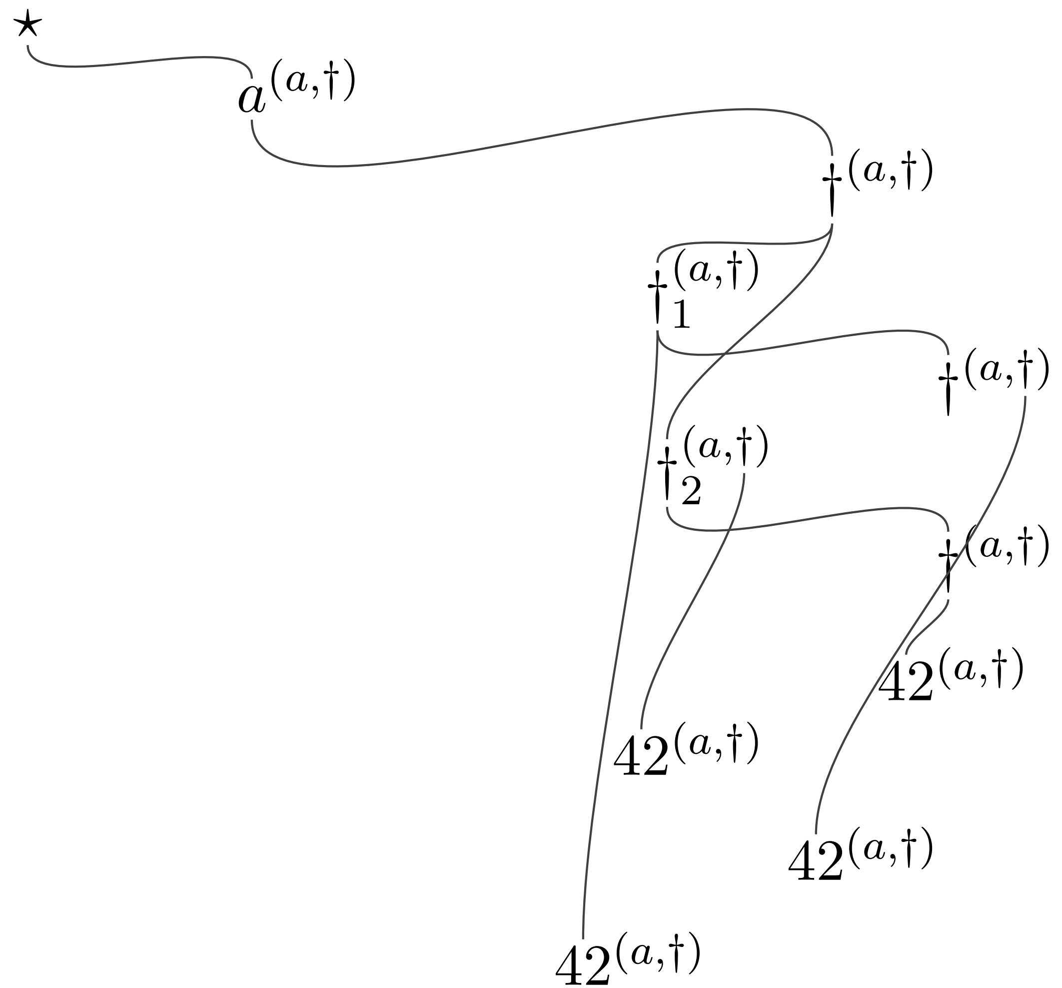


Composition: more copycat conditions

$\sigma = [\![\mathsf{ref}(\lambda x^{\mathsf{int}}.\, x : \mathsf{int} \to \mathsf{int})]\!]$

$\tau = [\![r : \mathsf{ref}(\mathsf{int}{\to}\mathsf{int}) \vdash \lambda f^{\mathsf{int}\to\mathsf{int}}.\, \mathsf{let}\, f_{old} = !r\; \mathsf{in}\;\; r := f;\;\; f_{old} : (\mathsf{int}{\to}\mathsf{int}){\to}(\mathsf{int} \to \mathsf{int})]\!]$

$$1 \xrightarrow{\;\sigma\;} \mathbb{A}_{\mathsf{int}\to\mathsf{int}} \xrightarrow{\;\tau\;} (\mathbb{Z} \Rightarrow \mathbb{Z}) \Rightarrow (\mathbb{Z} \Rightarrow \mathbb{Z})$$



Results

- the game model for RefML if fully abstract

$$\text{comp}([\![M]\!]) = \text{comp}([\![N]\!]) \iff M \cong N$$

(and we also have correctness and adequacy)

- conceptually:

$$\text{Higher-order references} \iff \text{Loosen visibility}$$

Something different: operational games

We started from informal dialogues and moved to formal games via:

- defining the rules of the games (arenas, moves, plays, etc.)
- defining the translation of basic terms (e.g. constant, variables, etc.)
- defining composition and other syntactic constructs
- combine the above to produce the semantics of terms

Something different: operational games

We started from informal dialogues and moved to formal games via:

- defining the rules of the games (arenas, moves, plays, etc.)
- defining the translation of basic terms (e.g. constant, variables, etc.)
- defining composition and other syntactic constructs
- combine the above to produce the semantics of terms

An alternative idea is:

just execute terms operationally to produce their plays

- reduce the term internally via its operational semantics
- when an external function needs to be called, play a move
- and take it on from there

Example

$\vdash \lambda y^{\mathsf{int}}. 2 * y + 1 : \mathsf{int} \rightarrow \mathsf{int} :$

- what is the result?
- it is a function m

Example

$\vdash \lambda y^{\mathsf{int}}. 2 * y + 1 : \mathsf{int} \to \mathsf{int}$:

- what is the result?
- it is a function m
  - what is the result of m on 42?
  - (...) it is 85

Example

$\vdash \lambda y^{\text{int}}. 2 * y + 1$: int → int :

- ○ what is the result?
- • it is a function m
  - ○ what is the result of m on 42?
  - • (...) it is 85
  - ○ what is the result of m on 22?
  - • (...) it is 45

Example

$\vdash \lambda y^{\text{int}}.\, 2 * y + 1 : \text{int} \to \text{int}$:

- ○ what is the result?
- • it is a function m
  - ○ what is the result of m on 42?
  - • (...) it is 85
  - ○ what is the result of m on 22?
  - • (...) it is 45

    ...

An operational reading of the game semantics is:

$$\text{call}\,()\;\; \text{ret}\,(m)\;\; \text{call}\,m(42)\;\; \text{ret}\,m(85)\;\; \text{call}\,m(22)\;\; \text{ret}\,m(25)\;\; \cdots$$

m here is a function **name** (think of it as a **method** name)

Example II

f : int $\to$ int $\vdash$ $\lambda x^{\mathsf{int}}.\, f x + 1$: int $\to$ int :

- given m (for f), what is the result?
- it is a function m'

Example II

$f : \mathsf{int} \to \mathsf{int} \vdash \lambda x^{\mathsf{int}}.\, f x + 1 : \mathsf{int} \to \mathsf{int}$:

- given m (for f), what is the result?
- it is a function m'
  - what is the result of m' on 42?
  - what is the result of m on 42?
  - it is 85

Example II

$f : \mathsf{int} \to \mathsf{int} \vdash \lambda x^{\mathsf{int}}.\, f x + 1 : \mathsf{int} \to \mathsf{int}$:

- given m (for f), what is the result?
- it is a function m'
  - what is the result of m' on 42?
  - what is the result of m on 42?
  - it is 85
  - (...) it is 86

Example II

f : int $\to$ int $\vdash\ \lambda x^{\text{int}}.\, f x + 1$: int $\to$ int :

- ◦ given m (for f), what is the result?
- • it is a function m'
  - ◦ what is the result of m' on 42?
  - • what is the result of m on 42?
  - ◦ it is 85
  - • (...) it is 86
  - ◦ what is the result of m' on 22?
  - • what is the result of m on 22?
  - ◦ it is 25

Example II

$f : \mathsf{int} \to \mathsf{int} \vdash \lambda x^{\mathsf{int}}.\, f x + 1 : \mathsf{int} \to \mathsf{int}$:

- ○ given m (for f), what is the result?
- • it is a function m'
  - ○ what is the result of m' on 42?
  - • what is the result of m on 42?
  - ○ it is 85
  - • (...) it is 86
  - ○ what is the result of m' on 22?
  - • what is the result of m on 22?
  - ○ it is 25
  - • (...) it is 26

Example II

f : int $\to$ int $\vdash\ \lambda x^{\text{int}}.\, f x + 1$: int $\to$ int :

- ○ given m (for f), what is the result?
- • it is a function m'
  - ○ what is the result of m' on 42?
  - • what is the result of m on 42?
  - ○ it is 85
  - • (...) it is 86
  - ○ what is the result of m' on 22?
  - • what is the result of m on 22?
  - ○ it is 25
  - • (...) it is 26

call (m) ret (m') call $m'(42)$ call $m(42)$ ret $m(85)$ ret $m'(86)$ $\cdots$

How to obtain the games operationally

- For $\vdash \lambda y^{\mathsf{int}}.\, 2 * y + 1 : \mathsf{int} \to \mathsf{int}$. Start with initial move:

$$(\vdash \lambda y^{\mathsf{int}}.\, 2 * y + 1 : \mathsf{int} \to \mathsf{int}) \xrightarrow{\mathsf{call}\,()} (\lambda y^{\mathsf{int}}.\, 2 * y + 1, S_0, R_0)$$

where S_0 an empty store, and R_0 an empty **repository** (stores method names and their definitions)

How to obtain the games operationally

- For $\vdash\ \lambda y^{\mathsf{int}}.\, 2 * y + 1 : \mathsf{int} \to \mathsf{int}$. Start with initial move:

$$(\vdash \lambda y^{\mathsf{int}}.\, 2 * y + 1 : \mathsf{int} \to \mathsf{int}) \xrightarrow{\mathsf{call}\,()} (\lambda y^{\mathsf{int}}.\, 2 * y + 1, S_0, R_0)$$

  where S_0 an empty store, and R_0 an empty **repository** (stores method names and their definitions)

- proceed to evaluation, producing a name for the function:

$$(\lambda y^{\mathsf{int}}.\, 2 * y + 1, S_0, R_0) \longrightarrow (m, S_0, \{m \mapsto \lambda y.2 * y + 1\})$$

How to obtain the games operationally

- For $\vdash \lambda y^{\mathsf{int}}. 2 * y + 1 : \mathsf{int} \to \mathsf{int}$. Start with initial move:

$$(\vdash \lambda y^{\mathsf{int}}. 2 * y + 1 : \mathsf{int} \to \mathsf{int}) \xrightarrow{\mathsf{call}\,()} (\lambda y^{\mathsf{int}}. 2 * y + 1, S_0, R_0)$$

  where S_0 an empty store, and R_0 an empty **repository** (stores method names and their definitions)

- proceed to evaluation, producing a name for the function:

$$(\lambda y^{\mathsf{int}}. 2 * y + 1, S_0, R_0) \longrightarrow (m, S_0, \{m \mapsto \lambda y.2 * y + 1\})$$

- return the function name:

$$(m, S_0, \{m \mapsto \lambda y.2 * y + 1\}) \xrightarrow{\mathsf{ret}\,(m)} (\circ, S_0, \{m \mapsto \lambda y.2 * y + 1\})$$

How to obtain the games operationally

- For $\vdash \lambda y^{\mathsf{int}}.\, 2 * y + 1 : \mathsf{int} \to \mathsf{int}$. Start with initial move:

$$(\vdash \lambda y^{\mathsf{int}}.\, 2 * y + 1 : \mathsf{int} \to \mathsf{int}) \xrightarrow{\mathsf{call}\,()} (\lambda y^{\mathsf{int}}.\, 2 * y + 1, S_0, R_0)$$

  where S_0 an empty store, and R_0 an empty **repository** (stores method names and their definitions)

- proceed to evaluation, producing a name for the function:

$$(\lambda y^{\mathsf{int}}.\, 2 * y + 1, S_0, R_0) \longrightarrow (m, S_0, \{m \mapsto \lambda y.2 * y + 1\})$$

- return the function name:

$$(m, S_0, \{m \mapsto \lambda y.2 * y + 1\}) \xrightarrow{\mathsf{ret}\,(m)} (\circ, S_0, \{m \mapsto \lambda y.2 * y + 1\})$$

- now it is Opponent's turn:

$$(\circ, S_0, \{m \mapsto \lambda y.2 * y + 1\}) \xrightarrow{\mathsf{call}\, m(42)} (m\, 42, S_0, \{m \mapsto \lambda y.2 * y + 1\})$$

How to obtain the games operationally

- For $\vdash \lambda y^{\mathsf{int}}.\, 2 * y + 1 : \mathsf{int} \to \mathsf{int}$. Start with initial move:

$$(\vdash \lambda y^{\mathsf{int}}.\, 2 * y + 1 : \mathsf{int} \to \mathsf{int}) \xrightarrow{\mathsf{call}\,()} (\lambda y^{\mathsf{int}}.\, 2 * y + 1, S_0, R_0)$$

  where S_0 an empty store, and R_0 an empty **repository** (stores method names and their definitions)

- proceed to evaluation, producing a name for the function:

$$(\lambda y^{\mathsf{int}}.\, 2 * y + 1, S_0, R_0) \longrightarrow (m, S_0, \{m \mapsto \lambda y.2 * y + 1\})$$

- return the function name:

$$(m, S_0, \{m \mapsto \lambda y.2 * y + 1\}) \xrightarrow{\mathsf{ret}\,(m)} (\circ, S_0, \{m \mapsto \lambda y.2 * y + 1\})$$

- now it is Opponent's turn:

$$(\circ, S_0, \{m \mapsto \lambda y.2 * y + 1\}) \xrightarrow{\mathsf{call}\, m(42)} (m\, 42, S_0, \{m \mapsto \lambda y.2 * y + 1\})$$

- compute and return:

$$(m\, 42, S_0, \{m \mapsto \lambda y.2{*}y{+}1\}) \longrightarrow (2{*}42{+}1, \ldots) \longrightarrow (85, \ldots) \xrightarrow{\mathsf{ret}\, m(85)} (\circ, \ldots)$$

In short

$$(\vdash \lambda y^{\mathsf{int}}.\, 2 * y + 1 : \mathsf{int} \to \mathsf{int})$$

$$\xrightarrow{\mathsf{call}\,()} (\lambda y^{\mathsf{int}}.\, 2 * y + 1, S_0, R_0)$$

$$\longrightarrow (m, S_0, R_1) \qquad R_1 = \{m \mapsto \lambda y.2 * y + 1\}$$

$$\xrightarrow{\mathsf{ret}\,(m)} (\circ, S_0, R_1)$$

$$\xrightarrow{\mathsf{call}\,m(42)} (m\,42, S_0, R_1)$$

$$\longrightarrow (2 * 42 + 1, S_0, R_1) \longrightarrow (85, S_0, R_1) \xrightarrow{\mathsf{ret}\,m(85)} (\circ, S_0, R_1)$$

$$\xrightarrow{\mathsf{call}\,m(22)} (m\,22, S_0, R_1)$$

$$\longrightarrow (2 * 22 + 1, S_0, R_1) \longrightarrow (45, S_0, R_1) \xrightarrow{\mathsf{ret}\,m(45)} (\circ, S_0, R_1)$$

$\ldots$

Example II

$$(f : \mathsf{int} \to \mathsf{int} \vdash \lambda x^{\mathsf{int}}.\, f x + 1 : \mathsf{int} \to \mathsf{int})$$

$$\xrightarrow{\mathsf{call}\,(m)} (\lambda x^{\mathsf{int}}.\, m x + 1, S_0, R_0)$$

$$\longrightarrow (m', S_0, R_1) \qquad R_1 = \{m' \mapsto \lambda x^{\mathsf{int}}.\, m x + 1\}$$

$$\xrightarrow{\mathsf{ret}\,(m')} (\circ, S_0, R_1)$$

$$\xrightarrow{\mathsf{call}\,m'(42)} (m'\,42, S_0, R_1) \longrightarrow (m\,42 + 1, S_0, R_1)$$

$$\xrightarrow{\mathsf{call}\,m(42)} (\bullet + 1, S_0, R_1)$$

$$\xrightarrow{\mathsf{ret}\,m(85)} (85 + 1, S_0, R_1) \longrightarrow (86, S_0, R_1)$$

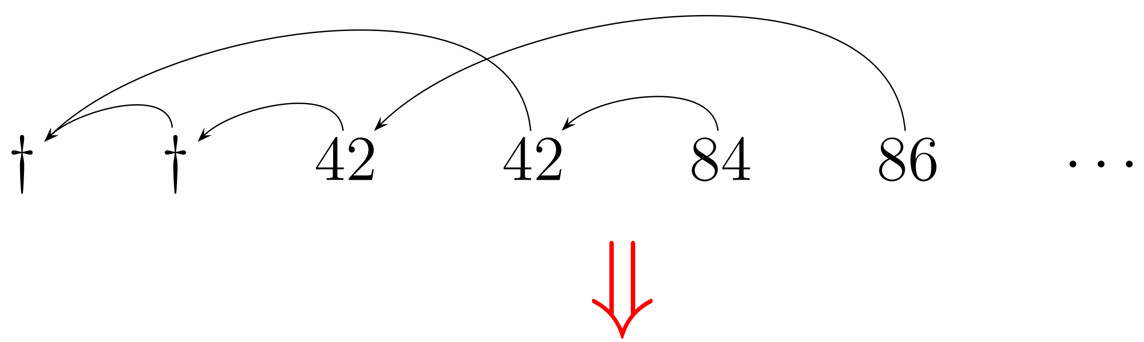
$$\xrightarrow{\mathsf{ret}\,m'(86)} (\circ, S_0, R_1)$$

$$\ldots$$

Plays vs traces

To distinguish them from plays we call these sequences of calls and returns **traces**.

There is a correspondence between plays and traces:



call (m) ret (m') call $m'(42)$ call $m(42)$ ret $m(85)$ ret $m'(86)$ $\cdots$

Traces more formally

Traces are based on **configurations**:

$$(M, \mathcal{E}, S, R, \mathcal{P}) \text{ or } (\circ, \mathcal{E}, S, R, \mathcal{P})$$

Traces more formally

Traces are based on **configurations**:

$$(M, \mathcal{E}, S, R, \mathcal{P}) \text{ or } (\circ, \mathcal{E}, S, R, \mathcal{P})$$

these are P and O configurations respectively
and:

- M is a term
- $\mathcal{E}$ is a stack of evaluation contexts and method names (we'll see why)
- S is a store
- R is a repository
- $\mathcal{P}$ is a map of *public* names: names of P that have been made public, or names revealed by O

Example formally

$$(\vdash \lambda y^{\mathsf{int}}.\, 2 * y + 1 : \mathsf{int} \to \mathsf{int})$$

$$\xrightarrow{\mathsf{call}\,()} (\lambda y^{\mathsf{int}}.\, 2 * y + 1, \epsilon, \emptyset, \emptyset, \emptyset)$$

$$\longrightarrow (m, \epsilon, \emptyset, R_1, \emptyset) \qquad R_1 = \{m \mapsto \lambda y.2 * y + 1\}$$

$$\xrightarrow{\mathsf{ret}\,(m)} (\circ, \epsilon, \emptyset, R_1, \mathcal{P}_1) \qquad \mathcal{P}_1 = \{m \mapsto P\}$$

$$\xrightarrow{\mathsf{call}\,m(42)} (m\,42, m, \emptyset, , R_1, \mathcal{P}_1)$$

$$\longrightarrow (2 * 42 + 1, \cdots) \longrightarrow (85, \cdots) \xrightarrow{\mathsf{ret}\,m(85)} (\circ, \epsilon, \emptyset, R_1, \mathcal{P}_1)$$

$$\xrightarrow{\mathsf{call}\,m(22)} (m\,22, m, \emptyset, R_1, \mathcal{P}_1)$$

$$\longrightarrow (2 * 22 + 1, \cdots) \longrightarrow (45, \cdots) \xrightarrow{\mathsf{ret}\,m(45)} (\circ, \epsilon, \emptyset, R_1, \mathcal{P}_1)$$

$$\ldots$$

Example II formally

$$(f : \mathsf{int} \to \mathsf{int} \vdash \lambda x^{\mathsf{int}}.\, f x + 1 : \mathsf{int} \to \mathsf{int})$$

$$\xrightarrow{\mathsf{call}\,(m)} (\lambda x^{\mathsf{int}}.\, m x + 1, \epsilon, \emptyset, \emptyset, \mathcal{P}_1) \qquad \mathcal{P}_1 = \{m \mapsto O\}$$

$$\longrightarrow (m', \epsilon, \emptyset, R_1, \mathcal{P}_1) \qquad R_1 = \{m' \mapsto \lambda x^{\mathsf{int}}.\, m x + 1\}$$

$$\xrightarrow{\mathsf{ret}\,(m')} (\circ, \epsilon, \emptyset, R_1, \mathcal{P}_2) \qquad \mathcal{P}_2 = \mathcal{P}_1[m' \mapsto P]$$

$$\xrightarrow{\mathsf{call}\, m'(42)} (m'\, 42, m', \emptyset, R_1, \mathcal{P}_2) \longrightarrow (m\, 42 + 1, m', \emptyset, R_1, \mathcal{P}_2)$$

$$\xrightarrow{\mathsf{call}\, m(42)} (\circ, (m, \bullet + 1) :: m', \emptyset, R_1, \mathcal{P}_2)$$

$$\xrightarrow{\mathsf{ret}\, m(85)} (85 + 1, m', \emptyset, R_1, \mathcal{P}_2) \longrightarrow (86, m', \emptyset, R_1, \mathcal{P}_2)$$

$$\xrightarrow{\mathsf{ret}\, m'(86)} (\circ, \epsilon, \emptyset, R_1, \mathcal{P}_2)$$

$$\ldots$$

Example III

The evaluation stack is needed in order to stack method calls:

$$(x : \mathsf{int} \vdash \lambda f^{\mathsf{int}\to\mathsf{int}}.\, f x + 1 : (\mathsf{int} \to \mathsf{int}) \to \mathsf{int})$$

$$\xrightarrow{\mathsf{call}\,(42)} (\lambda f^{\mathsf{int}\to\mathsf{int}}.\, f\, 42 + 1, \epsilon, \emptyset, \emptyset, \emptyset)$$

$$\longrightarrow (m, \epsilon, \emptyset, R_1, \emptyset) \qquad R_1 = \{m \mapsto \lambda f. f 42 + 1\}$$

$$\xrightarrow{\mathsf{ret}\,(m)} (\circ, \epsilon, \emptyset, R_1, \mathcal{P}_1) \qquad \mathcal{P}_1 = \{m \mapsto P\}$$

$$\xrightarrow{\mathsf{call}\, m(m_1)} (m\, m_1, m, \emptyset, R_1, \mathcal{P}_1)$$

$$\longrightarrow (m_1 42 + 1, m, \emptyset, R_1, \mathcal{P}_1)$$

$$\xrightarrow{\mathsf{call}\, m_1(42)} (\circ, (m_1, \bullet + 1) :: m, \emptyset, R_1, \mathcal{P}_1)$$

Example III

The evaluation stack is needed in order to stack method calls:

$$(x : \mathsf{int} \vdash \lambda f^{\mathsf{int}\to\mathsf{int}}.\, f x + 1 : (\mathsf{int} \to \mathsf{int}) \to \mathsf{int})$$

$$\xrightarrow{\mathsf{call}\,(42)} (\lambda f^{\mathsf{int}\to\mathsf{int}}.\, f\, 42 + 1, \epsilon, \emptyset, \emptyset, \emptyset)$$

$$\longrightarrow\xrightarrow{\mathsf{ret}\,(m)} (\circ, \epsilon, \emptyset, R_1, \mathcal{P}_1) \qquad R_1 = \{m \mapsto \lambda f. f 42 + 1\}$$

$$\xrightarrow{\mathsf{call}\, m(m_1)} (m\, m_1, m, \emptyset, R_1, \mathcal{P}_1)$$

$$\longrightarrow\xrightarrow{\mathsf{call}\, m_1(42)} (\circ, (m_1, \bullet + 1) :: m, \emptyset, R_1, \mathcal{P}_1)$$

Example III

The evaluation stack is needed in order to stack method calls:

$$(x : \mathsf{int} \vdash \lambda f^{\mathsf{int}\to\mathsf{int}}.\, f x + 1 : (\mathsf{int} \to \mathsf{int}) \to \mathsf{int})$$

$$\xrightarrow{\mathsf{call}\,(42)} (\lambda f^{\mathsf{int}\to\mathsf{int}}.\, f\, 42 + 1, \epsilon, \emptyset, \emptyset, \emptyset)$$

$$\longrightarrow \xrightarrow{\mathsf{ret}\,(m)} (\circ, \epsilon, \emptyset, R_1, \mathcal{P}_1) \qquad R_1 = \{m \mapsto \lambda f. f 42 + 1\}$$

$$\xrightarrow{\mathsf{call}\, m(m_1)} (m\, m_1, m, \emptyset, R_1, \mathcal{P}_1)$$

$$\longrightarrow \xrightarrow{\mathsf{call}\, m_1(42)} (\circ, (m_1, \bullet + 1) :: m, \emptyset, R_1, \mathcal{P}_1)$$

$$\xrightarrow{\mathsf{ret}\, m_1(85)} (85 + 1, m, \emptyset, R_1, \mathcal{P}_1) \longrightarrow \cdots$$

Example III

The evaluation stack is needed in order to stack method calls:

$$(x : \mathsf{int} \vdash \lambda f^{\mathsf{int}\to\mathsf{int}}.\, f x + 1 : (\mathsf{int} \to \mathsf{int}) \to \mathsf{int})$$

$$\xrightarrow{\mathsf{call}\,(42)} (\lambda f^{\mathsf{int}\to\mathsf{int}}.\, f\, 42 + 1, \epsilon, \emptyset, \emptyset, \emptyset)$$

$$\longrightarrow \xrightarrow{\mathsf{ret}\,(m)} (\circ, \epsilon, \emptyset, R_1, \mathcal{P}_1) \qquad R_1 = \{m \mapsto \lambda f.f 42 + 1\}$$

$$\xrightarrow{\mathsf{call}\, m(m_1)} (m\, m_1, m, \emptyset, R_1, \mathcal{P}_1)$$

$$\longrightarrow \xrightarrow{\mathsf{call}\, m_1(42)} (\circ, (m_1, \bullet + 1) :: m, \emptyset, R_1, \mathcal{P}_1)$$

$$\xrightarrow{\mathsf{call}\, m(m_2)} (m\, m_2, m :: (m_1, \bullet + 1) :: m, \emptyset, R_1, \mathcal{P}_1)$$

$$\longrightarrow (m_2\, 42 + 1, m :: (m_1, \bullet + 1) :: m, \emptyset, R_1, \mathcal{P}_1)$$

$$\xrightarrow{\mathsf{call}\, m_2(42)} (\circ, (m_2, \bullet + 1) :: m :: (m_1, \bullet + 1) :: m, \emptyset, R_1, \mathcal{P}_1)$$

$\ldots$

Formalising the trace rules

The language we examine is RefML.

First, we modify the operational semantics to include method names:

$$V \ ::= \ () \mid i \mid a \mid \langle V, V \rangle \mid m$$

and modify the operational semantics by adding repositories:

Formalising the trace rules

The language we examine is RefML.

First, we modify the operational semantics to include method names:

$$V \ ::= \ () \mid i \mid a \mid \langle V, V \rangle \mid m$$

and modify the operational semantics by adding repositories:

$$\begin{array}{rcll}
(i \oplus j, R, S) & \longrightarrow & (k, R, S) & (k = i \oplus j) \\
(\pi_i \langle V_1, V_2 \rangle, R, S) & \longrightarrow & (V_i, R, S) & \\
(\text{if } 0 \text{ then } M \text{ else } M', R, S) & \longrightarrow & (M', R, S) & \\
(\text{if } i \text{ then } M \text{ else } M', R, S) & \longrightarrow & (M, R, S) & (i > 0) \\
(\text{while}(M), R, S) & \longrightarrow & (\text{if } M \text{ then while}(M) \text{ else } (), R, S) & \\
(a = b, R, S) & \longrightarrow & (0, R, S) & (a \neq b) \\
(a = a, R, S) & \longrightarrow & (1, R, S) & \\
(!a, R, S) & \longrightarrow & (S(a), R, S) & \\
(a := V, R, S) & \longrightarrow & ((), R, S[a \mapsto V]) & \\
(\text{ref}(V), R, S) & \longrightarrow & (a', R, S[a' \mapsto V]) & (a' \notin \text{dom}(S))
\end{array}$$

Formalising the trace rules

The language we examine is RefML.

First, we modify the operational semantics to include method names:

$$V \ ::= \ () \mid i \mid a \mid \langle V, V \rangle \mid m$$

and modify the operational semantics by adding repositories:

$$((\lambda x.M)V, R, S) \quad \longrightarrow \quad (m, R[m \mapsto \lambda x.M], S) \qquad (a \notin \mathsf{dom}(R))$$

$$(m, R, S) \quad \longrightarrow \quad (M[V/x], R, S) \qquad (R(m) = \lambda x.M)$$

$$\frac{(M, R, S) \ \longrightarrow \ (M', R, S')}{(E[M], R, S) \ \longrightarrow \ (E[M'], R, S')}$$

Trace rules – internal

$$\frac{(M, R, S) \longrightarrow (M', R', S')}{(M, \mathcal{E}, R, S, \mathcal{P}) \longrightarrow (M', \mathcal{E}, R', S', \mathcal{P})}$$

so long all fresh names created are fresh for $\mathcal{E}, \mathcal{P}$ as well.

Trace rules – PQ

$$(E[mv], \mathcal{E}, R, S, \mathcal{P}) \xrightarrow{\text{call}\, m(v')^{S'}} (\circ, (m, E) :: \mathcal{E}, R \uplus R', S, \mathcal{P} \uplus \mathcal{P}')$$

where $\mathcal{P}(m) = O$ and:

- if v not a method name, then $v' = v$
- if $v = m$, then $v' = m'$ (fresh) and:
  - in R' we include $\{m' \mapsto \lambda x.mx\}$
  - in $\mathcal{P}'$ we include $\{m' \mapsto P\}$
- S' is S restricted to public reference names (i.e. those on P), with method names refreshed as above

Trace rules – OA

$$(\circ, (m, E) :: \mathcal{E}, R, S, \mathcal{P}) \xrightarrow{\mathsf{ret}\, m(v)^{S'}} (E[v], \mathcal{E}, R, S[S'], \mathcal{P} \uplus \mathcal{P}')$$

- if $v = m$ then it must be fresh and:
  - in $\mathcal{P}'$ we include $\{m' \mapsto O\}$
- S' can differ from S only for public reference names (i.e. those on P), with any method names being refresh as above

Trace rules – OQ

$$(\circ, \mathcal{E}, R, S, \mathcal{P}) \xrightarrow{\text{call}\, m(v)^{S'}} (mv, m :: \mathcal{E}, R, S[S'], \mathcal{P} \uplus \mathcal{P}')$$

where $\mathcal{P}(m) = P$ and:

- if $v = m$ then it must be fresh and:
  - in $\mathcal{P}'$ we include $\{m' \mapsto O\}$
- S' can differ from S only for public reference names (i.e. those on P), with any method names being refresh as above

Trace rules – PA

$$(v, m :: \mathcal{E}, R, S, \mathcal{P}) \xrightarrow{\mathsf{ret}\, m(v')^{S'}} (\circ, m :: \mathcal{E}, R \uplus R', S, \mathcal{P} \uplus \mathcal{P}')$$

- if v not a method name, then $v' = v$
- if $v = m$, then $v' = m'$ (fresh) and:
  - in R' we include $\{m' \mapsto \lambda x.mx\}$
  - in $\mathcal{P}'$ we include $\{m' \mapsto P\}$
- S' is S restricted to public reference names (i.e. those on P), with method names refreshed as above

Results

We define:

$$[\![\mathrm{U}, \Gamma \vdash M : \theta]\!]' = \{\ t \mid (\mathrm{U}, \Gamma \vdash M : \theta) \xrightarrow{t} (\circ, \mathcal{E}, R, S, \mathcal{P})\ \}$$

Results

We define:

$$[\![\mathrm{U}, \Gamma \vdash M : \theta]\!]' = \{\ t \mid (\mathrm{U}, \Gamma \vdash M : \theta) \xrightarrow{t} (\circ, \mathcal{E}, R, S, \mathcal{P})\ \}$$

What we can show:

- Correctness and adequacy (by construction)
- Soundness: if comp$([\![M]\!]')$ = comp$([\![N]\!]')$ then $M \cong N$
  - problem: the model is not compositional, so we cannot relate $[\![C[M]]\!]'$ with $[\![M]\!]'$ and $[\![C]\!]'$
  - we actually need to *prove* it is compositional (hard)
- Full abstraction (via definability and compositionality)

Further game models (operationally)

We can capture more paradigms:

- Objects: $\text{Cell} : \{\text{get} : \text{unit} \to \text{int}, \text{set} : \text{int} \to \text{unit}\}$

$$[\![x : \text{Cell} \vdash x.\text{set}(2 * x.\text{get}()) : \text{void}]\!] =$$
$$a \;\; \text{call}\, a.\text{get}() \;\; \text{ret}\, a.\text{get}(42) \;\; \text{call}\, a.\text{set}(84) \;\; \text{ret}\, a.\text{set}() \;\; \star$$

Further game models (operationally)

We can capture more paradigms:

- Objects: $\text{Cell} : \{\text{get} : \text{unit} \to \text{int}, \text{set} : \text{int} \to \text{unit}\}$

$$[\![x : \text{Cell} \vdash x.\text{set}(2 * x.\text{get}()) : \text{void}]\!] =$$
$$a \;\; \text{call}\, a.\text{get}() \;\; \text{ret}\, a.\text{get}(42) \;\; \text{call}\, a.\text{set}(84) \;\; \text{ret}\, a.\text{set}() \;\; \star$$

- Exceptions:

$$[\![\vdash \lambda x^{\text{int}}.\, \text{if } x \text{ then } 1/x \text{ else raise(DivZero)} : \text{int} \to \text{ratio}]\!] =$$
$$\text{call}\, () \;\; \text{ret}\, (m) \;\; \text{call}\, m(5) \;\; \text{ret}\, m(1/5) \;\; \text{call}\, m(0) \;\; \text{ret}\, m(e!)^{(e:\text{DivZero})} \ldots$$

Further game models (operationally)

We can capture more paradigms:

- Objects: $\mathsf{Cell} : \{\mathsf{get} : \mathsf{unit} \to \mathsf{int}, \mathsf{set} : \mathsf{int} \to \mathsf{unit}\}$

$$[\![x : \mathsf{Cell} \vdash x.\mathsf{set}(2 * x.\mathsf{get}()) : \mathsf{void}]\!] =$$

$$a \;\; \mathsf{call}\, a.\mathsf{get}() \;\; \mathsf{ret}\, a.\mathsf{get}(42) \;\; \mathsf{call}\, a.\mathsf{set}(84) \;\; \mathsf{ret}\, a.\mathsf{set}() \;\; \star$$

- Exceptions:

$$[\![\vdash \lambda x^{\mathsf{int}}.\, \mathsf{if}\ x\ \mathsf{then}\ 1/x\ \mathsf{else}\ \mathsf{raise}(\mathsf{DivZero}) : \mathsf{int} \to \mathsf{ratio}]\!] =$$

$$\mathsf{call}\,() \;\; \mathsf{ret}\,(m) \;\; \mathsf{call}\, m(5) \;\; \mathsf{ret}\, m(1/5) \;\; \mathsf{call}\, m(0) \;\; \mathsf{ret}\, m(e!)^{(e:\mathsf{DivZero})} \ldots$$

- Polymorphism:

$$[\![\vdash \Lambda X.\lambda x^{X}.\, x : \forall X.X \to X]\!] =$$

$$\mathsf{call}\,() \;\; \mathsf{ret}\,(m) \;\; \mathsf{call}\, m(\alpha) \;\; \mathsf{ret}\, m(m')^{(m':\alpha\to\alpha)} \;\; \mathsf{call}\, m'(p)^{\cdots,(p:\alpha)} \;\; \mathsf{ret}\, m'(p)^{\cdots}$$

Overview

We presented a general framework for constructing accurate models of programming languages:

- produces (in many cases the only) fully abstract models for a range of languages
- geared towards higher-order (realistic) languages: we saw up to RefML, but we can also model objects, exceptions, polymorphism, non-determinism, probability, etc.
- combines traditional denotational (based on strategy composition) with operational (based on traces) presentations