Funcons: Reusable and Modular Semantic Components

Neil Sculthorpe

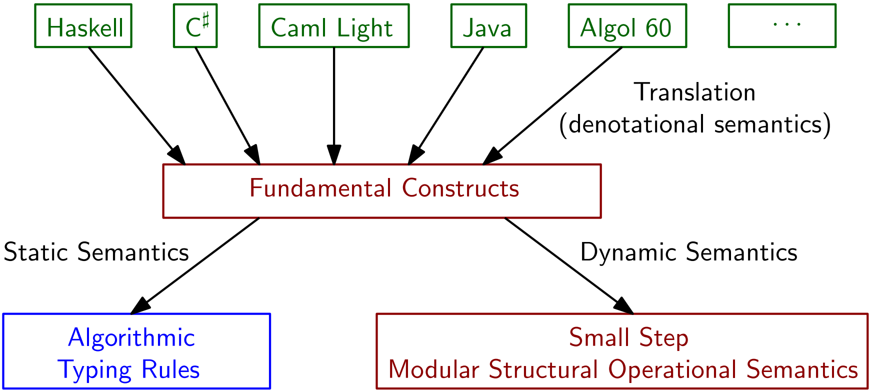
(describing work by Martin Churchill, Peter D. Mosses and Paolo Torrini)

Theory Group
Department of Computer Science
Swansea University
N.A.Sculthorpe@swansea.ac.uk

Nottingham, England
18th September 2014

A Component-based Language for Formal Semantics

- Aim: Making formal semantics easier to specify
- Approach: A component-based language of fundamental constructs



Fundamental Constructs (funcons)

- Each funcon defines a programming concept, e.g.
  - function application
  - declaration scoping
  - command sequencing
  - variable assignment
- Funcons are similar to language constructs to facilitate translation . . .
- . . . but general enough to be reusable for many languages.

An Open Collection of Modular Funcons

. . . assign scope bind-value print list forall

close assigned-value apply while-true abstraction

bound-value if-true equal skip record throw

sequential effect alloc vector catch null

not fail curry else let match stuck id

rectype fold unfold tuple store option . . .

An Open Collection of Modular Funcons

- The funcon language is open
- Each funcon:
  - is modular
  - has fixed syntax and semantics
- New funcons can be added, but existing funcons cannot be modified
- If a programming language changes, the translation to funcons changes

Case Studies

- Caml Light (almost complete)
- Algol 60 (in progress)
- C$^\sharp$ (in progress)
- Java (future work)

Example Translation (1)

- Source language syntax:

$$\begin{aligned} \textbf{expr} &\rightarrow \textbf{expr} \ ? \ \textbf{expr} : \textbf{expr} \\ &\mid \ \textbf{identifier} \\ &\mid \ \ldots \end{aligned}$$

- Translation to funcons:

$$\begin{aligned} [\![\ E_1 \ ? \ E_2 : E_3\ ]\!] &= \textbf{if-true}([\![E_1]\!], [\![E_2]\!], [\![E_3]\!]) \\ [\![\ I\ ]\!] &= \textbf{bound-value}(\textbf{id}(I)) \end{aligned}$$

Example Translation (2)

- Source language syntax:

$$\begin{aligned} \textbf{comm} \rightarrow\ & \textbf{if}\ (\ \textbf{expr}\ )\ \textbf{then}\ \textbf{comm}\ \textbf{else}\ \textbf{comm} \\ |\ \ & \textbf{identifier} := \textbf{expr}\ ; \\ |\ \ & \ldots \end{aligned}$$

- Translation to funcons:

$$\begin{aligned} [\![\ \textbf{if}\ (\ E\ )\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ ]\!] &= \textbf{if-true}(\textbf{not}(\textbf{equal}([\![E]\!], \textbf{0})), [\![C_1]\!], [\![C_2]\!]) \\ [\![\ I := E\ ;\ ]\!] &= \textbf{assign}(\textbf{bound-value}(\textbf{id}(I)), [\![E]\!]) \end{aligned}$$

Semantics of if-true (Verbose version)

- Sort Signature:

if-true(**computes**(**booleans**), **computes**(T), **computes**(T)) : **computes**(T)

Semantics of if-true (Verbose version)

- Sort Signature:

$$\textbf{if-true}(\textbf{computes}(\textbf{booleans}), \textbf{computes}(T), \textbf{computes}(T)) : \textbf{computes}(T)$$

- Typing Rules:

$$\frac{B : \textbf{booleans} \qquad X_1 : T \qquad X_2 : T}{\textbf{if-true}(B, X_1, X_2) : T}$$

Semantics of if-true (Verbose version)

- Sort Signature:

$$\textbf{if-true}(\textbf{computes}(\textbf{booleans}), \textbf{computes}(T), \textbf{computes}(T)) : \textbf{computes}(T)$$

- Typing Rules:

$$\frac{B : \textbf{booleans} \qquad X_1 : T \qquad X_2 : T}{\textbf{if-true}(B, X_1, X_2) : T}$$

- Operational Semantics:

$$\frac{B \longrightarrow B'}{\textbf{if-true}(B, X_1, X_2) \longrightarrow \textbf{if-true}(B', X_1, X_2)}$$

$$\textbf{if-true}(\textbf{true}, X_1, X_2) \longrightarrow X_1$$
$$\textbf{if-true}(\textbf{false}, X_1, X_2) \longrightarrow X_2$$

Semantics of if-true (Concise version)

- Sort Signature:

$$\textbf{if-true}(\textbf{booleans}, \textbf{computes}(T), \textbf{computes}(T)) : \textbf{computes}(T)$$

- Typing Rules:

$$\frac{B : \textbf{booleans} \qquad X_1 : T \qquad X_2 : T}{\textbf{if-true}(B, X_1, X_2) : T}$$

- Operational Semantics:

$$\textbf{if-true}(\textbf{true}, X_1, X_2) \longrightarrow X_1$$
$$\textbf{if-true}(\textbf{false}, X_1, X_2) \longrightarrow X_2$$

Semantics of bound-value

- Sort Signature:

$$\textbf{bound-value}(\textbf{ids}) : \textbf{computes}(\textbf{values})$$

- Typing Rules:

$$\frac{\Gamma(I) = T}{\textbf{typenv}\ \Gamma \vdash \textbf{bound-value}(I) : T}$$

- Operational Semantics:

$$\frac{\rho(I) = V}{\textbf{env}\ \rho \vdash \textbf{bound-value}(I) \longrightarrow V}$$

Semantics of assign

- Sort Signature:

$$\textbf{assign}(\textbf{variables}, \textbf{values}) : \textbf{commands}$$

- Typing Rules:

$$\frac{\mathit{Var} : \textbf{variable}(T) \qquad \mathit{Val} : T}{\textbf{assign}(\mathit{Var}, \mathit{Val}) : \textbf{unit}}$$

- Operational Semantics:

$$\frac{\sigma[\mathit{Var} \mapsto \mathit{Val}] = \sigma'}{(\textbf{assign}(\mathit{Var}, \mathit{Val}),\ \textbf{store}\ \sigma) \longrightarrow (\textbf{skip},\ \textbf{store}\ \sigma')}$$

Example Funcon Sorts

$$\textbf{computes}(\textbf{types}) : \textbf{sorts}$$

$$T <: \textbf{computes}(T)$$

$$\textbf{expressions} = \textbf{computes}(\textbf{values})$$

$$\textbf{commands} = \textbf{computes}(\textbf{unit})$$

$$\textbf{declarations} = \textbf{computes}(\textbf{environments})$$

$$\textbf{functions} = \textbf{abstractions}(\textbf{values}, \textbf{expressions})$$

$$\textbf{procedures} = \textbf{abstractions}(\textbf{values}, \textbf{commands})$$

$$\textbf{patterns} = \textbf{abstractions}(\textbf{values}, \textbf{environments})$$

$$\textbf{environments} = \textbf{maps}(\textbf{ids}, \textbf{values})$$

Example Funcons - Control Flow

$$\textbf{skip} : \textbf{unit}$$

$$\textbf{sequential}(\textbf{unit}, \textbf{computes}(T)) : \textbf{computes}(T)$$

$$\textbf{if-true}(\textbf{booleans}, \textbf{computes}(T), \textbf{computes}(T)) : \textbf{computes}(T)$$

$$\textbf{while-true}(\textbf{computes}(\textbf{booleans}), \textbf{commands}) : \textbf{commands}$$

$$\textbf{effect}(T) : \textbf{commands}$$

Example Funcons - Abstraction and Application

apply(**functions**, **values**) : **expressions**

abstraction(**expressions**) : **functions**

given : **expressions**

patt-abstraction(**patterns**, **expressions**) : **functions**

close(**functions**) : **computes**(**functions**)

Example Funcons - Binding and Scoping

$$\textbf{scope}(\textbf{environments}, \textbf{computes}(T)) : \textbf{computes}(T)$$

$$\textbf{bind-value}(\textbf{ids}, \textbf{values}) : \textbf{environments}$$

$$\textbf{bound-value}(\textbf{ids}) : \textbf{expressions}$$

Example Funcons - Pattern Matching

any : **patterns**

only(**values**) : **patterns**

bind(**ids**) : **patterns**

prefer-over(**patterns**, **patterns**) : **patterns**

Example Translation - Pattern Matching

- Source language syntax:

$$\textbf{expr} \rightarrow \lambda\ \textbf{patt}\ .\ \textbf{expr} \mid \ldots$$
$$\textbf{patt} \rightarrow \textbf{identifier} \mid \textbf{literal} \mid \_ \mid (\ \textbf{patt} \mid \textbf{patt}\ )$$

- Translation to funcons:

$$[\![\ \lambda\ P\ .\ E\ ]\!] = \textbf{patt-abstraction}([\![P]\!], [\![E]\!])$$

$$[\![\ I\ ]\!] = \textbf{bind}(\textbf{id}(I))$$
$$[\![\ L\ ]\!] = \textbf{only}([\![\ L\ ]\!])$$
$$[\![\ \_\ ]\!] = \textbf{any}$$
$$[\![\ (\ P_1 \mid P_2\ )\ ]\!] = \textbf{prefer-over}([\![\ P_1\ ]\!], [\![\ P_2\ ]\!])$$

Semantics of any

- Sort Signature:

$$\textbf{any} : \textbf{patterns}$$

- Typing Rules:

$$\textbf{any} : T \rightarrow \{\ \}$$

- Operational Semantics:

$$\textbf{any} \longrightarrow \textbf{abstraction}(\textbf{empty})$$

(Reminder: $\textbf{patterns} = \textbf{abstractions}(\textbf{values}, \textbf{environments})$)

Semantics of only

- Sort Signature:

$$\textbf{only}(\textbf{values}) : \textbf{patterns}$$

- Typing Rules:

$$\frac{V : T}{\textbf{only}(V) : T \rightarrow \{\ \}}$$

- Operational Semantics:

$$\textbf{only}(V) \longrightarrow \textbf{abstraction}(\textbf{if-true}(\textbf{equal}(\textbf{given}, V), \textbf{empty}, \textbf{fail}))$$

Semantics of prefer-over

- Sort Signature:

$$\textbf{prefer-over}(\textbf{patterns}, \textbf{patterns}) : \textbf{patterns}$$

- Typing Rules:

$$\frac{P_1 : T \rightarrow \mathit{EnvType} \qquad P_2 : T \rightarrow \mathit{EnvType}}{\textbf{prefer-over}(P_1, P_2) : T \rightarrow \mathit{EnvType}}$$

- Operational Semantics:

$$\textbf{prefer-over}(P_1, P_2) \longrightarrow \textbf{abstraction}(\textbf{else}(\textbf{apply}(P_1, \textbf{given}), \textbf{apply}(P_2, \textbf{given})))$$

Semantics of bind

- Sort Signature:

$$\textbf{bind}(\textbf{ids}) : \textbf{patterns}$$

- Typing Rules:

$$\textbf{bind}(I) : T \to \{\ I \mapsto T\ \}$$

- Operational Semantics:

$$\textbf{bind}(I) \longrightarrow \textbf{abstraction}(\textbf{bind-value}(I, \textbf{given}))$$

A real example: while loops in C$^\sharp$

- C$^\sharp$ syntax:

$$\textbf{statement} \rightarrow \textbf{while} \; (\; \textbf{expression} \;) \; \textbf{statement}$$

A real example: while loops in C$^\sharp$

- C$^\sharp$ syntax:

$$\textbf{statement} \rightarrow \textbf{while} \ (\ \textbf{expression} \ ) \ \textbf{statement}$$

- Translation to funcons:

$$[\![\ \textbf{while} \ (\ E \ ) \ S \ ]\!] = $$
$$\textbf{exit-on}('\mathit{break}', \textbf{while-true}([\![\ E \ ]\!], \textbf{exit-on}('\mathit{continue}', [\![\ S \ ]\!])))$$

A real example: while loops in C$^\sharp$

- C$^\sharp$ syntax:

$$\textbf{statement} \rightarrow \textbf{while} \; (\; \textbf{expression} \;) \; \textbf{statement}$$

- Translation to funcons:

$$[\![\; \textbf{while} \; (\; E \;) \; S \;]\!] =$$
$$\quad \textbf{exit-on}('break', \textbf{while-true}([\![\; E \;]\!], \textbf{exit-on}('continue', [\![\; S \;]\!])))$$

- Auxiliary funcon:

$$\textbf{exit-on}(\textbf{exceptions}, \textbf{commands}) : \textbf{commands}$$
$$\textbf{exit-on}(X, C) =$$
$$\quad \textbf{catch-else-rethrow}(C, \textbf{abstraction}(\textbf{if-true}(\textbf{equal}(\textbf{given}, X), \textbf{skip}, \textbf{fail})))$$

Summary

- Funcons are reusable semantic components
- The funcon language is open and modular
- Our goal is to provide a practical tool for formally specifying real programming languages

Funcon Publications

Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini.
Reusable components of semantic specifications.
In *Transactions on Aspect-Oriented Software Development XII*, volume 8989 of *Lecture Notes in Computer Science*, pages 132–179. Springer, 2015.

Peter D. Mosses and Ferdinand Vesely.
FunKons: Component-based semantics in K.
In *International Workshop on Rewriting Logic and its Applications*, volume 8663 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2014.