Basic mechanics of operational semantics

David Van Horn

UM

What is an operational semantics?

A method of defining the meaning of programs by describing the actions carried out during a program's execution.

There are many different flavors:

- Evaluator semantics
- Natural semantics, big-step
- Reduction semantics
- Abstract machine semantics

What is an operational semantics used for?

- Specifying a programming language
- Communicating language design ideas
- Validating claims about languages
- Validating claims about type systems, etc
- Proving correctness of a compiler
- ...

Context-Free Session Types

Peter Thiemann
Universität Freiburg, Germany
thiemann@acm.org

Vasco T. Vasconcelos
LaSIGE and University of Lisbon, Portugal
vmvasconcelos@ciencias.ulisboa.pt

Abstract

Session types describe structured communication on heterogeneously typed channels at a high level. Their tail-recursive structure imposes a protocol that can be described by a regular language. The types of transmitted values are drawn from the underlying functional language, abstracting from the details of serializing values of structured data types.

Context-free session types extend session types by allowing nested protocols that are not restricted to tail recursion. Nested protocols correspond to deterministic context-free languages. Such protocols are interesting in their own right, but they are particularly suited to describe the low-level serialization of tree-structured data in a type-safe way.

We establish the metatheory of context-free session types, prove that they properly generalize standard (two-party) session types, and take first steps towards type checking by showing that type equivalence is decidable.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; D.3.1 [*Formal Definitions and Theory*]

Keywords session types, semantics, type checking

1. Introduction

Session types have been discovered by Kohei Honda as a means to describe the structured interaction of processes via typed communication channels [13, 21]. While connections are homogeneously typed in languages like Concurrent ML [19], session types provide a heterogeneous type discipline for a protocol on a bidirectional connection: each message has an individual direction and type and there are choice points where a sender can make a choice and a receiver has to follow. While session types have been conceived for process calculi, they provide precise typings for communication channels in any programming language. They fit particularly well with strongly typed functional languages from the ML family.

The type structure of a functional language with session types typically comes with two layers, regular types and session types:

$$T ::= S \mid \mathsf{unit} \mid B \mid T \to T \mid \dots \quad (1)$$
$$S ::= \mathsf{end} \mid ?T.S \mid !T.S \mid \&\{l_i\colon S_i\} \mid \oplus\{l_i\colon S_i\} \mid z \mid \mu z.S$$

ICFP'16, September 18–24, 2016, Nara, Japan
© 2016 ACM. 978-1-4503-4219-3/16/09...$15.00
http://dx.doi.org/10.1145/2951913.2951926

A type T is either a session type S, a unit type, a base type B, a function type, and so on. Session types S are attached to communication channels. They denote denote different states of the channel. The type end indicates the end of a session, $?T.S$ ($!T.S$) indicates readiness to receive (send) a value of type T and continuing with S, the branching operators $\&\{l_i\colon S_i\}_{i\in I}$ and $\oplus\{l_i\colon S_i\}_{i\in I}$ indicate receiving and sending labels, where the label l_i selects the protocol S_i from a finite number of possibilities $i \in I$ for the subsequent communication on the channel. For example, the session type

$$\&\{\mathit{add}\colon ?\mathsf{int}.?\mathsf{int}.!\mathsf{int}.\mathsf{end}, \mathit{neg}\colon ?\mathsf{int}.!\mathsf{int}.\mathsf{end}\}$$

is the type of a server that accepts two commands *add* and *neg*, then reads the appropriate number of arguments and returns the result of the command. The session variable z and the operator $\mu z.S$ serve to introduce recursive protocols, for example, to read a list of numbers:

$$\mu z.\&\{\mathit{stop}\colon \mathsf{end}, \mathit{more}\colon ?\mathsf{int}.z\}$$

Session types are well suited to document high-level communication protocols and there is a whole range of extensions to make them amenable to deal with realistic situations, for example, multi-party session types [15], session types for distributed object-oriented programming [12], or for programming web services [6]. However, there is a fundamental limitation in their structure that makes it impossible to describe efficient low-level serialization (marshalling, pickling, ...) of tree structured data in a type-safe way, as we demonstrate with the following example.

Let's assume that a single communication operation can only transmit a label or a base type value to model the real-world restriction that data structures need to be serialized to a wire format before they can be sent over a network connection. Formally, it is sufficient to restrict the session type formation for sending and receiving data to base types: $!B.S$ and $?B.S$. Now suppose we want to transmit binary trees where the internal nodes contain a number. A recursive type for such trees can be defined as follows:

```
type Tree = Leaf
          | Node int Tree Tree
```

To serialize such a structure, we traverse it in some order and transmit a sequence of labels `Leaf` and `Node` and **`int`** values as they are visited by the traversal. The set of serialization sequences corresponding to a pre-order traversal of a tree may be described by the following context-free grammar.

$$N ::= \mathsf{Leaf} \mid \mathsf{Node}\ \mathsf{int}\ N\ N \quad (2)$$

Listing 1 contains a function `sendTree` that performs a pre-order traversal of a tree and sends correctly serialized output on a channel. The function relies on typical operations in a functional session type calculus like GV [11]: the **`select`** operation takes a label and a channel, outputs the label, and returns the (updated) channel. The `send` operation takes a value and a channel, outputs the value, and returns the channel. Ignore the type signature for a moment.

ICFP'16 (Wed. afternoon)

ICFP'16 (Wed. afternoon)

Contributions and overview

- We introduce context-free session types that extend the expressiveness of regular session types to capture the type-safe serialization of recursive datatypes. They further enable the type-safe implementation of remote operations on recursive datatypes that either traverse the structure eagerly or on demand.

Context-Free Session Types

Peter Thiemann
Universität Freiburg, Germany
thiemann@acm.org

Abstract

Session types describe structured communication on heterogeneously typed channels at a high level. Their tail-recursive structure imposes a protocol that can be described by a regular language. The types of transmitted values are drawn from the underlying functional language, abstracting from the details of serializing values of structured data types.

Context-free session types extend session types by allowing nested protocols that are not restricted to tail recursion. Nested protocols correspond to deterministic context-free languages. Such protocols are interesting in their own right, but they are particularly suited to describe the low-level serialization of tree-structured data in a type-safe way.

We establish the metatheory of context-free session types, prove that they properly generalize standard (two-party) session types, and take first steps towards type checking by showing that type equivalence is decidable.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; D.3.1 [*Formal Definitions and Theory*]

Keywords session types, semantics, type checking

1. Introduction

Session types have been discovered by Kohei Honda as a means to describe the structured interaction of processes via typed communication channels [13, 21]. While connections are homogeneously typed in languages like Concurrent ML [19], session types provide a heterogeneous type discipline for a protocol on a bidirectional connection: each message has an individual direction and type and there are choice points where a sender can make a choice and a receiver has to follow. While session types have been conceived for process calculi, they provide precise typings for communication channels in any programming language. They fit particularly well with strongly typed functional languages from the ML family.

The type structure of a functional language with session types typically comes with two layers, regular types and session types:

$$T ::= S \mid \mathsf{unit} \mid B \mid T \to T \mid \dots \quad (1)$$
$$S ::= \mathsf{end} \mid ?T.S \mid !T.S \mid \&\{l_i\colon S_i\} \mid \oplus\{l_i\colon S_i\} \mid z \mid \mu z.S$$

ICFP'16, September 18–24, 2016, Nara, Japan
© 2016 ACM. 978-1-4503-4219-3/16/09...$15.00
http://dx.doi.org/10.1145/2951913.2951926

A t
B, a
com
the
$(!T.$
and
$\oplus\{l$
labe
$i \in$
exa

is th
then
resu
$\mu z.$
a list of numbers:

$$\mu z.\&\{\mathit{stop}\colon \mathsf{end}, \mathit{more}\colon ?\,\mathsf{int}.z\}$$

Session types are well suited to document high-level communication protocols and there is a whole range of extensions to make them amenable to deal with realistic situations, for example, multi-party session types [15], session types for distributed object-oriented programming [12], or for programming web services [6]. However, there is a fundamental limitation in their structure that makes it impossible to describe efficient low-level serialization (marshalling, pickling, ...) of tree structured data in a type-safe way, as we demonstrate with the following example.

Let's assume that a single communication operation can only transmit a label or a base type value to model the real-world restriction that data structures need to be serialized to a wire format before they can be sent over a network connection. Formally, it is sufficient to restrict the session type formation for sending and receiving data to base types: $!\,B.S$ and $?\,B.S$. Now suppose we want to transmit binary trees where the internal nodes contain a number. A recursive type for such trees can be defined as follows:

```
type Tree = Leaf
          | Node int Tree Tree
```

To serialize such a structure, we traverse it in some order and transmit a sequence of labels `Leaf` and `Node` and **`int`** values as they are visited by the traversal. The set of serialization sequences corresponding to a pre-order traversal of a tree may be described by the following context-free grammar.

$$N ::= \mathsf{Leaf} \mid \mathsf{Node\ int}\ N\ N \quad (2)$$

Listing 1 contains a function `sendTree` that performs a pre-order traversal of a tree and sends correctly serialized output on a channel. The function relies on typical operations in a functional session type calculus like GV [11]: the **`select`** operation takes a label and a channel, outputs the label, and returns the (updated) channel. The `send` operation takes a value and a channel, outputs the value, and returns the channel. Ignore the type signature for a moment.

ICFP'16 (Wed. afternoon)

Context-Free Session Types

Peter Thiemann
Universität Freiburg, Germany
thiemann@acm.org

Abstract

Session types describe structured communication on heterogeneously typed channels at a high level. Their tail-recursive structure imposes a protocol that can be described by a regular language. The types of transmitted values are drawn from the underlying functional language, abstracting from the details of serializing values of structured data types.

Context-free session types extend session types by allowing nested protocols that are not restricted to tail recursion. Nested protocols correspond to deterministic context-free languages. Such protocols are interesting in their own right, but they are particularly suited to describe the low-level serialization of tree-structured data in a type-safe way.

We establish the metatheory of context-free session types, prove that they properly generalize standard (two-party) session types, and take first steps towards type checking by showing that type equivalence is decidable.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; D.3.1 [*Formal Definitions and Theory*]

Keywords session types, semantics, type checking

1. Introduction

Session types have been discovered by Kohei Honda as a means to describe the structured interaction of processes via typed communication channels [13, 21]. While connections are homogeneously typed in languages like Concurrent ML [19], session types provide a heterogeneous type discipline for a protocol on a bidirectional connection: each message has an individual direction and type and there are choice points where a sender can make a choice and a receiver has to follow. While session types have been conceived for process calculi, they provide precise typings for communication channels in any programming language. They fit particularly well with strongly typed functional languages from the ML family.

The type structure of a functional language with session types typically comes with two layers, regular types and session types:

$$T ::= S \mid \mathsf{unit} \mid B \mid T \to T \mid \dots \qquad (1)$$
$$S ::= \mathsf{end} \mid ?T.S \mid !T.S \mid \&\{l_i\colon S_i\} \mid \oplus\{l_i\colon S_i\} \mid z \mid \mu z.S$$

ICFP'16, September 18–24, 2016, Nara, Japan
© 2016 ACM. 978-1-4503-4219-3/16/09...$15.00
http://dx.doi.org/10.1145/2951913.2951926

A ty
B, a
com
the
$(!T.$
and
$\oplus\{l$
labe
$i \in$
exar

is th
then
resu
$\mu z.$
a list of numbers:

$$\mu z.\&\{\mathit{stop}\colon \mathsf{end}, \mathit{more}\colon ?\,\mathsf{int}.z\}$$

Session types are well suited to docu
nication protocols and there is a whole
make them amenable to deal with realis
ple, multi-party session types [15], sess
object-oriented programming [12], or fo
vices [6]. However, there is a fundamenta
ture that makes it impossible to describe
ization (marshalling, pickling, ...) of tree
safe way, as we demonstrate with the foll

Let's assume that a single communic
transmit a label or a base type value to
striction that data structures need to be s
before they can be sent over a network c
sufficient to restrict the session type form
ceiving data to base types: $!\,B.S$ and $?\,B.$
to transmit binary trees where the internal
A recursive type for such trees can be def

```
type Tree = Leaf
          | Node int Tree Tre
```

To serialize such a structure, we traver
transmit a sequence of labels `Leaf` and
they are visited by the traversal. The set
corresponding to a pre-order traversal of
by the following context-free grammar.

$$N ::= \mathsf{Leaf} \mid \mathsf{Node}\ \mathsf{int}$$

Listing 1 contains a function `sendTree` t
traversal of a tree and sends correctly ser
nel. The function relies on typical operatic
type calculus like GV [11]: the **select** ope
channel, outputs the label, and returns th
`send` operation takes a value and a chann
returns the channel. Ignore the type signa

Contributions and overview

- We introduce context-free session types that extend the expressiveness of regular session types to capture the type-safe serialization of recursive datatypes. They further enable the type-safe implementation of remote operations on recursive datatypes that either traverse the structure eagerly or on demand.

$$(\lambda a.e)v \to e[v/a] \qquad \mathsf{let}\, a, b = (u, v)\, \mathsf{in}\, e \to e[u/a][v/b]$$
$$\mathsf{match}\,(\mathsf{in}\, l_j\, v)\,\mathsf{with}\,[l_i \to e_i] \to e_j v \qquad \mathsf{fix}\, a.e \to e[\mathsf{fix}\, a.e/a]$$
$$\frac{e_1 \to e_2}{E[e_1] \to E[e_2]} \qquad E[\mathsf{fork}\, e] \to E[()] \mid e$$
$$E[\mathsf{new}] \to (\nu a, b)E[(a, b)]$$
$$(\nu a, b)(E_1[\mathsf{send}\, v\, a] \mid E_2[\mathsf{receive}\, b]) \to (\nu a, b)(E_1[a] \mid E_2[(v, b)])$$
$$(\nu a, b)(E_1[\mathsf{select}\, l_j\, a] \mid E_2[\mathsf{case}\, b\, \mathsf{of}\, \{l_i \to e_i\}]) \to (\nu a, b)(E_1[a] \mid E_2[e_j b])$$
$$\frac{p \to p'}{p \mid q \to p' \mid q} \qquad \frac{p \to p'}{(\nu\, a, b)p \to (\nu\, a, b)p'} \qquad \frac{p \equiv q \quad q \to q'}{p \to q'}$$

Context E_1 (resp. E_2, resp. E) does not bind a (resp. b, resp. a and b).
Dual $(\nu b, a)$ rules for send/receive and select/case omitted.

Figure 8. Reduction relation

Arithmetic

- Syntax
- Semantics
  - Natural, big-step
  - Evaluator
  - Structured, small-step
  - Reduction
  - Standard reduction
  - Abstract machine

Syntax of $\mathcal{A}$

3 ways: 1/3

$$i \in \mathbb{Z} \Rightarrow i \in \mathcal{A}$$

$$e \in \mathcal{A} \Rightarrow \mathit{Pred}(e) \in \mathcal{A}$$

$$e \in \mathcal{A} \Rightarrow \mathit{Succ}(e) \in \mathcal{A}$$

$$e_1 \in \mathcal{A} \wedge e_2 \in \mathcal{A} \Rightarrow \mathit{Plus}(e_1, e_2) \in \mathcal{A}$$

$$e_1 \in \mathcal{A} \wedge e_2 \in \mathcal{A} \Rightarrow \mathit{Mult}(e_1, e_2) \in \mathcal{A}$$

Syntax of $\mathcal{A}$

3 ways: 2/3

$$
\begin{array}{lllll}
\mathbb{Z} & i & ::= & \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \\
\mathcal{A} & e & ::= & i \\
& & \mid & \mathit{Pred}(e) \\
& & \mid & \mathit{Succ}(e) \\
& & \mid & \mathit{Plus}(e, e) \\
& & \mid & \mathit{Mult}(e, e)
\end{array}
$$

Inference rules

$$\frac{H_1 \qquad H_2 \qquad \ldots \qquad H_n}{C}$$

Inference rules

$$\frac{H_1 \qquad H_2 \qquad \ldots \qquad H_n}{C}$$

$$H_1 \wedge H_2 \wedge \cdots \wedge H_n \Rightarrow C$$

Syntax of $\mathcal{A}$

3 ways: 3/3

$$\frac{i \in \mathbb{Z}}{i \in \mathcal{A}}\ (1) \qquad \frac{e \in \mathcal{A}}{\mathit{Pred}(e) \in \mathcal{A}}\ (2) \qquad \frac{e \in \mathcal{A}}{\mathit{Succ}(e) \in \mathcal{A}}\ (3)$$

$$\frac{e_1 \in \mathcal{A} \qquad e_2 \in \mathcal{A}}{\mathit{Plus}(e_1, e_2) \in \mathcal{A}}\ (4) \qquad \frac{e_1 \in \mathcal{A} \qquad e_2 \in \mathcal{A}}{\mathit{Mult}(e_1, e_2) \in \mathcal{A}}\ (5)$$

Proof of $\mathit{Plus}(4, \mathit{Succ}(2)) \in \mathcal{A}$

$$\dfrac{\dfrac{4 \in \mathbb{Z}}{4 \in \mathcal{A}} \qquad \dfrac{\dfrac{2 \in \mathbb{Z}}{2 \in \mathcal{A}}}{\mathit{Succ}(2) \in \mathcal{A}}}{\mathit{Plus}(4, \mathit{Succ}(2)) \in \mathcal{A}}$$

Natural semantics of $\mathcal{A}$

$$\Downarrow \subseteq \mathcal{A} \times \mathbb{Z}$$

$$\mathit{Plus}(4, \mathit{Succ}(2)) \Downarrow 7$$

Natural semantics of $\mathcal{A}$

$$\frac{}{i \Downarrow i} \qquad \frac{e \Downarrow i}{\mathit{Pred}(e) \Downarrow i-1} \qquad \frac{e \Downarrow i}{\mathit{Succ}(e) \Downarrow i+1}$$

$$\frac{e_1 \Downarrow i \qquad e_2 \Downarrow j}{\mathit{Plus}(e_1, e_2) \Downarrow i+j} \qquad \frac{e_1 \Downarrow i \qquad e_2 \Downarrow j}{\mathit{Mult}(e_1, e_2) \Downarrow i \cdot j}$$

Natural semantics of $\mathcal{A}$

$$\frac{e_1 \Downarrow i \qquad e_2 \Downarrow j}{\mathit{Plus}(e_1, e_2) \Downarrow i + j}$$

$$\frac{e_1 \Downarrow i \qquad e_2 \Downarrow j \qquad k = i + j}{\mathit{Plus}(e_1, e_2) \Downarrow k}$$

Proof of $\mathit{Plus}(4, \mathit{Succ}(2)) \Downarrow 7$

$$\dfrac{\dfrac{}{4 \Downarrow 4} \qquad \dfrac{\dfrac{}{2 \Downarrow 2}}{\mathit{Succ}(2) \Downarrow 3}}{\mathit{Plus}(4, \mathit{Succ}(2)) \Downarrow 7}$$

Evaluator semantics of $\mathcal{A}$

```
type arith = Int of int
           | Pred of arith
           | Succ of arith
           | Plus of arith * arith
           | Mult of arith * arith
let rec eval (e : arith) : int =
    match e with
      Int i -> i
    | Pred e -> (eval e) - 1
    | Succ e -> (eval e) + 1
    | Plus (e1, e2) -> (eval e1) + (eval e2)
    | Mult (e1, e2) -> (eval e1) * (eval e2)

# eval (Plus (Int 4, Succ (Int 2)));;
- : int = 7
```

$\Downarrow \subseteq \mathcal{A} \times \mathbb{Z}$

SOS semantics of $\mathcal{A}$

~~$\Downarrow \subseteq \mathcal{A} \times \mathbb{Z}$~~

$$\rightarrow \subset \mathcal{A} \times \mathcal{A}$$

$$Plus(4, Succ(2)) \rightarrow Plus(4, 3)$$

$$Plus(4, 3) \rightarrow 7$$

each step has a proof

SOS semantics of $\mathcal{A}$

~~$\Downarrow \subseteq \mathcal{A} \times \mathbb{Z}$~~

$$\rightarrow \subset \mathcal{A} \times \mathcal{A}$$

$$Plus(4, Succ(2)) \rightarrow Plus(4, 3) \qquad \text{X}$$

$$Plus(4, 3) \rightarrow 7$$

each step has a proof

SOS semantics of $\mathcal{A}$

Part 1: axioms

$$\frac{}{\mathit{Pred}(i) \to i - 1} \qquad \frac{}{\mathit{Succ}(i) \to i + 1}$$

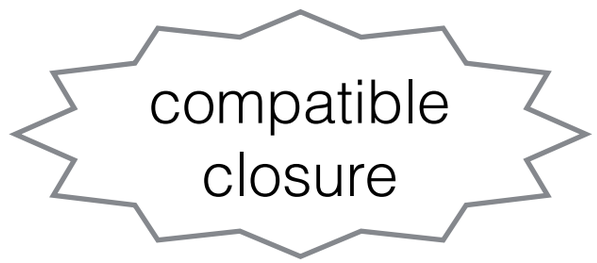
$$\frac{}{\mathit{Plus}(i, j) \to i + j} \qquad \frac{}{\mathit{Mult}(i, j) \to i \cdot j}$$

SOS semantics of $\mathcal{A}$

Part 2: contexts

$$\frac{e \to e'}{\mathit{Pred}(e) \to \mathit{Pred}(e')} \qquad \frac{e \to e'}{\mathit{Succ}(e) \to \mathit{Succ}(e')} \qquad \frac{e_1 \to e_1'}{\mathit{Plus}(e_1, e_2) \to \mathit{Plus}(e_1', e_2)}$$

$$\frac{e_2 \to e_2'}{\mathit{Plus}(e_1, e_2) \to \mathit{Plus}(e_1, e_2')} \qquad \frac{e_1 \to e_1'}{\mathit{Mult}(e_1, e_2) \to \mathit{Mult}(e_1', e_2)} \qquad \frac{e_2 \to e_2'}{\mathit{Mult}(e_1, e_2) \to \mathit{Mult}(e_1, e_2')}$$



Proof of each step

$$\dfrac{\dfrac{}{\mathit{Succ}(2) \to 3}}{\mathit{Plus}(4, \mathit{Succ}(2)) \to \mathit{Plus}(4, 3)} \qquad \dfrac{}{\mathit{Plus}(4, 3) \to 7}$$

Different steps

$$\dfrac{\dfrac{}{\mathit{Succ}(4) \to 5}}{\mathit{Plus}(\mathit{Succ}(4), \mathit{Pred}(3)) \to \mathit{Plus}(5, \mathit{Pred}(3))}$$

$$\dfrac{\dfrac{}{\mathit{Pred}(3) \to 2}}{\mathit{Plus}(\mathit{Succ}(4), \mathit{Pred}(3)) \to \mathit{Plus}(\mathit{Succ}(4), 2)}$$

SOS semantics of $\mathcal{A}$

$$\frac{e \to e'}{e \to^\star e'}$$

$$\frac{}{e \to^\star e}$$

reflexive closure

$$\frac{e \to^\star e' \qquad e' \to^\star e''}{e \to^\star e''}$$

transitive closure

Relating natural and SOS

Claim:

$$e \Downarrow i \iff e \to^\star i$$

Reduction semantics

Every proof of one-step reduction looks like:

$$\dfrac{\dfrac{\dfrac{}{e\ \mathbf{a}\ e'}}{\ldots e \ldots \to \ldots e' \ldots}}{\vdots}$$
$$\ldots(\ldots e \ldots) \ldots \to \ldots(\ldots e' \ldots) \ldots$$

Reduction axioms

$$\frac{}{Pred(i)\ \mathbf{a}\ i-1} \qquad \frac{}{Succ(i)\ \mathbf{a}\ i+1}$$

$$\frac{}{Plus(i,j)\ \mathbf{a}\ i+j} \qquad \frac{}{Mult(i,j)\ \mathbf{a}\ i\cdot j}$$

Reduction semantics

Every proof of one-step reduction looks like:

$$\dfrac{\dfrac{\dfrac{}{e\ \mathbf{a}\ e'}}{\ldots e \ldots \to \ldots e' \ldots}}{\vdots}$$
$$\ldots(\ldots e \ldots) \ldots \to \ldots(\ldots e' \ldots) \ldots$$

Reduction semantics

$$\begin{array}{lcl} \mathit{Context} \quad \mathcal{C} & = & \Box \\ & \mid & \mathit{Pred}(\mathcal{C}) \mid \mathit{Succ}(\mathcal{C}) \\ & \mid & \mathit{Plus}(\mathcal{C}, e) \mid \mathit{Plus}(e, \mathcal{C}) \\ & \mid & \mathit{Mult}(\mathcal{C}, e) \mid \mathit{Mult}(e, \mathcal{C}) \end{array}$$

$$\mathcal{C}[e]$$

$$\dfrac{\dfrac{\dfrac{}{e \; \mathbf{a} \; e'}}{\ldots e \ldots \rightarrow \ldots e' \ldots}}{\vdots}$$
$$\ldots(\ldots e \ldots) \ldots \rightarrow \ldots(\ldots e' \ldots) \ldots$$

$$\dfrac{e \; \mathbf{a} \; e'}{\mathcal{C}[e] \rightarrow \mathcal{C}[e']}$$

Standard reductions

$$\frac{e \ \mathbf{a} \ e'}{e \longmapsto e'}$$

$$\frac{e \longmapsto e'}{\mathit{Pred}(e) \longmapsto \mathit{Pred}(e')} \qquad \frac{e \longmapsto e'}{\mathit{Succ}(e) \longmapsto \mathit{Succ}(e')}$$

$$\frac{e \longmapsto e'}{\mathit{Plus}(v, e) \longmapsto \mathit{Plus}(v, e')} \qquad \frac{e_1 \longmapsto e_1'}{\mathit{Plus}(e_1, e_2) \longmapsto \mathit{Plus}(e_1', e_2)}$$

$$\frac{e_1 \longmapsto e_1'}{\mathit{Mult}(e_1, e_2) \longmapsto \mathit{Mult}(e_1', e_2)} \qquad \frac{e \longmapsto e'}{\mathit{Mult}(v, e) \longmapsto \mathit{Mult}(v, e')}$$

Standard reductions

$$\frac{e \ \mathbf{a} \ e'}{\mathcal{E}[e] \longmapsto \mathcal{E}[e']}$$

$$\begin{array}{lcll} \mathit{EvalContext} & \mathcal{E} & = & \square \\ & & \mid & \mathit{Pred}(\mathcal{E}) \mid \mathit{Succ}(\mathcal{E}) \\ & & \mid & \mathit{Plus}(\mathcal{E}, e) \mid \mathit{Plus}(v, \mathcal{E}) \\ & & \mid & \mathit{Mult}(\mathcal{E}, e) \mid \mathit{Mult}(v, \mathcal{E}) \end{array}$$

Relating reductions

Claim:

$$e \longmapsto^\star i \iff e \to^\star i$$

Abstract (stack) machine

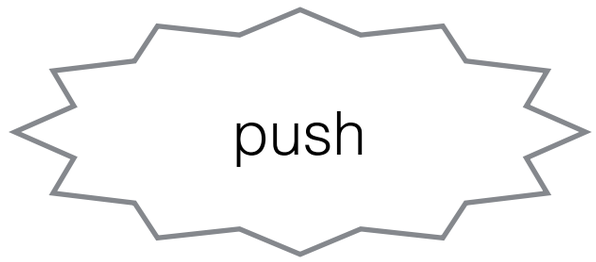
$$
\begin{array}{rrcl}
\textit{Frame} & \mathcal{F} & = & \textit{Pred}(\square) \mid \textit{Succ}(\square) \\
 & & \mid & \textit{Plus}(\square, e) \mid \textit{Plus}(v, \square) \\
 & & \mid & \textit{Mult}(\square, e) \mid \textit{Mult}(v, \square) \\
\textit{Stack} & \mathcal{S} & = & [\,] \mid \mathcal{F} :: \mathcal{S} \\
 & & & \\
\textit{Serious} & s & \in & \mathcal{A} \setminus \mathbb{Z}
\end{array}
$$

Abstract (stack) machine

$$\frac{e \ \mathbf{a} \ e'}{e, \mathcal{S} \rightsquigarrow e', \mathcal{S}}$$

Abstract (stack) machine

$$\frac{}{Pred(s), \mathcal{S} \rightsquigarrow s, Pred(\square) :: \mathcal{S}}$$



$$\frac{}{Mult(s, e), \mathcal{S} \rightsquigarrow s, Mult(\square, e) :: \mathcal{S}}$$

$$\frac{}{Mult(v, s), \mathcal{S} \rightsquigarrow s, Mult(v, \square) :: \mathcal{S}}$$

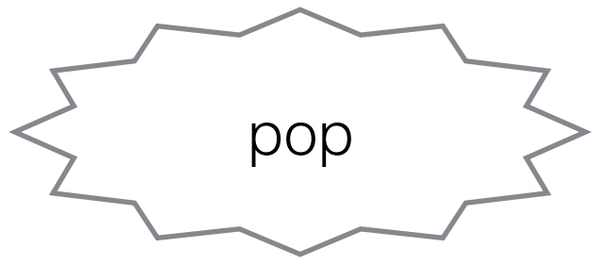
(Not showing similar rules for Succ, Plus)

Abstract (stack) machine

$$\frac{}{v, \mathit{Pred}(\square) :: \mathcal{S} \rightsquigarrow \mathit{Pred}(v), \mathcal{S}}$$

$$\frac{}{v, \mathit{Mult}(\square, e) :: \mathcal{S} \rightsquigarrow \mathit{Mult}(v, e), \mathcal{S}}$$

$$\frac{}{v, \mathit{Mult}(e, \square) :: \mathcal{S} \rightsquigarrow \mathit{Mult}(e, v), \mathcal{S}}$$



Relating reductions

Claim:

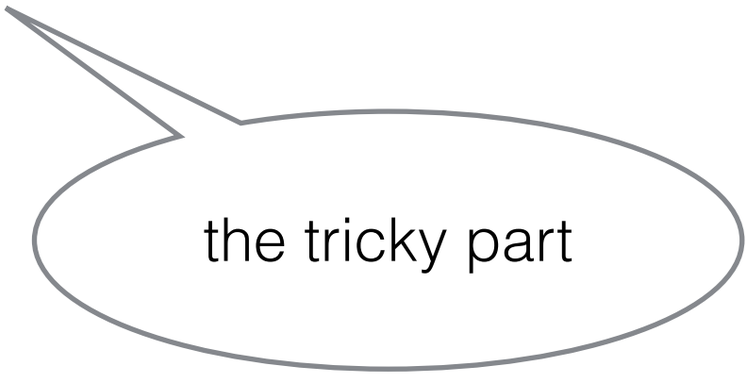
$$e \longmapsto^\star i \iff e, [\,] \rightsquigarrow^\star i, [\,]$$

Functions

$$
\begin{array}{rcl}
e & = & \ldots \\
 & | & \mathit{App}(e, e) \\
 & | & \mathit{Fun}(x, e) \\
 & | & \mathit{Var}(x) \\
x & = & \mathtt{x} \mid \mathtt{y} \mid \mathtt{z} \mid \ldots \\
v & = & i \mid \mathit{Fun}(x, e)
\end{array}
$$

Substitution

$$
\begin{aligned}
\mathit{Var}(x')[e/x] &= \begin{cases} e, & \text{if } x = x' \\ \mathit{Var}(x'), & \text{otherwise} \end{cases} \\
\mathit{Succ}(e_0)[e/x] &= \mathit{Succ}(e_0[e/x]) \\
\mathit{Plus}(e_0, e_1)[e/x] &= \mathit{Plus}(e_0[e/x], e_1[e/x]) \\
&\vdots \quad\quad \vdots \\
\mathit{Fun}(x', e_0)[e/x] &= \ldots
\end{aligned}
$$



Natural semantics

$$\frac{e_0 \Downarrow \mathit{Fun}(x, e) \qquad e[e_1/x] \Downarrow v}{\mathit{App}(e_0, e_1) \Downarrow v}$$

call-by-*name*

Natural semantics

$$\frac{e_0 \Downarrow \boldsymbol{Fun}(x, e) \qquad e_1 \Downarrow v_1 \qquad e[v_1/x] \Downarrow v}{App(e_0, e_1) \Downarrow v}$$

Reduction semantics

$$\frac{}{App(Fun(x, e), e') \ \boldsymbol{\beta} \ e[e'/x]}$$

$$\begin{array}{lcll} Context & \mathcal{C} & = & \ldots \\ & & | & Fun(x, \mathcal{C}) \\ & & | & App(\mathcal{C}, e) \mid App(e, \mathcal{C}) \end{array}$$

$$\frac{e \ (\mathbf{a} \cup \boldsymbol{\beta}) \ e'}{\mathcal{C}[e] \rightarrow \mathcal{C}[e']}$$

Reduction semantics

$$\frac{}{\mathit{App}(\mathit{Fun}(x, e), v)\ \boldsymbol{\beta}_{\mathbf{v}}\ e[v/x]}$$

$$\begin{array}{lcll} \mathit{Context} & \mathcal{C} & = & \ldots \\ & & \mid & \mathit{Fun}(x, \mathcal{C}) \\ & & \mid & \mathit{App}(\mathcal{C}, e) \mid \mathit{App}(e, \mathcal{C}) \end{array}$$

$$\frac{e\ (\mathbf{a} \cup \boldsymbol{\beta}_{\mathbf{v}})\ e'}{\mathcal{C}[e] \rightarrow_{\mathbf{v}} \mathcal{C}[e']}$$

Standard reductions

$$\frac{}{App(Fun(x, e), e') \ \beta \ e[e'/x]}$$

$$\begin{array}{rrcl} Context & \mathcal{C} & = & \ldots \\ & & \mid & Fun(x, \mathcal{C}) \\ & & \mid & App(\mathcal{C}, e) \mid App(e, \mathcal{C}) \\ EvalContext & \mathcal{E} & = & \ldots \\ & & \mid & App(\mathcal{E}, e) \end{array}$$

$$\frac{e \ (\mathbf{a} \cup \beta) \ e'}{\mathcal{E}[e] \longmapsto \mathcal{E}[e']}$$

Standard reductions

$$\frac{}{App(Fun(x, e), v)\ \beta_{\mathbf{v}}\ e[v/x]}$$

$$\begin{array}{rcll} Context & \mathcal{C} & = & \ldots \\ & & | & Fun(x, \mathcal{C}) \\ & & | & App(\mathcal{C}, e) \mid App(e, \mathcal{C}) \\ EvalContext & \mathcal{E} & = & \ldots \\ & & | & App(\mathcal{E}, e) \mid App(v, \mathcal{E}) \end{array}$$

$$\frac{e\ (\mathbf{a} \cup \beta_{\mathbf{v}})\ e'}{\mathcal{E}[e] \longmapsto_{\mathbf{v}} \mathcal{E}[e']}$$

Abstract machine

$$\begin{array}{lcl} \mathit{Frame}\quad \mathcal{F} & = & \ldots \\ & \mid & \mathit{App}(\square, e) \mid \mathit{App}(v, \square) \end{array}$$

remove for CbN

Push, pop, reduce same as before *mutatis mutandis*

Exceptions

$$
\begin{array}{rcll}
e & = & \ldots & \\
 & | & \mathit{Raise}(e) & \\
 & | & \mathit{Try}(e, x, e) &
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{EvalContext} & \mathcal{E} & = & \ldots \\
 & & | & \mathit{Raise}(\mathcal{E}) \\
 & & | & \mathit{Try}(\mathcal{E}, x, e) \\
\mathit{TryContext} & \mathcal{T} & \in & \mathcal{E} \setminus \mathit{Try}(\mathcal{E}, x, e)
\end{array}
$$

Exceptions

$$\frac{}{\mathit{Try}(v, x, e)\ \boldsymbol{\tau}\ v}$$

$$\frac{}{\mathit{Try}(\mathcal{T}[\mathit{Raise}(v)], x, e)\ \boldsymbol{\tau}\ e[v/x]}$$

$$\frac{e\ (\mathbf{a} \cup \boldsymbol{\beta} \cup \boldsymbol{\tau})\ e'}{\mathcal{E}[e] \longmapsto \mathcal{E}[e']}$$

Call/cc

$$
\begin{array}{rcl}
e & = & \ldots \\
 & | & \mathit{Callcc}(x, e) \mid \mathit{Halt}(e) \\
\mathcal{E} & = & \ldots \\
 & | & \mathit{Halt}(\mathcal{E})
\end{array}
$$

$$\frac{}{\mathcal{E}[\mathit{Halt}(v)] \longmapsto v}$$

$$\frac{}{\mathcal{E}[\mathit{Callcc}(x, e)] \longmapsto e[\mathit{Fun}(x', \mathit{Halt}(\mathcal{E}[x']))/x]}$$

Operational semantics: A method of defining the meaning of programs by describing the actions carried out during a program's execution.

Useful for:

- Specifying a PL
- Communicating ideas
- Validating claims
- ...

What you've seen:

- Syntax
- Semantics
  - Natural, big-step
  - Evaluator
  - Structured, small-step
  - Reduction
  - Standard reduction
  - Abstract machine

$$(\lambda a.e)v \to e[v/a] \qquad \mathsf{let}\, a, b = (u, v)\, \mathsf{in}\, e \to e[u/a][v/b]$$

$$\mathsf{match}\, (\mathsf{in}\, l_j\, v)\, \mathsf{with}\, [l_i \to e_i] \to e_j v \qquad \mathsf{fix}\, a.e \to e[\mathsf{fix}\, a.e/a]$$

$$\frac{e_1 \to e_2}{E[e_1] \to E[e_2]} \qquad E[\mathsf{fork}\, e] \to E[()] \mid e$$

$$E[\mathsf{new}] \to (\nu a, b)E[(a, b)]$$

$$(\nu a, b)(E_1[\mathsf{send}\, v\, a] \mid E_2[\mathsf{receive}\, b]) \to (\nu a, b)(E_1[a] \mid E_2[(v, b)])$$

$$(\nu a, b)(E_1[\mathsf{select}\, l_j\, a] \mid E_2[\mathsf{case}\, b\, \mathsf{of}\, \{l_i \to e_i\}]) \to (\nu a, b)(E_1[a] \mid E_2[e_j b])$$

$$\frac{p \to p'}{p \mid q \to p' \mid q} \qquad \frac{p \to p'}{(\nu\, a, b)p \to (\nu\, a, b)p'} \qquad \frac{p \equiv q \quad q \to q'}{p \to q'}$$

Context E_1 (resp. E_2, resp. E) does not bind a (resp. b, resp. a and b).
Dual $(\nu b, a)$ rules for send/receive and select/case omitted.

Figure 8. Reduction relation

$$(\lambda a.e)v \to e[v/a] \qquad \mathsf{let}\, a, b = (u, v)\, \mathsf{in}\, e \to e[u/a][v/b]$$

$$\mathsf{match}\, (\mathsf{in}\, l_j\, v)\, \mathsf{with}\, [l_i \to e_i] \to e_j v \qquad \mathsf{fix}\, a.e \to e[\mathsf{fix}\, a.e/a]$$

$$\frac{e_1 \to e_2}{E[e_1] \to E[e_2]} \qquad E[\mathsf{fork}\, e] \to E[()] \mid e$$

$$E[\mathsf{new}] \to (\nu a, b)E[(a, b)]$$

$$(\nu a, b)(E_1[\mathsf{send}\, v\, a] \mid E_2[\mathsf{receive}\, b]) \to (\nu a, b)(E_1[a] \mid E_2[(v, b)])$$

$$(\nu a, b)(E_1[\mathsf{select}\, l_j\, a] \mid E_2[\mathsf{case}\, b\, \mathsf{of}\, \{l_i \to e_i\}]) \to (\nu a, b)(E_1[a] \mid E_2[e_j b])$$

$$\frac{p \to p'}{p \mid q \to p' \mid q} \qquad \frac{p \to p'}{(\nu\, a, b)p \to (\nu\, a, b)p'} \qquad \frac{p \equiv q \quad q \to q'}{p \to q'}$$

Context E_1 (resp. E_2, resp. E) does not bind a (resp. b, resp. a and b).
Dual $(\nu b, a)$ rules for send/receive and select/case omitted.

Figure 8. Reduction relation