# Towards Incremental Language Definition with Reusable Components

Damian Frölich and L. Thomas van Binsbergen

Informatics Institute, University of Amsterdam
{dfrolich,ltvanbinsbergen}@acm.org

September 3, 2021 – IFL 2021

# Definitional interpreters

$$\textit{interpreter} : \textit{program} \times \textit{config} \rightarrow \textit{config}$$

# Definitional interpreters

A **language** $L$ is a structure $\langle P, \Gamma, \gamma^0, I \rangle$ with:

- $P$ a set of programs,
- $\Gamma$ a set of configurations (containing semantic entities, attributes, algebraic effects, etc..),
- $\gamma^0$ an initial configuration with $\gamma^0 \in \Gamma$ and
- $I$ a definitional interpreter assigning to each program $p \in P$ a function $I_p : \Gamma \to \Gamma$.

$$\mathit{interpreter} : \mathit{program} \times \mathit{config} \to \mathit{config}$$

# Definitional interpreters

A **language** $L$ is a structure $\langle P, \Gamma, \gamma^0, I\rangle$ with:

- $P$ a set of programs,
- $\Gamma$ a set of configurations (containing semantic entities, attributes, algebraic effects, etc..),
- $\gamma^0$ an initial configuration with $\gamma^0 \in \Gamma$ and
- $I$ a definitional interpreter assigning to each program $p \in P$ a function $I_p : \Gamma \to \Gamma$.

$$\mathit{interpreter} : \mathit{program} \times \mathit{config} \to \mathit{config}$$

Note that the interpreter can be applied repeatedly, i.e. that effects can be composed

# Deriving REPLs and Notebooks for DSLs

**From DSL Specification to Interactive Computer Programming Environment**


Pierre Jeanjean
Inria, Univ Rennes, CNRS, IRISA
Rennes, France
pierre.jeanjean@inria.fr

Benoit Combemale
University of Toulouse
Toulouse, France
benoit.combemale@irit.fr

Olivier Barais
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
olivier.barais@irisa.fr


Figure: SLE2019

**Bacatá: Notebooks for DSLs, Almost for Free**


Mauricio Verano Merino[a,d], Jurgen Vinju[a,b], and Tijs van der Storm[b,c]

a Eindhoven University of Technology, The Netherlands

b Centrum Wiskunde & Informatica, The Netherlands

c University of Groningen, The Netherlands

d Océ Technologies B.V., The Netherlands


Figure: Art, Science, and Engineering of Programming

# Deriving REPL/Notebook – commonalities

- READ: Identify entry points, i.e. the alternatives in syntactic root
- EVAL: Connect entry points with evaluation function in DSL interpreter
- PRINT: Specify function to convert evaluation result to string
- LOOP:

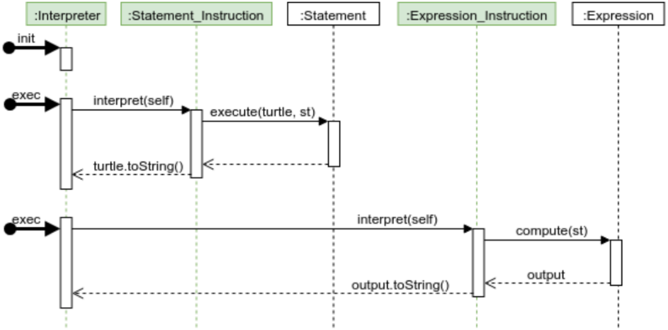



**Figure 8.** Overall Execution Flow for Logo

# Deriving REPL/Notebook – commonalities

- READ: Identify entry points, i.e. the alternatives in syntactic root
- EVAL: Connect entry points with evaluation function in DSL interpreter
- PRINT: Specify function to convert evaluation result to string
- LOOP:

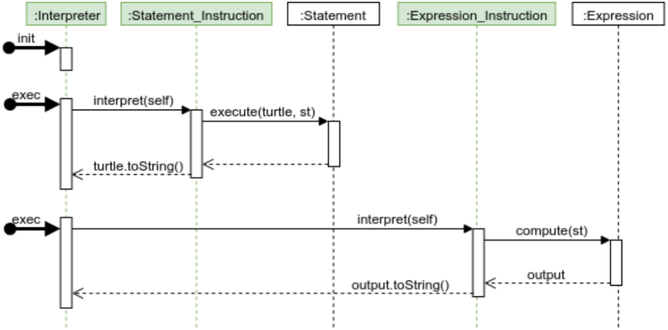



**Figure 8.** Overall Execution Flow for Logo

How does one execution affect the next?

# Idea..!

*Distinguish between REPL language and base language (e.g. JShell vs Java)*

## A Principled Approach to REPL Interpreters


L. Thomas van Binsbergen
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
ltvanbinsbergen@acm.org

Mauricio Verano Merino
Eindhoven University of Technology
Eindhoven, The Netherlands
m.verano.merino@tue.nl

Pierre Jeanjean
Inria, University of Rennes, CRNS,
IRISA
Rennes, France
pierre.jeanjean@inria.fr

Tijs van der Storm
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
University of Groningen
Groningen, The Netherlands
storm@cwi.nl

Benoit Combemale
University of Rennes, Inria, CNRS,
IRISA
Rennes, France
benoit.combemale@irit.fr

Olivier Barais
University of Rennes, Inria, CNRS,
IRISA
Rennes, France
olivier.barais@irisa.fr


Figure: Onward!2020

# Observation..!

*REPLs with incremental execution implement a language with the following property:*

# Observation..!

*REPLs with incremental execution implement a language with the following property:*

A **sequential language** is a language in which $p_1; p_2$ is a (syntactically) valid program iff $p_1$ and $p_2$ are valid programs and iff $p_1; p_2$ is equivalent to 'doing' $p_1$ and then $p_2$

# Observation..!

*REPLs with incremental execution implement a language with the following property:*

A **sequential language** is a language in which $p_1; p_2$ is a (syntactically) valid program iff $p_1$ and $p_2$ are valid programs and iff $p_1; p_2$ is equivalent to 'doing' $p_1$ and then $p_2$

$$\mathit{interpreter}(p_1; p_2)(\gamma) == \mathit{interpreter}(p_2)(\mathit{interpreter}(p_1)(\gamma))$$

# Observation..!

*REPLs with incremental execution implement a language with the following property:*

A **sequential language** is a language in which $p_1; p_2$ is a (syntactically) valid program iff $p_1$ and $p_2$ are valid programs and iff $p_1; p_2$ is equivalent to 'doing' $p_1$ and then $p_2$

$$\mathit{interpreter}(p_1; p_2)(\gamma) == \mathit{interpreter}(p_2)(\mathit{interpreter}(p_1)(\gamma))$$

$$[\![p_1; p_2]\!] = [\![p_2]\!] \circ [\![p_1]\!]$$

# Observation..!

*REPLs with incremental execution implement a language with the following property:*

A **sequential language** is a language in which $p_1; p_2$ is a (syntactically) valid program iff $p_1$ and $p_2$ are valid programs and iff $p_1; p_2$ is equivalent to 'doing' $p_1$ and then $p_2$

$$\mathit{interpreter}(p_1; p_2)(\gamma) == \mathit{interpreter}(p_2)(\mathit{interpreter}(p_1)(\gamma))$$

$$[\![p_1; p_2]\!] = [\![p_2]\!] \circ [\![p_1]\!]$$

*A REPL is a monoid homomorphism between programs and their effects*

# REPLization in Onward!2020

Replization *is: extending a base language to a sequential variant*

# REPLization in Onward!2020

Replization *is: extending a base language to a sequential variant*

1. Define the syntax of the extended language (**phrases**/**entry points**)

# REPLization in Onward!2020

Replization *is: extending a base language to a sequential variant*

1. Define the syntax of the extended language (**phrases**/**entry points**)
2. Implement definitional interpreter by choosing $\Gamma$ and in terms of base interpreter

# REPLization in Onward!2020

Replization *is: extending a base language to a sequential variant*

1. Define the syntax of the extended language (**phrases**/**entry points**)
2. Implement definitional interpreter by choosing Γ and in terms of base interpreter
3. Add phrase composition operator to the language (it is now sequential by definition)

$$[\![p_1 \otimes p_2]\!] = [\![p_2]\!] \circ [\![p_1]\!]$$

# REPLization in Onward!2020

Replization *is: extending a base language to a sequential variant*

1. Define the syntax of the extended language (**phrases**/**entry points**)
2. Implement definitional interpreter by choosing $\Gamma$ and in terms of base interpreter
3. Add phrase composition operator to the language (it is now sequential by definition)

$$[\![p_1 \otimes p_2]\!] = [\![p_2]\!] \circ [\![p_1]\!]$$

- The effect of one phrase on the next is determined by its modifications to $\gamma \in \Gamma$

# REPLization in Onward!2020

Replization *is: extending a base language to a sequential variant*

1. Define the syntax of the extended language (**phrases**/**entry points**)
2. Implement definitional interpreter by choosing $\Gamma$ and in terms of base interpreter
3. Add phrase composition operator to the language (it is now sequential by definition)

$$[\![p_1 \otimes p_2]\!] = [\![p_2]\!] \circ [\![p_1]\!]$$

- The effect of one phrase on the next is determined by its modifications to $\gamma \in \Gamma$
- REPL-first: what if we design all our languages as sequential languages?

# Onward!2020 (MiniJava case study)

```
Config eval((Phrase)`<Expression e> ;`, Config c)
  = catchExceptions(collectBindings(
      setOutput(createBinding(eval(c, e)))));

Config eval((Phrase)`<Statement s>`, Config c)
  = catchExceptions(collectBindings(
      setOutput(exec(s, c))));

Config eval((Phrase)`<ClassDecl cd>`, Config c)
  = catchExceptions(collectBindings(
      declareClass(cd, c)));

Config eval((Phrase)`<VarDecl vd>`, Config c)
  = catchExceptions(collectBindings(
      declareVariables(vd, c)));

Config eval((Phrase)`<MethodDecl md>`, Config c)
  = catchExceptions(collectBindings(
      declareGlobalMethod(md, c)));

Config eval((Phrase)`<Phrase p1> <Phrase p2>`, Config c)
  = eval(p2, eval(p1, c));
```

# Onward!2020 (MiniJava case study)

```
Config eval((Phrase)`<Expression e> ;`, Config c)
  = catchExceptions(collectBindings(
      setOutput(createBinding(eval(c, e)))));

Config eval((Phrase)`<Statement s>`, Config c)
  = catchExceptions(collectBindings(
      setOutput(exec(s, c))));

Config eval((Phrase)`<ClassDecl cd>`, Config c)
  = catchExceptions(collectBindings(
      declareClass(cd, c)));

Config eval((Phrase)`<VarDecl vd>`, Config c)
  = catchExceptions(collectBindings(
      declareVariables(vd, c)));

Config eval((Phrase)`<MethodDecl md>`, Config c)
  = catchExceptions(collectBindings(
      declareGlobalMethod(md, c)));

Config eval((Phrase)`<Phrase p1> <Phrase p2>`, Config c)
  = eval(p2, eval(p1, c));
```

## What if?

- The chosen entry points came from different languages?

# Onward!2020 (MiniJava case study)

```
Config eval((Phrase)`<Expression e> ;`, Config c)
  = catchExceptions(collectBindings(
      setOutput(createBinding(eval(c, e)))));

Config eval((Phrase)`<Statement s>`, Config c)
  = catchExceptions(collectBindings(
      setOutput(exec(s, c))));

Config eval((Phrase)`<ClassDecl cd>`, Config c)
  = catchExceptions(collectBindings(
      declareClass(cd, c)));

Config eval((Phrase)`<VarDecl vd>`, Config c)
  = catchExceptions(collectBindings(
      declareVariables(vd, c)));

Config eval((Phrase)`<MethodDecl md>`, Config c)
  = catchExceptions(collectBindings(
      declareGlobalMethod(md, c)));

Config eval((Phrase)`<Phrase p1> <Phrase p2>`, Config c)
  = eval(p2, eval(p1, c));
```

## What if?

- The chosen entry points came from different languages?

$\hookrightarrow$ ‘coarse-grained’ language composition

# Onward!2020 (MiniJava case study)

```
Config eval((Phrase)`<Expression e> ;`, Config c)
  = catchExceptions(collectBindings(
      setOutput(createBinding(eval(c, e)))));

Config eval((Phrase)`<Statement s>`, Config c)
  = catchExceptions(collectBindings(
      setOutput(exec(s, c))));

Config eval((Phrase)`<ClassDecl cd>`, Config c)
  = catchExceptions(collectBindings(
      declareClass(cd, c)));

Config eval((Phrase)`<VarDecl vd>`, Config c)
  = catchExceptions(collectBindings(
      declareVariables(vd, c)));

Config eval((Phrase)`<MethodDecl md>`, Config c)
  = catchExceptions(collectBindings(
      declareGlobalMethod(md, c)));

Config eval((Phrase)`<Phrase p1> <Phrase p2>`, Config c)
  = eval(p2, eval(p1, c));
```

## What if?

- The chosen entry points came from different languages?

$\hookrightarrow$ 'coarse-grained' language composition

- And had different configurations? i.e. other semantic entities?

# Onward!2020 (MiniJava case study)

```
Config eval((Phrase)`<Expression e> ;`, Config c)
  = catchExceptions(collectBindings(
      setOutput(createBinding(eval(c, e)))));

Config eval((Phrase)`<Statement s>`, Config c)
  = catchExceptions(collectBindings(
      setOutput(exec(s, c))));

Config eval((Phrase)`<ClassDecl cd>`, Config c)
  = catchExceptions(collectBindings(
      declareClass(cd, c)));

Config eval((Phrase)`<VarDecl vd>`, Config c)
  = catchExceptions(collectBindings(
      declareVariables(vd, c)));

Config eval((Phrase)`<MethodDecl md>`, Config c)
  = catchExceptions(collectBindings(
      declareGlobalMethod(md, c)));

Config eval((Phrase)`<Phrase p1> <Phrase p2>`, Config c)
  = eval(p2, eval(p1, c));
```

## What if?

- The chosen entry points came from different languages?

$\hookrightarrow$ 'coarse-grained' language composition

- And had different configurations? i.e. other semantic entities?

$\hookrightarrow$ We need modular interpreters

# Onward!2020 (MiniJava case study)

```
Config eval((Phrase)`<Expression e> ;`, Config c)
  = catchExceptions(collectBindings(
      setOutput(createBinding(eval(c, e)))));

Config eval((Phrase)`<Statement s>`, Config c)
  = catchExceptions(collectBindings(
      setOutput(exec(s, c))));

Config eval((Phrase)`<ClassDecl cd>`, Config c)
  = catchExceptions(collectBindings(
      declareClass(cd, c)));

Config eval((Phrase)`<VarDecl vd>`, Config c)
  = catchExceptions(collectBindings(
      declareVariables(vd, c)));

Config eval((Phrase)`<MethodDecl md>`, Config c)
  = catchExceptions(collectBindings(
      declareGlobalMethod(md, c)));

Config eval((Phrase)`<Phrase p1> <Phrase p2>`, Config c)
  = eval(p2, eval(p1, c));
```

## What if?

- The chosen entry points came from different languages?

$\hookrightarrow$ 'coarse-grained' language composition

- And had different configurations? i.e. other semantic entities?

$\hookrightarrow$ We need modular interpreters

- We had a shared notion of configuration for all languages?

# Onward!2020 (MiniJava case study)

```
Config eval((Phrase)`<Expression e> ;`, Config c)
  = catchExceptions(collectBindings(
      setOutput(createBinding(eval(c, e)))));

Config eval((Phrase)`<Statement s>`, Config c)
  = catchExceptions(collectBindings(
      setOutput(exec(s, c))));

Config eval((Phrase)`<ClassDecl cd>`, Config c)
  = catchExceptions(collectBindings(
      declareClass(cd, c)));

Config eval((Phrase)`<VarDecl vd>`, Config c)
  = catchExceptions(collectBindings(
      declareVariables(vd, c)));

Config eval((Phrase)`<MethodDecl md>`, Config c)
  = catchExceptions(collectBindings(
      declareGlobalMethod(md, c)));

Config eval((Phrase)`<Phrase p1> <Phrase p2>`, Config c)
  = eval(p2, eval(p1, c));
```

## What if?

- The chosen entry points came from different languages?

$\hookrightarrow$ 'coarse-grained' language composition

- And had different configurations? i.e. other semantic entities?

$\hookrightarrow$ We need modular interpreters

- We had a shared notion of configuration for all languages?

$\hookrightarrow$ Such as suggested for 'funcons'

# PLanCompS project (2011-2015...) – `http://plancomps.org`

- 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 funcons and object languages[1]
  - A method for translating funcon definitions to executable micro-interpreters[1]

- Funcons are defined in I-MSOS with a fixed set of entity *classes*

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

# What is the state of the funcon library?

**Verified and available: `https://plancomps.github.io/CBS-beta/Funcons-beta/`**

- 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 (i.e. not used in large case studies)**

- 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

```
Rule
  initialise[[ 'function' Id '(' Ids? ')' Block ]] =
    assign(
      bound(id[[ Id ]]),
      function closure(
        scope(
          match(given,tuple(patts[[ Ids? ]])),
          handle-return(exec[[ Block ]]))))
```

```
Rule
  rval[[ Exp '(' Exps? ')' ]] = apply(rval[[ Exp ]], tuple(rvals[[ Exps? ]]) )
```

# Incremental language definition with reusable components

**Modular reusable operators definitions, determining:**

- The arity and signature (sorts) of an operator, i.e. abstract syntax
- A semantic function expressing a translation to funcons
- Optionally: a context-free grammar production rule

# Incremental language definition with reusable components

## Modular reusable operators definitions, determining:

- The arity and signature (sorts) of an operator, i.e. abstract syntax
- A semantic function expressing a translation to funcons
- Optionally: a context-free grammar production rule

## Language definition

A language is defined by (in the context of some operator declarations):

- Assigning operators to the 'top-level', e.g. the entry-points (coarse-grained composition)
- Assigning operators to operand positions (fine-grained composition)

$\hookrightarrow$ Determines the structure of the abstract syntax and a denotational semantics

# Incremental language definition with reusable components

## Modular reusable operators definitions, determining:

- The arity and signature (sorts) of an operator, i.e. abstract syntax
- A semantic function expressing a translation to funcons
- Optionally: a context-free grammar production rule

## Language definition

A language is defined by (in the context of some operator declarations):

- Assigning operators to the 'top-level', e.g. the entry-points (coarse-grained composition)
- Assigning operators to operand positions (fine-grained composition)

$\hookrightarrow$ Determines the structure of the abstract syntax and a denotational semantics

## Incremental? Language experimentation in a REPL/Notebook

Develop the specification as a sequence of operator declarations and sort constraints

# Example

## Conventional approach (e.g. ADTs or Variants)

$$
\begin{aligned}
\mathit{Var}_{\mathscr{O}} &: \mathit{String} \rightarrow \mathit{Expr} \\
\mathit{Abs}_{\mathscr{O}} &: \mathit{String} \times \mathit{Expr} \rightarrow \mathit{Expr} \\
\mathit{App}_{\mathscr{O}} &: \mathit{Expr} \times \mathit{Expr} \rightarrow \mathit{Expr}
\end{aligned}
$$

# Example

## Conventional approach (e.g. ADTs or Variants)

$$Var_{\mathscr{O}} : String \to Expr$$

$$Abs_{\mathscr{O}} : String \times Expr \to Expr$$

$$App_{\mathscr{O}} : Expr \times Expr \to Expr$$

## Alternative approach

$$Var_{\mathscr{O}} : String$$

$$Abs_{\mathscr{O}} : String \times AbsBody$$

$$App_{\mathscr{O}} : AppAbs \times AppArg$$

# Example

## Conventional approach (e.g. ADTs or Variants)

$$\begin{aligned} Var_{@} &: String \to Expr \\ Abs_{@} &: String \times Expr \to Expr \\ App_{@} &: Expr \times Expr \to Expr \end{aligned}$$

## Alternative approach

$$\begin{aligned} Var_{@} &: String \\ Abs_{@} &: String \times AbsBody \\ App_{@} &: AppAbs \times AppArg \end{aligned}$$

$$\begin{aligned} Var_{@} &\in Expr \\ App_{@} &\in Expr \\ Abs_{@} &\in Expr \\ Expr &\subseteq AbsBody \\ Expr &\subseteq AppAbs \\ Expr &\subseteq AppArg \end{aligned}$$

# Incremental language development (abstract syntax)

$$
\begin{aligned}
Var_{\mathscr{O}} &: String \\
Abs_{\mathscr{O}} &: String \times AbsBody \\
App_{\mathscr{O}} &: AppAbs \times AppArg
\end{aligned}
$$

Operator declarations introduce operators, arities and name ‘operand positions’

$$
\begin{aligned}
Var_{\mathscr{F}}(lit) &= \textbf{bound string } lit \\
Abs_{\mathscr{F}}(x, b) &= \textbf{function closure scope}(\textbf{bind}(\textbf{string } x, \textbf{given}), b) \\
App_{\mathscr{F}}(abs, arg) &= \textbf{apply}(abs, arg)
\end{aligned}
$$

Semantic functions translate operator occurrences to funcon terms (semantic domain).

# Incremental language development (language construction)

Sort constraints assign (one or more) operators to (possibly new) sorts.

$$
\begin{aligned}
\mathit{Var}_{\mathscr{O}} &\in \mathit{Expr} \\
\mathit{App}_{\mathscr{O}} &\in \mathit{Expr} \\
\mathit{Abs}_{\mathscr{O}} &\in \mathit{Expr} \\
\mathit{Expr} &\subseteq \mathit{AbsBody} \\
\mathit{Expr} &\subseteq \mathit{AppAbs} \\
\mathit{Expr} &\subseteq \mathit{AppArg}
\end{aligned}
$$

Sort constraints determine the precise relations between operators and operands

# Glue code (problem)

$$
\begin{aligned}
\mathit{Var}_{\mathscr{F}}(\mathit{lit}) &= \textbf{bound string } \mathit{lit} \\
\mathit{Abs}_{\mathscr{F}}(x, b) &= \textbf{function closure scope}(\textbf{bind}(\textbf{string } x, \textbf{given}), b) \\
\mathit{App}_{\mathscr{F}}(\mathit{abs}, \mathit{arg}) &= \textbf{apply}(\mathit{abs}, \mathit{arg})
\end{aligned}
$$

What if the body of an abstract can terminate abruply? e.g. due to a return command.

$$
\mathit{Abs}_{\mathscr{F}}(x, b) = \textbf{function closure scope}(\textbf{bind}(\textbf{string } x, \textbf{given}), \textbf{handle-return } b)
$$

# Glue code (possible solution)

Associating 'wrapper funcon terms' as part of sort constraints

$$
\begin{aligned}
Return_{\mathscr{O}} &: ReturnVal && \text{(Operator declaration)}\\
Return_{\mathscr{F}}(val) &= \textbf{return}\ val && \text{(Semantic function)}\\
Return_{\mathscr{O}} &\in Command && \text{(Sort constraint)}\\
Command &\subseteq AbsBody && \text{(Sort constraint with glue code)}\\
&\hookrightarrow \textbf{handle-return}(Command_{\mathscr{F}}) && \text{(glue code)}
\end{aligned}
$$

# Full paper

## Realisation

- Haskell EDSL implementation reflecting our approach
- Building on from (Swierstra 2008) and (Bahr & Hvitved 2011)
- Enforce sort constraints and language definedness through Haskell's type system
- Optionally: GLL combinators for concrete syntax (Van Binsbergen et al. 2018)

## Evaluation

- Case studies to demonstrate: use of glue code, language variations, etc.
- Positioning within meta-language analysis frameworks of (Erdweg et al. 2012), (Méndez-Acuña et al. 2016), and/or (Leduc et al. 2019)
- Comparison with related work

# Towards Incremental Language Definition with Reusable Components

Damian Frölich and L. Thomas van Binsbergen

Informatics Institute, University of Amsterdam
{dfrolich,ltvanbinsbergen}@acm.org

September 3, 2021 – IFL 2021

Techniques vary in how effects (entities below) are implicitly propagated:

Techniques vary in how effects (entities below) are implicitly propagated:

- **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

Techniques vary in how effects (entities below) are implicitly propagated:

- **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
- **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

Techniques vary in how effects (entities below) are implicitly propagated:

- **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
- **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)

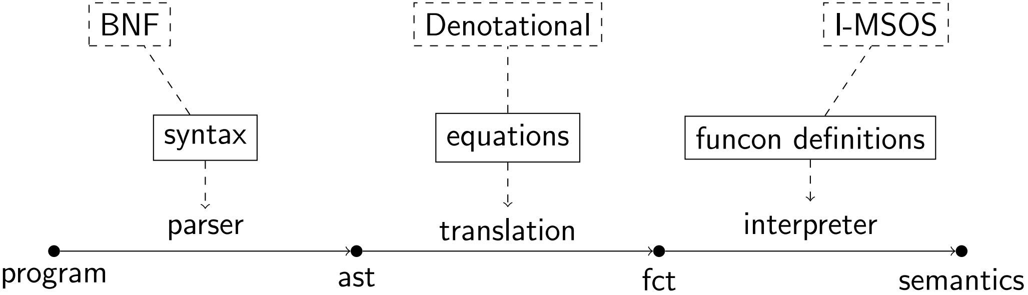
Techniques vary in how effects (entities below) are implicitly propagated:

- **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
- **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)
- **CBS & funcons** (topic of next slides):
  I-MSOS $+$ monolithic monad implementing the entity classes

Techniques vary in how effects (entities below) are implicitly propagated:

- **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
- **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)
- **CBS & funcons** (topic of next slides):
  I-MSOS + monolithic monad implementing the entity classes
- **Implicit equations in Attribute Grammars (e.g. UUAG)**: Every entity is an attribute.
  Missing attribute equations are generated according to built-in schemes

# Language Engineering with Funcons




*Can this pipeline support modular, incremental language development?*

... a requirement for Agile Language Engineering

- Funcons also have informal semantics (no need to always worry about the details!)
- I-MSOS funcon definitions serve as a reference

# Incremental language development (concrete syntax)

$$\mathit{Var}_{\mathscr{S}}(\mathit{lit}) ::= \mathit{lit} \qquad \text{(Syntax declaration)}$$

$$\mathit{App}_{\mathscr{S}}(\mathit{abs}, \mathit{arg}) ::= \mathit{abs}\ \mathit{arg} \qquad \text{(Syntax declaration)}$$

$$\mathit{Abs}_{\mathscr{S}}(\mathit{param}, \mathit{body}) ::= \texttt{'('}\ \texttt{'\textbackslash'}\ \mathit{param}\ \texttt{"->"}\ \mathit{body}\ \texttt{')'} \qquad \text{(Syntax declaration)}$$

In a syntax declaration, the operands are names for nonterminals, whose productions rules are determined by sort constraints and (other) syntax declarations.

# REPL feature model