Game Semantics, Open Systems and Components

Samson Abramsky

The Game Plan

- Overview of work in game semantics, including some recent developments. Applied and algorithmic aspects.
- Some tentative thoughts about connections to specification and refinement of components.

Generalities

Game semantics is centrally concerned with the **compositional modelling of interactive systems**.

Two faces of game semantics:

- Games as a highly structured mathematical universe, an intensional analogue of **domains** as a setting for denotational semantics.
- Games as **concrete objects** (representable as graphs, automata etc.), suitable for algorithmic manipulation.

Basic Ideas

- Types of a programming language are interpreted as 2-person games: the Player is the System (program fragment) currently under consideration, while the Opponent is the Environment or context.
- Programs are **strategies** for these games.

So game semantics is inherently a semantics of **open systems**; the meaning of a program is given by its potential interactions with its environment.

- Compositionality. The key operation is plugging two strategies together, so that each **actualizes** part of the environment of the other. (Usual game idea corresponds to a **closed** system, with no residual environment). This exploits the game-theoretic P/O duality.

Types as Games

- A simple example of a basic datatype of natural numbers:

  $$\mathbf{nat} = \{q \cdot n \mid n \in \mathbb{N}\}$$

  Note a further classification of moves, orthgonal to the P/O duality; q is a **question**, n are **answers**. This turns out to be important for capturing **control features** of programming languages.

- Forming function or procedure types $A \Rightarrow B$. We form a new game from disjoint copies of A and B, **with P/O roles in** A **reversed**. Thus we think of $A \Rightarrow B$ as a **structured interface** to the Environment; in B, we interact with the caller of the procedure, **covariantly**, while in A, we interact with the argument supplied to the procedure call, **contravariantly**.

Example

Strategy for $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2$.

$$(\quad \mathbf{nat} \quad \Rightarrow \quad \mathbf{nat} \quad) \quad \Rightarrow \quad \mathbf{nat} \quad \Rightarrow \quad \mathbf{nat}$$

Example

Strategy for $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2$.

$$(\quad \mathbf{nat} \quad \Rightarrow \quad \mathbf{nat} \quad) \quad \Rightarrow \quad \mathbf{nat} \quad \Rightarrow \quad \mathbf{nat}$$

O $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ q

Example

Strategy for $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2.$

$$\begin{array}{cccccccccc}
 & (& \mathbf{nat} & \Rightarrow & \mathbf{nat} &) & \Rightarrow & \mathbf{nat} & \Rightarrow & \mathbf{nat} \\
O & & & & & & & & & \mathrm{q} \\
P & & & & \mathrm{q} & & & & &
\end{array}$$

Example

Strategy for $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2$.

$$\begin{array}{cccccccccc}
 & (& \mathbf{nat} & \Rightarrow & \mathbf{nat} &) & \Rightarrow & \mathbf{nat} & \Rightarrow & \mathbf{nat} \\
O & & & & & & & & & \mathrm{q} \\
P & & & & \mathrm{q} & & & & & \\
O & & \mathrm{q} & & & & & & &
\end{array}$$

Example

Strategy for $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2$.

	(	$\mathbf{nat}$	$\Rightarrow$	$\mathbf{nat}$	)	$\Rightarrow$	$\mathbf{nat}$	$\Rightarrow$	$\mathbf{nat}$
O									q
P				q					
O		q							
P							q		

Example

Strategy for $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2$.

$$
\begin{array}{lccccccccc}
 & (& \mathbf{nat} & \Rightarrow & \mathbf{nat} &) & \Rightarrow & \mathbf{nat} & \Rightarrow & \mathbf{nat} \\
O & & & & & & & & & \mathrm{q} \\
P & & & & \mathrm{q} & & & & & \\
O & & \mathrm{q} & & & & & & & \\
P & & & & & & & \mathrm{q} & & \\
O & & & & & & & \mathrm{n} & &
\end{array}
$$

Example

Strategy for $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2$.

$$
\begin{array}{cccccccccc}
 & (& \mathbf{nat} & \Rightarrow & \mathbf{nat} &) & \Rightarrow & \mathbf{nat} & \Rightarrow & \mathbf{nat} \\
O & & & & & & & & & \mathrm{q} \\
P & & & & \mathrm{q} & & & & & \\
O & & \mathrm{q} & & & & & & & \\
P & & & & & & & \mathrm{q} & & \\
O & & & & & & & \mathrm{n} & & \\
P & & \mathrm{n} & & & & & & &
\end{array}
$$

Example

Strategy for $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2$.

	($\mathbf{nat}$	$\Rightarrow$	$\mathbf{nat}$)	$\Rightarrow$	$\mathbf{nat}$	$\Rightarrow$	$\mathbf{nat}$
O							q
P			q				
O	q						
P					q		
O					n		
P	n						
O			m				

Example

Strategy for $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2$.

	(	**nat**	$\Rightarrow$	**nat**	)	$\Rightarrow$	**nat**	$\Rightarrow$	**nat**
O									q
P				q					
O		q							
P							q		
O							n		
P		n							
O				m					
P									m $+ 2$

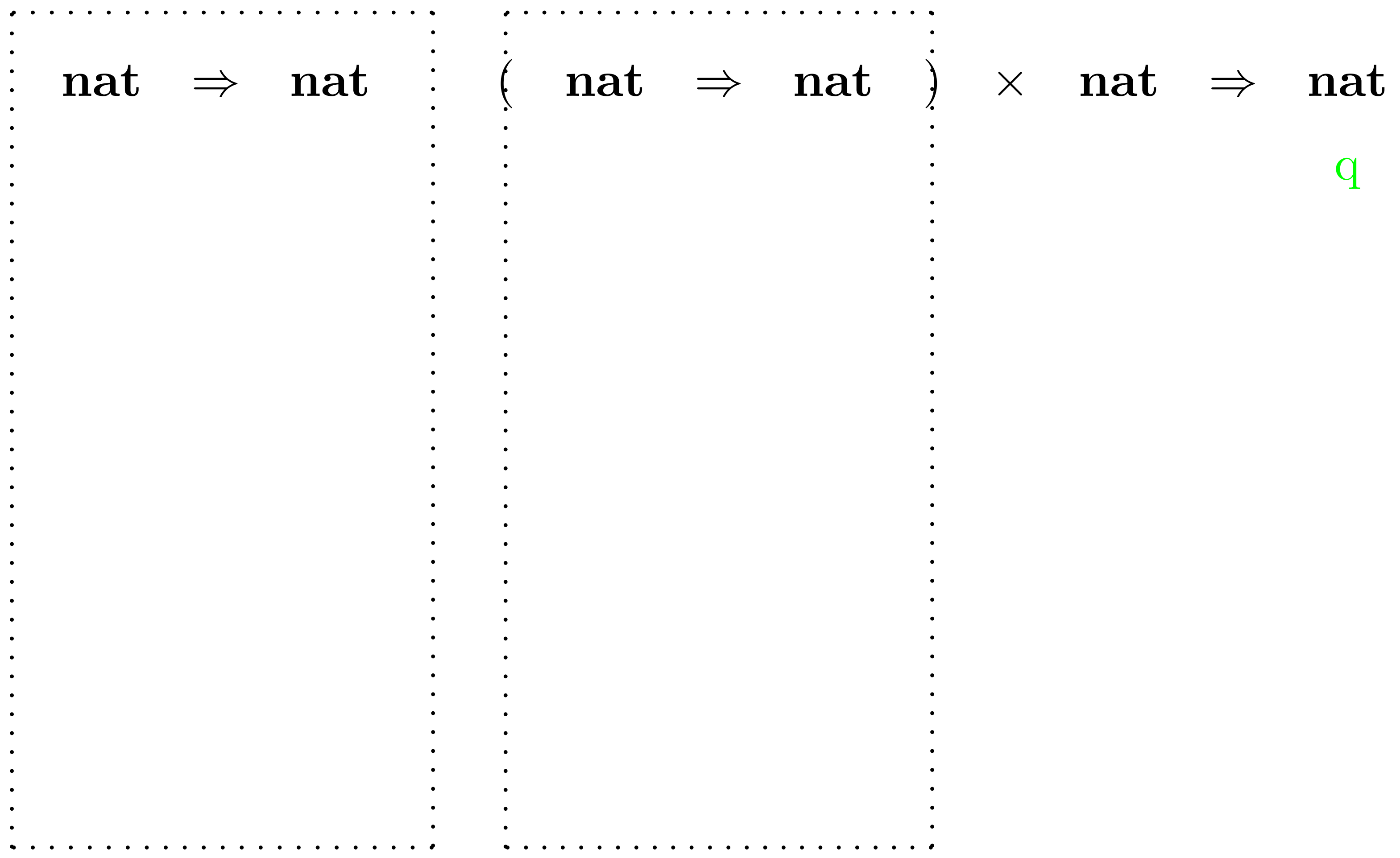
Composition

Apply $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2$ to $\lambda x : \mathbf{nat}.\, x^2$.

$$\boxed{\mathbf{nat} \Rightarrow \mathbf{nat}} \quad \boxed{(\ \mathbf{nat} \Rightarrow \mathbf{nat}\ )} \times \mathbf{nat} \Rightarrow \mathbf{nat}$$

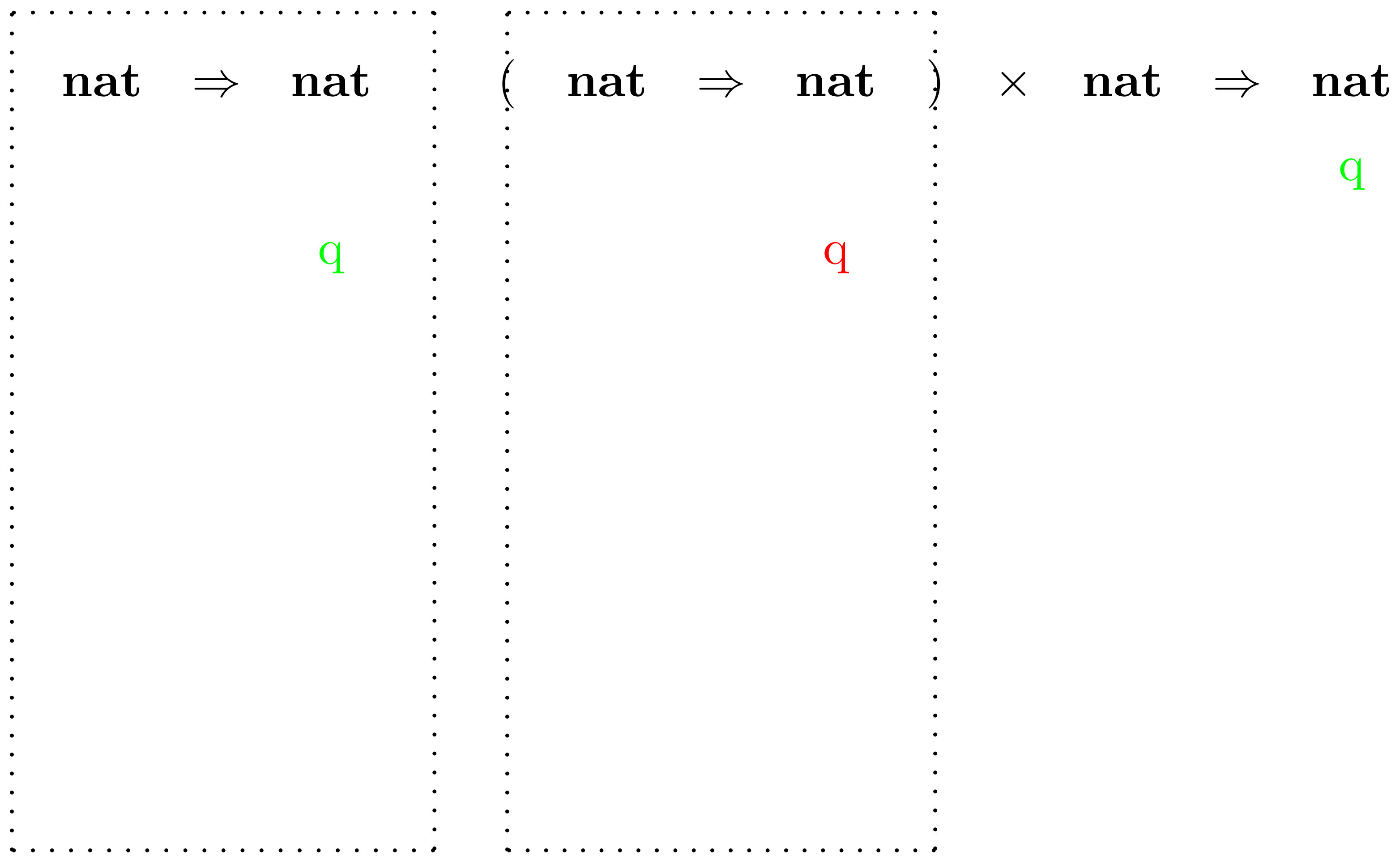
Composition

Apply $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2$ to $\lambda x : \mathbf{nat}.\, x^2$.


$\mathbf{nat} \Rightarrow \mathbf{nat}$
$(\mathbf{nat} \Rightarrow \mathbf{nat}) \times \mathbf{nat} \Rightarrow \mathbf{nat}$
q

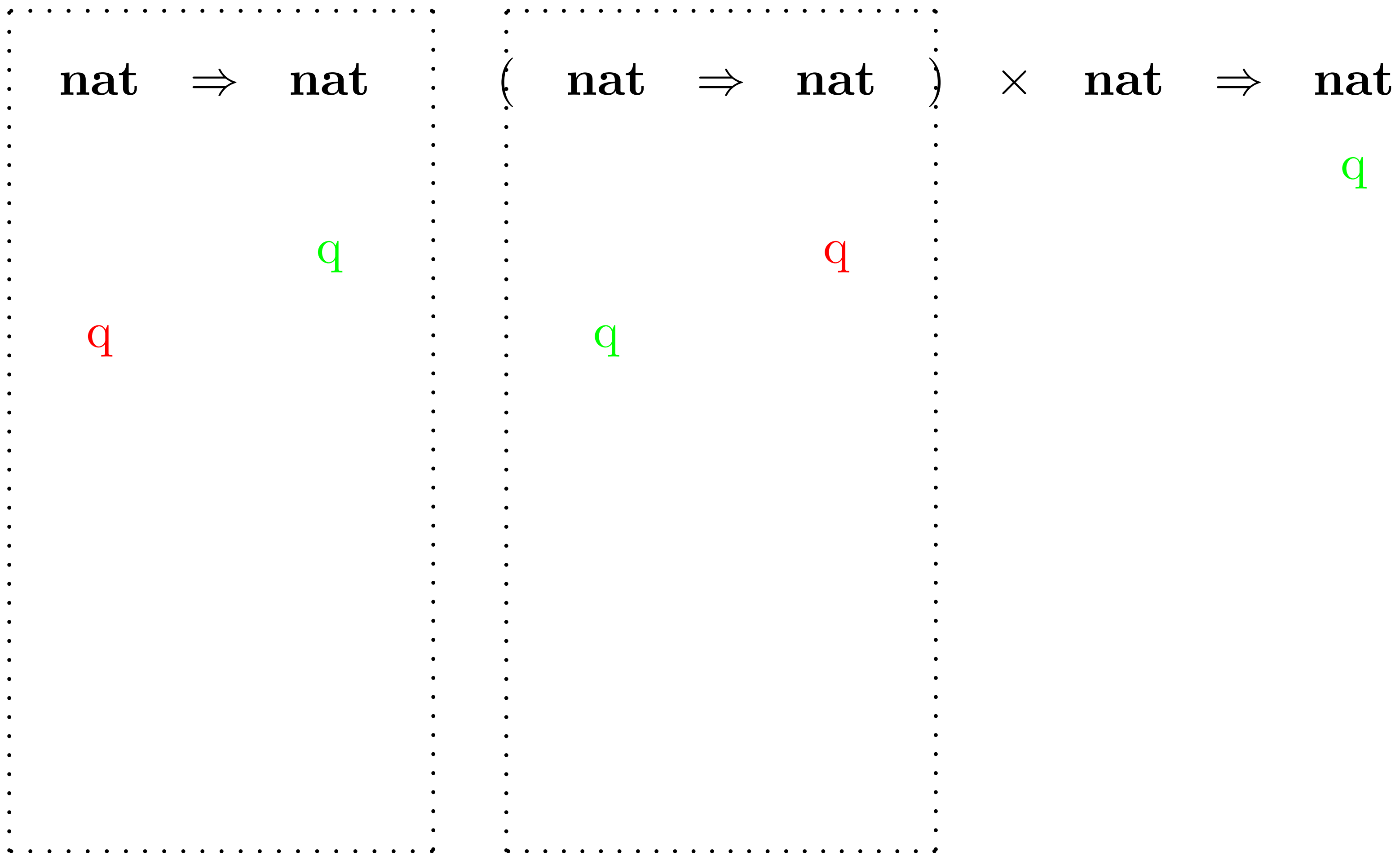
Composition

Apply $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2$ to $\lambda x : \mathbf{nat}.\, x^2$.



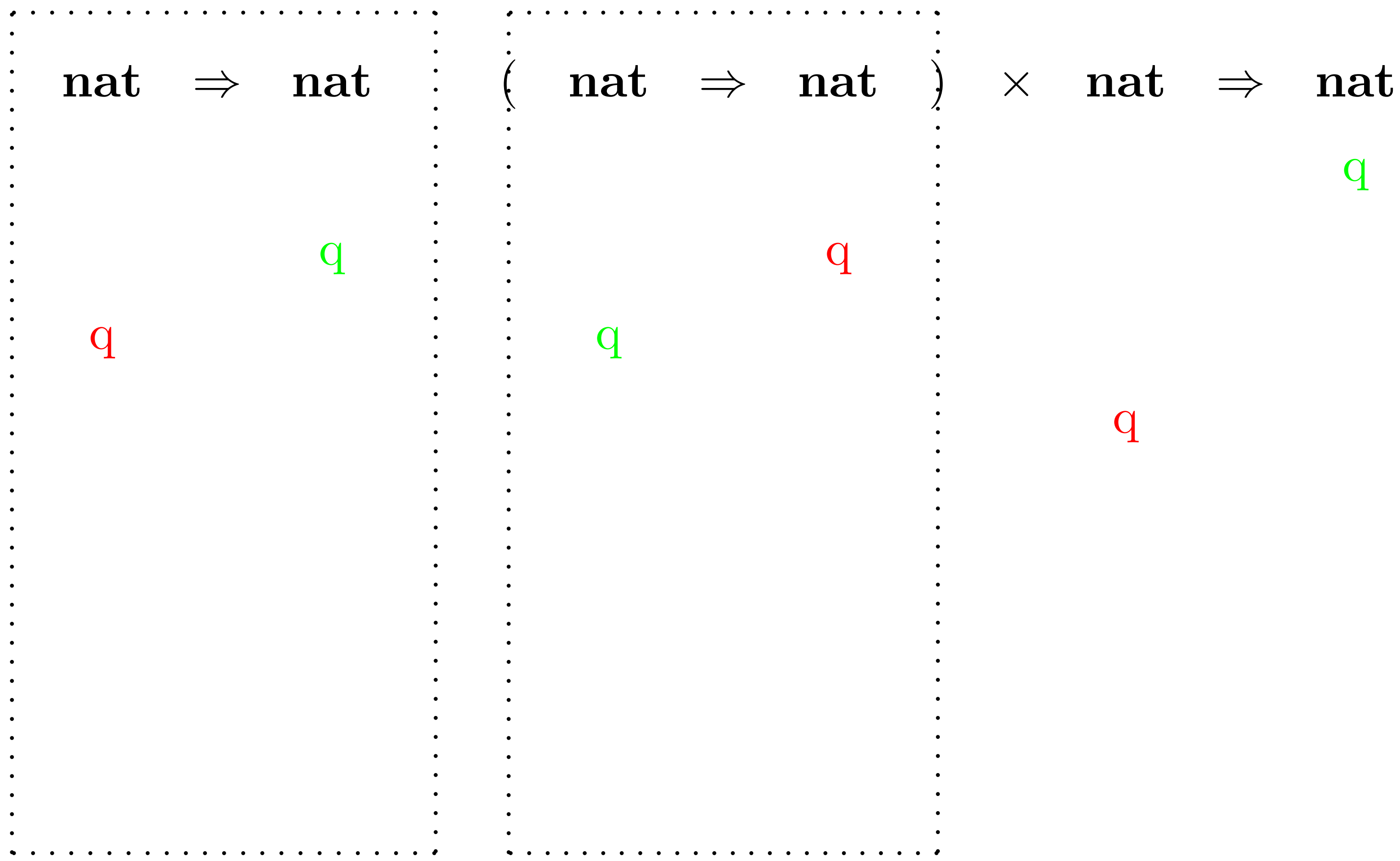
Composition

Apply $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2$ to $\lambda x : \mathbf{nat}.\, x^2$.



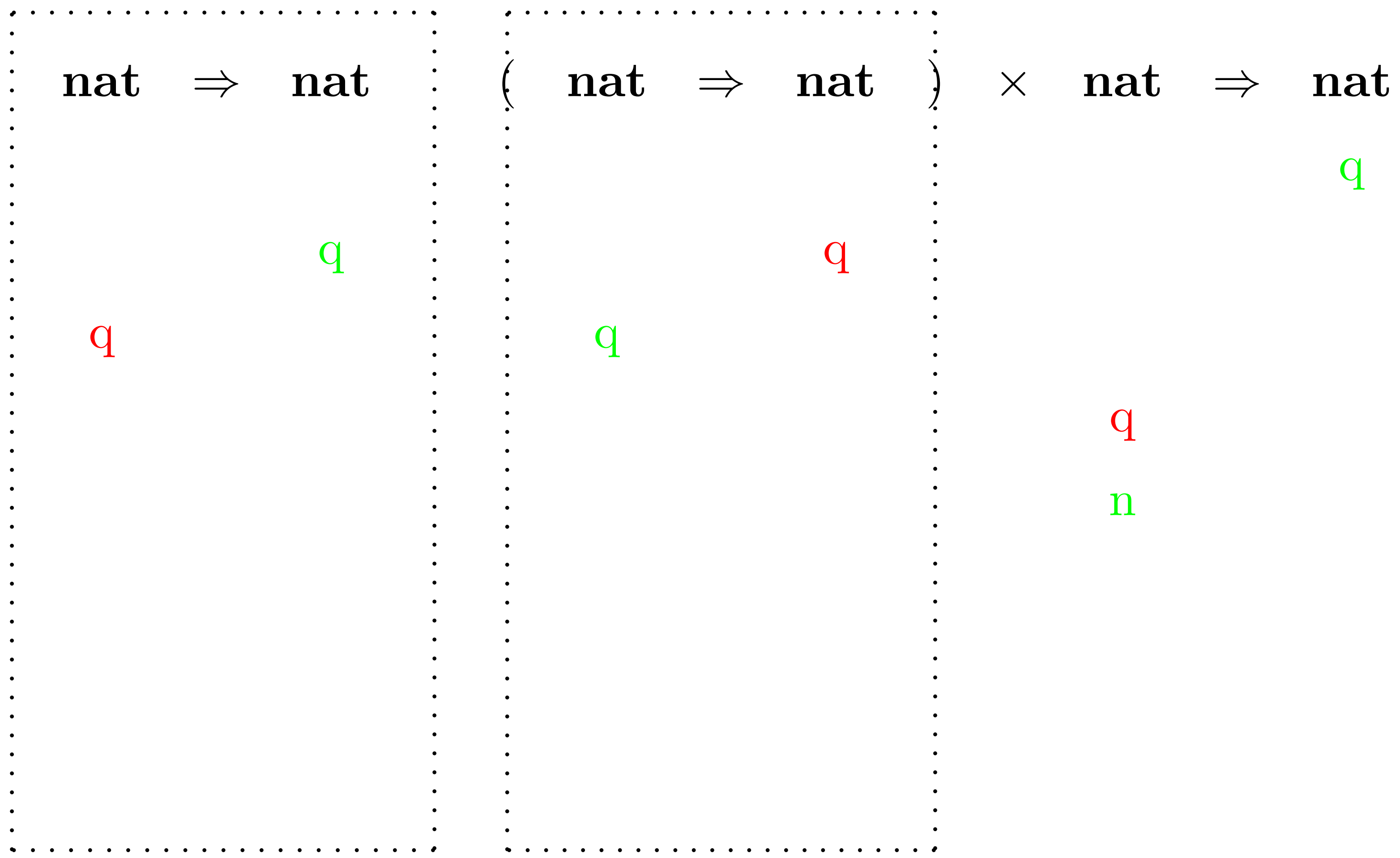
Composition

Apply $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2$ to $\lambda x : \mathbf{nat}.\, x^2$.



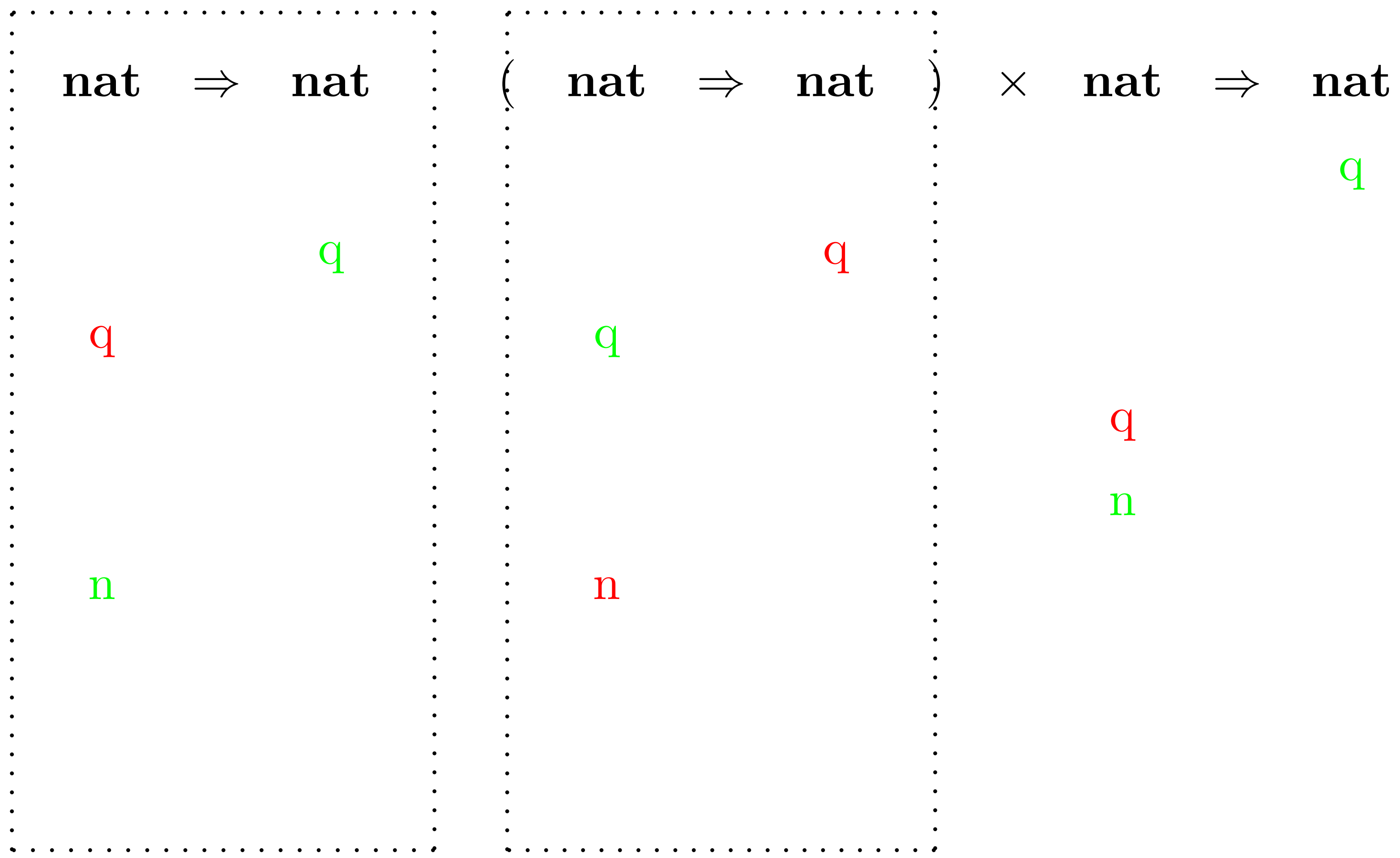
Composition

Apply $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2$ to $\lambda x : \mathbf{nat}.\, x^2$.



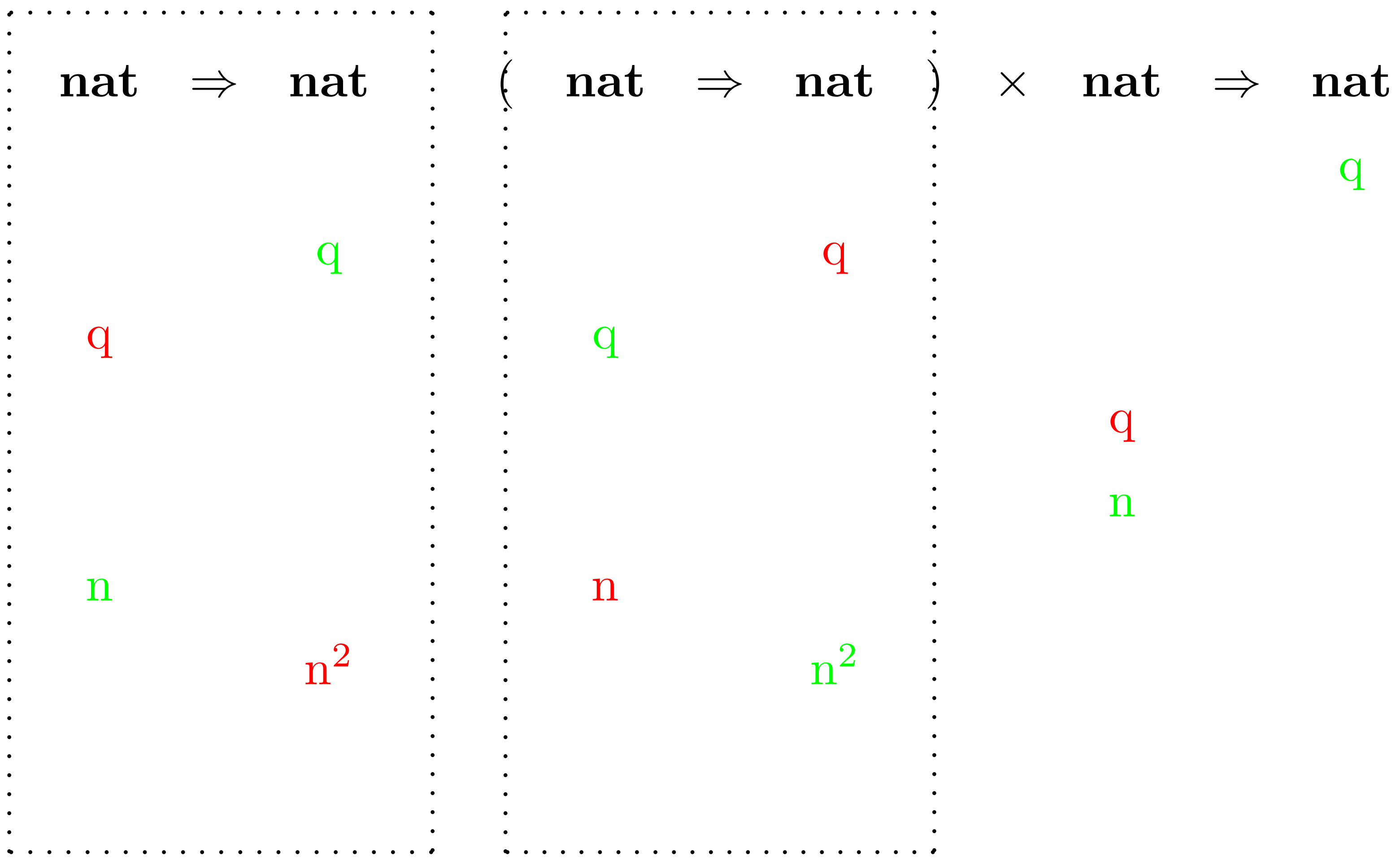
Composition

Apply $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2$ to $\lambda x : \mathbf{nat}.\, x^2$.



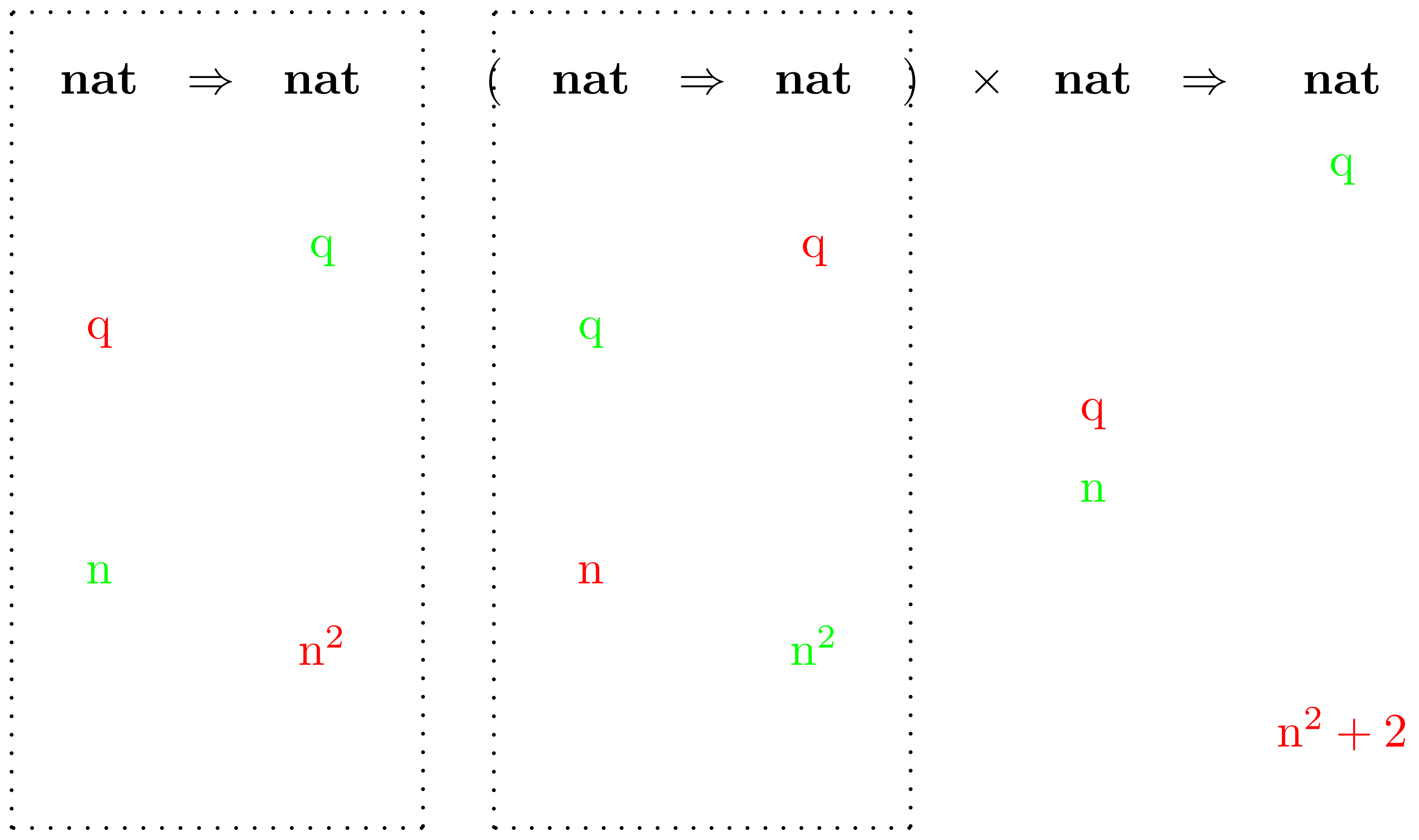
Composition

Apply $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2$ to $\lambda x : \mathbf{nat}.\, x^2$.



Composition

Apply $\lambda f : \mathbf{nat} \Rightarrow \mathbf{nat}.\, \lambda x : \mathbf{nat}.\, f(x) + 2$ to $\lambda x : \mathbf{nat}.\, x^2$.



Imperative variables

To interpret mutable variables, we will take an "object-oriented view", in which a variable (say for example being used to store values of type **nat**) is seen as an object with two methods:

- the "read method", for dereferencing, giving rise to an operation of type $\mathbf{var}[\mathbf{nat}] \Rightarrow \mathbf{nat}$;
- the "write method", for assignment, giving an operation of type $\mathbf{var}[\mathbf{nat}] \Rightarrow \mathbf{nat} \Rightarrow \mathbf{com}$.

We **identify** the type of variables with the product of the types of these methods, setting

$$\mathbf{var}[\mathbf{nat}] = (\mathbf{nat} \Rightarrow \mathbf{com}) \times \mathbf{nat}.$$

Now assignment and dereferencing are just the two projections, and we can interpret a command `x:=!x+1` as the strategy

$$
\begin{array}{ccccccccc}
(\mathbf{nat} & \Rightarrow & \mathbf{com}) & \times & \mathbf{nat} & \Longrightarrow & \mathbf{com} \\
 & & & & & & \texttt{run} \\
 & & & & \texttt{read} & & \\
 & & & & n & & \\
 & & \texttt{write} & & & & \\
q & & & & & & \\
n+1 & & & & & & \\
 & & \texttt{ok} & & & & \\
 & & & & & & \texttt{ok}
\end{array}
$$

Block structure

The key point is to interpret the **allocation** of variables correctly, so that if the variable x in the above example has been bound to a genuine storage cell, the various reads and writes made to it have the expected relationship. In general, a term M with a free variable x will be interpreted as a strategy for $\mathbf{var}[\mathbf{nat}] \Rightarrow A$, where A is the type of M. We must interpret `new` x `in` M as a strategy for A by "binding x to a memory cell". With game semantics, this is easy! We just need to define a suitable strategy cell, and use our general operation of composition of strategies:

$$[\![\mathtt{new}\ x\ \mathtt{in}\ M]\!] = [\![M]\!] \circ \mathsf{cell}.$$

The General Picture

Games and strategies organize themselves into mathematical structures (categories of various kinds) suitable for modelling programming languages.

By imposing various **structural constraints** on strategies, exact matches can be found with various **computational features** as embodied in key programming language constructs, leading to **full abstraction** results.

The first success story: purely functional languages, fully abstract models for PCF etc. (In retrospect, the **least applicable** results!) Constraints: strategies have restricted information available ("history-free" = memory-less, or "innocent"), corresponding to statelessness; and also satisfy a properly nested call-return discipline (no jumps).

Relaxing these constraints leads to fully abstract models for languages with (locally scoped) state, or control operators, or both.

The Game Semantics Landscape

Game semantics has proved to be a flexible and powerful paradigm for constructing highly structured fully abstract semantics for languages with a wide range of computational features:

- (higher-order) functions and procedures
- call by name and call by value
- locally scoped state
- general reference types
- control features (continuations, exceptions)
- non-determinism, probabilities
- concurrency (GM, FOSSACS 04)
- names and freshness (AGMOS, LiCS 04)

In many cases, game semantics have yielded the first, and often still the only, semantics construction of a fully abstract model for the language in question.

Where sufficient computational features (typically state or control) are present, the game semantics captures the fully abstract model directly, without the need for any quotient.

Algorithmic Game Semantics

We can take advantage of the **concrete nature** of game semantics. A play is a sequence of moves, so a strategy can be represented by the set of its plays, i.e. by a **language** over the alphabet of moves, and hence by an automaton.

There are significant finite-state fragments of the semantics for various interesting languages, as first observed by Ghica and McCusker (ICALP 00).

This means we can **compositionally construct** automata as (representations of) the meanings of open (incomplete) programs, giving a powerful basis for compositional software model-checking.

The Algorithmic Game Semantics Project

EPSRC-funded project at Oxford University Computing Laboratory

S.A., Luke Ong, Dan Ghica and Andrzej Murawski

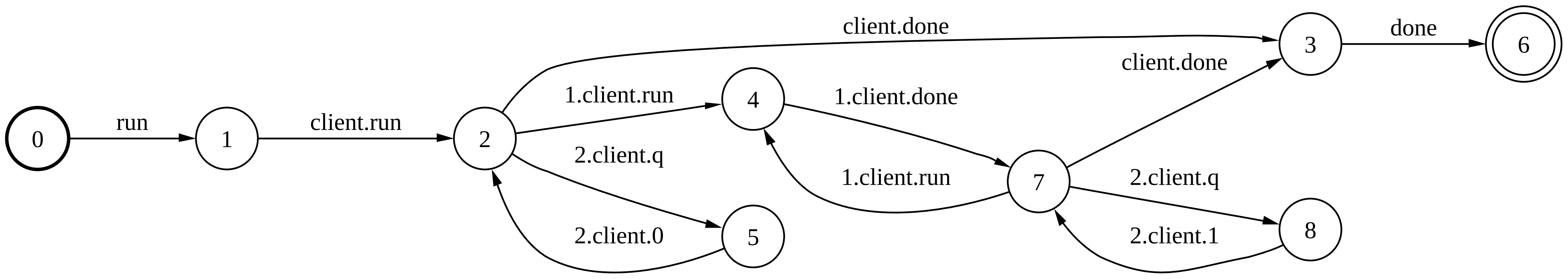
- Theoretical results on complexity, decidability of various fragments.
- Extending the games model, e.g. for concurrency (Idealized Parallel Algol), names and freshness, etc.
- Implementation of a 'game semantics compiler', which compiles Finitary IA programs into FSM's.
- Case studies using the compiler.

Implementation notes

- parser + type inference + (some) back-end processing: CAML
- (most) back-end RL processing: AT&T FSM Library
- output: AT&T GraphViz and dot packages

Warm-up example

```
client : com -> exp -> com |-
        new var v:= 0 in
        let set be v:=1 in
        let get be !v in
        client (set, get): com.
```



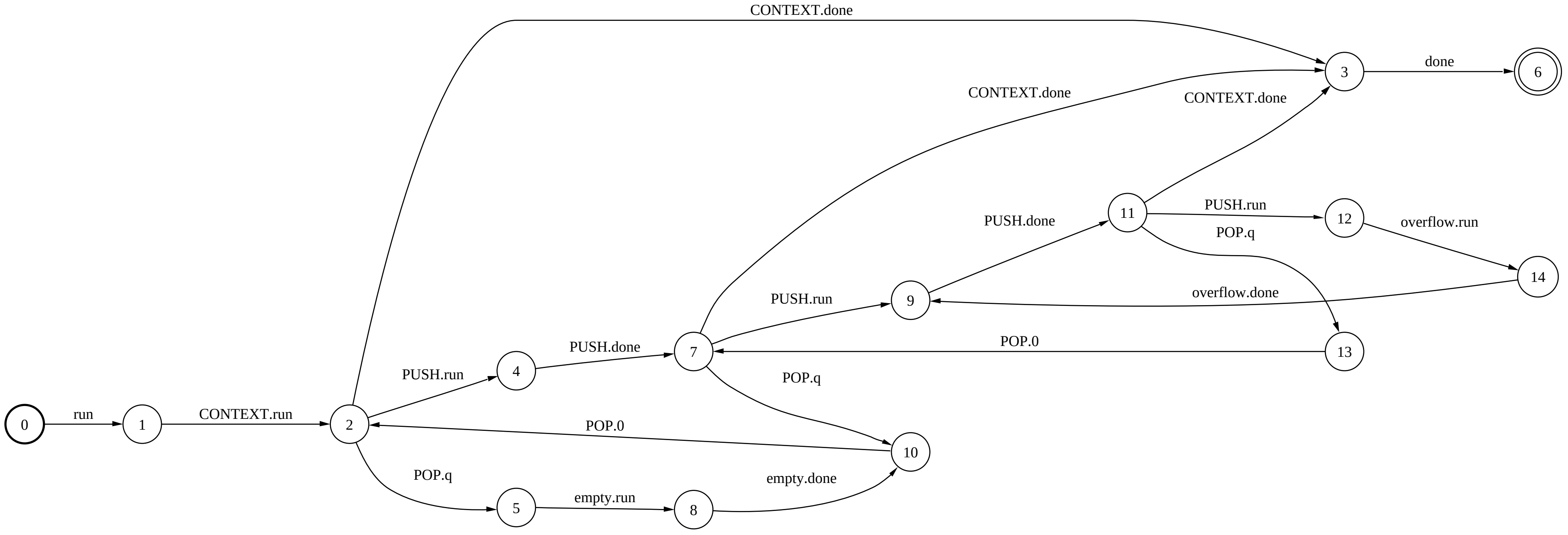
Case studies [SAVCBS'03, TACAS'04]

- safety properties for ADTs
- models of algorithm implementations (sorting)
- models of protocol implementations
- etc (not problem specific)

Modeling ADTs: a stack module

```
empty : com, overflow : com, m : exp, CONTEXT : com -> exp -> com |-
      array buffer[\(n\)] in
      let size be \(n\) in
      new var crt := 0 in
      let isempty be !crt = 0 in
      let isfull be !crt = size in
      let push be fun x : exp.
          new var temp := x in
          if isfull then overflow
          else buffer[!crt] := !temp;
            crt := !crt + 1 fi in
      let pop be
          if isempty then empty; 0
          else crt := !crt - 1; !buffer[!crt] fi in
      CONTEXT(push (m), pop): com.
```

Model of size-2 unary data stack



Modeling safety properties

- safety properties = RLs [Manna & Pnueli 90]
- safety properties = assertions (standard in SMC)

```
let assert be fun a:exp.
    if a then skip else error fi
```

- traces containing `error` actions are counterexamples

ADT invariants as safety properties

```
...
      VERIFY(push (m), pop,
        assert(
           new var x := m in
           push(!x);
           pop = !x
        ))
: com.
```

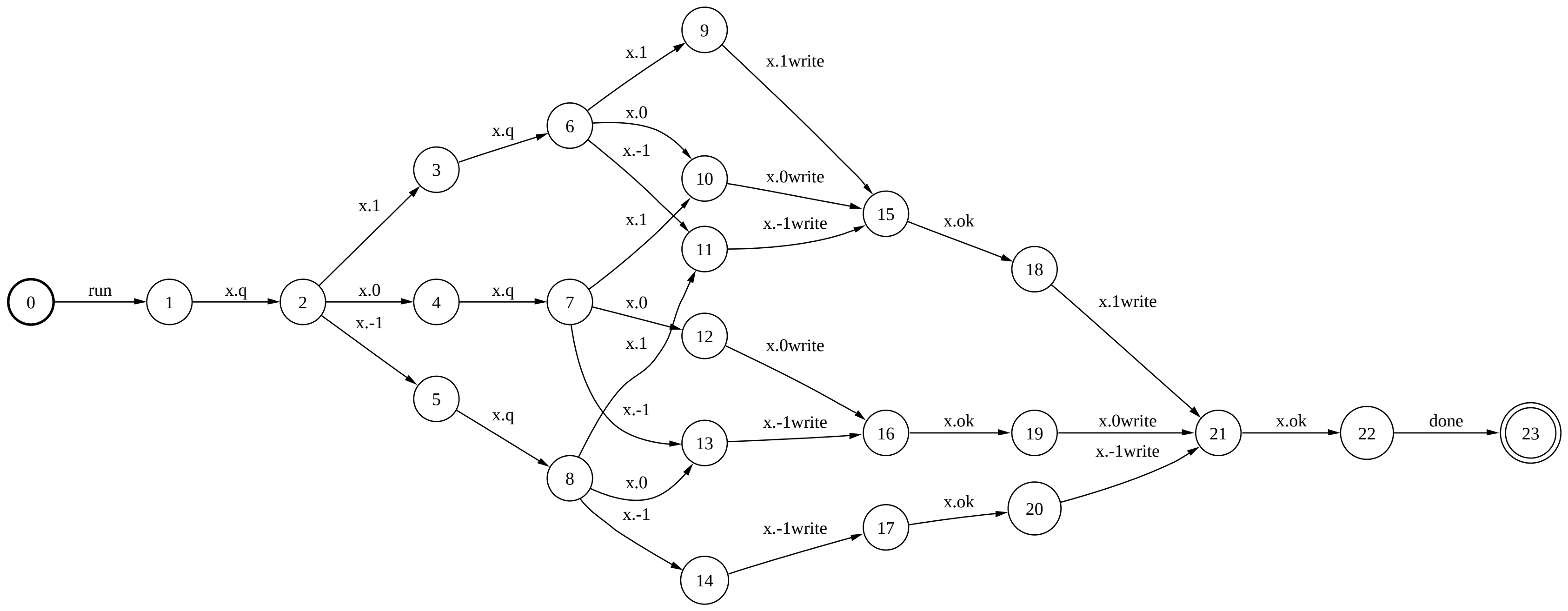
Extracting the shortest diagnostic trace

```
0       1       run
1       2       VERIFY.run
2       3       1.VERIFY.run
3       4       m.q
4       5       m.1
5       6       1.VERIFY.done
6       7       1.VERIFY.run
7       8       m.q
8       9       m.1
9       10      1.VERIFY.done
10      11      3.VERIFY.run
11      12      m.q
12      13      m.0
13      14      overflow.run
14      15      overflow.done
15      16      error.run
16      17      error.done [...]
```

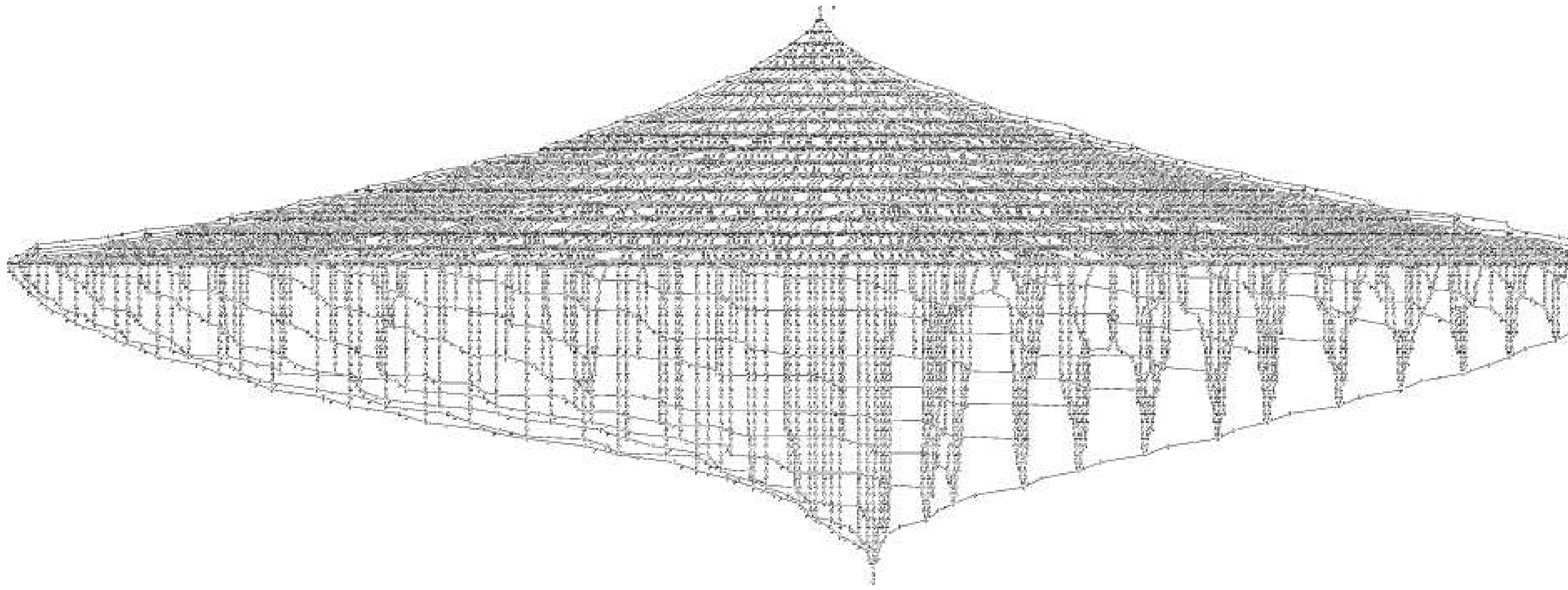
Sorting: bubblesort

```
x:var |-
  array a[\(n\)] in
  new var i:=0 in
  while !i < \(n\) do a[!i]:=!x; i:=!i+1 od;
  new var flag:=1 in
  while !flag do
    new var i:=0 in
    flag:=0;
    while !i < \(n\) - 1 do
      if !a[!i] > !a[!i+1] then
        flag:=1;
        new var temp:=!a[!i] in
          a[!i]:=!a[!i+1]; a[!i+1]:=!temp
        else skip fi;
      i:=!i+1 od od;
  new var i:=0 in
  while !i < \(n\) do x:=!a[!i]; i:=!i+1 od : com.
```

2-element array of integers %2



20-element array of integers %2



On modeling sorting programs

“[...] it seems impossible to use model-checking to verify that a sorting algorithm is correct since sorting correctness is a data-oriented property involving several quantifications and data structures.” [Bandera user manual]

Why does it work?

- program state-space: 5.5×10^{12} states
- model: $6,393$ states
- max space: $1,153,240$ states

Hiding local state!

Further Directions

- Extend compiler to handle concurrency and other features.
- Use FDR (CSP model-checker) as a back end.

Reactive Refinement

We shall now look at how a simple approach to components leads very naturally to Game semantics ideas.

We will also look at a natural game notion of reactive refinement, which seems promising for use with reactive components.

A simple model of ‘components’ or ‘objects’

Basic data types D, *e.g.* **int**, **bool**.

Method types:

$$T \quad = \quad D_1 \times \cdots \times D_k \longrightarrow D'_1 \times \cdots \times D'_l$$

Signature (of a **closed** component)

$$\vdash m_1 : T_1, \ldots, m_k : T_k$$

Examples

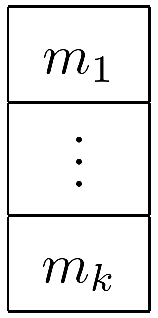
- A `counter` component has a method

$$\texttt{inc} : \mathbf{unit} \longrightarrow \mathbf{nat}$$

- A `stack` component has methods

$$\begin{array}{lcl} \texttt{push} & : & D \longrightarrow \mathbf{unit} \\ \texttt{pop} & : & \mathbf{unit} \longrightarrow \mathbf{unit} \\ \texttt{top} & : & \mathbf{unit} \longrightarrow D \end{array}$$

Component Environment



Basic ‘events’ or ‘actions’ of a component

$$m : D_1 \times \cdots \times D_k \longrightarrow D'_1 \times \cdots \times D'_l$$

Action type	Notation	Performed by
Method call	$m?(d_1, \ldots, d_k)$	Environment
Method return	$m!(d'_1, \ldots, d'_l)$	Component

A **trace** or **history** of the component is a sequence

$$\langle m_1?(\vec{d_1}), m_1!(\vec{d'}_1), \ldots, m_k?(\vec{d_k}), m_k!(\vec{d'}_k)\rangle$$

of alternating calls and returns.

We can **specify** a component in terms of the set of its possible histories. This allows **state-dependent behaviour** to be captured (even though no variables appear explicitly in our model).

Example Possible histories for the counter module have the form

$$\langle \mathtt{inc}?, \mathtt{inc}!0, \mathtt{inc}?, \mathtt{inc}!1, \ldots\rangle$$

Open Components

The general case: **open components**

An open component both:

- **uses** some methods to be supplied by the Environment
- **provides** some methods which can be used by the Environment

General component signature:

$$\begin{array}{ccc} \text{Uses} & \vdash & \text{Provides} \\ m_1 : T_1, \ldots, m_k : T_k & \vdash & m'_1 : T'_1, \ldots, m'_l : T'_l \end{array}$$

General classification of action types

There are now **four kinds** of action or event:

• Environment call	of a provided method
• Component return	“
• Component call	of a used method
• Environment return	“

The general structure of a component history is now as follows: a sequence of ‘blocks’

$$B_1 \cdots B_k$$

where each block has the form

$m?(\vec{d})$	Environment call
$m_1?(\vec{d}_1)$	Component call
$m_1!(\vec{d}'_1)$	Environment return
$\vdots$	
$m_k?(\vec{d}_k)$	Component call
$m_k!(\vec{d}'_k)$	Environment return
$m!(\vec{d}')$	Component return

NB: this is really a bit over-simplied - does not allow for re-entrancy, concurrent activations ...

Composition: plugging components together

Given component specifications

$$\begin{aligned} C_1 : &\qquad \Gamma_1, \Theta_1 \vdash \Delta_1, \Theta_2 \\ C_2 : &\qquad \Gamma_2, \Theta_2 \vdash \Delta_2, \Theta_1 \end{aligned}$$

we want to form a new component C with signature

$$\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2$$

Here

Θ_1 are the methods which C_1 requires **and** C_2 provides.
Θ_2 are the methods which C_2 requires **and** C_1 provides.

The histories of C are obtained in two stages:

1. 'Parallel Composition': form all histories in the signature

$$\Gamma, \Theta \vdash \Delta, \Theta$$

   (where $X = X_1 \cup X_2$, $X \in \{\Gamma, \Delta, \Theta\}$)
   such that:

   - The restriction to $\Gamma_1, \Theta_1 \vdash \Delta_1, \Theta_2$ is a history of C_1
   - The restriction to $\Gamma_2, \Theta_2 \vdash \Delta_2, \Theta_1$ is a history of C_2.

2. 'Hiding': erase all events in Θ.

Connection with game semantics

Environment calls	Opponent Questions
Component returns	Player Answers
Component calls	Player Questions
Environment returns	Opponent Answers

The structure of histories falls out from the game semantics for first-order procedures.

Composition of components is composition of strategies.

However: game semantics focusses on **types** and **programs** — modelled as **games** and **strategies** respectively.

We wish to have a broader range of **specifications**, with a spectrum

$$[\underbrace{\text{Types}\cdots\cdots\text{Programs}}_{\mathit{Specifications}}]$$

mediated by a suitable notion of **Refinement**.

Standard view of refinement

(See e.g. the books by Carroll Morgan and by Back and Von Wright).

A specification can be regarded as a **contract** between a "client" and a "programmer". Consider firstly the case of relational specifications. We think of a program as a relation $R \subseteq I \times O$ between a set of input values I and a set of output values O. A specification for such a program can be expressed as a precondition-postcondition pair (ϕ, ψ) where ϕ is a predicate on inputs, $\phi \subseteq I$, and ψ is a predicate on outputs, $\psi \subseteq O$. The intended interpretation is:

when the input satisfies ϕ*, then the output must satisfy* ψ

or more formally

$$\forall x.\, x \in \phi \;\Rightarrow\; R(x) \subseteq \psi.$$

Note that the programmer and the client have different—in fact complementary—aspects of the situation under their control. The client can control which inputs are supplied to the program when it is put into use; while the programmer's code will determine, for a given input, which outputs can be produced. We can say that the client generates the inputs, but then can only observe the outputs, while the programmer's view is dual: he observes the inputs, and must then generate the outputs. From the client's point of view, the **less** the specification constrains the inputs, and the **more** it constrains the outputs, the more chances he has to show that the program is at fault; the better his interests are protected. The programmer's view is again dual: the more the specification constrains the inputs, and the less it constrains the outputs, the easier his task becomes.

Cf. also subtyping.

This leads to the following standard notion of refinement of specifications:

$$(\phi_1, \psi_1) \sqsubseteq (\phi_2, \psi_2) \iff \phi_1 \Rightarrow \phi_2 \wedge \psi_2 \Rightarrow \psi_1.$$

Thus (ϕ_2, ψ_2) refines (ϕ_1, ψ_1) if, whenever the client is happy with (ϕ_1, ψ_1), then he will certainly be happy with (ϕ_2, ψ_2). In passing to the refined specification, the programmer is not allowed to make his task easier, either by over-constraining the environment in which the program will run—in this case, just the inputs which may be supplied to it—or by under-constraining the behaviour of the system—in this case the outputs which may be produced from given inputs.

Reactive refinement for component specifications

Suppose that S and T are component specifications over the same signature. We say that S is refined by T if:

The Environment is **less** constrained by T than by S
The Component is **more** constrained by T than by S.

With repeated interactions between the component and the environment, we need appropriate **conditioning** of the constraints.

Definition of Reactive Refinement

Fix a signature $\Gamma \vdash \Delta$, and let $\mathcal{H}$ be the set of all possible histories over this signature. A is the set of all actions (of all four types). Note that each history is an **alternating sequence** of moves by the Environment (at odd-numbered positions in the sequence) and by the Component (at even-numbered positions). We write $\mathcal{H}^{\mathrm{even}}$ for the set of even-length sequences in $\mathcal{H}$, and $\mathcal{H}^{\mathrm{odd}}$ for the set of odd-length sequences.

We define

$$S \sqsubseteq T \iff$$

$$\forall s \in \mathcal{H}^{\mathrm{even}}, \forall a \in A.\, s \cdot a \in S \wedge s \in T \ \Rightarrow s \cdot a \in T$$

$$\wedge$$

$$\forall t \in \mathcal{H}^{\mathrm{odd}}, \forall a \in A.\, t \cdot a \in T \wedge t \in S \ \Rightarrow t \cdot a \in S$$

'At Environment moves, $S \subseteq T$, and at Component moves, $S \supseteq T$'.

The Refinement Lattice

Given a family $\{S_i\}_{i \in I}$ of specifications, we can form their meet under the refinement ordering by the following inductive definition:

$$\begin{array}{rcl} \bigsqcap_{i \in I} S_i & = & \mu X. \{\varepsilon\} \\ & & \cup \{s \cdot a \mid s \in X^{\mathrm{even}} \ \wedge \ \forall i \in I. s \in S_i \ \Rightarrow \ s \cdot a \in S_i\} \\ & & \cup \{s \cdot a \mid s \in X^{\mathrm{odd}} \ \wedge \ \exists i \in I. s \in S_i \ \wedge \ s \cdot a \in S_i\} \end{array}$$

This lattice strcture has already been used in giving semantics of polymorphism and sub-typing (SA, Semantics of Interaction, Juliusz Chroboczek Ph.D. thesis).

Current direction (SA, Jan Jurjens): study general use in component refinement, applications to security properties.