The CBS Framework

Peter D. Mosses

Swansea University (emeritus)
TU Delft (visitor)

IFIP WG 2.11 Meeting, Kyoto, June 2018

CBS: Component-Based Semantics

Main goal:

Make formal semantics as popular as BNF !

Encourage language developers *to use formal semantics:*

- *documentation* of language features, design decisions
- *generation* of (prototype) implementations

CBS: Component-Based Semantics

Reusable components: 'funcons'

- specifications of *fundamental programming constructs*
- independent of particular programming languages

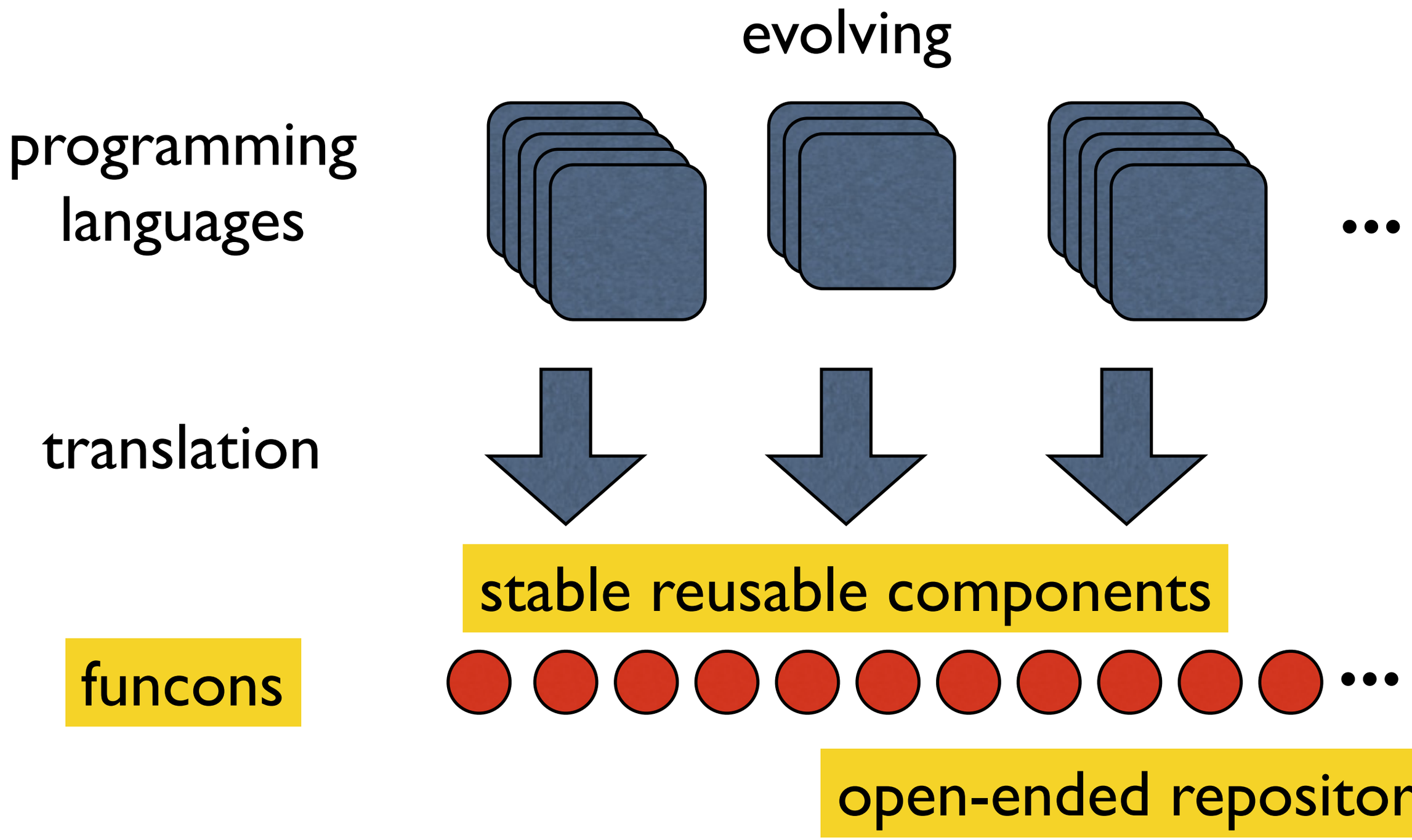
Language semantics

- ***translation*** from language constructs to funcon terms
- language semantics ***derived*** from funcon semantics

Conjecture

Using ***component-based*** semantics can significantly reduce the effort of language specification

Component-based semantics



Language specification in CBS

Syntax

```
E : exp ::= ... | 'let' id '=' exp 'in' exp | ...
```

Semantics

```
eval[[ _ : exp ]] : =>exp-values
```

Rule

```
eval[[ 'let' I '=' E1 'in' E2 ]] =
  scope ( bind-value ( I, eval[[E1]] ), eval[[E2]] )
```

Language specification in CBS

co-evolution

Syntax

```
S : stm ::= ... | 'while' '(' exp ')' stm | ...
```

Semantics

```
exec[[ _ : stm ]] : =>null-type
```

Rule

```
exec[[ 'while' '(' E ')' S ]] =
                    while-true ( eval[[E]]  , exec[[S]] )
```

Language specification in CBS

co-evolution

Syntax

```
S : stm ::= ... | 'while' '(' exp ')' stm | ...
```

Semantics

```
exec[[ _ : stm ]] : =>null-type
```

Rule

```
exec[[ 'while' '(' E ')' S ]] =
        while-true ( not is-eq( 0, eval[[E]] ), exec[[S]] )
```

Language specification in CBS

co-evolution

Syntax

```
S : stm ::= ... | 'while' '(' exp ')' stm | 'break' | ...
```

Semantics

```
exec[[ _ : stm ]] : =>null-type
```

Rule

```
exec[[ 'while' '(' E ')' S ]] =
        handle-break ( while-true ( eval[[E]], exec[[S]] ) )
```

Rule

```
exec[[ 'break' ]] = break
```

Funcons

Funcons

Fundamental programming constructs

- language-***independent***
- (mostly) ***specified independently***
- have ***fixed behaviour***
- ***extensible***

Funcons

Computations

- **Normal:** flowing, giving, binding, storing, linking, generating, interacting, …

- **Abnormal:** failing, throwing, breaking, continuing, returning, controlling, …

- (Concurrent: not yet specified)

Funcons

Values *(some types are built-in)*

- **Primitive:** atoms, booleans, integers, floats, characters, null, …

- **Composite:** algebraic datatypes, tuples, lists, vectors, sets, multisets, maps, pointers, references, variants, …

- **Abstractions:** closures, thunks, functions, patterns, …

Funcon specification in CBS

Normal computation: **flowing**

Funcon

```
while-true ( _: =>booleans, _: =>null-type ) : =>null-type
```

Rule

```
while-true ( X, Y )
  ~> if-true-else (
       X,
       sequential ( Y, while-true ( X, Y ) ),
       null-value )
```

Funcon specification in CBS

Normal computation: **flowing**

```
Funcon
  if-true-else ( _: booleans, _: =>T, _: =>T) : =>T

Rule
  if-true-else ( true, X, Y ) ~> X

Rule
  if-true-else ( false, X, Y ) ~> Y
```

Funcon specification in CBS

Normal computation: **flowing**

```
Funcon
  sequential ( _: null-type, _: =>T ) : =>T

Rule
sequential ( null-value, Y ) ~> Y
```

Funcon specification in CBS

Normal computation: **binding**

Type

```
envs ~> maps(ids, values?)
```

Funcon

```
bind-value ( _: ids, _: values ) : =>envs
```

Rule

```
bind-value ( I: ids, V: values ) ~> { I |-> V }
```

Funcon specification in CBS

Normal computation: **binding**

```
Entity
  env(_: envs) |- _ ---> _

Funcon
  scope( _: envs, _: =>T ) : =>T

Rule
  env(map-override(Rho1,Rho0)) |- X ---> X'
  ----------------------------------------
  env(Rho0) |- scope ( Rho1:envs, X ) ---> scope ( Rho1, X')

Rule
  scope ( _:envs, V:T ) ~> V
```

CBS foundations
main features

- ***I-MSOS:*** *SOS* notation, but interpreted as *Modular SOS*
- ***strictness annotations*** in signatures (_ : T vs _ : **=>**T)
- ***value-computation systems***
  - transitions (—>) and rewriting (~>)
- ***bisimulation congruence format*** for rules
- ***bisimulation preserved*** by disjoint extension

CBS beta-release

plancomps.github.io/CBS-beta

Funcons-beta made available **for review**

- *changes possible until final release (autumn 2018)*

Languages-beta illustrates CBS using Funcons-beta

- *simple languages: IMP, SIMPLE, SL*
- *sub-languages: MiniJava, OCaml Light*
- *may need to change before final release*

CBS beta-release

plancomps.github.io/CBS-beta

Demonstrated:

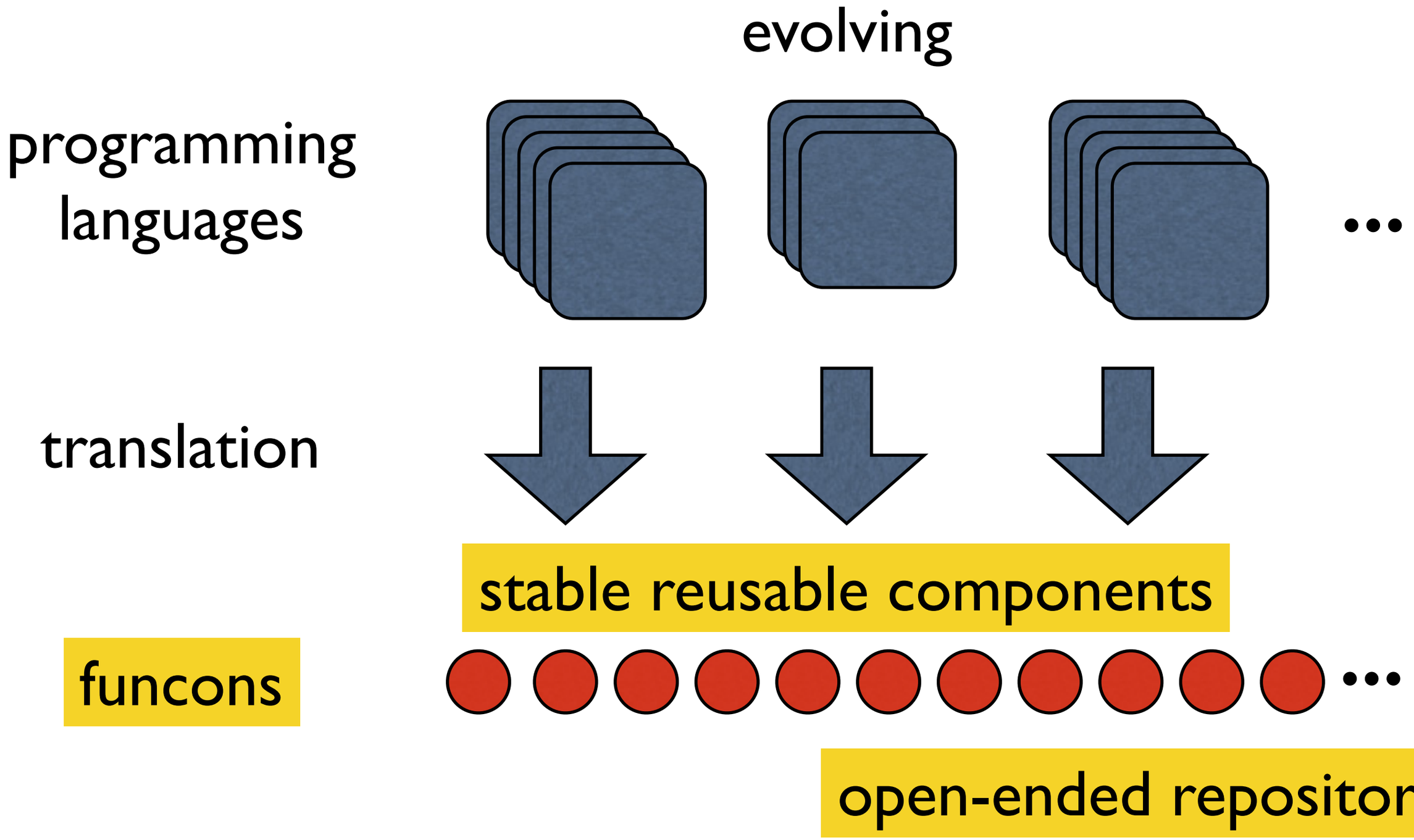
- benefits of funcons: reuse, simplicity
- generation of editors, prototype interpreters

Not yet demonstrated:

- co-evolution, scaling-up, DSLs, modeling languages
- funcon algebra

Conclusion

Language specification in CBS



Conjecture

Using ***component-based semantics*** can significantly reduce the effort of language specification

Static semantics questions

- ***modular static semantics*** for funcons?
  - unrestricted effects, co-effects
- ***type-directed translation*** of language constructs?
  - separate conventional static semantics?
  - exploitation of scope graphs?

PLANCOMPS

plancomps.org

"Programming Language Components and Specifications"

Funded project 2011–16: EPSRC Engineering and Physical Sciences Research Council

- at Swansea, Royal Holloway (RHUL), City, Newcastle

Current participants:

- A. Johnstone, E.A. Scott, L.T. van Binsbergen (RHUL)
  N. Sculthorpe (NTU), C. Bach Poulsen, PDM (Delft)

New participants are welcome !