Programs and Strategies
Guy McCusker
Logic and Interaction 2012

Alternative title

The semantics game
Every program corresponds to a something

Game Semantics
Every program corresponds to a strategy

Definability
Every program corresponds to a strategy
... and every (finite) strategy comes from a program

Game Semantics
- We will see how game semantics gives models with definability for:
  - pure functional programs
  - imperative programs
  - programs with control operators
  - programs with higher-order store

Strategies and games
- Game semantics models a program as a strategy for a game.
- Games define what moves are available.
- Constraints on strategies limit their behaviour.

Constraints and effects

<table>
<thead>
<tr>
<th>Constraining strategies with</th>
<th>means you don't get</th>
</tr>
</thead>
<tbody>
<tr>
<td>Innocence</td>
<td>Store</td>
</tr>
<tr>
<td>Bracketing</td>
<td>Control</td>
</tr>
<tr>
<td>Visibility</td>
<td>Higher-order store</td>
</tr>
</tbody>
</table>
Definability

Every program corresponds to a strategy...
...and every (finite) strategy comes from a program...
...and behavioural properties of strategies classify programs

1. Pure functional programs

Typed lambda calculus

- Terms: $M ::= x | \lambda x.M | MM$
- Types: $A ::= \gamma | A \to A$
- Normal forms: $\lambda x_1x_2...x_n . x_i M_1M_2...M_k$

Normal forms as trees

- Normal forms as trees
- Normal forms as trees

A path in the tree

- Root node
- Choice of variable
- Choice of argument
- Choice of variable
- ...

The path as a picture

- The path as a picture
- The path as a picture

Games on arenas

- View the type as a tree: each $\to$ connects a node to its parent.
- Describe a term in normal form by playing a two-player game:
  - Opponent interrogates the term by choosing branches
  - Player represents the term by choosing head-variables
Games on arenas

- At Opponent's turn, he chooses a descendant of Player's last move.
- At Player's turn, he chooses a descendant of any previous O-move.
- The justification pointer tells us which one.

Strategies

- A strategy $\sigma$ is a set of even-length plays of the game:
  - non-empty and even-prefix-closed
  - deterministic: $sab, sac \in \sigma \Rightarrow b=c$.
- Set of plays $\approx$ tree $\equiv$ (partial, infinite) normal form.

Definability...

- A strategy $\sigma$ is total if it always has a response:
  $$s \in \sigma, sa \text{ legal } \Rightarrow \exists sab \in \sigma$$
- Finite, total strategies correspond to normal forms.

... but not yet a model

- We can interpret every normal form as a strategy.
- We do not yet have an interpretation of arbitrary $\lambda$-terms.
- We want one! And we want it to be compositional.

Problem: asymmetry

- These plays and strategies are asymmetric:
  - Opponent always has to move directly down the tree
  - Player can backtrack.
- Felscher (1985) referred to these as $E$-strategies.

Composition as interaction?

- The natural way to compose strategies would be by interaction.
- The asymmetry means our strategies cannot interact properly.
A failing interaction

\((\gamma \to \gamma) \to (\gamma \to \gamma)\)

\(\lambda f. \lambda z. f(f(z))\)

Supply \(\lambda f. \lambda z. f(f(z))\) as argument to \(\lambda F. \lambda x. F(\lambda y. y)(x)\)

A failing interaction

\((\gamma \to \gamma) \to (\gamma \to \gamma) \to (\gamma \to \gamma)\)

Supply \(\lambda f. \lambda z. f(f(z))\) as argument to \(\lambda F. \lambda x. F(\lambda y. y)(x)\)

A failing interaction

\((\gamma \to \gamma) \to (\gamma \to \gamma) \to (\gamma \to \gamma)\)

Supply \(\lambda f. \lambda z. f(f(z))\) as argument to \(\lambda F. \lambda x. F(\lambda y. y)(x)\)

Restoring symmetry: innocence

- The solution to this asymmetry was discovered by Hyland and Ong (Inf. Comp. 2000), and was also present in the work of Coquand (JSL 1995).
- The idea is:
  - let both players backtrack

Restoring symmetry: innocence

- The solution to this asymmetry was discovered by Hyland and Ong (Inf. Comp. 2000), and was also present in the work of Coquand (JSL 1995).
- The idea is:
  - let both players backtrack

Restoring symmetry: innocence

- The solution to this asymmetry was discovered by Hyland and Ong (Inf. Comp. 2000), and was also present in the work of Coquand (JSL 1995).
- The idea is:
  - let both players backtrack
  - recover definability by constraining the
Restoring symmetry:
innocence

- The solution to this asymmetry was discovered by Hyland and Ong (Inf. Comp. 2000), and was also present in Coquand (JSL 1995).
- The idea is:
  - let both players backtrack
  - recover definability by constraining the

Arenas and plays

- An arena is a forest (collection of trees) of moves, labelled as Opponent and Player moves.
- O / P alternate down the trees.
- A play is a sequence of moves-with-pointers.
- Pointer-chains are paths in the arena.

Views

- We want Player to behave as though O were not backtracking.
- At any point in the play, we can erase certain moves to give a P-view which disguises backtracking:

Innocent strategy

- An innocent strategy $\sigma$ on an arena is a set of even-length plays such that:
  - $\sigma$ is non-empty and closed under even-prefix
  - $\sigma$ is deterministic
  - if $sa \in \sigma$, $ta \in \sigma$, and $\text{view}(sa) = \text{view}(ta)$ then $ta \vDash \sigma$
Example: a non-innocent strategy

(((y→y) → (y→y)) → (y→y)) → (y→y)

Example: a non-innocent strategy

(((y→y) → (y→y)) → (y→y)) → (y→y)

Example: a non-innocent strategy

(((y→y) → (y→y)) → (y→y)) → (y→y)
Example: a non-innocent strategy

We have a model

We have a model

We have a model

We have a model

Soundness

Fact [cf. Hyland-Ong 2000]

If $M : A \rightarrow B$ and $N : A$ are normal forms, the strategy obtained by

- allowing the strategies for $M$ and $N$ to interact
- hiding the play in $A$

is the total, innocent strategy corresponding to the normal form of $MN$. 

\[ \lambda x.x \]
Soundness

- Proving soundness is hard work.
- We approach it by showing that innocent strategies have the structure of a Cartesian closed category.
- CCCs are just what is needed to make a sound model of the $\lambda$-calculus.

A category

- We can build a category of arenas and innocent strategies:
  - Objects are arenas
  - Morphisms are innocent strategies
  - Composition is interaction plus hiding
  - Identity is the *copycat strategy*

A category

- We can build a category of arenas and innocent strategies:
  - Objects are arenas
  - Morphisms are innocent strategies
  - Composition is interaction plus hiding
  - Identity is the *copycat strategy*

A category

- We can build a category of arenas and innocent strategies:
  - Objects are arenas
  - Morphisms are innocent strategies
  - Composition is interaction plus hiding
  - Identity is the *copycat strategy*

A category

- We can build a category of arenas and innocent strategies:
  - Objects are arenas
  - Morphisms are innocent strategies
  - Composition is interaction plus hiding
  - Identity is the *copycat strategy*

Copycat strategies

$\lambda f f. \lambda x. fx$

$((Y \rightarrow Y) \rightarrow (Y \rightarrow Y))$

Copycat strategies

$\lambda f f. \lambda x. fx$

$((Y \rightarrow Y) \rightarrow (Y \rightarrow Y))$

Copycat strategies

$\lambda f f. \lambda x. fx$

$((Y \rightarrow Y) \rightarrow (Y \rightarrow Y))$
Copycat strategies
\[ \lambda f. \lambda x. fx \]
\[ ((\gamma \rightarrow \gamma) \rightarrow (\gamma \rightarrow \gamma)) \]

Cartesian closure
Finally we show that the category has products and exponentials:

\[ A \times B \]
\[ A \Rightarrow B \]

After proving that these behave properly, we get soundness.

Full completeness

• Full completeness (definability) of the model is now easy:
Full completeness

- Full completeness (definability) of the model is now easy:
  - an innocent strategy is determined by the E-strategy it contains

Literature note 1: abstract machines

The soundness of interaction of strategies as a model of λ-application means that it can be used as a kind of abstract machine for computation. See e.g.

Danos, Herbelin and Regnier, Games Semantics and Abstract Machines, 1996.

Literature note 2: innocence semantically

- We have presented innocence as a syntactically-inspired condition.
- The work of Melliès on Asynchronous Games shows that it can be recovered from semantically-inspired considerations to do with permutability of moves.


PCF

- So far we have only considered logic rather than programming.
- Plotkin's language PCF is a prototypical functional programming language.
  - Typed λ-calculus with base types for numeric and boolean values.
  - Constants for arithmetic and boolean operations.
  - Recursion.

A Game for the Booleans

- We introduce an arena with data to model the Booleans:

- Note: this is the same as the arena for the type γ → γ → γ which encodes Booleans in the λ-calculus.

if-then-else

B → B → B → B

q tt tt ff q
Exercise

• In the \( \lambda \)-calculus, write down the term corresponding to if-then-else, when Booleans are encoded using \( Y \to Y \to Y \to Y \).
  • true is \( \lambda x.\lambda y.x \), false is \( \lambda x.\lambda y.y \).
  • if-then-else (over Booleans) is...

Recursion

• Strategies are sets of plays.
• Directed unions of innocent strategies are innocent strategies, and composition preserves them.
• We can therefore define least fixed points, which lets us interpret recursion in the usual way.
• Of course, we abandon totality.

Definability for PCF?

• The CCC of arenas and innocent strategies therefore contains a model of PCF.
• Does it have the definability property?
• Is there a class of "normal form" PCF terms that correspond to the finite innocent strategies?

Normal Forms

• What might a normal form look like?
• Something like this:
  \[ \lambda x_1 x_2 \ldots x_n. \text{if } x_i \text{ then } M_1 \ldots M_k \text{ else } N_1 \text{ else } N_2 \]
• Does every strategy behave like this?

Early exits

Consider a strategy which plays as shown below:

\[ (B \to B) \to B \]

\[
\begin{array}{c}
q \\
\uparrow \\
q \\
\downarrow \\
q \\
\end{array}
\]

This does not correspond to any PCF-definable term.

Bracketing condition

Our normal forms

\[ \lambda x_1 x_2 \ldots x_n. \text{if } x_i \text{ then } M_1 \ldots M_k \text{ then } N_1 \text{ else } N_2 \]

and in fact all PCF terms, satisfy a bracketing condition:

no question \( q \) is answered until all questions asked after \( q \) have been answered.

Bracketing condition

• Add a label to moves in arenas: every move is a question or an answer. In a play, an answer move answers the question it points to.
• A play \( s \) satisfies \( P \)-bracketing if and only if:
  for all prefixes \( t \) of \( s \), where \( a \) is a \( P \)-answer, \( a \) answers the last unanswered question in \( \text{view}(a) \).

Formalities

• An innocent strategy \( \sigma \) is well-bracketed if every play \( s \in \sigma \) is well-bracketed.
• Composition of well-bracketed strategies yields a well-bracketed strategy.
• The category of arenas and well-bracketed innocent strategies is a CCC; it contains our model of PCF.

Definability for PCF

Theorem [Hyland-Ong 2000]

Every finite, innocent, well-bracketed strategy corresponds to a term of PCF.

• "Finite" means "finite as an E-strategy", i.e. containing a finite number of distinct views.
• The proof is a direct extension of the one for \( \lambda \)-calculus.
Definability for PCF

Innocent, well-bracketed strategies

Add a control operator

• Add your favourite control operator to PCF. For example, add an empty type \( \bot \) and a constant

\[
\text{callcc} : (\lambda B. B \to \bot) \to B
\]

• How do we model this?

3. Control

Bracketing vs Control

• The bracketing condition restricts functions to a stack discipline for calls and returns.
• In programming terms, this means the absence of control operators.
• Can we make that correspondence precise?

callcc

Interpreting \( \bot \) as the one-move arena, we model callcc with the following strategy:

\[
(\lambda B. B \to \bot) \to B
\]
callcc

Interpreting \( \bot \) as the one-move arena, we model callcc with the following strategy:
\[
((B \rightarrow \bot) \rightarrow B) \rightarrow B
\]

Interpreting \( \bot \) as the one-move arena, we model callcc with the following strategy:
\[
((B \rightarrow \bot) \rightarrow B) \rightarrow B
\]

Interpreting \( \bot \) as the one-move arena, we model callcc with the following strategy:
\[
((B \rightarrow \bot) \rightarrow B) \rightarrow B
\]

Definability

This gives us a model of PCF + callcc, with a definability property.

Theorem [Laird 1997]
Every finite, innocent, (not necessarily well-bracketed) strategy corresponds to a term of PCF + callcc.

Definability for PCF + callcc

Games and linear continuations

- This result demonstrates that the games model is a continuations-based model, slightly disguised.
- Laird (2005) shows that the well-bracketed strategies are exactly those obeying a certain linear continuation passing regime:
  - answers are continuations that can be invoked at most once.

4. Beyond innocence: state
Non-innocent strategies

- Strategies without the innocence constraint form another CCC.
- Our analysis so far has exploited a tight correspondence between views and paths in syntax trees.
- If we abandon innocence, what do we get?

Non-innocence

\[((Y \rightarrow Y) \rightarrow (Y \rightarrow Y)) \rightarrow (Y \rightarrow Y)\]

\[\lambda F. \lambda x. \]

\[\lambda y. \]

\[y \]

\[\lambda y. \]

\[F \]

Changing minds

- How can a program change its responses like this?
- Using state!
Changing minds

• How can a program change its responses like this?
• Using state!

\[ \lambda F. \lambda x. \text{new } v := \text{true in } \]
\[ F(\lambda y. \text{if } !v \text{ then } v := \text{false}; \text{return } y \]
\[ \text{else } F(\ldots) ) \]

Stateful strategies

• Non-innocent strategies directly represent stateful computation.
• The state itself is implicit: what we see in the strategy is the behaviour implemented by using the state.
• Contrast with explicit-state models, e.g. those based on a state monad.

Stateful strategies

• Non-innocent strategies directly represent stateful computation.
• The state itself is implicit: what we see in the strategy is the behaviour implemented by using the state.
• Contrast with explicit-state models, e.g. those based on a state monad.

this idea was first explored by Uday Reddy (1993)

Idealised Algol

• Reynolds's Idealised Algol is a prototypical higher-order imperative programming language

\[ \text{IA} = \text{PCF + assignable variables + block structure} \]

Idealised Algol

• Reynolds's Idealised Algol is a prototypical higher-order imperative programming language

\[ \text{IA} = \text{simple while programs + block structure + } \lambda \text{-calculus} \]

Idealised Algol

• Types: \( A ::= \text{comm | exp | var | } A \rightarrow A \)
• Terms: \( M ::= \text{PCF | x:= M | !x | M ; M | new x in M} \)

Commands and variables

• To interpret commands and variables, we can no longer rely on the analogy with normal-form trees.
• We have to do some semantics!
• Consider the observable actions available for each type, and build an appropriate arena.

Commands

• With no explicit store, what can we observe of a command!
Commands

- With no explicit store, what can we observe of a command?
  - We can try to run it: “run”
  - We will notice when it terminates: “done”.
  - That’s all.

Sequential composition

\[
\text{comm} \rightarrow \text{comm} \rightarrow \text{comm} \\
\text{r} \rightarrow \text{d} \rightarrow \text{r} \\
\text{d} \rightarrow \text{r} \\
\text{d} \rightarrow \text{d}
\]

Variables

- Variables are more complex:
  - We can try to read a value, and get something back
  - We can try to store a value; this is like a command, so we just observe termination.

Assignment

\[
\text{var} \rightarrow \text{exp} \rightarrow \text{comm} \\
\text{q} \rightarrow \text{r} \\
\text{a} \rightarrow \text{r} \\
\text{write(a)} \rightarrow \text{d} \\
\text{ok} \rightarrow \text{d}
\]
Example
\[ x: \text{var}, \quad y: \text{var} \quad \vdash \quad x := \text{not} (\neg y) : \text{comm} \]

Example
\[ x: \text{var}, \quad y: \text{var} \quad \vdash \quad x := \text{not} (\neg y) : \text{comm} \]

Example
\[ x: \text{var}, \quad y: \text{var} \quad \vdash \quad x := \text{not} (\neg y) : \text{comm} \]

Example
\[ x: \text{var}, \quad y: \text{var} \quad \vdash \quad x := \text{not} (\neg y) : \text{comm} \]

Read
\[ b \]

Write
\[ \text{not } b \]

Ok

Example
\[ x: \text{var}, \quad y: \text{var} \quad \vdash \quad x := \text{not} (\neg y) : \text{comm} \]

Another example
\[ c: \text{comm}, \quad x: \text{var} \quad \vdash \quad x := \text{true}; \quad x := \text{not} (\neg x) : \text{comm} \]

Another example
\[ c: \text{comm}, \quad x: \text{var} \quad \vdash \quad x := \text{true}; \quad x := \text{not} (\neg x) : \text{comm} \]
Another example

c:comm, x: var ⊢ x := true; c; x := not(!x) : comm

doesn’t have to be “true”

doesn’t have to be “true”

write(true)

write(true)

write(true)

read

read

read

r

r

r

ok

ok

ok

d

d

r

r

r

d

d

b

b

b

Another example

c:comm, x: var ⊢ x := true; c; x := not(!x) : comm

doesn’t have to be “true”

doesn’t have to be “true”

write(true)

write(true)

write(true)

read

read

read

r

r

r

d

d

b

b

b

Another example

c:comm, x: var ⊢ x := true; c; x := not(!x) : comm

doesn’t have to be “true”

doesn’t have to be “true”

write(true)

write(true)

write(true)

read

read

read

r

r

r

d

d

b

b

b

Another example

c:comm, x: var ⊢ x := true; c; x := not(!x) : comm

doesn’t have to be “true”

doesn’t have to be “true”

write(true)

write(true)

write(true)

read

read

read

r

r

r

d

d

b

b

b

Another example

c:comm, x: var ⊢ x := true; c; x := not(!x) : comm

doesn’t have to be “true”

doesn’t have to be “true”

write(true)

write(true)

write(true)

read

read

read

r

r

r

d

d

b

b

b

Another example

c:comm, x: var ⊢ x := true; c; x := not(!x) : comm

doesn’t have to be “true”

doesn’t have to be “true”

write(true)

write(true)

write(true)

read

read

read

r

r

r

d

d

b

b

b

Another example

c:comm, x: var ⊢ x := true; c; x := not(!x) : comm

doesn’t have to be “true”

doesn’t have to be “true”

write(true)

write(true)

write(true)

read

read

read

r

r

r

d

d

b

b

b

Another example

c:comm, x: var ⊢ x := true; c; x := not(!x) : comm

doesn’t have to be “true”

doesn’t have to be “true”

write(true)

write(true)

write(true)

read

read

read

r

r

r

d

d

b

b

b

Another example

c:comm, x: var ⊢ x := true; c; x := not(!x) : comm

doesn’t have to be “true”

doesn’t have to be “true”

write(true)

write(true)

write(true)

read

read

read

r

r

r

d

d

b

b

b

Another example

c:comm, x: var ⊢ x := true; c; x := not(!x) : comm

doesn’t have to be “true”

doesn’t have to be “true”

write(true)

write(true)

write(true)

read

read

read

r

r

r

d

d

b

b

b

Another example

c:comm, x: var ⊢ x := true; c; x := not(!x) : comm

doesn’t have to be “true”

doesn’t have to be “true”

write(true)

write(true)

write(true)

read

read

read

r

r

r

d

d

b

b

b

Another example

c:comm, x: var ⊢ x := true; c; x := not(!x) : comm

doesn’t have to be “true”

doesn’t have to be “true”

write(true)

write(true)

write(true)

read

read

read

r

r

r

d

d

b

b

b
Another example

c: comm, x: var \vdash x:=true; c; x := not(!x) : comm

read(true)  
ok

write(true)  
ok

doesn't have to be  
"true"

write(not b)  
ok

d

Bad variable behaviour

• Why don’t we try to restrict plays so that the b in the previous example is always “true”?

The command c could later become bound to something that alters x, so it is vital to allow the read to return any value.

Local variable behaviour

• If we wrap the term in a variable allocation
  new x in x:=true; c; x := not(!x)
  it is no longer possible for c to alter x.

• The variable x becomes a good variable.

• It also becomes hidden: the environment should not know it is there.

Variable allocation

• The command new x in M is just like M, but:
  x is bound to a storage cell, so M’s interactions with x should be variable-like.
  the outside world should no longer see x.

• How can we model this? Interaction plus hiding!

A cell strategy

\[
(var \rightarrow \text{comm}) \rightarrow \text{comm}
\]

A cell strategy

\[
(var \rightarrow \text{comm}) \rightarrow \text{comm}
\]

Allocation in action

\[
(var \rightarrow \text{comm}) \rightarrow \text{comm}
\]

any "good" sequence of reads and writes
Allocation in action

\[ \text{comm} \rightarrow (\text{var} \rightarrow \text{comm}) \rightarrow \text{comm} \]

write(true) \[ r \]
ok \[ r \]
read \[ r \]
true \[ d \]
write(false) \[ r \]
ok \[ d \]

A remark

- In this model, only variable allocation requires a non-innocent strategy: terms without `new x in` ... are interpreted innocently.
- This is a marked difference from explicit state models, where assignment and lookup operations access the state.

Definability?

- We have a model of Idealised Algol, and we can prove that it is sound.
- Does it have the definability property?

Visibility

- Our non-innocent strategies are very powerful. E.g.:
\[ (A \rightarrow (A \rightarrow \text{comm}) \rightarrow \text{comm}) \rightarrow \text{comm} \]
• Our non-innocent strategies are very powerful. E.g.:

$\text{(A} \rightarrow (\text{A} \rightarrow \text{comm}) \rightarrow \text{comm}) \rightarrow \text{comm}$

• No Idealised Algol program has this behaviour.

• To eliminate this strategy, we impose a new constraint: visibility.

A play $s$ satisfies P-visibility if, for every prefix $tm$ where $m$ is a P-move, the justifier of $m$ is in view($t$).

(All plays in innocent strategies satisfy this automatically.)
Eliminating such plays lets us recover definability. But how to prove it?

**Factorization**

**Theorem** [Abramsky + M 1996]

Every well-bracketed, P-visible strategy on $A \rightarrow \text{comm}$ is of the form $\sigma ; \text{cell}$ for some innocent $\sigma : A \rightarrow (\text{var} \rightarrow \text{comm})$.

**Proof:** Innocently simulate the non-innocent strategy by storing the history in the cell.

**Another definability result**

**Theorem** [Abramsky + M 1996]

Every finite, well-bracketed, P-visible strategy on $A \rightarrow \text{comm}$ is definable by a term of Idealised Algol.

**Definability for Idealised Algol**

- If $M$ is a term defining $\sigma$, then new $x$ in $M$ defines $\sigma ; \text{cell}$.
Another definability result?

Probably a theorem that I don’t think anyone has actually bothered to prove.

Every finite (not necessarily well-bracketed) $P$-visible strategy on $A \rightarrow \text{comm}$ is definable by a term of Idealised Algol + callcc.

5. Beyond visibility: more state

What can general strategies express?

- I am afraid I can no longer pretend to make the progression seem logical; I’ll just have to tell you.
- Strategies that may break visibility correspond exactly to programs with higher order store.

Ground vs higher order store

- Because ground-type data can be completely evaluated, assignment was easy to interpret.

```
var \rightarrow \exp \rightarrow \text{comm}
```

```
q \quad r
a
write(a)
ok
d
```

- How could we do this for a general type $A$?

```
var[A] \rightarrow A \rightarrow \text{comm}
```

```
q \quad r
q'
write(a)
ok
d
```

Ground vs higher order store

- How can we store a program of type $A$ in a variable?

```
exp \times (\exp \rightarrow \text{comm})
```

- We can hardly have a move

```
write(\sigma)
```

where $\sigma$ is a strategy.

What is var[A] anyway?

- An alternate var type, for ground data, is given by

```
\exp \times (\exp \rightarrow \text{comm})
```

- Think of this as the product of

  - the “read method”, returning an exp
  - the “write method”, taking an exp and returning the assignment command

```
q \quad r
q'
\text{take two}
```

var take two
Definability again

- This gives another way to interpret Idealised Algol, and we can obtain another definability result.
- Perhaps this version of `var` will generalise to arbitrary types? Define
  
  \[ \text{var}[A] = A \times (A \to \text{comm}). \]

Assignment and lookup

- The type for the assignment constant is now
  
  \[ (A \times (A \to \text{comm})) \to A \to \text{comm} \]
- Now it's easy to interpret, using projection.
- Lookup is similarly simple: just use the other projection
  
  \[ (A \times (A \to \text{comm})) \to A \]

A cell strategy?

- How can we implement a cell strategy of type
  
  \[ (\text{var}[A] \to \text{comm}) \to \text{comm} \]

\[ r \]

\[ \cdots \]

\[ s \]

\[ \cdots \]

\[ d \]

\[ d \]

Cells

\[ (A \times (A \to \text{comm}) \to \text{comm}) \to \text{comm} \]

Cells

\[ (A \times (A \to \text{comm}) \to \text{comm}) \to \text{comm} \]

Cells

\[ (A \times (A \to \text{comm}) \to \text{comm}) \to \text{comm} \]
Factorization again

**Theorem (approx)**

[ Abramsky, Honda, M 1998]

Every finite well-bracketed strategy on a type $A \rightarrow \text{comm}$ is of the form $\sigma \cdot \text{cell}^n$ for some $P$-visible $\sigma : \text{var} \rightarrow \text{comm}$. 

\[ \sigma : A \rightarrow (\text{var} \rightarrow \text{comm}) \]

\[ \sigma ; \text{cell}^n \]

Proof: turn all the violations of $P$-visibility into violations of $O$-visibility performed by the cell.
Factorization again

**Theorem (approx)**
[Abramsky, Honda, M 1998]
Every finite well-bracketed strategy on a type \( A \rightarrow \text{comm} \) is of the form \( \sigma ; \text{cell}^n \) for some \( P \)-visible \( \sigma : A \rightarrow (\text{var[comm]}^n \rightarrow \text{comm}) \).

This means that arbitrary well-bracketed strategies can be expressed as a composition of cells storing \text{comm} and boolean types.

Definability again

- Our cell strategies allow us to interpret a language where terms of any type can be stored in the variables.
- The factorization result means that we have a definability result once more: every finite strategy is the denotation of some term in this language.

Literature note 3:

- Laird has shown how the ideas at play here can be expressed algebraically, in terms of his sequoidal categories.

Definability for Higher-Order Store

- Programs with ho-store
- Well-bracketed strategies

Closing remarks

Conditions classify programs

- \( \lambda \)-calculus \( \longrightarrow \) finite, total, innocent
- PCF \( \longrightarrow \) innocent, \( P \)-bracketed
- PCF+callcc \( \longrightarrow \) innocent
- Idealised Algol \( \longrightarrow \) \( P \)-bracketed, \( P \)-visible
- Higher-type Store \( \longrightarrow \) \( P \)-bracketed

**O is unconstrained**

- Our behavioural constraints have all been expressed as conditions on strategies.
- \( O \) is free to behave as he wishes.
- This means it makes sense to allow, e.g., a PCF term to interact in an IA context.

Full abstraction

- In each case, we can lift our full completeness (definability) result to a full abstraction result by means of a quotient:
  \[ \sigma \equiv \tau : A \text{ if and only if for all } \alpha : A \rightarrow \text{comm}, \sigma ; \alpha \equiv \tau ; \alpha. \]
- In the case of the imperative languages, this quotient can be defined directly.