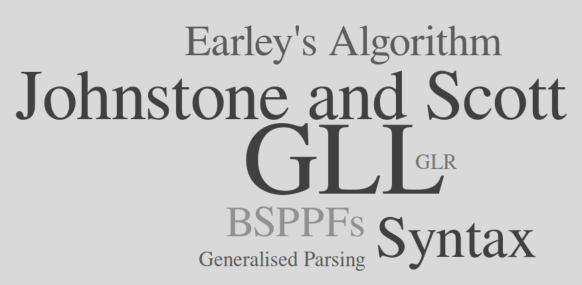
A Guided Tour of Initial Algebra Semantics

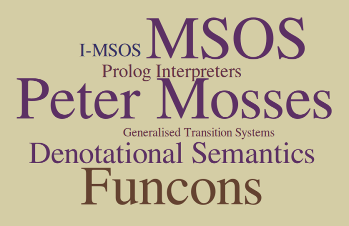
L. Thomas van Binsbergen
`https://plancomps.github.io`

Centrum Wiskunde & Informatica
ltvanbinsbergen@acm.org

17 December, 2019

Centrum Wiskunde & Informatica


Earley's Algorithm
Johnstone and Scott
GLL
GLR
BSPPFs
Syntax
Generalised Parsing


I-MSOS
MSOS
Prolog Interpreters
Peter Mosses
Generalised Transition Systems
Denotational Semantics
Funcons


Attribute Grammars
Atze Dijkstra
Haskell
Doaitse Swierstra
Parser Combinators

1. Introduce funcons
2. Discuss modularity in some instances of "Initial Algebra Semantics"
3. Discuss other pragmatic considerations

- Direct implementation as sets and pure functions (denotational)
- M-SOS, I-MSOS (operational semantics)
- Funcon translation (component-based semantics)
- Attribute grammars (syntax-directed translation)

PLanCompS project (2011-2015...)

- Component-based approach towards formal, dynamic semantics

Main contributions:

- A library of highly reusable, *fun*damental *con*structs (*funcons*)
- The meta-language CBS for defining component-based semantics[1]

- `http://plancomps.org` or `https://plancomps.github.io`

[1] *Executable Component-Based Semantics*. Van Binsbergen, Sculthorpe, Mosses. JLAMP 2019

What is the state of the funcon library?

Verified and available at `https://plancomps.github.io/`

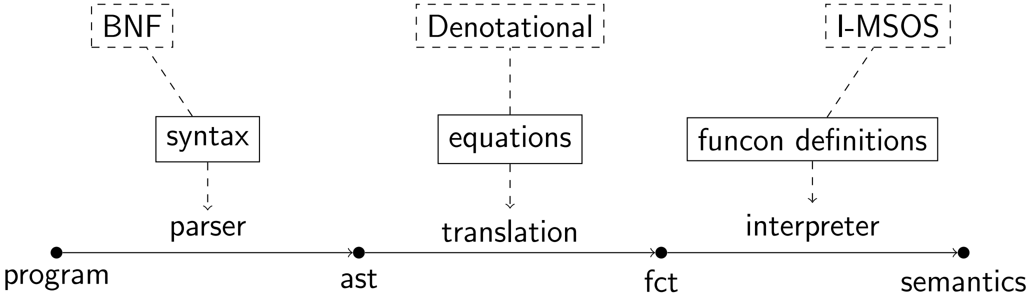
- Procedural: procedures, references, scoping, iteration
- Functional: functions, bindings, datatypes, pattern matching
- Object-oriented: objects, classes, inheritance
- Abnormal control: exceptions, break/continue, delimited continuations

Unverified as of yet (prototype phase)

- Concurrency: multi-threading
- Logical programming: backtracking, unification
- Meta-programming: AST conversions, staged evaluation[2]

[2] *Funcons for Homogeneous Generative Meta-Programming*. Van Binsbergen. GPCE 2018

Funcons pipeline - CBS



WHILE - Expressions

$$\textbf{plus}(e_1, e_2) = \textbf{integer-add}(e_1, e_2)$$
$$\textbf{leq}(e_1, e_2) = \textbf{is-less-or-equal}(e_1, e_2)$$
$$\textbf{int}(i) = i$$
$$\textbf{bool}(b) = b$$
$$\textbf{ident}(x) = \textbf{assigned}(\textbf{bound}(x))$$

WHILE - Commands

$$
\begin{aligned}
\textbf{seq}(c_1, c_2) &= \textbf{accumulate}(c_1, c_2) \\
\textbf{print}(e) &= \mathit{command}(\textbf{seq}(\textbf{print}(\textbf{to-string}(e)), \textbf{print}(\textbf{line-feed}))) \\
\textbf{assign}(x, e) &= \textbf{else}(\mathit{command}(\textbf{assign}(\textbf{bound}(x), e)), \textbf{bind}(x, \textbf{alloc-init}(\textbf{values}, e))) \\
\textbf{while}(e, c) &= \mathit{command}(\textbf{while}(e, \textbf{handle-thrown}(\textbf{effect}(c), \textbf{null}))) \\
\textbf{continue}\ () &= \textbf{throw}(\texttt{"continue"}) \\
\mathit{command}(e) &= \textbf{seq}(\textbf{effect}(e), \textbf{map-empty})
\end{aligned}
$$

WHILE - Programs

$$program(c) = \textbf{initialise-binding}(\textbf{initialise-storing}(\textbf{finalise-throwing}(c)))$$

WHILE - Programs

$$\textit{program}(c) = \textbf{initialise-binding}(\textbf{initialise-storing}(\textbf{finalise-throwing}(c)))$$

We have seen:

- an example of an algebra
- a(n) (in)formal semantic specification
- agile language engineering with funcons as funcons are executable

Initial Algebra Semantics and Continuous Algebras

J. A. GOGUEN

UCLA, Los Angeles, California

AND

J. W. THATCHER, E. G. WAGNER, AND J. B. WRIGHT

IBM Thomas J Watson Research Center, Yorktown Heights, New York

ABSTRACT   Many apparently divergent approaches to specifying formal semantics of programming languages are applications of initial algebra semantics In this paper an overview of initial algebra semantics is provided

Signature

- A set S of *sorts*
- A set of *operations* written $f : (s_1, \ldots, s_n) \to s_0$ with $s_i \in S$

Signature

- A set S of *sorts*
- A set of *operations* written $f : (s_1, \ldots, s_n) \to s_0$ with $s_i \in S$

Example

$$S = \{\textbf{commands}, \textbf{expressions}, \textbf{programs}, \textbf{ints}, \textbf{bools}, \textbf{ids}\}$$

$$\textbf{print} : (\textbf{expressions}) \to \textbf{commands}$$

$$\textbf{assign} : (\textbf{ids}, \textbf{expressions}) \to \textbf{commands}$$

$$\textbf{leq} : (\textbf{expressions}, \textbf{expressions}) \to \textbf{bools}$$

$$\ldots$$

Signature

- A set S of *sorts*
- A set of *operations* written $f : (s_1, \ldots, s_n) \to s_0$ with $s_i \in S$

Example

$$S = \{\textbf{commands}, \textbf{expressions}, \textbf{programs}, \textbf{ints}, \textbf{bools}, \textbf{ids}\}$$

$$\textbf{print} : (\textbf{expressions}) \to \textbf{commands}$$

$$\textbf{assign} : (\textbf{ids}, \textbf{expressions}) \to \textbf{commands}$$

$$\textbf{leq} : (\textbf{expressions}, \textbf{expressions}) \to \textbf{bools}$$

$$\ldots$$

A signature captures the abstract syntax of a language

Algebra A for a given signature

- A *carrier set* A_s for each sort $s \in S$
- An *evaluation function* f_A of type $A_{s_1} \times \ldots \times A_{s_n} \to A_{s_0}$ for each $f : (s_1, \ldots, s_n) \to s_0$

Algebra A for a given signature

- A *carrier set* A_s for each sort $s \in S$
- An *evaluation function* f_A of type $A_{s_1} \times \ldots \times A_{s_n} \to A_{s_0}$ for each $f : (s_1, \ldots, s_n) \to s_0$

An algebra is one semantics for the language; multiple can be defined

Algebra A for a given signature

- A *carrier set* A_s for each sort $s \in S$
- An *evaluation function* f_A of type $A_{s_1} \times \ldots \times A_{s_n} \to A_{s_0}$ for each $f : (s_1, \ldots, s_n) \to s_0$

Example

type *Sem_Cmds* = *Funcons* -- carrier of **commands** is the set of funcon terms
type *Sem_Exprs* = *Funcons* -- carrier of **expressions** is the set of funcon terms
type *Sem_Ids* = *String* -- carrier of **ids** is the set of Haskell strings
...

sem_assign :: *Sem_Ids* → *Sem_Expr* → *Sem_Command*

...

An algebra is one semantics for the language; multiple can be defined

How to connect **concrete syntax** *to* **abstract syntax***?*

How to define the carriers and the evaluation functions?

How to connect **concrete syntax** *to* **abstract syntax***?*

How to connect **concrete syntax** *to* **abstract syntax***?*

Syntax definition formalisms enable users to attach operations to productions

How to connect **concrete syntax** *to* **abstract syntax***?*

Syntax definition formalisms enable users to attach operations to productions

problem: There is often a significant gap between concrete and abstract syntax

How to connect **concrete syntax** *to* **abstract syntax***?*

Syntax definition formalisms enable users to attach operations to productions

problem: There is often a significant gap between concrete and abstract syntax

possible solutions:

- If sufficient, simply ignore keywords and separators
- Introduce one or more intermediate syntaxes to bridge the gap
- Apply generalised parsing technologies, shrinking the gap

```
syn_command :: BNF Token Sem_Command
syn_command = "command"
  <::=> sem_seq       <$$> syn_command <** keychar ';' <<<**> syn_command
  <||>  sem_assign    <$$> id_lit <** keyword ":=" <**> syn_expr
  <||>  sem_print     <$$  keyword "print" <**> syn_expr
  <||>  sem_while     <$$  keyword "while" <**> syn_expr <** keyword "do" <**>
                             syn_command <** optional (keychar ';') <**
                             keyword "done"
  <||>  sem_continue  <$$  keyword "continue"
```

Package GLL on Hackage:

- Evaluation functions are applied in so-called “semantic actions”
- Ignore the output of certain symbols in the right-hand side of productions
- Uses generalised top-down (GLL) parsing under the hood
- Might require the invocation of ambiguity reduction strategies

How to define the carrier sets and the evaluation functions?

How to define the carrier sets and the evaluation functions?

problems:

1. Defining pure evaluation functions for operations with effects

How to define the carrier sets and the evaluation functions?

problems:

1. Defining pure evaluation functions for operations with effects
   solution: auxiliary semantic entities (carrier sets become functions)

How to define the carrier sets and the evaluation functions?

problems:

1. Defining pure evaluation functions for operations with effects
   solution: auxiliary semantic entities (carrier sets become functions)
2. Composing carrier sets when composing languages or language fragments

How to define the carrier sets and the evaluation functions?

problems:

1. Defining pure evaluation functions for operations with effects
   solution: auxiliary semantic entities (carrier sets become functions)
2. Composing carrier sets when composing languages or language fragments

$$\textbf{combine } Sem\_Expr :: Env \rightarrow Val \textbf{ with } Sem\_Cmd :: Sto \rightarrow Sto \textbf{ so that}$$
$$Sem\_Cmd :: Env \rightarrow Sto \rightarrow (Val, Sto)$$

How to define the carrier sets and the evaluation functions?

problems:

1. Defining pure evaluation functions for operations with effects
   solution: auxiliary semantic entities (carrier sets become functions)
2. Composing carrier sets when composing languages or language fragments

$$\textbf{combine } \mathit{Sem\_Expr} :: \mathit{Env} \rightarrow \mathit{Val} \textbf{ with } \mathit{Sem\_Cmd} :: \mathit{Sto} \rightarrow \mathit{Sto} \textbf{ so that}$$
$$\mathit{Sem\_Cmd} :: \mathit{Env} \rightarrow \mathit{Sto} \rightarrow (\mathit{Val}, \mathit{Sto})$$

   solutions: object algebras? effect handlers? carrier gen, fixed entity classes

How to define the carrier sets and the evaluation functions?

problems:

1. Defining pure evaluation functions for operations with effects
   solution: auxiliary semantic entities (carrier sets become functions)
2. Composing carrier sets when composing languages or language fragments

   $$\textbf{combine } \mathit{Sem\_Expr} :: \mathit{Env} \rightarrow \mathit{Val} \textbf{ with } \mathit{Sem\_Cmd} :: \mathit{Sto} \rightarrow \mathit{Sto} \textbf{ so that } \mathit{Sem\_Cmd} :: \mathit{Env} \rightarrow \mathit{Sto} \rightarrow (\mathit{Val}, \mathit{Sto})$$

   solutions: object algebras? effect handlers? carrier gen, fixed entity classes
3. Defining *modular*, pure evaluation functions for operations with effects

How to define the carrier sets and the evaluation functions?

problems:

1. Defining pure evaluation functions for operations with effects
   solution: auxiliary semantic entities (carrier sets become functions)
2. Composing carrier sets when composing languages or language fragments

$$\textbf{combine } \mathit{Sem\_Expr} :: \mathit{Env} \rightarrow \mathit{Val} \textbf{ with } \mathit{Sem\_Cmd} :: \mathit{Sto} \rightarrow \mathit{Sto} \textbf{ so that}$$
$$\mathit{Sem\_Cmd} :: \mathit{Env} \rightarrow \mathit{Sto} \rightarrow (\mathit{Val}, \mathit{Sto})$$

   solutions: object algebras? effect handlers? carrier gen, fixed entity classes
3. Defining *modular*, pure evaluation functions for operations with effects
   solution: implicit propagation schemes for auxiliary semantic entities

Different techniques vary wildly in how propagation schemes are defined/implemented:

Different techniques vary wildly in how propagation schemes are defined/implemented:

- **MSOS**: Every entity is an instance of a category $\mathbb{C}$.
  The composition operator of the category determines how values are propagated.
  All entities together form a product category

Different techniques vary wildly in how propagation schemes are defined/implemented:

- **MSOS**: Every entity is an instance of a category $\mathbb{C}$.
  The composition operator of the category determines how values are propagated.
  All entities together form a product category
- **I-MSOS**: The formalism chooses certain MSOS categories and provides syntax to indicate for each entity of which category it is an instance of (entity classes)

Different techniques vary wildly in how propagation schemes are defined/implemented:

- **MSOS**: Every entity is an instance of a category $\mathbb{C}$.
  The composition operator of the category determines how values are propagated.
  All entities together form a product category
- **I-MSOS**: The formalism chooses certain MSOS categories and provides syntax to indicate for each entity of which category it is an instance of (entity classes)
- **Monads/Monad transformers**: Every entity is an instance of a monad.
  The bind operator defines how its values are propagated.
  All entities are composed by either defining a monolithic super-monad or by composing monad-transformers

Different techniques vary wildly in how propagation schemes are defined/implemented:

- **MSOS**: Every entity is an instance of a category $\mathbb{C}$.
  The composition operator of the category determines how values are propagated.
  All entities together form a product category
- **I-MSOS**: The formalism chooses certain MSOS categories and provides syntax to indicate for each entity of which category it is an instance of (entity classes)
- **Monads/Monad transformers**: Every entity is an instance of a monad.
  The bind operator defines how its values are propagated.
  All entities are composed by either defining a monolithic super-monad or by composing monad-transformers
- **Utrecht University Attribute Grammars (UUAGs)**: Every entity is an attribute. Missing attribute equations are generated according to built-in schemes

Different techniques vary wildly in how propagation schemes are defined/implemented:

- **MSOS**: Every entity is an instance of a category $\mathbb{C}$.
  The composition operator of the category determines how values are propagated.
  All entities together form a product category
- **I-MSOS**: The formalism chooses certain MSOS categories and provides syntax to indicate for each entity of which category it is an instance of (entity classes)
- **Monads/Monad transformers**: Every entity is an instance of a monad.
  The bind operator defines how its values are propagated.
  All entities are composed by either defining a monolithic super-monad or by composing monad-transformers
- **Utrecht University Attribute Grammars (UUAGs)**: Every entity is an attribute. Missing attribute equations are generated according to built-in schemes
- **CBS & funcons** implementation: I-MSOS + monolithic super-monad

formalism	**entity classes**				
CBS & funcons	contextual	mutable	output	control	_
MSOS (categories)	discrete	preorder	monoidal	abrupt term.	...
I-MSOS (2008)	read-only	updateable	emittable	_	_
Haskell Monads	reader	state	writer	exception	...
Attribute grammars	inherited	chained	synthesized	_	_

formalism	**entity classes**				
CBS & funcons	contextual	mutable	output	control	_
MSOS (categories)	discrete	preorder	monoidal	abrupt term.	...
I-MSOS (2008)	read-only	updateable	emittable	_	_
Haskell Monads	reader	state	writer	exception	...
Attribute grammars	inherited	chained	synthesized	_	_

Language aspects covered by CBS & funcons

- Procedural: procedures, references, scoping, iteration
- Functional: functions, bindings, datatypes, pattern matching
- Object-oriented: objects, classes, inheritance
- Abnormal control: exceptions, break/continue, delimited continuations
- Concurrency: multi-threading
- Logical programming: backtracking, unification
- Meta-programming: AST conversions, staged evaluation

To conclude, *I am an old-fashioned guy*:

- Grammar-first
- (Modular) Structural Operational Semantics, (Modular) Attribute Grammars
- Answer to every question is a collection of pure, higher-order functions
- (ideally with a strong static types)

But eager to learn new things: object algebras, meta-modelling with Ecore and Ale(x)

A Guided Tour of Initial Algebra Semantics

L. Thomas van Binsbergen
`https://plancomps.github.io`

Centrum Wiskunde & Informatica
ltvanbinsbergen@acm.org

17 December, 2019

Centrum Wiskunde & Informatica

Contextual - Contextual information only, no effects

For example: environments collecting bindings active only in certain scopes

$$x := 0;\ \textbf{while}\ x \leqslant 10\ \textbf{do}\ x := x + 1; \textbf{print}\ x\ \textbf{done}$$

Contextual - Contextual information only, no effects

For example: environments collecting bindings active only in certain scopes

$$x := 0;\ \textbf{while}\ x \leqslant 10\ \textbf{do}\ x := x + 1; \textbf{print}\ x\ \textbf{done}$$

$$\begin{aligned}
\textbf{seq}(c_1, c_2) &= \textbf{accumulate}(c_1, c_2) \\
\textbf{assign}(x, e) &= \textbf{else}(\mathit{command}(\textbf{assign}(\textbf{bound}(x), e)), \textbf{bind}(x, \textbf{alloc-init}(\textbf{values}, e))) \\
\mathit{command}(e) &= \textbf{seq}(\textbf{effect}(e), \textbf{map-empty})
\end{aligned}$$

Contextual - Contextual information only, no effects

For example: environments collecting bindings active only in certain scopes

$$x := 0;\ \textbf{while}\ x \leqslant 10\ \textbf{do}\ x := x + 1; \textbf{print}\ x\ \textbf{done}$$

$$\begin{aligned}
\textbf{seq}(c_1, c_2) &= \textbf{accumulate}(c_1, c_2) \\
\textbf{assign}(x, e) &= \textbf{else}(\mathit{command}(\textbf{assign}(\textbf{bound}(x), e)), \textbf{bind}(x, \textbf{alloc-init}(\textbf{values}, e))) \\
\mathit{command}(e) &= \textbf{seq}(\textbf{effect}(e), \textbf{map-empty})
\end{aligned}$$

Contextual entity propagation

- Contextual entities appear as parameters to evaluation functions
- Automatically copied from ‘parent’ to ‘children’

Mutable - Contextual information, may mutate

For example: mutable references in a store, fresh atom generation

$$x := 0;\ \textbf{while}\ x \leqslant 10\ \textbf{do}\ x := x + 1; \textbf{print}\ x\ \textbf{done}$$

$$\begin{aligned}
\textbf{seq}(c_1, c_2) &= \textbf{accumulate}(c_1, c_2)\\
\textbf{assign}(x, e) &= \textbf{else}(\mathit{command}(\textbf{assign}(\textbf{bound}(x), e)), \textbf{bind}(x, \textbf{alloc-init}(\textbf{values}, e)))\\
\mathit{command}(e) &= \textbf{seq}(\textbf{effect}(e), \textbf{map-empty})
\end{aligned}$$

Mutable entity propagation

- Mutable entities appear as parameters and results
- Deterministic semantics require a linear (evaluation) order over operands
- Automatically copied in ‘around-the-clock’ fashion, determined by linear order

Output - Accumulating effects only

For example: printed values, reporting errors/warnings in a static analysis

$$\mathbf{print}(e) = \mathit{command}(\mathbf{seq}(\mathbf{print}(\mathbf{to\text{-}string}(e)), \mathbf{print}(\mathbf{line\text{-}feed})))$$

Output - Accumulating effects only

For example: printed values, reporting errors/warnings in a static analysis

$$\textbf{print}(e) = \textit{command}(\textbf{seq}(\textbf{print}(\textbf{to-string}(e)), \textbf{print}(\textbf{line-feed})))$$

Output entity propagation

- Output entities appear as results only
- Output needs to form a monoid with associative $\otimes$ and identity element
- Deterministic semantics require a linear (evaluation) order over operands (unless $\otimes$ commutative)

Control - Halting effects only, maybe "handled" by context

For example: pattern match failure, exceptions, continue/break/return statements

$$x := 0;\ \textbf{while}\ x \leqslant 10\ \textbf{do}\ x := x + 1;\ \textbf{continue}\ ; x := x + 1; \textbf{print}\ x\ \textbf{done}$$

$$\textbf{while}(e, c) = \textit{command}(\textbf{while}(e, \textbf{handle-thrown}(\textbf{effect}(c), \textbf{null})))$$

$$\textbf{continue}\ () = \textbf{throw}(\texttt{"continue"})$$

Control - Halting effects only, maybe "handled" by context

For example: pattern match failure, exceptions, continue/break/return statements

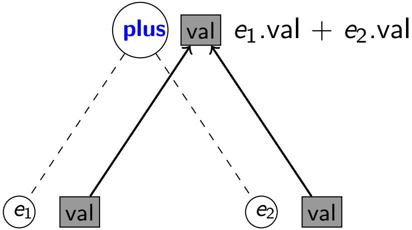
$$x := 0;\ \textbf{while}\ x \leqslant 10\ \textbf{do}\ x := x + 1;\ \textbf{continue}\ ; x := x + 1; \textbf{print}\ x\ \textbf{done}$$

$$\textbf{while}(e, c) = \textit{command}(\textbf{while}(e, \textbf{handle-thrown}(\textbf{effect}(c), \textbf{null})))$$

$$\textbf{continue}\ () = \textbf{throw}(\texttt{"continue"})$$

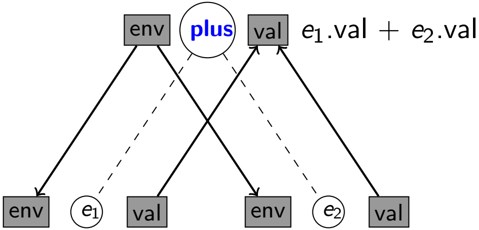
Control entity propagation

- Control entities are optional values that appear as results only
- The presence of a control entity halts evaluation,
- The 'closest' handler-operator will remove the entity and invoke its handler
- Deterministic semantics require linear order over operands



$$\textbf{plus}(e_1, e_2) = e_1() + e_2()$$

Inherited entities



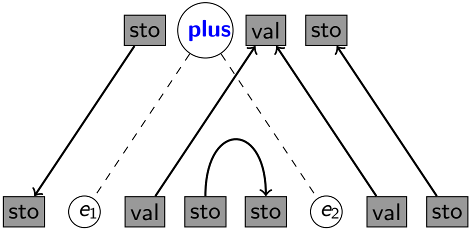
$$\textbf{plus}(e_1, e_2)(\rho) = e_1(\rho) + e_2(\rho)$$

Inherited entity propagation

- Inherited entities appear as parameters to evaluation functions
- Values are copied from ‘parent’ to ‘children’ (occurrence of operation to operands)

$$\textbf{seq}(c_1, c_2)(\rho) = c_2(\rho \cdot \rho')$$
$$\textbf{where } \rho' = c_1(\rho)$$

Mutable entities

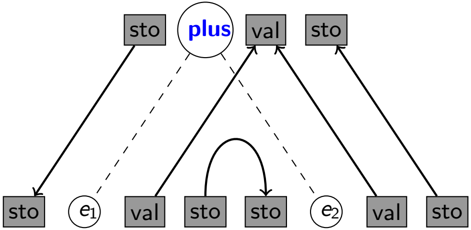


$$\textbf{plus}(e_1, e_2)(\sigma_0) = \langle v_1 + v_2, \sigma_2\rangle$$
$$\textbf{where}\ \langle v_1, \sigma_1\rangle = e_1(\sigma_0)$$
$$\textbf{and}\quad \langle v_2, \sigma_2\rangle = e_2(\sigma_1)$$

Mutable entity propagation

- Mutable entities appear as parameters and results
- Deterministic semantics require a linear order over operands
- Values are copied in 'counter-clockwise' fashion, determined by linear order

Mutable entities

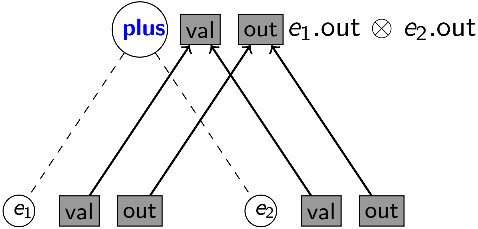


$$\textbf{plus}(e_1, e_2)(\sigma_0) = \langle v_1 + v_2, \sigma_2\rangle$$
$$\textbf{where } \langle v_1, \sigma_1\rangle = e_1(\sigma_0)$$
$$\textbf{and } \langle v_2, \sigma_2\rangle = e_2(\sigma_1)$$

$$\textbf{assign}(x, e)(\rho, \sigma) = \begin{cases} \langle\{x \mapsto r\}, \sigma[r \mapsto v]\rangle & \bot = \rho(x), r \textbf{ fresh in } \sigma \\ \langle\emptyset, \sigma[r \mapsto v]\rangle & r = \rho(x) \end{cases}$$

$$\textbf{where } v = e(\rho, \sigma)$$

Output entities



$$\textbf{plus}(e_1, e_2) = \langle v_1 + v_2, \alpha \otimes \beta \rangle$$
$$\textbf{where } \langle v_1, \alpha \rangle = e_1()$$
$$\textbf{and } \langle v_2, \beta \rangle = e_2()$$

Output entity propagation

- Output entities appear as results only
- Output needs to form a monoid with associative $\otimes$ and identity element
- Deterministic semantics require a linear order over operands (unless $\otimes$ commutative)