Topics in Computer Science: Semantics of programming languages

Dr Philip Saville

UNIVERSITY OF SUSSEX

Theoretical computer science:

- Foundations: a mathematical theory of computation
- Algorithms and complexity: what can be computed? How fast?

Theoretical computer science:

- Foundations: a mathematical theory of computation
- Algorithms and complexity: what can be computed? How fast?
- **Semantics**: what is the mathematical <u>structure</u> of computation?

Theoretical computer science:

- Foundations: a mathematical theory of computation
- Algorithms and complexity: what can be computed? How fast?
- **Semantics**: what is the mathematical structure of computation?

In other words, what is the meaning of a program?

When we write programs, we have some sense of what they mean.

We sometimes write this ‘meaning’ down quite precisely.

When we write programs, we have some sense of what they mean.

We sometimes write this ‘meaning’ down quite precisely.

```python
def a_function(x, y):
  return x + y
```

When we write programs, we have some sense of what they mean.

We sometimes write this ‘meaning’ down quite precisely.

```
def a_function(x, y):
  return x + y
```

A mathematical function:

$$(x, y) \mapsto x + y$$

When we write programs, we have some sense of what they mean.

We sometimes write this ‘meaning’ down quite precisely.

```
def a_function(x, y):
  return x + y
```

A mathematical function:
$$(x, y) \mapsto x + y$$

For us:

- A set is a collection of elements;
- A function between two sets, written $A \to B$, is a rule assigning to each element of A an element of B.

When we write programs, we have some sense of what they mean.

We sometimes write this ‘meaning’ down quite precisely.

```
def a_function(x, y):
  return x + y
```

A mathematical function:

$$(x, y) \mapsto x + y$$

For us:

- A set is a collection of elements;
- A function between two sets, written $A \to B$, is a rule assigning to each element of A an element of B.

For example, writing $\mathbb{N}$ for the set of natural numbers $0, 1, 2, \ldots$:

1. We get a function $\mathbb{N} \to \mathbb{N}$ as the rule $x \mapsto x + 1$;
2. We get a different function $\mathbb{N} \to \mathbb{N}$ as the rule $x \mapsto 1$.

When we write programs, we have some sense of what they mean.

We sometimes write this ‘meaning’ down quite precisely.

```
def a_function(x, y):
  return x + y
```

A mathematical function:

$$(x, y) \mapsto x + y$$

When we write programs, we have some sense of what they mean.

We sometimes write this ‘meaning’ down quite precisely.

```
def a_function(x, y):
  return x + y
```

A mathematical function:

$$(x, y) \mapsto x + y$$

But this quickly gets more difficult:

When we write programs, we have some sense of what they mean.

We sometimes write this ‘meaning’ down quite precisely.

```
def a_function(x, y):
  return x + y
```

A mathematical function:
$$(x, y) \mapsto x + y$$

But this quickly gets more difficult:

```
  def another_function(x,y):
    print("hello")
    return x + y
```

???

When we write programs, we have some sense of what they mean.

We sometimes write this ‘meaning’ down quite precisely.

```
def a_function(x, y):
  return x + y
```

A mathematical function:

$$(x, y) \mapsto x + y$$

But this quickly gets more difficult:

```
    def another_function(x,y):
      print("hello")
      return x + y
```

???

Aim: give you some sense of why we want to do this, and how we do it

Aims of the next four lectures

At the end of the next four lectures you should be able to:

- Explain why we do semantics
- Distinguish between operational and denotational approaches
- Be able to prove basic semantic facts about some toy languages
- Outline some directions of where the field goes next

Why do we do semantics?

:(

Your PC ran into a problem that it couldn't handle, and now it needs to restart.

You can search for the error online: HAL_INITIALIZATION_FAILED

Why do we do semantics?

Reason 1: if we understand what programs mean,
we can verify they do what they're supposed to

Why do we do semantics?

Reason 1: if we understand what programs mean,
we can verify they do what they're supposed to

> Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example driver verification, we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability.
>
> Bill Gates, April 18, 2002. Keynote address at WinHec 2002

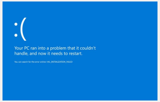
Why do we do semantics?

Reason 1: if we understand what programs mean,
we can verify they do what they're supposed to
…and hopefully make them more reliable

> Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example driver verification, we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability.
>
> Bill Gates, April 18, 2002. Keynote address at WinHec 2002

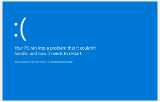
Reliability really matters

In our day-to-day lives software failures are often just annoying:



Reliability really matters

In our day-to-day lives software failures are often just annoying:



But they can also be life-threatening and catastrophic:

Reliability really matters

Software is at the heart of the modern world.

It's crucial we understand what it does!

Why do we do semantics?

Reason 1: to verify they do what they're supposed to
. . . and hopefully make them more reliable

Reason 2: to write tools for existing languages (e.g. a compiler)
we need to understand their behaviour

Why do we do semantics?

Reason 1: to verify they do what they're supposed to
. . . and hopefully make them more reliable

Reason 2: to write tools for existing languages (e.g. a compiler)
we need to understand their behaviour

> Higher-order languages. . . encourage the programmer to build abstractions by composing functions. A good compiler must inline many of these calls to recover an efficiently executable program. In principle, inlining is dead simple: just replace the call of a function by an instance of its body. But any compiler-writer will tell you that inlining is a black art, full of delicate compromises that work together to give good performance without unnecessary code bloat.
>
> Simon Peyton-Jones, one of the designers of Haskell

Why do we do semantics?

Reason 1: to verify they do what they're supposed to
. . . and hopefully make them more reliable

Reason 2: to write tools for existing languages (e.g. a compiler)
we need to understand their behaviour

Reason 3: to design new programming languages and paradigms

(a) by thoroughly understanding the structure of programs,
we can discover useful new constructs
example: effect handlers in Jax and Pyro

(b) we can also make sure new features behave correctly

A cautionary tale

The ML language has a range of powerful features that give strong guarantees on program behaviour

(e.g. strict typing, type inference, polymorphism ...)

A cautionary tale

The ML language has a range of powerful features that give strong guarantees on program behaviour

(e.g. strict typing, type inference, polymorphism ...)

... but an apparently-innocuous combination of its features broke these guarantees!

Making new features behave correctly

We want programs to do increasingly-fancy things.

Example 1: specify probabilistic models

e.g. for pandemic modelling

```
normalize(
  x = sample(bernoulli (2/7));
  r = if x then 3 else 10;
  observe 4 from poisson(r);
  return (x)
)
```

Making new features behave correctly

We want programs to do increasingly-fancy things.

Example 1: specify probabilistic models

e.g. for pandemic modelling

```
normalize(
  x = sample(bernoulli (2/7));
  r = if x then 3 else 10;
  observe 4 from poisson(r);
  return (x)
)
```

Example 2: specify smooth functions, with a derivative we can compute algorithmically

e.g. for writing neural networks

Designing new programming features

We want programs to do increasingly-fancy things.

Example 1: specify probabilistic models
e.g. for pandemic modelling

Example 2: specify smooth functions, with a derivative we can compute algorithmically
e.g. for writing neural networks

Both **probabilistic programs** and **differentiable programs** involve

1. Powerful program features (e.g. loops, recursion, higher-order functions, ...)
2. Sophisticated mathematics (e.g. probability, measure theory, differentiability, ...)

Making sure these combine to do the right thing is both subtle and difficult!

Why do we do semantics?

Reason 1: to verify they do what they're supposed to
. . . and hopefully make them more reliable

Reason 2: to write tools for existing languages (e.g. a compiler)
we need to understand their behaviour

Reason 3: to design new programming languages and paradigms

(a) by thoroughly understanding the structure of programs,
we can discover useful new constructs
example: effect handlers in Jax and Pyro

(b) we can also make sure new features behave correctly

Do we need semantics?

Can't we just...?

Can't we just ... use the compiler?

Suggestion: define the meaning of a program as the mapping that the compiler defines from the source code to machine code.

Does this work?

Can't we just ... use the compiler?

Suggestion: define the meaning of a program as the mapping that the compiler defines from the source code to machine code.

Does this work?

1. It is machine dependent.

Can't we just . . . use the compiler?

Suggestion: define the meaning of a program as the mapping that the compiler defines from the source code to machine code.

Does this work?

1. It is machine dependent.
2. It provides the wrong level of abstraction for the programmer, to the point it becomes useless. (Try to understand the meaning of a program by reading its translation to assembly code!)

Can't we just ... use the compiler?

Suggestion: define the meaning of a program as the mapping that the compiler defines from the source code to machine code.

Does this work?

1. It is machine dependent.
2. It provides the wrong level of abstraction for the programmer, to the point it becomes useless. (Try to understand the meaning of a program by reading its translation to assembly code!)
3. What if the compiler has bugs? It might translate a program in a different way from what it was supposed to do according to the language designers.

Can't we just . . . use the compiler?

Suggestion: define the meaning of a program as the mapping that the compiler defines from the source code to machine code.

Does this work?

1. It is machine dependent.
2. It provides the wrong level of abstraction for the programmer, to the point it becomes useless. (Try to understand the meaning of a program by reading its translation to assembly code!)
3. What if the compiler has bugs? It might translate a program in a different way from what it was supposed to do according to the language designers.
4. It doesn't help us to do any formal analysis of properties.

Can't we just . . . use the compiler?

Suggestion: define the meaning of a program as the mapping that the compiler defines from the source code to machine code.

Does this work?

1. It is machine dependent.
2. It provides the wrong level of abstraction for the programmer, to the point it becomes useless. (Try to understand the meaning of a program by reading its translation to assembly code!)
3. What if the compiler has bugs? It might translate a program in a different way from what it was supposed to do according to the language designers.
4. It doesn't help us to do any formal analysis of properties.

Can't we just … write out what we mean?

Suggestion: define the meaning of a program as how it behaves, which we write down in words as part of the specification of the language.

Can't we just . . . write out what we mean?

Suggestion: define the meaning of a program as how it behaves, which we write down in words as part of the specification of the language.

Every language manual contains descriptions in plain English of the meaning of the various constructs, how they should be used, and example code fragments (*pragmatics*).

Can't we just ... write out what we mean?

Suggestion: define the meaning of a program as how it behaves, which we write down in words as part of the specification of the language.

Every language manual contains descriptions in plain English of the meaning of the various constructs, how they should be used, and example code fragments (*pragmatics*).

4.7.3.3 Body replacement and execution. Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed. If the procedure is called from a place outside the scope of any non-local quantity of the procedure body, the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

From the Revised Report of Algol 60.

The limitations of writing things in words

Side effects = changes to variables resulting from the evaluation of an expression for which the variable is not local.

An Algol 60 program

begin **integer** a;
  integer **procedure** $f(x, y)$; **value** y, x; **integer** y, x;
    $a := f := x + 1$;
  integer **procedure** $g(x)$; **integer** x; $x := g := a + 2$;
    $a := 0$; $outreal\big(1,\ a\ +\ f(a, g(a))\,/g(a)\big)$ **end**

The limitations of writing things in words

Side effects = changes to variables resulting from the evaluation of an expression for which the variable is not local.

An Algol 60 program

begin integer a;
  integer procedure $f(x, y)$; **value** y, x; **integer** y, x;
    $a := f := x + 1$;
  integer procedure $g(x)$; **integer** x; $x := g := a + 2$;
    $a := 0$; $outreal\big(1,\ a\ +\ f(a, g(a))\ /g(a)\big)$ **end**

Depending on the order of evaluation, the final result can be $4\,{}^1\!/_2$, $\frac{1}{3}$, $\frac{3}{5}$, $\frac{3}{2}$, $\frac{5}{2}$, $\frac{4}{3}$, $3\,{}^3\!/_5$, $3\,{}^1\!/_3$, $5\,{}^3\!/_5$, $3\,{}^1\!/_2$, and $7\,{}^1\!/_2$!

The limitations of writing things in words

Side effects = changes to variables resulting from the evaluation of an expression for which the variable is not local.

An ALGOL 60 program

begin integer a;
  integer procedure $f(x, y)$; **value** y, x; **integer** y, x;
    $a := f := x + 1$;
  integer procedure $g(x)$; **integer** x; $x := g := a + 2$;
    $a := 0$; $outreal\big(1,\ a\ +\ f(a, g(a))\ /g(a)\big)$ **end**

Depending on the order of evaluation, the final result can be $4\,{}^1\!/_2$, $\frac{1}{3}$, $\frac{3}{5}$, $\frac{3}{2}$, $\frac{5}{2}$, $\frac{4}{3}$, $3\,{}^3\!/_5$, $3\,{}^1\!/_3$, $5\,{}^3\!/_5$, $3\,{}^1\!/_2$, and $7\,{}^1\!/_2$!

The Algol Report doesn't say whether procedures are "allowed" side effects or not.

The limitations of writing things in words

Side effects = changes to variables resulting from the evaluation of an expression for which the variable is not local.

An ALGOL 60 program

begin integer a;
  integer procedure $f(x, y)$; **value** y, x; **integer** y, x;
    $a := f := x + 1$;
  integer procedure $g(x)$; **integer** x; $x := g := a + 2$;
    $a := 0$; $outreal\big(1,\ a\ +\ f(a, g(a))\ /g(a)\big)$ **end**

Depending on the order of evaluation, the final result can be $4\,{}^1\!/_2$, $\frac{1}{3}$, $\frac{3}{5}$, $\frac{3}{2}$, $\frac{5}{2}$, $\frac{4}{3}$, $3\,{}^3\!/_5$, $3\,{}^1\!/_3$, $5\,{}^3\!/_5$, $3\,{}^1\!/_2$, and $7\,{}^1\!/_2$!

The Algol Report doesn't say whether procedures are "allowed" side effects or not.

See also: the huge on-going effort to verify chip architectures.

To do semantics, we want something that is

- *formal* (not just in English),
- *precise* (no ambiguity)
- *concise* (readable to humans!)

...and which can define the behaviour of *any* (syntactically correct) program.

To do semantics, we want something that is

- *formal* (not just in English),
- *precise* (no ambiguity)
- *concise* (readable to humans!)

…and which can define the behaviour of *any* (syntactically correct) program.

In other words, we want a **mathematical model** of programs.

Aside: syntax vs semantics

Every programming language comes with

1. **Syntax** = the sequences of symbols that are valid in the language
   (e.g. where { and } go in Java)
2. **Semantics** = the meaning of (correctly-written) programs.

Ways of doing semantics

= different ways of modelling programs

Ways of doing semantics

- **Operational**: a program's meaning is given in terms of the steps of computation the program makes when you run it. Heavily syntax-oriented and rather intuitive. Very useful in implementation.

Ways of doing semantics

- **Operational**: a program's meaning is given in terms of the steps of computation the program makes when you run it. Heavily syntax-oriented and rather intuitive. Very useful in implementation.
- **Denotational**: a program's meaning is given abstractly as an element of some mathematical structure (e.g. some kind of set). Provides the deepest and most widely applicable techniques, borrowed from mathematics.

Ways of doing semantics

- **Operational**: a program's meaning is given in terms of the steps of computation the program makes when you run it. Heavily syntax-oriented and rather intuitive. Very useful in implementation.
- **Denotational**: a program's meaning is given abstractly as an element of some mathematical structure (e.g. some kind of set). Provides the deepest and most widely applicable techniques, borrowed from mathematics.
- **Axiomatic**: a program's meaning is given indirectly in terms of the collection of properties it satisfies; these properties are defined via a collection of axioms and rules.

Ways of doing semantics

Different approaches complement one another:

1. correctness of the proof rules of an axiomatic semantics relies on an underlying denotational or operational semantics,
2. correctness of an implementation with respect to a denotational semantics requires a proof that the operational and denotational semantics agree,
3. in proving facts about an operational semantics it can be of great help to use a denotational semantics, which abstracts away from unimportant, implementation details.

Time to see some semantics!

What we'll see next:

> operational and denotational semantics for a very basic language of numerical expressions.

Time to see some semantics!

What we'll see next:

> operational and denotational semantics for a very basic language of numerical expressions.

This is a short course! The aim is to give a flavour of how things go;
at the end we'll add some features and think how to adapt things for that

The language `NumExp`

A VERY basic language NumExp

$$
\begin{array}{llll}
\mathit{Num} & \mathtt{n} & ::= & \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \ldots \\
\mathit{Exp} & \mathtt{e} & ::= & \mathtt{n} \mid \mathtt{e} \oplus \mathtt{e} \mid \mathtt{e} \otimes \mathtt{e}
\end{array}
$$

A VERY basic language `NumExp`

$$
\begin{array}{llcl}
\mathit{Num} & \mathtt{n} & ::= & \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \ldots \\
\mathit{Exp} & \mathtt{e} & ::= & \mathtt{n} \mid \mathtt{e} \oplus \mathtt{e} \mid \mathtt{e} \otimes \mathtt{e}
\end{array}
$$

The fonts matter! We distinguish between

1. numerals like $\mathtt{3}$ (syntax, part of the language)
2. numbers like 3 (semantics, what we usually mean by numbers)

A VERY basic language $\texttt{NumExp}$

$$
\begin{array}{llcl}
\mathit{Num} & \texttt{n} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \\
\mathit{Exp} & \texttt{e} & ::= & \texttt{n} \mid \texttt{e} \oplus \texttt{e} \mid \texttt{e} \otimes \texttt{e}
\end{array}
$$

The fonts matter! We distinguish between

1. numerals like $\texttt{3}$ (syntax, part of the language)
2. numbers like 3 (semantics, what we usually mean by numbers)

We only care about valid syntax, so we have to use brackets to make this clear.

We will never write $\texttt{1} \oplus \texttt{2} \otimes \texttt{3}$, only well-formed expressions like $(\texttt{4} \oplus \texttt{5}) \otimes \texttt{6}$.

A VERY basic language `NumExp`

$$
\begin{array}{llll}
\textit{Num} & \mathtt{n} & ::= & \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \ldots \\
\textit{Exp} & \mathtt{e} & ::= & \mathtt{n} \mid \mathtt{e} \oplus \mathtt{e} \mid \mathtt{e} \otimes \mathtt{e}
\end{array}
$$

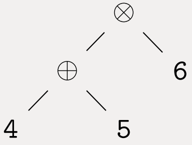
The fonts matter! We distinguish between

1. numerals like $\mathtt{3}$ (syntax, part of the language)
2. numbers like 3 (semantics, what we usually mean by numbers)

We only care about valid syntax, so we have to use brackets to make this clear.

We will never write $\mathtt{1} \oplus \mathtt{2} \otimes \mathtt{3}$, only well-formed expressions like $(\mathtt{4} \oplus \mathtt{5}) \otimes \mathtt{6}$.

To be precise, these represent *abstract sytax* trees, like:



Our informal reading of what `NumExp` programs mean:

1. Every numeral $\mathtt{n}$ is evaluated to the corresponding number n;
2. To find the value associated with an expression of the form $\mathtt{e}_0 \oplus \mathtt{e}_1$ we evaluate the expressions $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the sum of the results;
3. To find the value associated with an expression of the form $\mathtt{e}_0 \otimes \mathtt{e}_1$ we evaluate the expressions $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the product of the results

These are choices, we could do something different!

Our informal reading of what `NumExp` programs mean:

1. Every numeral $\texttt{n}$ is evaluated to the corresponding number n;
2. To find the value associated with an expression of the form $\texttt{e}_0 \oplus \texttt{e}_1$ we evaluate the expressions $\texttt{e}_0$ and $\texttt{e}_1$ and take the sum of the results;
3. To find the value associated with an expression of the form $\texttt{e}_0 \otimes \texttt{e}_1$ we evaluate the expressions $\texttt{e}_0$ and $\texttt{e}_1$ and take the product of the results

These are choices, we could do something different!

Examples:

- $\texttt{42}$ is evaluated to 42,
- $(\texttt{1} \oplus \texttt{2}) \otimes \texttt{3}$ is evaluated to 9,
- $(\texttt{1} \oplus \texttt{2}) \otimes (\texttt{3} \oplus \texttt{4})$ is evaluated to 21.

Operational semantics for `NumExp`

Giving operational semantics to `NumExp`

We want to *precisely* and *unambiguously* describe the steps needed to evaluate an arbitrary expression.

Giving operational semantics to `NumExp`

We want to *precisely* and *unambiguously* describe the steps needed to evaluate an arbitrary expression.

We describe how individual steps of a computation take place on an abstract device, but ignore details such as the use of registers and storage addresses.

Giving operational semantics to `NumExp`

We want to *precisely* and *unambiguously* describe the steps needed to evaluate an arbitrary expression.

We describe how individual steps of a computation take place on an abstract device, but ignore details such as the use of registers and storage addresses.

We do this using *logical rules*:

Giving operational semantics to `NumExp`

We want to *precisely* and *unambiguously* describe the steps needed to evaluate an arbitrary expression.

We describe how individual steps of a computation take place on an abstract device, but ignore details such as the use of registers and storage addresses.

We do this using *logical rules*:

$$\mathsf{e} \to \mathsf{e}'$$

"after evaluating e by one step, the expression e' remains to be evaluated".

Giving operational semantics to `NumExp`

We want to *precisely* and *unambiguously* describe the steps needed to evaluate an arbitrary expression.

We describe how individual steps of a computation take place on an abstract device, but ignore details such as the use of registers and storage addresses.

We do this using *logical rules*:

$$\mathsf{e} \to \mathsf{e}'$$

"after evaluating e by one step, the expression e' remains to be evaluated".

$$\frac{\text{premise}_1 \quad \ldots \quad \text{premise}_n}{\text{conclusion}} \text{ side-condition (rule name)}$$

"if we have all the premises, and the side-condition holds, we get the conclusion"

Examples of logical rules

$$\frac{}{\mathtt{n}_0 \oplus \mathtt{n}_1 \rightarrow \mathtt{n}}\; n = n_0 + n_1 (sum) \qquad \frac{\mathsf{e}_0 \rightarrow \mathsf{e}_0'}{\mathsf{e}_0 \oplus \mathsf{e}_1 \rightarrow \mathsf{e}_0' \oplus \mathsf{e}_1}\;(sumL)$$

Examples of logical rules

$$\frac{}{\mathtt{n}_0 \oplus \mathtt{n}_1 \to \mathtt{n}}\ n = n_0 + n_1 (sum) \qquad \frac{\mathtt{e}_0 \to \mathtt{e}_0'}{\mathtt{e}_0 \oplus \mathtt{e}_1 \to \mathtt{e}_0' \oplus \mathtt{e}_1}\ (sumL)$$

It's sometimes more natural to read from top to bottom: *sumL* says

> To evaluate $\mathtt{e}_0 \oplus \mathtt{e}_1$ one step, first evaluate $\mathtt{e}_0$ one step.

Small-step operational semantics for `NumExp`

Small-step = we describe every step the computation makes,
not just its final result.

$$\frac{}{\mathtt{n}_0 \oplus \mathtt{n}_1 \to \mathtt{n}}\; n = n_0 + n_1 (sum) \qquad \frac{\mathsf{e}_0 \to \mathsf{e}_0'}{\mathsf{e}_0 \oplus \mathsf{e}_1 \to \mathsf{e}_0' \oplus \mathsf{e}_1}\;(sumL) \qquad \frac{\mathsf{e}_1 \to \mathsf{e}_1'}{\mathsf{e}_0 \oplus \mathsf{e}_1 \to \mathsf{e}_0 \oplus \mathsf{e}_1'}\;(sumR)$$

$$\frac{}{\mathtt{n}_0 \otimes \mathtt{n}_1 \to \mathtt{n}}\; n = n_0 \times n_1 (prod) \qquad \frac{\mathsf{e}_0 \to \mathsf{e}_0'}{\mathsf{e}_0 \otimes \mathsf{e}_1 \to \mathsf{e}_0' \otimes \mathsf{e}_1}\;(prodL) \qquad \frac{\mathsf{e}_1 \to \mathsf{e}_1'}{\mathsf{e}_0 \otimes \mathsf{e}_1 \to \mathsf{e}_0 \otimes \mathsf{e}_1'}\;(prodR)$$

Derivations

We can now start to see when running one program gives another.

Derivations

We can now start to see when running one program gives another.

We build *derivations* by correctly stacking rules on top of one another. For example:

$$\dfrac{\dfrac{}{\mathsf{1} \oplus \mathsf{2} \to \mathsf{3}}\ {\scriptstyle 3=1+2}\,(sum)}{(\mathsf{1} \oplus \mathsf{2}) \otimes (\mathsf{3} \oplus \mathsf{4}) \to \mathsf{3} \otimes (\mathsf{3} \oplus \mathsf{4})}\ (prodL)$$

Derivations

We can now start to see when running one program gives another.

We build *derivations* by correctly stacking rules on top of one another. For example:

$$\dfrac{\dfrac{}{\mathsf{1} \oplus \mathsf{2} \to \mathsf{3}}\ {\scriptstyle 3=1+2\,(sum)}}{(\mathsf{1} \oplus \mathsf{2}) \otimes (\mathsf{3} \oplus \mathsf{4}) \to \mathsf{3} \otimes (\mathsf{3} \oplus \mathsf{4})}\ (prodL)$$

We think of this as saying

> If we start from $(1 \oplus 2) \otimes (3 \oplus 4)$ then $3 \otimes (3 \oplus 4)$ is an intermediate result in the program's execution.

Derivations

Definition

A judgement $\mathsf{e} \rightarrow \mathsf{e}'$ is *derivable* if there is a derivation whose conclusion is $\mathsf{e} \rightarrow \mathsf{e}'$.

Derivations

Definition

A judgement $\mathtt{e} \to \mathtt{e}'$ is *derivable* if there is a derivation whose conclusion is $\mathtt{e} \to \mathtt{e}'$.

Both of the following judgements are derivable:

$$(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to \mathtt{3} \otimes (\mathtt{3} \oplus \mathtt{4}) \qquad (\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to (\mathtt{1} \oplus \mathtt{2}) \otimes \mathtt{7}$$

and here are their derivations:

$$\dfrac{\dfrac{}{\mathtt{1} \oplus \mathtt{2} \to \mathtt{3}}\ {}^{3=1+2\,(sum)}}{(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to \mathtt{3} \otimes (\mathtt{3} \oplus \mathtt{4})}\ (prodL) \qquad \dfrac{\dfrac{}{\mathtt{3} \oplus \mathtt{4} \to \mathtt{7}}\ {}^{7=3+4\,(sum)}}{(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to (\mathtt{1} \oplus \mathtt{2}) \otimes \mathtt{7}}\ (prodR)$$

Derivations

Definition

A judgement $\mathsf{e} \to \mathsf{e}'$ is *derivable* if there is a derivation whose conclusion is $\mathsf{e} \to \mathsf{e}'$.

Both of the following judgements are derivable:

$$(1 \oplus 2) \otimes (3 \oplus 4) \to 3 \otimes (3 \oplus 4) \qquad (1 \oplus 2) \otimes (3 \oplus 4) \to (1 \oplus 2) \otimes 7$$

and here are their derivations:

$$\frac{\overline{1 \oplus 2 \to 3}\ {}^{3=1+2\,(sum)}}{(1 \oplus 2) \otimes (3 \oplus 4) \to 3 \otimes (3 \oplus 4)}\ (prodL) \qquad \frac{\overline{3 \oplus 4 \to 7}\ {}^{7=3+4\,(sum)}}{(1 \oplus 2) \otimes (3 \oplus 4) \to (1 \oplus 2) \otimes 7}\ (prodR)$$

Question: Is $3 \otimes 2 \to 5$ derivable? What about $(1 \oplus 2) \otimes (3 \oplus 4) \to 3 \otimes 7$?

Specifying an evaluation order

Our rules so far specify that the left-most operand always gets evaluated first:

$$\frac{\mathsf{e}_1 \to \mathsf{e}_1'}{\mathsf{e}_0 \oplus \mathsf{e}_1 \to \mathsf{e}_0 \oplus \mathsf{e}_1'}\ (sumR) \quad \text{and} \quad \frac{\mathsf{e}_1 \to \mathsf{e}_1'}{\mathsf{e}_0 \otimes \mathsf{e}_1 \to \mathsf{e}_0 \otimes \mathsf{e}_1'}\ (prodR)$$

Specifying an evaluation order

Our rules so far specify that the left-most operand always gets evaluated first:

$$\frac{\mathtt{e}_1 \rightarrow \mathtt{e}_1'}{\mathtt{e}_0 \oplus \mathtt{e}_1 \rightarrow \mathtt{e}_0 \oplus \mathtt{e}_1'} \ (sumR) \quad \text{and} \quad \frac{\mathtt{e}_1 \rightarrow \mathtt{e}_1'}{\mathtt{e}_0 \otimes \mathtt{e}_1 \rightarrow \mathtt{e}_0 \otimes \mathtt{e}_1'} \ (prodR)$$

We could instead choose to evaluate the right-most operand first:

$$\frac{\mathtt{e}_1 \rightarrow \mathtt{e}_1'}{\mathtt{n} \oplus \mathtt{e}_1 \rightarrow \mathtt{n} \oplus \mathtt{e}_1'} \ (sumR) \quad \text{and} \quad \frac{\mathtt{e}_1 \rightarrow \mathtt{e}_1'}{\mathtt{n} \otimes \mathtt{e}_1 \rightarrow \mathtt{n} \otimes \mathtt{e}_1'} \ (prodR)$$

Specifying an evaluation order

Our rules so far specify that the left-most operand always gets evaluated first:

$$\frac{\mathtt{e}_1 \to \mathtt{e}_1'}{\mathtt{e}_0 \oplus \mathtt{e}_1 \to \mathtt{e}_0 \oplus \mathtt{e}_1'}\ (sumR) \quad \text{and} \quad \frac{\mathtt{e}_1 \to \mathtt{e}_1'}{\mathtt{e}_0 \otimes \mathtt{e}_1 \to \mathtt{e}_0 \otimes \mathtt{e}_1'}\ (prodR)$$

We could instead choose to evaluate the right-most operand first:

$$\frac{\mathtt{e}_1 \to \mathtt{e}_1'}{\mathtt{n} \oplus \mathtt{e}_1 \to \mathtt{n} \oplus \mathtt{e}_1'}\ (sumR) \quad \text{and} \quad \frac{\mathtt{e}_1 \to \mathtt{e}_1'}{\mathtt{n} \otimes \mathtt{e}_1 \to \mathtt{n} \otimes \mathtt{e}_1'}\ (prodR)$$

With these rules $(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to (\mathtt{1} \oplus \mathtt{2}) \otimes \mathtt{7}$ is no longer derivable.

Exercise: show that the following judgements are now derivable:

$$(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \to \mathtt{3} \otimes (\mathtt{3} \oplus \mathtt{4}) \qquad \mathtt{3} \otimes (\mathtt{3} \oplus \mathtt{4}) \to \mathtt{3} \otimes \mathtt{7} \qquad \mathtt{3} \otimes \mathtt{7} \to \mathtt{21}$$

Specifying an evaluation order — these choices can be **very** important!

Our rules so far specify that the left-most operand always gets evaluated first:

$$\frac{\mathtt{e}_1 \rightarrow \mathtt{e}_1'}{\mathtt{e}_0 \oplus \mathtt{e}_1 \rightarrow \mathtt{e}_0 \oplus \mathtt{e}_1'}\ (sumR) \quad \text{and} \quad \frac{\mathtt{e}_1 \rightarrow \mathtt{e}_1'}{\mathtt{e}_0 \otimes \mathtt{e}_1 \rightarrow \mathtt{e}_0 \otimes \mathtt{e}_1'}\ (prodR)$$

We could instead choose to evaluate the right-most operand first:

$$\frac{\mathtt{e}_1 \rightarrow \mathtt{e}_1'}{\mathtt{n} \oplus \mathtt{e}_1 \rightarrow \mathtt{n} \oplus \mathtt{e}_1'}\ (sumR) \quad \text{and} \quad \frac{\mathtt{e}_1 \rightarrow \mathtt{e}_1'}{\mathtt{n} \otimes \mathtt{e}_1 \rightarrow \mathtt{n} \otimes \mathtt{e}_1'}\ (prodR)$$

With these rules $(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \rightarrow (\mathtt{1} \oplus \mathtt{2}) \otimes \mathtt{7}$ is no longer derivable.

Exercise: show that the following judgements are now derivable:

$$(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \rightarrow \mathtt{3} \otimes (\mathtt{3} \oplus \mathtt{4}) \qquad \mathtt{3} \otimes (\mathtt{3} \oplus \mathtt{4}) \rightarrow \mathtt{3} \otimes \mathtt{7} \qquad \mathtt{3} \otimes \mathtt{7} \rightarrow \mathtt{21}$$

Composing evaluation steps

So far we have said what steps our programs can take, one by one.

Now we want to talk about *sequences* of those steps, and their eventual outcomes.

Composing evaluation steps

So far we have said what steps our programs can take, one by one.

Now we want to talk about *sequences* of those steps, and their eventual outcomes.

Definition

We define the *multiple-step evaluation* relation $\rightarrow^\star$ as follows. We write $\mathtt{e} \rightarrow^\star \mathtt{e}'$ if either:

1. $\mathtt{e} = \mathtt{e}'$ or
2. there is a finite sequence $\mathtt{e} \rightarrow \mathtt{e}_1 \rightarrow \mathtt{e}_2 \rightarrow \cdots \rightarrow \mathtt{e}_k \rightarrow \mathtt{e}'$.

The relation $\rightarrow^\star$ is called *the reflexive and transitive closure of* $\rightarrow$.

If $\mathtt{e} \rightarrow^\star \mathtt{n}$ we say that $\mathtt{n}$ is the *final answer* of $\mathtt{e}$.

Composing evaluation steps

So far we have said what steps our programs can take, one by one.

Now we want to talk about *sequences* of those steps, and their eventual outcomes.

Definition

We define the *multiple-step evaluation* relation $\rightarrow^\star$ as follows. We write $\mathtt{e} \rightarrow^\star \mathtt{e}'$ if either:

1. $\mathtt{e} = \mathtt{e}'$ or
2. there is a finite sequence $\mathtt{e} \rightarrow \mathtt{e}_1 \rightarrow \mathtt{e}_2 \rightarrow \cdots \rightarrow \mathtt{e}_k \rightarrow \mathtt{e}'$.

The relation $\rightarrow^\star$ is called *the reflexive and transitive closure of* $\rightarrow$.

If $\mathtt{e} \rightarrow^\star \mathtt{n}$ we say that $\mathtt{n}$ is the *final answer* of $\mathtt{e}$.

Question: what is the final answer of $(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4})$?

Composing evaluation steps

So far we have said what steps our programs can take, one by one.

Now we want to talk about *sequences* of those steps, and their eventual outcomes.

Definition

We define the *multiple-step evaluation* relation $\rightarrow^\star$ as follows. We write $\mathtt{e} \rightarrow^\star \mathtt{e}'$ if either:

1. $\mathtt{e} = \mathtt{e}'$ or
2. there is a finite sequence $\mathtt{e} \rightarrow \mathtt{e}_1 \rightarrow \mathtt{e}_2 \rightarrow \cdots \rightarrow \mathtt{e}_k \rightarrow \mathtt{e}'$.

The relation $\rightarrow^\star$ is called *the reflexive and transitive closure of* $\rightarrow$.

If $\mathtt{e} \rightarrow^\star \mathtt{n}$ we say that $\mathtt{n}$ is the *final answer* of $\mathtt{e}$.

Question: what is the final answer of $(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4})$?

Question: is it true that every expression has a final answer? Are there expressions with more than one final answer?

Summarising small-step operational semantics

- We describe individual computation steps using the $\to$ relation;
- We understand the final answer of a computation as the end-result of applying $\to$ as many times as we can.

Summarising small-step operational semantics

- We describe individual computation steps using the $\rightarrow$ relation;
- We understand the final answer of a computation as the end-result of applying $\rightarrow$ as many times as we can.

This is very fine-grained, but sometimes can be more than we need.

Next we'll look at the big-step approach, where we only describe the final answer of a computation.

Big-step Operational Semantics

Big-step semantics describes the overall result of an execution. We write

$$\mathtt{e} \Downarrow \mathtt{n}$$

to mean "$\mathtt{e}$ is eventually evaluated to $\mathtt{n}$".

Big-step Operational Semantics

Big-step semantics describes the overall result of an execution. We write

$$\texttt{e} \Downarrow \texttt{n}$$

to mean "$\texttt{e}$ is eventually evaluated to $\texttt{n}$".

Big-step rules

$$\frac{}{\texttt{n} \Downarrow \texttt{n}}\ (num) \qquad \frac{\texttt{e}_0 \Downarrow \texttt{n}_0 \quad \texttt{e}_1 \Downarrow \texttt{n}_1}{\texttt{e}_0 \oplus \texttt{e}_1 \Downarrow \texttt{n}}\ n{=}n_0{+}n_1\ (sum) \qquad \frac{\texttt{e}_0 \Downarrow \texttt{n}_0 \quad \texttt{e}_1 \Downarrow \texttt{n}_1}{\texttt{e}_0 \otimes \texttt{e}_1 \Downarrow \texttt{n}}\ n{=}n_0{\times}n_1\ (prod)$$

Big-step Operational Semantics

Big-step semantics describes the overall result of an execution. We write

$$\mathtt{e} \Downarrow \mathtt{n}$$

to mean "$\mathtt{e}$ is eventually evaluated to $\mathtt{n}$".

Big-step rules

$$\frac{}{\mathtt{n} \Downarrow \mathtt{n}}\ (num) \qquad \frac{\mathtt{e}_0 \Downarrow \mathtt{n}_0 \quad \mathtt{e}_1 \Downarrow \mathtt{n}_1}{\mathtt{e}_0 \oplus \mathtt{e}_1 \Downarrow \mathtt{n}}\ n{=}n_0{+}n_1\ (sum) \qquad \frac{\mathtt{e}_0 \Downarrow \mathtt{n}_0 \quad \mathtt{e}_1 \Downarrow \mathtt{n}_1}{\mathtt{e}_0 \otimes \mathtt{e}_1 \Downarrow \mathtt{n}}\ n{=}n_0{\times}n_1\ (prod)$$

This hides the information about evaluation order, but for our language we can prove it doesn't matter.

A benefit: much fewer rules!

Big-step operational semantics

We can prove that $(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4}) \Downarrow \mathtt{21}$:

Big-step operational semantics

We can prove that $(1 \oplus 2) \otimes (3 \oplus 4) \Downarrow 21$:

$$\dfrac{\dfrac{\dfrac{}{1 \Downarrow 1}\,(num) \quad \dfrac{}{2 \Downarrow 2}\,(num)}{1 \oplus 2 \Downarrow 3}\,{\scriptstyle 3=1+2}\,(sum) \qquad \dfrac{\dfrac{}{3 \Downarrow 3}\,(num) \quad \dfrac{}{4 \Downarrow 4}\,(num)}{3 \oplus 4 \Downarrow 7}\,{\scriptstyle 7=3+4}\,(sum)}{(1 \oplus 2) \otimes (3 \oplus 4) \Downarrow 21}\,{\scriptstyle 21=3\times 7}\,(prod)$$

Evaluating operational semantics

Pros:

- Fine-grained description of program behaviour
- Makes precise our intuition about which steps the program makes when: very useful when multiple features are involved
- Quite easy to write down (e.g. you could feed the rules into a proof assistant)

Evaluating operational semantics

Pros:

- Fine-grained description of program behaviour
- Makes precise our intuition about which steps the program makes when: very useful when multiple features are involved
- Quite easy to write down (e.g. you could feed the rules into a proof assistant)

Cons:

- Doesn't talk so much about the meaning of programs
- Harder to analyse programs' high-level structure
- More constrained set of proof techniques

Evaluating operational semantics

Pros:

- Fine-grained description of program behaviour
- Makes precise our intuition about which steps the program makes when: very useful when multiple features are involved
- Quite easy to write down (e.g. you could feed the rules into a proof assistant)

Cons:

- Doesn't talk so much about the meaning of programs
- Harder to analyse programs' high-level structure
- More constrained set of proof techniques

Next up: a denotational semantics for `NumExp`

Denotational semantics for `NumExp`

A denotational model attempts to say what a piece of program text “really means”.

Rather than thinking about the steps programs take, we think about their overall behaviour.

A denotational model attempts to say what a piece of program text “really means”.

Rather than thinking about the steps programs take, we think about their overall behaviour.

Remember our original intuitive description of what `NumExp`-expressions mean:

1. Every numeral $\mathtt{n}$ is evaluated to the corresponding number n;
2. To find the value associated with an expression of the form $\mathtt{e}_0 \oplus \mathtt{e}_1$ we evaluate the expressions $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the sum of the results;
3. To find the value associated with an expression of the form $\mathtt{e}_0 \otimes \mathtt{e}_1$ we evaluate the expressions $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the product of the results

A denotational model attempts to say what a piece of program text "really means".

Rather than thinking about the steps programs take, we think about their overall behaviour.

Remember our original intuitive description of what `NumExp`-expressions mean:

1. Every numeral $\mathtt{n}$ is evaluated to the corresponding number n;
2. To find the value associated with an expression of the form $\mathtt{e}_0 \oplus \mathtt{e}_1$ we evaluate the expressions $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the sum of the results;
3. To find the value associated with an expression of the form $\mathtt{e}_0 \otimes \mathtt{e}_1$ we evaluate the expressions $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the product of the results

What is a `NumExp`-expression "really" representing? A natural number!

From expressions to numbers

Let's turn our intuitive idea into something precise

Intuition

1. Every numeral `n` is evaluated to the corresponding number n;
2. To find the value associated with an expression of the form $\mathtt{e}_0 \oplus \mathtt{e}_1$ we evaluate the expressions $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the sum of the results;
3. To find the value associated with an expression of the form $\mathtt{e}_0 \otimes \mathtt{e}_1$ we evaluate the expressions $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the product of the results

Denotational semantics (idea: denotation = "true meaning")

From expressions to numbers

Let's turn our intuitive idea into something precise

Intuition

1. Every numeral $\mathtt{n}$ is evaluated to the corresponding number n;
2. To find the value associated with an expression of the form $\mathtt{e}_0 \oplus \mathtt{e}_1$ we evaluate the expressions $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the sum of the results;
3. To find the value associated with an expression of the form $\mathtt{e}_0 \otimes \mathtt{e}_1$ we evaluate the expressions $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the product of the results

Denotational semantics (idea: denotation = "true meaning")

1. The denotation of $\mathtt{n}$ is the corresponding number n;
2. To find the denotation of $\mathtt{e}_0 \oplus \mathtt{e}_1$ we find the denotations of $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the sum of the results;
3. To find the denotation of $\mathtt{e}_0 \otimes \mathtt{e}_1$ we find the denotations of $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the product of the results

From expressions to numbers

Let's turn our intuitive idea into something precise

Notation: we write $[\![\mathtt{e}]\!]$ for the meaning, or denotation, of $\mathtt{e}$

Intuition

1. Every numeral $\mathtt{n}$ is evaluated to the corresponding number n;
2. To find the value associated with an expression of the form $\mathtt{e}_0 \oplus \mathtt{e}_1$ we evaluate the expressions $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the sum of the results;
3. To find the value associated with an expression of the form $\mathtt{e}_0 \otimes \mathtt{e}_1$ we evaluate the expressions $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the product of the results

Denotational semantics (idea: denotation = "true meaning")

1. The denotation of $\mathtt{n}$ is the corresponding number n;
2. To find the denotation of $\mathtt{e}_0 \oplus \mathtt{e}_1$ we find the denotations of $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the sum of the results;
3. To find the denotation of $\mathtt{e}_0 \otimes \mathtt{e}_1$ we find the denotations of $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the product of the results

From expressions to numbers

Let's turn our intuitive idea into something precise

Notation: we write $[\![\mathtt{e}]\!]$ for the meaning, or denotation, of $\mathtt{e}$

Intuition

1. Every numeral $\mathtt{n}$ is evaluated to the corresponding number n;
2. To find the value associated with an expression of the form $\mathtt{e}_0 \oplus \mathtt{e}_1$ we evaluate the expressions $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the sum of the results;
3. To find the value associated with an expression of the form $\mathtt{e}_0 \otimes \mathtt{e}_1$ we evaluate the expressions $\mathtt{e}_0$ and $\mathtt{e}_1$ and take the product of the results

Denotational semantics (idea: denotation = "true meaning")

1. $[\![\mathtt{n}]\!] := n$;
2. $[\![\mathtt{e}_0 \oplus \mathtt{e}_1]\!] := [\![\mathtt{e}_0]\!] + [\![\mathtt{e}_1]\!]$;
3. $[\![\mathtt{e}_0 \otimes \mathtt{e}_1]\!] := [\![\mathtt{e}_0]\!] \times [\![\mathtt{e}_1]\!]$

Denotational semantics of `NumExp`

The denotation of an expression $\mathtt{e}$ is given *by recursion*:

1. $[\![\mathtt{n}]\!] := n$;
2. $[\![\mathtt{e}_0 \oplus \mathtt{e}_1]\!] := [\![\mathtt{e}_0]\!] + [\![\mathtt{e}_1]\!]$;
3. $[\![\mathtt{e}_0 \otimes \mathtt{e}_1]\!] := [\![\mathtt{e}_0]\!] \times [\![\mathtt{e}_1]\!]$

Denotational semantics of `NumExp`

The denotation of an expression $\mathtt{e}$ is given *by recursion*:

1. $[\![\mathtt{n}]\!] := n$;
2. $[\![\mathtt{e}_0 \oplus \mathtt{e}_1]\!] := [\![\mathtt{e}_0]\!] + [\![\mathtt{e}_1]\!]$;
3. $[\![\mathtt{e}_0 \otimes \mathtt{e}_1]\!] := [\![\mathtt{e}_0]\!] \times [\![\mathtt{e}_1]\!]$

Example:

$$\begin{aligned}
[\![(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4})]\!] &= [\![\mathtt{1} \oplus \mathtt{2}]\!] \times [\![\mathtt{3} \oplus \mathtt{4}]\!] \\
&= ([\![\mathtt{1}]\!] + [\![\mathtt{2}]\!]) \times ([\![\mathtt{3}]\!] + [\![\mathtt{4}]\!]) \\
&= (1 + 2) \times (3 + 4) \\
&= 21.
\end{aligned}$$

Denotational semantics of `NumExp`

The denotation of an expression $\mathtt{e}$ is given *by recursion*:

1. $[\![\mathtt{n}]\!] := n$;
2. $[\![\mathtt{e}_0 \oplus \mathtt{e}_1]\!] := [\![\mathtt{e}_0]\!] + [\![\mathtt{e}_1]\!]$;
3. $[\![\mathtt{e}_0 \otimes \mathtt{e}_1]\!] := [\![\mathtt{e}_0]\!] \times [\![\mathtt{e}_1]\!]$

The definition is compositional: the meaning of an expression is built up from the meanings of sub-expressions

Denotational semantics of `NumExp`

The denotation of an expression $\mathtt{e}$ is given *by recursion*:

1. $[\![\mathtt{n}]\!] := n$;
2. $[\![\mathtt{e}_0 \oplus \mathtt{e}_1]\!] := [\![\mathtt{e}_0]\!] + [\![\mathtt{e}_1]\!]$;
3. $[\![\mathtt{e}_0 \otimes \mathtt{e}_1]\!] := [\![\mathtt{e}_0]\!] \times [\![\mathtt{e}_1]\!]$

The definition is compositional: the meaning of an expression is built up from the meanings of sub-expressions

This defines a function $\mathit{Exp} \to \mathbb{N}$ from the set of `NumExp`-expressions to the set of natural numbers.

Here $\mathbb{N}$ is the semantic domain. By changing the semantic domain, or the denotation of expressions, we can study different kinds of properties.

Using the denotational semantics

Intuitively, the behaviour of an expression only using $\oplus$ shouldn't depend on the order of brackets.

Using the denotational semantics

Intuitively, the behaviour of an expression only using $\oplus$ shouldn't depend on the order of brackets.

We can make this intuition precise:

Proposition

For all `NumExp`-*expressions* $\mathtt{e}_1$, $\mathtt{e}_2$, $\mathtt{e}_3$, $[\![(\mathtt{e}_1 \oplus \mathtt{e}_2) \oplus \mathtt{e}_3]\!] = [\![\mathtt{e}_1 \oplus (\mathtt{e}_2 \oplus \mathtt{e}_3)]\!]$.

Using the denotational semantics

Intuitively, the behaviour of an expression only using $\oplus$ shouldn't depend on the order of brackets.

We can make this intuition precise:

Proposition

For all `NumExp`*-expressions* e_1, e_2, e_3, $[\![(\mathsf{e}_1 \oplus \mathsf{e}_2) \oplus \mathsf{e}_3]\!] = [\![\mathsf{e}_1 \oplus (\mathsf{e}_2 \oplus \mathsf{e}_3)]\!]$.

Proof.

$$\begin{aligned}
[\![(\mathsf{e}_1 \oplus \mathsf{e}_2) \oplus \mathsf{e}_3]\!] &= [\![\mathsf{e}_1 \oplus \mathsf{e}_2]\!] + [\![\mathsf{e}_3]\!] \\
&= ([\![\mathsf{e}_1]\!] + [\![\mathsf{e}_2]\!]) + [\![\mathsf{e}_3]\!] \\
&= [\![\mathsf{e}_1]\!] + ([\![\mathsf{e}_2]\!] + [\![\mathsf{e}_3]\!]) \\
&= [\![\mathsf{e}_1]\!] + [\![\mathsf{e}_2 \oplus \mathsf{e}_3]\!] \\
&= [\![\mathsf{e}_1 \oplus (\mathsf{e}_2 \oplus \mathsf{e}_3)]\!]
\end{aligned}$$

$\square$

Tying things together

We've seen two perspectives on `NumExp`-expressions:

1. Operational: the steps they take in running to a final answer;
2. Denotational: the meaning of the expression, expressed as a natural number.

Tying things together

We've seen two perspectives on `NumExp`-expressions:

1. Operational: the steps they take in running to a final answer;
2. Denotational: the meaning of the expression, expressed as a natural number.

How do these relate?

A correspondence theorem

Theorem

For all expressions $\mathtt{e}$ *and numbers* n, $[\![\mathtt{e}]\!] = n$ *if and only if* $\mathtt{e} \Downarrow \mathtt{n}$.

A correspondence theorem

Theorem

For all expressions $\texttt{e}$ *and numbers* n, $[\![\texttt{e}]\!] = n$ *if and only if* $\texttt{e} \Downarrow \texttt{n}$.

Proof.

By structural induction on $\texttt{e}$.

A correspondence theorem

Theorem

For all expressions $\mathtt{e}$ *and numbers* n, $[\![\mathtt{e}]\!] = n$ *if and only if* $\mathtt{e} \Downarrow \mathtt{n}$.

Proof.

By structural induction on $\mathtt{e}$.

Base case: if $\mathtt{e}$ is a numeral, say $\mathtt{n}$, then both $[\![\mathtt{e}]\!] = n$ and $\mathtt{e} \Downarrow \mathtt{n}$.

A correspondence theorem

Theorem

For all expressions $\texttt{e}$ *and numbers* n*,* $[\![\texttt{e}]\!] = n$ *if and only if* $\texttt{e} \Downarrow \texttt{n}$*.*

Proof.

By structural induction on $\texttt{e}$.
Base case: if $\texttt{e}$ is a numeral, say $\texttt{n}$, then both $[\![\texttt{e}]\!] = n$ and $\texttt{e} \Downarrow \texttt{n}$.
Inductive step: we suppose that $\texttt{e} = \texttt{e}_1 \oplus \texttt{e}_2$ and that the theorem holds for both $\texttt{e}_1$ and $\texttt{e}_2$, that is:

1. We assume that for all numbers k_1 and k_2, $[\![\texttt{e}_1]\!] = k_1$ if and only if $\texttt{e}_1 \Downarrow \texttt{k}_1$ and $[\![\texttt{e}_2]\!] = k_2$ if and only if $\texttt{e}_2 \Downarrow \texttt{k}_2$,
2. We prove that for all numbers n, $[\![\texttt{e}_1 \oplus \texttt{e}_2]\!] = n$ if and only if $\texttt{e}_1 \oplus \texttt{e}_2 \Downarrow \texttt{n}$.

A correspondence theorem

Proof.

1. We assume that for all numbers k_1 and k_2, $[\![\mathtt{e}_1]\!] = k_1$ if and only if $\mathtt{e}_1 \Downarrow \mathtt{k}_1$ and $[\![\mathtt{e}_2]\!] = k_2$ if and only if $\mathtt{e}_2 \Downarrow \mathtt{k}_2$,
2. We prove that for all numbers n, $[\![\mathtt{e}_1 \oplus \mathtt{e}_2]\!] = n$ if and only if $\mathtt{e}_1 \oplus \mathtt{e}_2 \Downarrow \mathtt{n}$.

A correspondence theorem

Proof.

1. We assume that for all numbers k_1 and k_2, $[\![\mathtt{e}_1]\!] = k_1$ if and only if $\mathtt{e}_1 \Downarrow \mathtt{k}_1$ and $[\![\mathtt{e}_2]\!] = k_2$ if and only if $\mathtt{e}_2 \Downarrow \mathtt{k}_2$,
2. We prove that for all numbers n, $[\![\mathtt{e}_1 \oplus \mathtt{e}_2]\!] = n$ if and only if $\mathtt{e}_1 \oplus \mathtt{e}_2 \Downarrow \mathtt{n}$.

Suppose that $[\![\mathtt{e}_1 \oplus \mathtt{e}_2]\!] = n$. By definition, $[\![\mathtt{e}_1 \oplus \mathtt{e}_2]\!] = [\![\mathtt{e}_1]\!] + [\![\mathtt{e}_2]\!]$. Then $[\![\mathtt{e}_1]\!]$ and $[\![\mathtt{e}_2]\!]$ are two numbers, let's call them k_1 and k_2, that add up to n. By (1), $\mathtt{e}_1 \Downarrow \mathtt{k}_1$ and $\mathtt{e}_2 \Downarrow \mathtt{k}_2$, which by the rule *sum* of big step operational semantics means that $\mathtt{e}_1 \oplus \mathtt{e}_2 \Downarrow \mathtt{n}$.

A correspondence theorem

Proof.

1. We assume that for all numbers k_1 and k_2, $[\![\texttt{e}_1]\!] = k_1$ if and only if $\texttt{e}_1 \Downarrow \texttt{k}_1$ and $[\![\texttt{e}_2]\!] = k_2$ if and only if $\texttt{e}_2 \Downarrow \texttt{k}_2$,
2. We prove that for all numbers n, $[\![\texttt{e}_1 \oplus \texttt{e}_2]\!] = n$ if and only if $\texttt{e}_1 \oplus \texttt{e}_2 \Downarrow \texttt{n}$.

Suppose that $[\![\texttt{e}_1 \oplus \texttt{e}_2]\!] = n$. By definition, $[\![\texttt{e}_1 \oplus \texttt{e}_2]\!] = [\![\texttt{e}_1]\!] + [\![\texttt{e}_2]\!]$. Then $[\![\texttt{e}_1]\!]$ and $[\![\texttt{e}_2]\!]$ are two numbers, let's call them k_1 and k_2, that add up to n. By (1), $\texttt{e}_1 \Downarrow \texttt{k}_1$ and $\texttt{e}_2 \Downarrow \texttt{k}_2$, which by the rule *sum* of big step operational semantics means that $\texttt{e}_1 \oplus \texttt{e}_2 \Downarrow \texttt{n}$.

Conversely, suppose that $\texttt{e}_1 \oplus \texttt{e}_2 \Downarrow \texttt{n}$. This can only be proved using the *sum* rule, so there must be numbers k_1 and k_2 such that $n = k_1 + k_2$ and $\texttt{e}_1 \Downarrow \texttt{k}_1$ and $\texttt{e}_2 \Downarrow \texttt{k}_2$. By (1) then $[\![\texttt{e}_1]\!] = k_1$ and $[\![\texttt{e}_2]\!] = k_2$. Hence

$$[\![\texttt{e}_1 \oplus \texttt{e}_2]\!] = [\![\texttt{e}_1]\!] + [\![\texttt{e}_2]\!] = k_1 + k_2 = n. \qquad \square$$

Theorem

For all $\texttt{NumExp}$*-expressions* $\texttt{e}$ *and numbers* n*,* $[\![\texttt{e}]\!] = n$ *if and only if* $\texttt{e} \Downarrow \texttt{n}$*.*

Theorem

For all $\texttt{NumExp}$*-expressions* $\texttt{e}$ *and numbers* n, $[\![\texttt{e}]\!] = n$ *if and only if* $\texttt{e} \Downarrow \texttt{n}$.

Corollary

For all $\texttt{NumExp}$*-expressions* $\texttt{e}_1$, $\texttt{e}_2$, $\texttt{e}_3$,

$$(\texttt{e}_1 \oplus \texttt{e}_2) \oplus \texttt{e}_3 \Downarrow \texttt{n} \quad \textit{if and only if} \quad \texttt{e}_1 \oplus (\texttt{e}_2 \oplus \texttt{e}_3) \Downarrow \texttt{n}$$

Theorem

For all $\texttt{NumExp}$*-expressions* $\mathtt{e}$ *and numbers* n, $[\![\mathtt{e}]\!] = n$ *if and only if* $\mathtt{e} \Downarrow \mathtt{n}$.

Corollary

For all $\texttt{NumExp}$*-expressions* $\mathtt{e}_1$, $\mathtt{e}_2$, $\mathtt{e}_3$,

$$(\mathtt{e}_1 \oplus \mathtt{e}_2) \oplus \mathtt{e}_3 \Downarrow \mathtt{n} \quad \textit{if and only if} \quad \mathtt{e}_1 \oplus (\mathtt{e}_2 \oplus \mathtt{e}_3) \Downarrow \mathtt{n}$$

Proof.

$$\begin{aligned} (\mathtt{e}_1 \oplus \mathtt{e}_2) \oplus \mathtt{e}_3 \Downarrow n \quad & \text{if and only if} \quad [\![(\mathtt{e}_1 \oplus \mathtt{e}_2) \oplus \mathtt{e}_3]\!] = n \\ & \text{if and only if} \quad [\![\mathtt{e}_1 \oplus (\mathtt{e}_2 \oplus \mathtt{e}_3)]\!] = n \\ & \text{if and only if} \quad \mathtt{e}_1 \oplus (\mathtt{e}_2 \oplus \mathtt{e}_3) \Downarrow \mathtt{n} \end{aligned}$$

□

What we've seen

For the language `NumExp`, defined by

$$\begin{array}{llll} \textit{Num} & \mathtt{n} & ::= & \mathtt{0} \mid \mathtt{1} \mid \mathtt{2} \mid \ldots \\ \textit{Exp} & \mathtt{e} & ::= & \mathtt{n} \mid \mathtt{e} \oplus \mathtt{e} \mid \mathtt{e} \otimes \mathtt{e} \end{array}$$

we've seen two semantic approaches:

1. **Operational:** the steps they take in running to a final answer (both small-step and big-step);
2. **Denotational:** the meaning of the expression, expressed as a natural number.

And we've proven these agree. This can be read in two ways:

1. Our operational semantics correctly captures the 'meaning' of programs;
2. Our denotational semantics correctly captures the way programs run;

`NumExp` is a very limited language!

What other features might we want?

`NumExp` is a very limited language!

What other features might we want?

Possible examples:

- variables
- other forms of data (arrays / lists, boolean values, . . .)
- other control structures (iteration / loops, if-statements, handlers, `call-cc`, . . .)
- procedures / methods / user-defined functions
- interaction with the world: state, I/O, printing, exceptions, probability. . .
- abstract data types (e.g. to define trees, lists, . . .)
- higher-order functions
- types
- polymorphism

`NumExp` is a very limited language!

What other features might we want?

Possible examples:

- variables
- other forms of data (arrays / lists, boolean values, ...)
- other control structures (iteration / loops, if-statements, handlers, `call-cc`, ...)
- procedures / methods / user-defined functions
- interaction with the world: state, I/O, printing, exceptions, probability...
- abstract data types (e.g. to define trees, lists, ...)
- higher-order functions
- types
- polymorphism

Challenge: think about how these things work *together*!

`NumExp` is a very limited language!

What other features might we want?

In what follows, we'll think about operational and denotational perspectives on some of the simpler cases.

Semantics for variables

As it stands, we can only write things like $(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4})$.

As it stands, we can only write things like $(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4})$.

But we know some things are true no matter what numbers we put in, such as:

- The associativity rule we proved above
- The meaning of $a \oplus b$ should be the same as that of $b \oplus a$ whatever a and b are

As it stands, we can only write things like $(\mathtt{1} \oplus \mathtt{2}) \otimes (\mathtt{3} \oplus \mathtt{4})$.

But we know some things are true no matter what numbers we put in, such as:

- The associativity rule we proved above
- The meaning of $a \oplus b$ should be the same as that of $b \oplus a$ whatever a and b are

To understand these things properly we need variables

What is a variable anyway?

We will use variables in our language as a way to stand for any possible number
(like in high-school algebra)

This is a *functional* perspective. It might not be what you're used to!

Contrast to an *imperative* language, where a variable is a pointer to a memory cell

Adding variables to NumExp

The language $\texttt{NumExp} + \texttt{Var}$ is defined by

$$
\begin{array}{llcl}
\mathit{Num} & \texttt{n} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \\
\mathit{Exp} & \texttt{e} & ::= & \texttt{x} \mid \texttt{n} \mid \texttt{e} \oplus \texttt{e} \mid \texttt{e} \otimes \texttt{e}
\end{array}
$$

We assume $\texttt{x}$ stands for anything in a fixed stock of variables $\texttt{x}, \texttt{y}, \texttt{z}, \ldots$.

Adding variables to $\texttt{NumExp}$

The language $\texttt{NumExp} + \texttt{Var}$ is defined by

$$
\begin{array}{llll}
\mathit{Num} & \texttt{n} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \\
\mathit{Exp} & \texttt{e} & ::= & \texttt{x} \mid \texttt{n} \mid \texttt{e} \oplus \texttt{e} \mid \texttt{e} \otimes \texttt{e}
\end{array}
$$

We assume $\texttt{x}$ stands for anything in a fixed stock of variables $\texttt{x}, \texttt{y}, \texttt{z}, \ldots$.

Now we can write things like

$$\texttt{x} \oplus \texttt{2} \qquad \texttt{y} \oplus \texttt{2} \qquad (\texttt{7} \oplus \texttt{x}) \otimes \texttt{y}$$

Adding variables to $\texttt{NumExp}$

The language $\texttt{NumExp} + \texttt{Var}$ is defined by

$$
\begin{array}{llcl}
\mathit{Num} & \texttt{n} & ::= & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \ldots \\
\mathit{Exp} & \texttt{e} & ::= & \texttt{x} \mid \texttt{n} \mid \texttt{e} \oplus \texttt{e} \mid \texttt{e} \otimes \texttt{e}
\end{array}
$$

We assume $\texttt{x}$ stands for anything in a fixed stock of variables $\texttt{x}, \texttt{y}, \texttt{z}, \ldots$.

Now we can write things like

$$\texttt{x} \oplus \texttt{2} \qquad \texttt{y} \oplus \texttt{2} \qquad (\texttt{7} \oplus \texttt{x}) \otimes \texttt{y}$$

An expression $\texttt{e}$ is open if it contains a variable (e.g. $\texttt{x}$, $\texttt{2} \oplus \texttt{y}$, $\ldots$)

If it contains no variables, it is closed (e.g. $\texttt{7} \oplus \texttt{3}$)

Operational semantics for variables

We already know how closed terms run. What about open ones?

Operational semantics for variables

We already know how closed terms run. What about open ones?

For example: how should $\mathtt{x} \oplus (\mathtt{3} \otimes \mathtt{7})$ run?

Operational semantics for variables

We already know how closed terms run. What about open ones?

For example: how should $\mathtt{x} \oplus (\mathtt{3} \otimes \mathtt{7})$ run?

If we know the value of $\mathtt{x}$, we know what to do

But if $\mathtt{x}$ can be anything, the final result can also be lots of things

(but not anything: why not?)

Operational semantics for variables

We already know how closed terms run. What about open ones?

For example: how should $\mathtt{x} \oplus (\mathtt{3} \otimes \mathtt{7})$ run?

If we know the value of `x`, we know what to do

But if `x` can be anything, the final result can also be lots of things
(but not anything: why not?)

So `x` is *incomplete*: we don't know how it runs till you tell me what `x` is

The small-step rules for `NumExp` + `Var` are therefore exactly those of `NumExp`:
we give no rules for running variables

Small-step rules for $\texttt{NumExp} + \texttt{Var}$

$$\frac{}{\mathtt{n}_0 \oplus \mathtt{n}_1 \to \mathtt{n}}\ n = n_0 + n_1 (sum) \qquad \frac{\mathsf{e}_0 \to \mathsf{e}_0'}{\mathsf{e}_0 \oplus \mathsf{e}_1 \to \mathsf{e}_0' \oplus \mathsf{e}_1}\ (sumL) \qquad \frac{\mathsf{e}_1 \to \mathsf{e}_1'}{\mathsf{e}_0 \oplus \mathsf{e}_1 \to \mathsf{e}_0 \oplus \mathsf{e}_1'}\ (sumR)$$

$$\frac{}{\mathtt{n}_0 \otimes \mathtt{n}_1 \to \mathtt{n}}\ n = n_0 \times n_1 (prod) \qquad \frac{\mathsf{e}_0 \to \mathsf{e}_0'}{\mathsf{e}_0 \otimes \mathsf{e}_1 \to \mathsf{e}_0' \otimes \mathsf{e}_1}\ (prodL) \qquad \frac{\mathsf{e}_1 \to \mathsf{e}_1'}{\mathsf{e}_0 \otimes \mathsf{e}_1 \to \mathsf{e}_0 \otimes \mathsf{e}_1'}\ (prodR)$$

Examples of valid $\texttt{NumExp} + \texttt{Var}$-derivations:

$$(\texttt{1} \oplus \texttt{2}) \otimes \texttt{x} \to \texttt{3} \otimes \texttt{x} \qquad\qquad (\texttt{1} \oplus \texttt{y}) \otimes (\texttt{3} \oplus \texttt{4}) \to (\texttt{1} \oplus \texttt{y}) \otimes \texttt{7}$$

with their derivations:

$$\dfrac{\dfrac{}{\texttt{1} \oplus \texttt{2} \to \texttt{3}}\ {\scriptstyle 3=1+2\,(sum)}}{(\texttt{1} \oplus \texttt{2}) \otimes \texttt{x} \to \texttt{3} \otimes \texttt{x}}\ (prodL) \qquad \dfrac{\dfrac{}{\texttt{3} \oplus \texttt{4} \to \texttt{7}}\ {\scriptstyle 7=3+4\,(sum)}}{(\texttt{1} \oplus \texttt{y}) \otimes (\texttt{3} \oplus \texttt{4}) \to (\texttt{1} \oplus \texttt{y}) \otimes \texttt{7}}\ (prodR)$$

Big-step rules for `NumExp` + `Var`

For the big-step rules, we want to say what the eventual ouput of an expression is.

For an open term we don't know what that is. So we restrict our big-step rules to closed terms.

Big-step rules for $\texttt{NumExp} + \texttt{Var}$

For the big-step rules, we want to say what the eventual ouput of an expression is.

For an open term we don't know what that is. So we restrict our big-step rules to closed terms.

Big-step rules (just as for $\texttt{NumExp}$)

$$\frac{}{\mathtt{n} \Downarrow \mathtt{n}}\ (\mathit{num}) \qquad \frac{\mathtt{e}_0 \Downarrow \mathtt{n}_0 \quad \mathtt{e}_1 \Downarrow \mathtt{n}_1}{\mathtt{e}_0 \oplus \mathtt{e}_1 \Downarrow \mathtt{n}}\ {\scriptstyle n = n_0 + n_1}\ (\mathit{sum}) \qquad \frac{\mathtt{e}_0 \Downarrow \mathtt{n}_0 \quad \mathtt{e}_1 \Downarrow \mathtt{n}_1}{\mathtt{e}_0 \otimes \mathtt{e}_1 \Downarrow \mathtt{n}}\ {\scriptstyle n = n_0 \times n_1}\ (\mathit{prod})$$

Big-step rules for $\texttt{NumExp}$ + $\texttt{Var}$

For the big-step rules, we want to say what the eventual ouput of an expression is.

For an open term we don't know what that is. So we restrict our big-step rules to closed terms.

Big-step rules (just as for $\texttt{NumExp}$)

$$\frac{}{\mathtt{n} \Downarrow \mathtt{n}}\ (\mathit{num}) \qquad \frac{\mathtt{e}_0 \Downarrow \mathtt{n}_0 \quad \mathtt{e}_1 \Downarrow \mathtt{n}_1}{\mathtt{e}_0 \oplus \mathtt{e}_1 \Downarrow \mathtt{n}}\ n{=}n_0{+}n_1\ (\mathit{sum}) \qquad \frac{\mathtt{e}_0 \Downarrow \mathtt{n}_0 \quad \mathtt{e}_1 \Downarrow \mathtt{n}_1}{\mathtt{e}_0 \otimes \mathtt{e}_1 \Downarrow \mathtt{n}}\ n{=}n_0{\times}n_1\ (\mathit{prod})$$

An alternative approach would be to say variables don't run:

$$\frac{}{\mathtt{x} \Downarrow \mathtt{x}}\ (\mathit{var})$$

Denotational semantics for variables

The meaning of a closed term is a number:

$$[\![\mathsf{7} \oplus (\mathsf{3} \otimes \mathsf{1})]\!] = [\![\mathsf{7}]\!] + [\![\mathsf{3} \otimes \mathsf{1}]\!] = [\![\mathsf{7}]\!] + ([\![\mathsf{3}]\!] \times [\![\mathsf{1}]\!]) = 21$$

Denotational semantics for variables

The meaning of a closed term is a number:

$$[\![\mathsf{7} \oplus (\mathsf{3} \otimes \mathsf{1})]\!] = [\![\mathsf{7}]\!] + [\![\mathsf{3} \otimes \mathsf{1}]\!] = [\![\mathsf{7}]\!] + ([\![\mathsf{3}]\!] \times [\![\mathsf{1}]\!]) = 21$$

What is the meaning of an open term?

Denotational semantics for variables

The meaning of a closed term is a number:

$$[\![\mathtt{7} \oplus (\mathtt{3} \otimes \mathtt{1})]\!] = [\![\mathtt{7}]\!] + [\![\mathtt{3} \otimes \mathtt{1}]\!] = [\![\mathtt{7}]\!] + ([\![\mathtt{3}]\!] \times [\![\mathtt{1}]\!]) = 21$$

What is the meaning of an open term?

It depends on the values of the variables: on the environment

Definition

An environment ρ for a $\mathtt{NumExp} + \mathtt{Var}$-expression $\mathtt{e}$ is a function assigning to each variable in $\mathtt{e}$ a natural number. Write $\mathit{Env}(\mathtt{e})$ for the set of environments of $\mathtt{e}$.

Formally: ρ is a partial function $\mathit{Vars} \rightharpoonup \mathbb{N}$ which is defined on all the variables in $\mathtt{e}$.

Denotational semantics for variables

Definition

An environment ρ for a $\texttt{NumExp} + \texttt{Var}$-expression $\texttt{e}$ is a function assigning to each variable in $\texttt{e}$ a natural number. Write $\mathit{Env}(\texttt{e})$ for the set of environments of $\texttt{e}$.

Formally: ρ is a partial function $\mathit{Vars} \rightharpoonup \mathbb{N}$ which is defined on just the variables in $\texttt{e}$.

Denotational semantics for variables

Definition

An environment ρ for a `NumExp` + `Var`-expression `e` is a function assigning to each variable in `e` a natural number. Write $\mathit{Env}(\mathtt{e})$ for the set of environments of `e`.

Formally: ρ is a partial function $\mathit{Vars} \rightharpoonup \mathbb{N}$ which is defined on just the variables in `e`.

Suppose the environment ρ assigns $\mathtt{x} \mapsto 1$. What is the meaning of `x` in this environment?

Denotational semantics for variables

Definition

An environment ρ for a $\texttt{NumExp} + \texttt{Var}$-expression $\texttt{e}$ is a function assigning to each variable in $\texttt{e}$ a natural number. Write $\mathit{Env}(\texttt{e})$ for the set of environments of $\texttt{e}$.

Formally: ρ is a partial function $\mathit{Vars} \rightharpoonup \mathbb{N}$ which is defined on just the variables in $\texttt{e}$.

Suppose the environment ρ assigns $\texttt{x} \mapsto 1$. What is the meaning of $\texttt{x}$ in this environment?

So the meaning of a variable $\texttt{x}$ depends on which environment you have: for each environment ρ we get a value $\rho(\texttt{x})$.

Denotational semantics for variables

Definition

An environment ρ for a $\texttt{NumExp} + \texttt{Var}$-expression $\texttt{e}$ is a function assigning to each variable in $\texttt{e}$ a natural number. Write $\mathit{Env}(\texttt{e})$ for the set of environments of $\texttt{e}$.

Formally: ρ is a partial function $\mathit{Vars} \rightharpoonup \mathbb{N}$ which is defined on just the variables in $\texttt{e}$.

Suppose the environment ρ assigns $\texttt{x} \mapsto 1$. What is the meaning of $\texttt{x}$ in this environment?

So the meaning of a variable $\texttt{x}$ depends on which environment you have: for each environment ρ we get a value $\rho(\texttt{x})$.

So $[\![\texttt{x}]\!]$ is a *function* $\mathit{Env}(\texttt{x}) \to \mathbb{N}$

Denotational semantics for variables

Given an environment ρ, what is the meaning of $\mathtt{7} \oplus (\mathtt{x} \oplus \mathtt{3})$?

Denotational semantics for variables

Given an environment ρ, what is the meaning of $\mathtt{7} \oplus (\mathtt{x} \oplus \mathtt{3})$?

It should be $7 + (\rho(\mathtt{x}) + 3)$.

This also depends on ρ!

Denotational semantics for variables

Given an environment ρ, what is the meaning of $\mathtt{7} \oplus (\mathtt{x} \oplus \mathtt{3})$?

It should be $7 + (\rho(\mathtt{x}) + 3)$.

This also depends on ρ!

The denotation of a `NumExp` + `Var`-expression `e` is a function $\mathit{Env}(\mathtt{e}) \to \mathbb{N}$.

Denotational semantics for variables

Given an environment ρ, what is the meaning of $\texttt{7} \oplus (\texttt{x} \oplus \texttt{3})$?

It should be $7 + (\rho(\texttt{x}) + 3)$.

This also depends on ρ!

The denotation of a `NumExp` + `Var`-expression `e` is a function $\mathit{Env}(\texttt{e}) \to \mathbb{N}$.

Example: given an environment ρ setting $\texttt{x} \mapsto 3$ and $\texttt{y} \mapsto 0$,
what is the denotation of $(\texttt{3} \otimes \texttt{x}) \otimes (\texttt{x} \oplus \texttt{y})$?

Denotational semantics for variables

Given an environment ρ, what is the meaning of $\texttt{7} \oplus (\texttt{x} \oplus \texttt{3})$?

It should be $7 + (\rho(\texttt{x}) + 3)$.

This also depends on ρ!

The denotation of a `NumExp` + `Var`-expression `e` is a function $\mathit{Env}(\texttt{e}) \to \mathbb{N}$.

Example: given an environment ρ setting $\texttt{x} \mapsto 3$ and $\texttt{y} \mapsto 0$,
what is the denotation of $(\texttt{3} \otimes \texttt{x}) \otimes (\texttt{x} \oplus \texttt{y})$?

More generally: what is the denotation of $(\texttt{3} \otimes \texttt{x}) \otimes (\texttt{x} \oplus \texttt{y})$?

Denotational semantics for `NumExp` + `Var`

Definition

An environment ρ for a `NumExp` + `Var`-expression `e` is a function assigning to each variable in `e` a natural number. Write $Env(\mathtt{e})$ for the set of environments of `e`.

Denotational semantics for NumExp + Var

Definition

An environment ρ for a $\texttt{NumExp} + \texttt{Var}$-expression $\texttt{e}$ is a function assigning to each variable in $\texttt{e}$ a natural number. Write $\mathit{Env}(\texttt{e})$ for the set of environments of $\texttt{e}$.

The denotation of a $\texttt{NumExp} + \texttt{Var}$-expression $\texttt{e}$ is a function $\mathit{Env}(\texttt{e}) \to \mathbb{N}$.
For an environment ρ in $\mathit{Env}(\texttt{e})$, the denotation of $\texttt{e}$ is given by recursion:

1. $[\![\texttt{x}]\!](\rho) := \rho(\texttt{x})$
2. $[\![\texttt{n}]\!](\rho) := n$;
3. $[\![\texttt{e}_0 \oplus \texttt{e}_1]\!](\rho) := [\![\texttt{e}_0]\!](\rho) + [\![\texttt{e}_1]\!](\rho)$;
4. $[\![\texttt{e}_0 \otimes \texttt{e}_1]\!](\rho) := [\![\texttt{e}_0]\!](\rho) \times [\![\texttt{e}_1]\!](\rho)$

Denotational semantics for $\texttt{NumExp} + \texttt{Var}$

Definition

An environment ρ for a $\texttt{NumExp} + \texttt{Var}$-expression $\texttt{e}$ is a function assigning to each variable in $\texttt{e}$ a natural number. Write $\mathit{Env}(\texttt{e})$ for the set of environments of $\texttt{e}$.

The denotation of a $\texttt{NumExp} + \texttt{Var}$-expression $\texttt{e}$ is a function $\mathit{Env}(\texttt{e}) \to \mathbb{N}$.
For an environment ρ in $\mathit{Env}(\texttt{e})$, the denotation of $\texttt{e}$ is given by recursion:

1. $[\![\texttt{x}]\!](\rho) := \rho(\texttt{x})$
2. $[\![\texttt{n}]\!](\rho) := n;$
3. $[\![\texttt{e}_0 \oplus \texttt{e}_1]\!](\rho) := [\![\texttt{e}_0]\!](\rho) + [\![\texttt{e}_1]\!](\rho);$
4. $[\![\texttt{e}_0 \otimes \texttt{e}_1]\!](\rho) := [\![\texttt{e}_0]\!](\rho) \times [\![\texttt{e}_1]\!](\rho)$

The definition is still compositional

Denotational semantics for $\texttt{NumExp} + \texttt{Var}$

The denotation of a $\texttt{NumExp} + \texttt{Var}$-expression $\texttt{e}$ is a function $\mathit{Env}(\texttt{e}) \to \mathbb{N}$.
For an environment ρ in $\mathit{Env}(\texttt{e})$, the denotation of $\texttt{e}$ is given by recursion:

1. $[\![\texttt{x}]\!](\rho) := \rho(\texttt{x})$
2. $[\![\texttt{n}]\!](\rho) := n$;
3. $[\![\texttt{e}_0 \oplus \texttt{e}_1]\!](\rho) := [\![\texttt{e}_0]\!](\rho) + [\![\texttt{e}_1]\!](\rho)$;
4. $[\![\texttt{e}_0 \otimes \texttt{e}_1]\!](\rho) := [\![\texttt{e}_0]\!](\rho) \times [\![\texttt{e}_1]\!](\rho)$

Example: what is the denotation of $\texttt{x} \oplus \texttt{y}$?

$$[\![\texttt{x} \oplus \texttt{y}]\!](\rho) = [\![\texttt{x}]\!](\rho) + [\![\texttt{y}]\!](\rho) = \rho(\texttt{x}) + \rho(\texttt{y})$$

What about $\texttt{x} \otimes \texttt{y}$?

Tying things together

Proposition

For any closed expression $\texttt{e}$*, we have* $\texttt{e} \Downarrow n$ *if and only if* $[\![\texttt{e}]\!] = \texttt{n}$.

Proven in the same way as for `NumExp`

Tying things together

Proposition

For any closed expression $\texttt{e}$, *we have* $\texttt{e} \Downarrow n$ *if and only if* $[\![\texttt{e}]\!] = \texttt{n}$.

Proven in the same way as for `NumExp`

Something more interesting:

Proposition

For any expression $\texttt{e}$ *containing the variables* $\texttt{x}_1, \ldots, \texttt{x}_k$, *and any environment* ρ *assigning* $\rho(\texttt{x}_1) = m_1, \ldots, \rho(\texttt{x}_k) = m_k$:

$$[\![\texttt{e}]\!](\rho) = \texttt{n} \quad \textit{if and only if} \quad \texttt{e}' \Downarrow \texttt{n}$$

where $\texttt{e}'$ *is obtained from* $\texttt{e}$ *by replacing each* $\texttt{x}_i$ *by* $\texttt{m}_\texttt{i}$.

Denotational semantics for `NumExp` + `Var`

When you have variables, the meaning of programs depends
on the values of the variables.

Open expressions don't run, but we can think about their operational semantics
by thinking about how we fill in the variables.

For the denotational semantics, we think of programs as depending on a choice of
environment. So programs are now interpreted as functions.

If we want to study different properties or features, we need different models. A lot
of the work in denotational semantics is coming up with these models: makes use of
powerful tools from mathematics (algebra, topology, logic, category theory, ...)