Funcons

Executable Component-Based Semantics

L. Thomas van Binsbergen,
Neil Sculthorpe, Peter D. Mosses

Royal Holloway, University of London

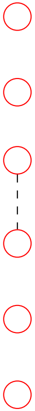
22 March, 2015

Section 1

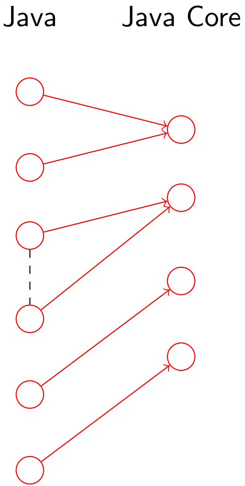
PLanCompS

Reusable Components: Funcons

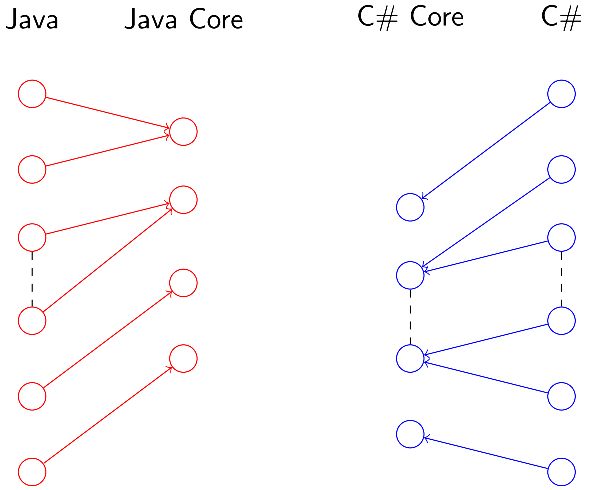
Java



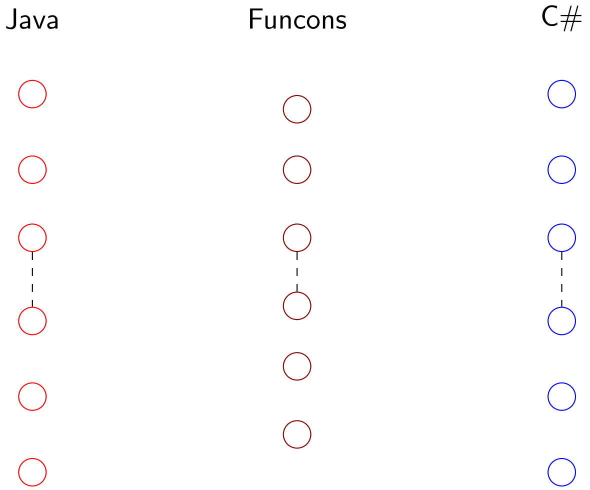
Reusable Components: Funcons



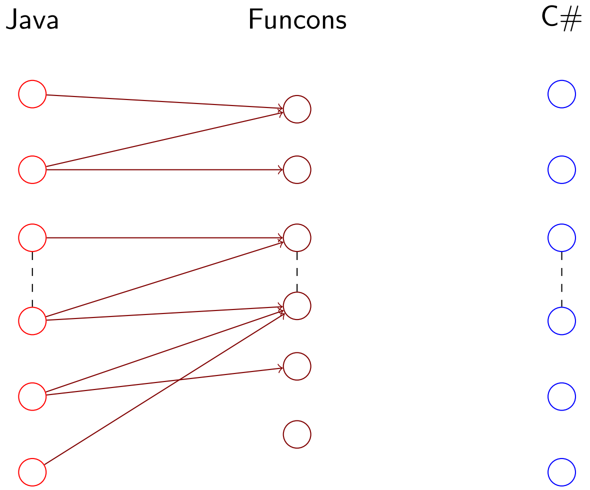
Reusable Components: Funcons



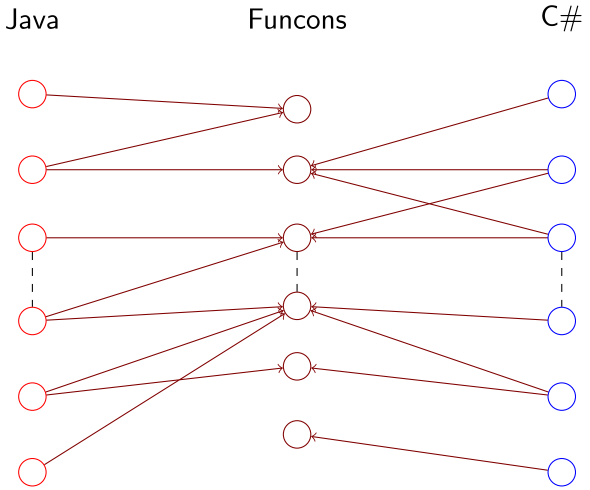
Reusable Components: Funcons



Reusable Components: Funcons



Reusable Components: Funcons



The PLanCompS Approach

- Component based approach towards formal semantics.
- The components are highly reusable constructions.
- We call them fundamental constructions or *funcons*.
- Each funcon has a formal definition in I-MSOS.

The CBS Language

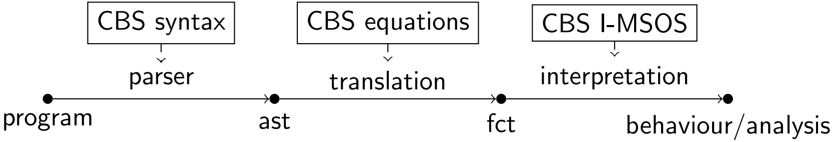


Figure : PLanCompS: generate interpreters from reusable specification.

Tool Support

Spoofax / Eclipse

- CBS is implemented in the Spoofax language workbench.
- IDE support, e.g. syntax-highlighting, declaration referencing.

Haskell

- Modular funcon definition (both manual and generated).
- Compositional interpreters (plug and play).
- Hackage: `funcons-tools`, `gll`.

Case Studies

- Caml Light case study completed (TAOSD2015).
- C# case study underway.
- Various small languages: imperative & logical programming.
- Website: `http://plancomps.org`

Section 2

CBS Funcon Compilation

Funcons in CBS

- Implicitly Modular Structural Operational Semantics.
- I-MSOS transition relations:
    - Context-free rewrites $X = Y$.
    - Context-aware small steps $X \to Y$.
- Semantic entitities propagate contextual information.
- Every funcon has 'zero or more' step and/or rewrite rules.
- Funcon terms are *values* or *computations*.

Example - If-Then-Else

$$\textit{Funcon } \mathbf{if\text{-}then\text{-}else}(\_ : \mathbf{booleans}, \_ : \Rightarrow T, \_ : \Rightarrow T) : \Rightarrow T$$

$$\textit{Rule } \mathbf{if\text{-}then\text{-}else}(\mathbf{true}, X, \_) = X$$

$$\textit{Rule } \mathbf{if\text{-}then\text{-}else}(\mathbf{false}, \_, Y) = Y$$

Example - Bound

Funcon **bound**(_ : **identifiers**) : ⇒**values**

$$\textit{Rule}\ \frac{\mathbf{lookup}(B, \rho) = V}{\mathrm{environment}(\rho) \vdash \mathbf{bound}(B) \longrightarrow V}$$

Funcons in Haskell

- Rules are implemented as sequences of monadic statements.
- Both transition relations have their own monad.
- Both monads propagate meta-information.
- Only the step-monad propagates semantic entities.
- An applicable rule is selected by backtracking.

Rewrite Rules

$$\frac{C_1 \quad \ldots \quad C_k}{f(P) = T}$$

Rewrite Rules

$$\frac{X : \mathit{booleans}}{f(P) = T}$$

Rewrite Rules

$$\frac{Y \equiv true}{f(P) = T}$$

Rewrite Rules

$$\frac{Z = [1, X]}{f(P) = T}$$

Rewrite Rules

$$\frac{C_1 \quad \ldots \quad C_k}{f(P) = T}$$

$$\begin{aligned}
&R = \textbf{do} \\
&\quad \textbf{let}\ env = emptyEnv \\
&\quad env \leftarrow fsMatch\ fargs\ P\ env \\
&\quad env \leftarrow sideCondition\ C_1\ env \\
&\quad \ldots \\
&\quad env \leftarrow sideCondition\ C_k\ env \\
&\quad substitute\ T\ env
\end{aligned}$$

Backtracking

- The statements of a rule can throw exceptions.
- Some exceptions indicate a rule is not applicable.
- Other exceptions indicate an internal error.
- A handler function backtracks between rules, until
  - A rule has been executed successfully (it was applicable).
  - A rule throws an internal error, which is then propagated.

Semantic Entities

- CBS supports the definition of *semantics entities*.
- Each belonging to one of five entity classes:
  - Inherited, e.g. **environment**
  - Mutable, e.g. **store**
  - Output, e.g. **standard-out**
  - Input, e.g. **standard-in**
  - Control, e.g. **thrown**
- In a single rule multiple entities of the same or different classes can be used.
- Each entity class implemented by a map, achieving modularity.

Inherited Entities

$$\frac{\ldots}{\textsf{environment}(\gamma) \vdash f(P) \longrightarrow T}$$

$$\begin{aligned}
S = \textbf{do}\\
&\textbf{let } \textit{env} = \textit{emptyEnv}\\
&\textit{env} \leftarrow \textit{fsMatch fargs } P \textit{ env}\\
&\textit{env} \leftarrow \textit{getInhPatt } \text{“environment”}\ \gamma\ \textit{env}\\
&\ldots\\
&\textit{substitute } T \textit{ env}
\end{aligned}$$

Inherited Entities as Premises

$$\frac{T \longrightarrow P}{\ldots}$$

$$\begin{array}{l} \ldots \\ env \leftarrow stepTerm\ T\ P\ env \\ \ldots \end{array}$$

Inherited Entities as Premises

$$\frac{\text{environment}(\gamma) \vdash T \longrightarrow P}{\ldots}$$

$\ldots$

$$\begin{aligned} &env \leftarrow withInhTerm \text{ “environment” } \gamma \; env \\ &\qquad (stepTerm \; T \; P \; env) \end{aligned}$$

$\ldots$

Mutable Entities

$$\frac{\ldots}{\langle f(P)\ ,\mathbf{store}(\sigma)\rangle \longrightarrow \langle T\ ,\mathbf{store}(\sigma')\rangle}$$

$$
\begin{aligned}
S = &\ \mathbf{do}\\
&\quad \mathbf{let}\ \mathit{env} = \mathit{emptyEnv}\\
&\quad \mathit{env} \leftarrow \mathit{fsMatch}\ \mathit{fargs}\ P\ \mathit{env}\\
&\quad \mathit{env} \leftarrow \mathit{getMutPatt}\ \text{“store”}\ \sigma\ \mathit{env}\\
&\quad \ldots\\
&\quad \mathit{putMutTerm}\ \text{“store”}\ \sigma'\ \mathit{env}\\
&\quad \mathit{substitute}\ T\ \mathit{env}
\end{aligned}
$$

Mutable Entities as Premises

$$\frac{\langle T\ , \mathbf{store}(\sigma')\rangle \longrightarrow \langle P\ , \mathbf{store}(\sigma)\rangle}{\ldots}$$

$$\begin{array}{l} \ldots \\ \mathit{putMutTerm}\ \text{“store”}\ \sigma'\ \mathit{env} \\ \mathit{env} \leftarrow \mathit{stepTerm}\ T\ P\ \mathit{env} \\ \mathit{getMutPatt}\ \text{“store”}\ \sigma\ \mathit{env} \\ \ldots \end{array}$$

Pattern Matching

- CBS supports meta-variables like X^*, Y^+ and $Z^?$.
- Example of an ambiguous pattern: X^*, Y^+.
- Greedy longest-match is no solution: X^*, true.

Pattern Matching Algorithm

- Assuming a matcher M for single terms and patterns.
- We can then match $t_1, \ldots, t_n$ with $p_1, \ldots, p_m$.
- Starting with $(0, 0, \mathit{emptyEnv}) \in \mathcal{R}$, until $\mathcal{R} \equiv \emptyset$:
  - Pop (i, j, env) from $\mathcal{R}$.
  - Return env if $i \equiv n + 1$ and $j \equiv m + 1$.
  - Next iteration if $i < n + 1$ and $j \equiv m + 1$.
  - If p_j is a simple pattern:
    - Add $(i + 1, j + 1, \mathit{env}') \in \mathcal{R}$ iff $\mathit{env}' = M(t_i, p_j)$.
  - Else p_j is a sequence meta-variable: ... (next slide)
- Pattern mismatch if no env was returned.

Pattern Matching Algorithm (2)

- If p_j is a sequence meta-variable:
  - Add $(k, j+1, env') \in \mathcal{R}$, with:
    - $\forall i \leqslant k \leqslant n+1$ (if p_j is X^*)
    - $\forall i < k \leqslant n+1$ (if p_j is X^+)
    - $\forall i \leqslant k \leqslant i+1$ (if p_j is $X^?$)
    - $env' = [X^* \mapsto t_i, \ldots, t_{k-1}]env$.