Funcons

Executable Component-Based Semantics

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

Royal Holloway University of London,
Swansea University

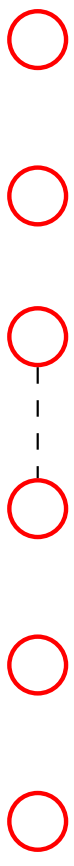
April 20, 2016

Section 1

PLanCompS

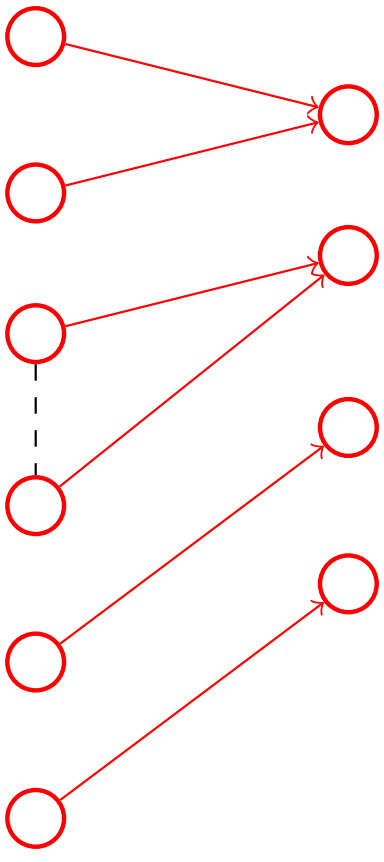
Reusable Components: Funcons

Java

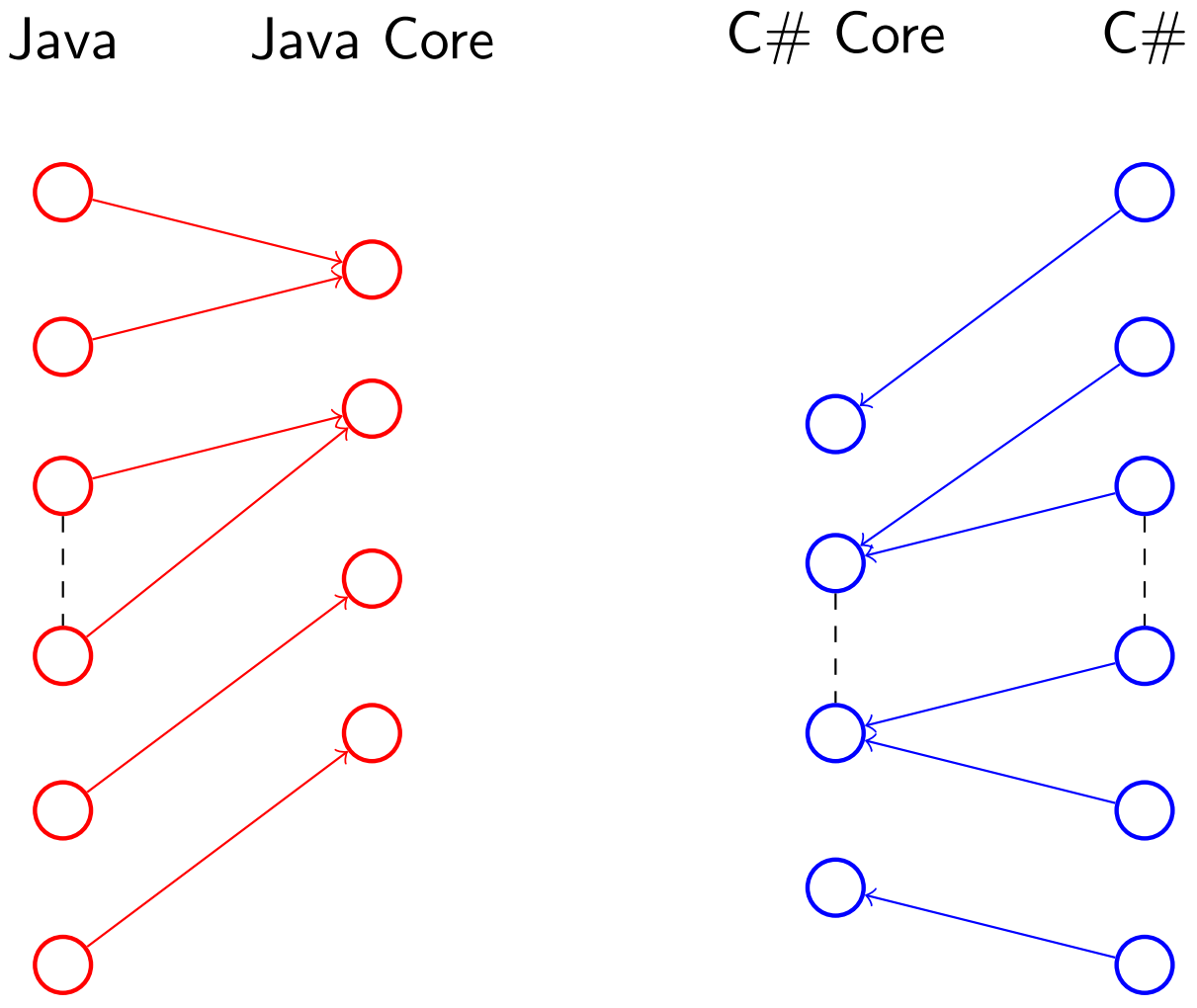


Reusable Components: Funcons

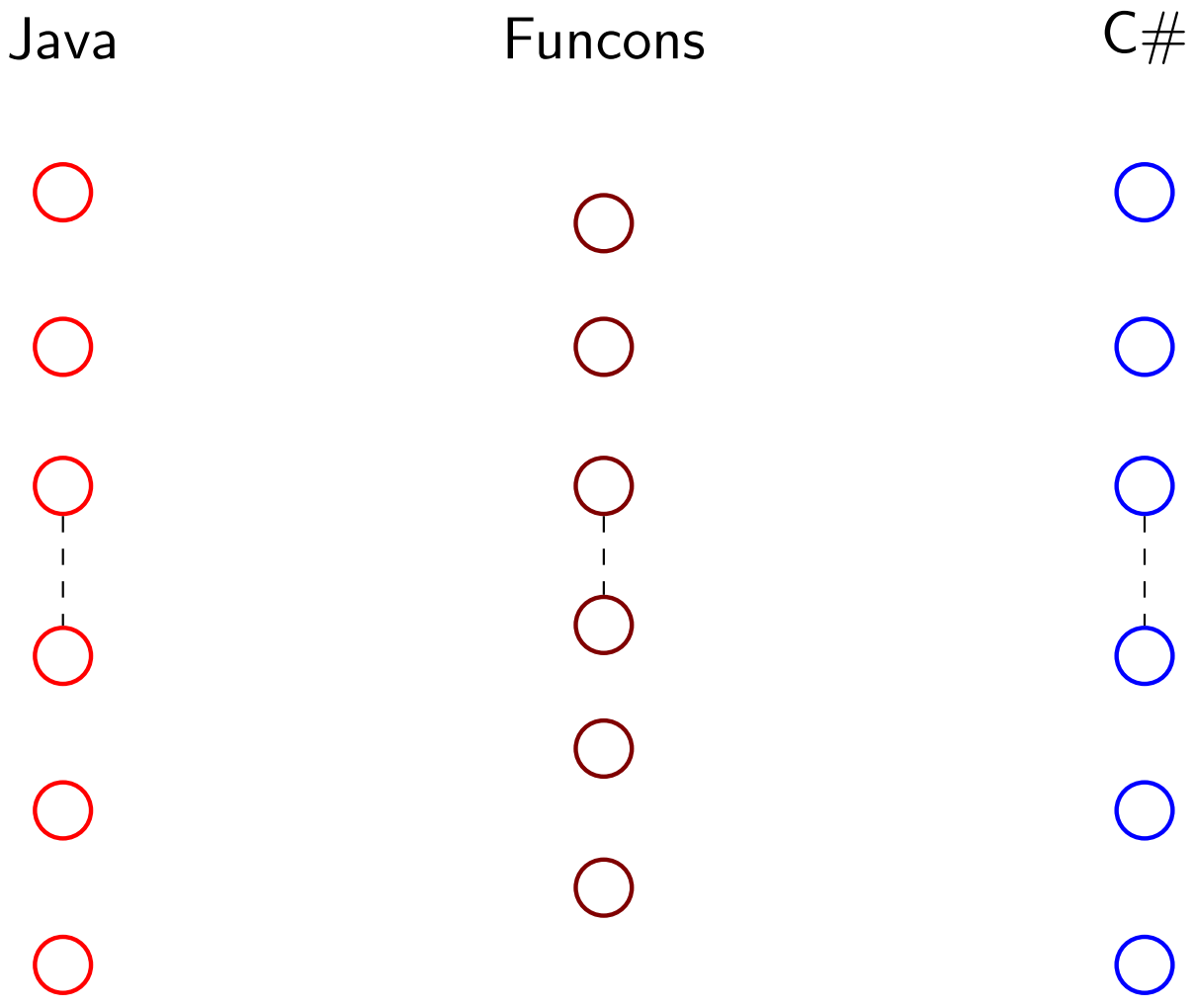
Java Java Core



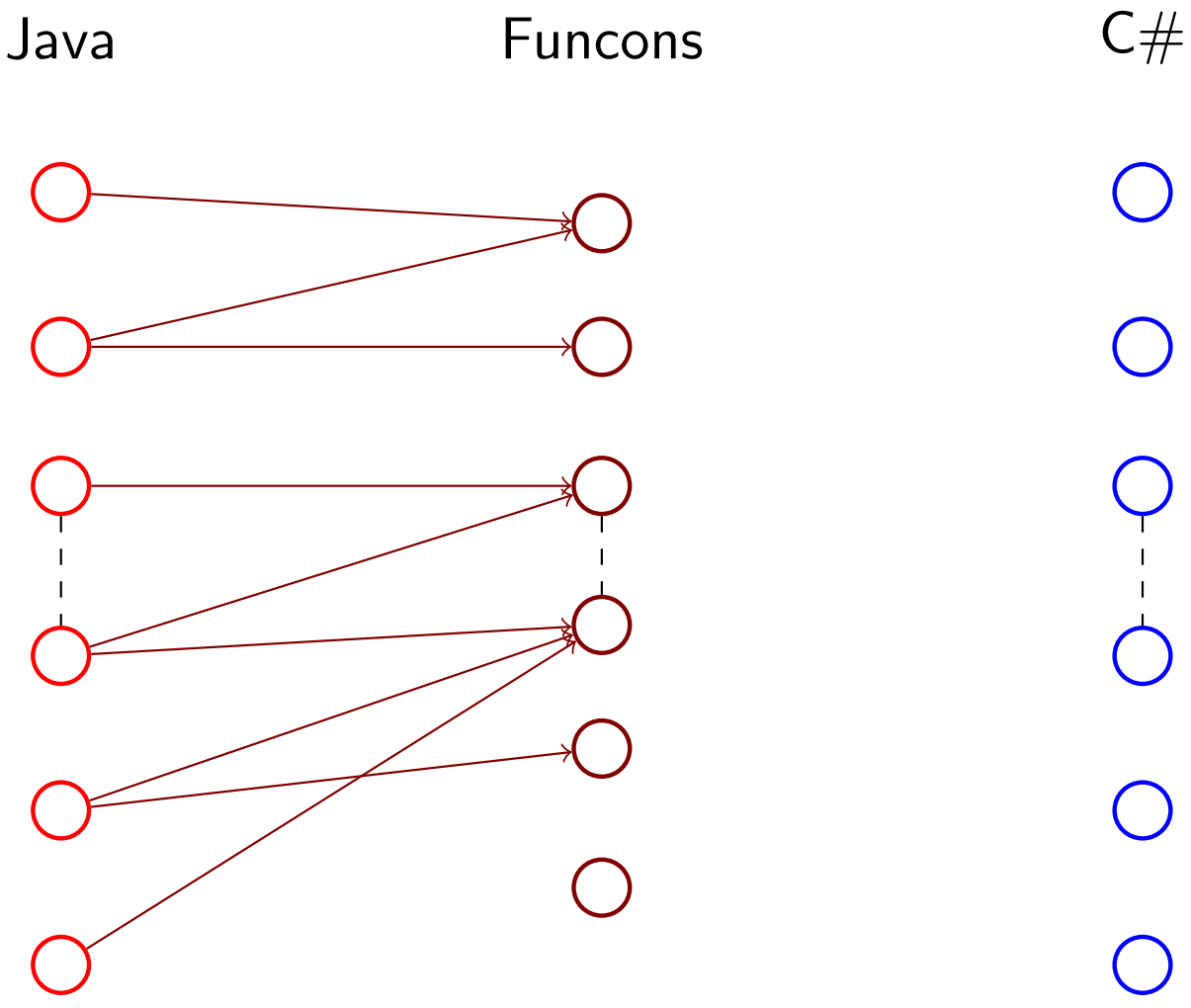
Reusable Components: Funcons



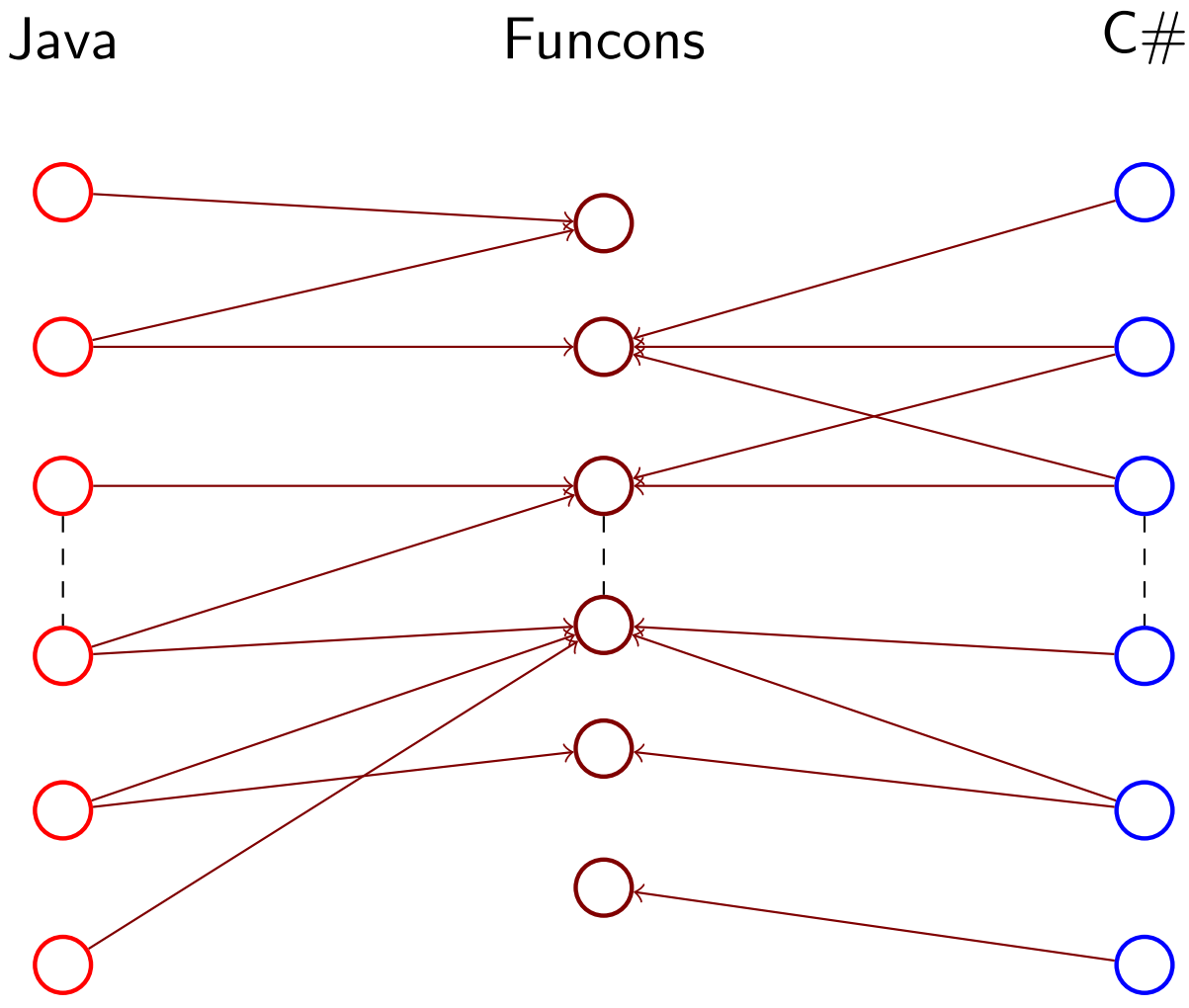
Reusable Components: Funcons



Reusable Components: Funcons



Reusable Components: Funcons



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

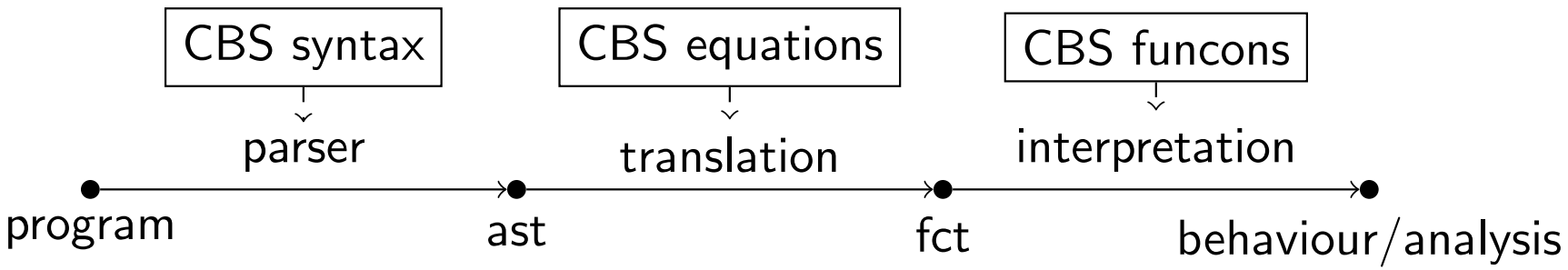


Figure : PLanCompS: generate interpreters from reusable specification.

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

print-variable1.himp

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

print-variable1.fct

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

Proble @ Javad Declar Search Consol Progre

<terminated> FCTInterpreter [Program] /home/thomas/plancomps/FunconInterpr

```
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

Funcons in CBS

- Implicitly Modular Structural Operational Semantics.
- Semantic entitities 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**$(\_ : \textbf{booleans}, \_ : \Rightarrow T, \_ : \Rightarrow T) : \Rightarrow T$

Rule **if-then-else**$(\textbf{true}, X, \_) = X$

Rule **if-then-else**$(\textbf{false}, \_, Y) = Y$

Example: Bound

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

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

Example: Scope

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

Rule $$\frac{\mathsf{environment}(\mathbf{map\text{-}override}(\rho_1, \rho_0)) \vdash X \longrightarrow X'}{\mathsf{environment}(\rho_0) \vdash \mathbf{scope}(\rho_1, X) \longrightarrow \mathbf{scope}(\rho_1, X')}$$

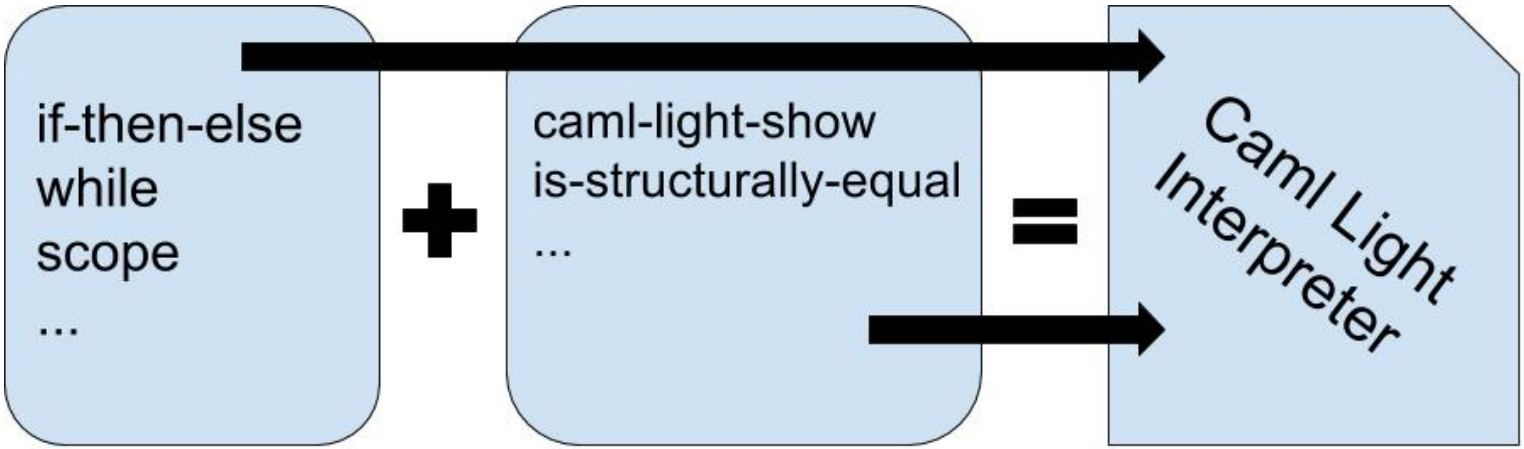
Rule $\mathbf{scope}(\_, V : \mathbf{values}) = V$

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).

Backtracking

- 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".

Rewrite Rules

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

Rewrite Rules

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

Rewrite Rules

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

Rewrite Rules

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

Rewrite Rules

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

$R =$ **do**
  let *env* $=$ *emptyEnv*
  env $\leftarrow$ *fsMatch fargs P env*
  env $\leftarrow$ *sideCondition* C_1 *env*
  ...
  env $\leftarrow$ *sideCondition* C_k *env*
  substitute T env

Rewrite Rules

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

$R =$ **do**
  let *env* $=$ *emptyEnv*
  env $\leftarrow$ *fsMatch fargs P env*
  env $\leftarrow$ *sideCondition* C_1 *env*
  ...
  env $\leftarrow$ *sideCondition* C_k *env*
  substitute T env

evalRules [*rewrite1*, *rewrite2*] [*step1*, *step2*, *step3*]

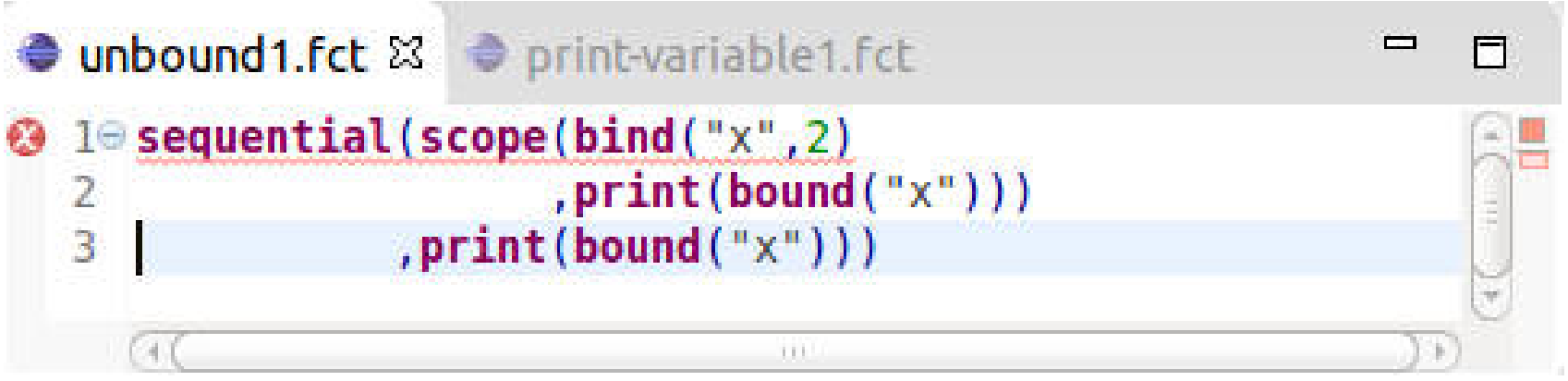
Semantic Entities

- 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 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

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

$S =$ **do**
  let *env* $=$ *emptyEnv*
  env $\leftarrow$ *fsMatch fargs P env*
  env $\leftarrow$ *getInhPatt* "environment" γ *env*
  ...
  substitute T env

Inherited Entities as Premises

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

$$\ldots$$
$$env \leftarrow stepTerm\ T\ P\ env$$
$$\ldots$$

Inherited Entities as Premises

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

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

Mutable Entities Example

print-variable1.himp

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

print-variable1.fct

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

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

Mutable Entities

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

$S =$ **do**
  let *env* $=$ *emptyEnv*
  env $\leftarrow$ *fsMatch fargs P env*
  env $\leftarrow$ *getMutPatt* "store" σ *env*
  ...
  putMutTerm "store" σ' *env*
  substitute T env

Mutable Entities as Premises

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

$\ldots$

putMutTerm "store" σ' *env*

$env \leftarrow$ *stepTerm* T P *env*

getMutPatt "store" σ *env*

$\ldots$

Section 3

CBS Funcon Compilation - Opportunities

Potential Efficiency Improvements

- Rewrite transitions are ‘unobservable’.
  - Can be applied at any time.
  - Effects can be *shared*.
- Pessimistic, but safe, refocusing.
- Rules can be factorised.

Small-Step Interpretation

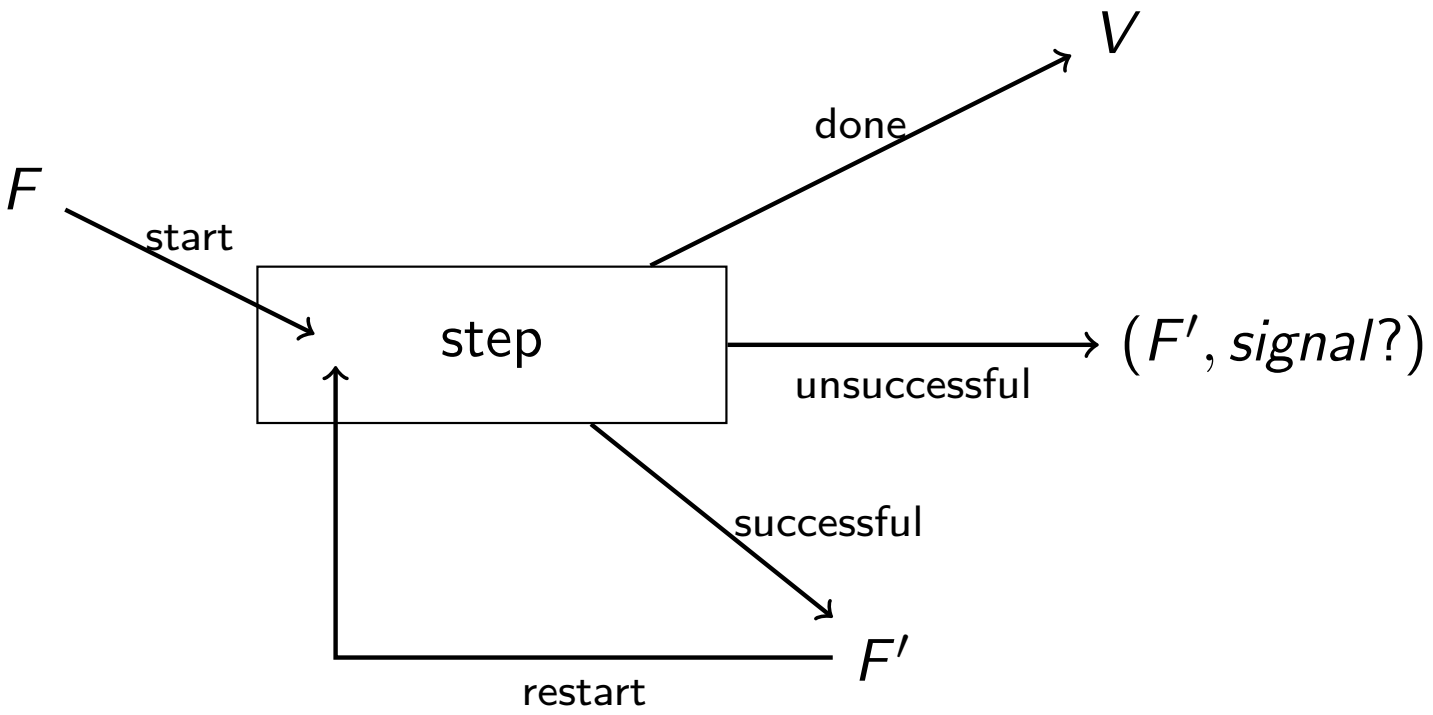
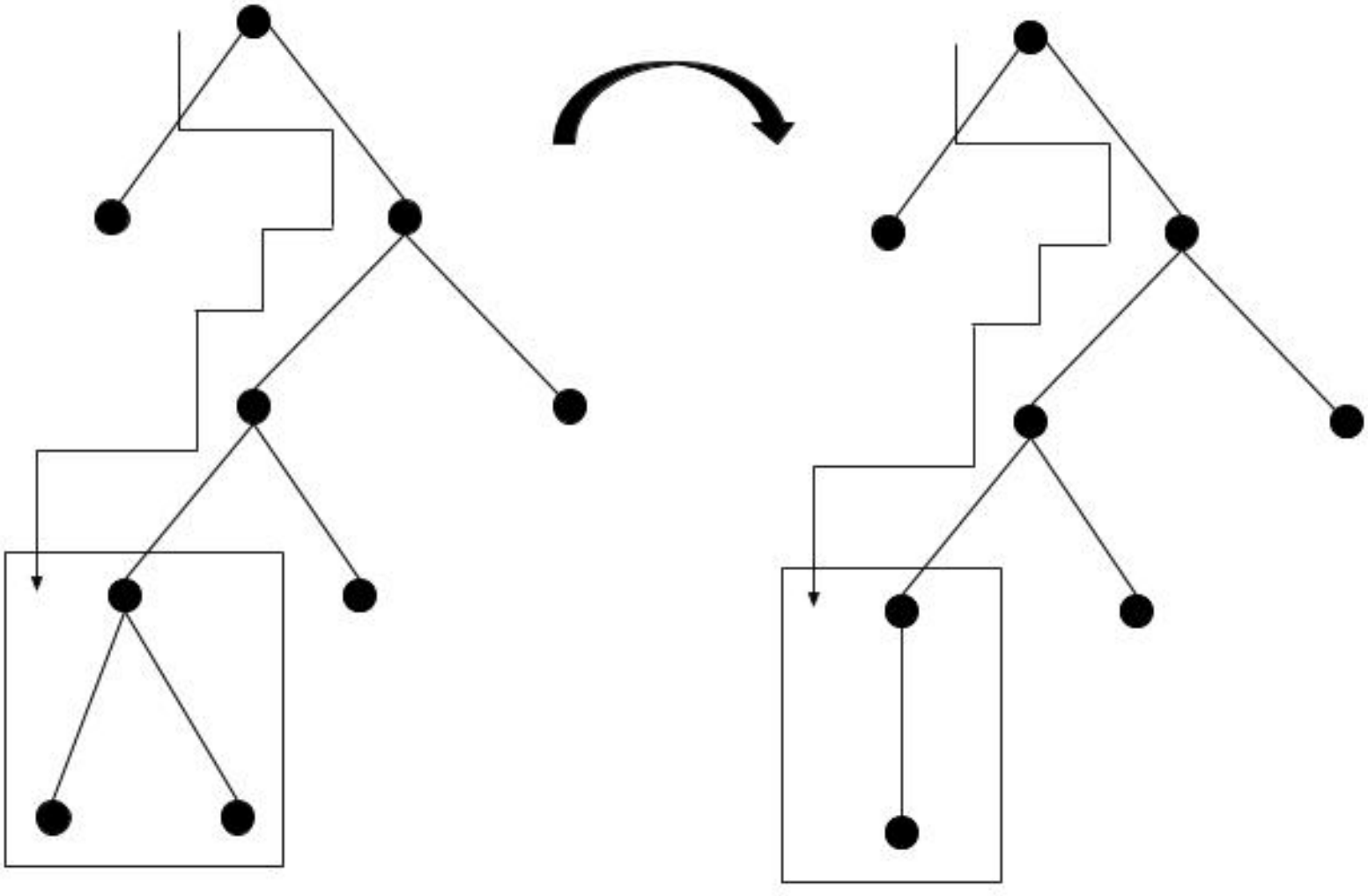


Figure : Diagram of small-step interpretation.

Refocusing (1)



Refocusing (2)

- 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.