Funcons
Executable Component-Based Semantics

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

Royal Holloway University of London,
Swansea University

April 20, 2016
Section 1

PLanCompS
Reusable Components: Funcons

Java
Reusable Components: Funcons

Java  Java Core

Diagram depicting the relationship between Java and Java Core.
Reusable Components: Funcons

Java       Java Core

C# Core  C#
Reusable Components: Funcons

Java       | Funcons       | C#
---        | ---           | ---

Diagram showing the relationships between Java, Funcons, and C# components.
Reusable Components: Funcons

Java | Funcons | C#

![Diagram of Java, Funcons, and C# relationships]
The PLanCompS Approach

- Component based approach towards formal semantics.
- Highly reusable, fundamental constructs: *funcons*.
- A language is defined formally via a translation to 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
- Compositional interpreters (plug and play).
- Hackage: funcons-tools, gll.
Example Translation

```himp
1 int x = 1;
2 x := 2;
3 print(x);
```

```fct
1 scope(bind("x", allocate-initialised-variable(integers, 1)),
   sequential(assign(bound("x"), 2),
   print(assigned(bound("x"))))
```

Result:
()

Output Entity: standard-out
[2]
Case Studies

- Caml Light case study (TAOSD2015).
- C# case study underway.
- Various small languages: IMP++, SIMPLE, LogiK, ...
- Website: http://plancomps.org
Section 2

CBS Funcon Compilation - Challenges
Implicitly Modular Structural Operational Semantics.

Semantic entities propagate contextual information.

I-MSOS relations:
- Context-free rewrites $X = Y$.
- Context-rich small-step transitions $X \rightarrow Y$.

Every funcon has ‘zero or more’ step and/or rewrite rules.

Funcon terms are values or computations.

Computations cannot be inspected.
Example: If-Then-Else

*Funcon* `if-then-else(_ : booleans, _ : ⇒T, _ : ⇒T) : ⇒T`

*Rule* `if-then-else(true, X, _) = X`

*Rule* `if-then-else(false, _, Y) = Y`
Example: Bound

Funcon bound(_: identifiers) : \Rightarrow values

Rule

\[ \text{lookup}(B, \rho) = V \]

\[ \text{environment}(\rho) \vdash \text{bound}(B) \rightarrow V \]
Example: Scope

Funcon \( \text{scope}(\_ : \text{environments}, \_ : \Rightarrow T) : \Rightarrow T \)

Rule

\[
\frac{\text{environment}(\text{map-override}(\rho_1, \rho_0)) \vdash X \rightarrow X'}{
\text{environment}(\rho_0) \vdash \text{scope}(\rho_1, X) \rightarrow \text{scope}(\rho_1, X')}
\]

Rule \( \text{scope}(\_ , V : \text{values}) = V \)
Challenges

- Inference rules are transactions.
  - Requires roll-back or persistent data.

- Immutable variables.
  - Requires persistent data.

- Complex reasons for a rule not to be applicable.
  - Requires exceptions.

- Funcons can be defined in isolation.
  - A CBS compiler should generate compilable code for individual files.

- Funcons depend on other components:
  - Directly related funcons.
  - Accessed semantic entities.
  - Builtin aspects, e.g. numbers, list-notation, set-notation, etc.
Funcon Isolation

- Generic funcon term representation based on strings/names.
- For each CBS file we generate a funcon library.
- A funcon library maps funcon names to evaluation functions.

Disadvantages

- No static guarantee that funcon exists at runtime.
- Libraries need to be explicitly managed (imported and joined).
The statements of a rule can throw exceptions.

- Some exceptions indicate a rule is not applicable.
- Other exceptions indicate an internal error.

Handlers backtrack between rules until:

- A rule has been fully executed (it was applicable).
- A rule throws an internal error, which is then propagated.

With persistent data, entities’ values are easily “reverted”.
\[
\frac{C_1 \ldots C_k}{f(P) = T}
\]
Rewrite Rules

$$X : \text{booleans}$$

$$f(P) = T$$
\[ Y \equiv true \]
\[ f(P) = T \]
Rewrite Rules

\[
Z = [1, X]
\]

\[
f(P) = T
\]
### Rewrite Rules

\[
C_1 \ldots C_k \\
\frac{f(P)}{T}
\]

\[
R = \text{do} \\
\begin{aligned}
\text{let } env &= \text{emptyEnv} \\
env &\leftarrow \text{fsMatch fargs } P \text{ env} \\
env &\leftarrow \text{sideCondition } C_1 \text{ env} \\
&\ldots \\
env &\leftarrow \text{sideCondition } C_k \text{ env} \\
\text{substitute } T \text{ env}
\end{aligned}
\]
\[
\begin{align*}
C_1 \ldots C_k \\
f(P) = T \\
\end{align*}
\]

\[
R = \text{do} \\
\text{let } env = \text{emptyEnv} \\
env \leftarrow \text{fsMatch fargs P env} \\
env \leftarrow \text{sideCondition } C_1 \text{ env} \\
\ldots \\
env \leftarrow \text{sideCondition } C_k \text{ env} \\
\text{substitute } T \text{ env}
\]

evalRules \[\text{rewrite1, rewrite2} \] \[\text{step1, step2, step3} \]
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 accessed.

Each entity class implemented by a map, with modular access.
Implicit Propagation

- 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.
  - Guarantees rewrites are context-free.
Inherited Entity Example

- **environment** is only locally overridden by **scope**.
Inherited Entities

...\[
\text{environment}(\gamma) \vdash f(P) \rightarrow T
\]

\[S = \text{do}
\]

\[\text{let } env = \text{emptyEnv}
\]

\[env \leftarrow \text{fsMatch fargs P env}
\]

\[env \leftarrow \text{getInhPatt “environment” } \gamma \text{ env}
\]

\[...\]

\[\text{substitute } T \text{ env}\]
Inherited Entities as Premises

\[ T \rightarrow P \]

\[ \text{env} \leftarrow \text{stepTerm} \ T \ P \text{ env} \]

\[ \ldots \]
Inherited Entities as Premises

\[
\text{environment}(\gamma) \vdash T \rightarrow P
\]

\[
\ldots
\]

\[
\text{env} \leftarrow \text{withInhTerm} \text{ “environment” } \gamma \text{ env} \\
(\text{stepTerm } T \ P \text{ env})
\]

\[
\ldots
\]
Mutable Entities Example

- Changes to the **store** are global.
Mutable Entities

\[ \langle f(P), store(\sigma) \rangle \rightarrow \langle T, store(\sigma') \rangle \]

\[ S = \text{do} \]
\[ \quad \text{let } env = \text{emptyEnv} \]
\[ \quad env \leftarrow \text{fsMatch fargs P env} \]
\[ \quad env \leftarrow \text{getMutPatt "store" } \sigma \text{ env} \]
\[ \quad \ldots \]
\[ \quad \text{putMutTerm "store" } \sigma' \text{ env} \]
\[ \quad \text{substitute } T \text{ env} \]
Mutable Entities as Premises

\[ \langle T, \text{store}(\sigma') \rangle \rightarrow \langle P, \text{store}(\sigma) \rangle \]

... putMutTerm “store” \( \sigma' \) env
env \( \leftarrow \) stepTerm \( T \) \( P \) env
getMutPatt “store” \( \sigma \) env

...
Section 3

CBS Funcon Compilation - Opportunities
Potential Efficiency Improvements

- Rewrite transitions are ‘unobservable’.
  - Can be applied at any time.
  - Effects can be \textit{shared}.
- Pessimistic, but safe, refocusing.
- Rules can be factorised.
Small-Step Interpretation

Figure: Diagram of small-step interpretation.
Refocusing (1)
Refocusing is safe after a rewrite transition.

Refocusing is safe after a step transition, if by the transition:
- No mutable entity has been modified.
- No output has been emitted.
- No signal has been raised.
- No input has been read.

How to improve the precision of the implemented check?

For example, by recording which entities are ‘listening’.
- e.g. **standard-out** is often modified but is rarely inspected.