Run Your Component-Based Semantics

Thomas van Binsbergen
Royal Holloway, University of London, UK
Peter Mosses, Neil Sculthorpe
Swansea University, UK

WG 2.11 Meeting, London, UK, Novenber 2015

Related work: Redex

POPL 2012:

Run Your Research

On the Effectiveness of Lightweight Mechanization

Casey Klein[1] John Clements[2] Christos Dimoulas[3] Carl Eastlund[3] Matthias Felleisen[3]
Matthew Flatt[4] Jay A. McCarthy[5] Jon Rafkind[4] Sam Tobin-Hochstadt[3] Robert Bruce Findler[1]
PLT

- found flaws in formal semantics used in nine ICFP papers
- models formulated in Redex
  - a domain-specific meta-language *embedded in Racket*
- DrRacket IDE runs programs based on their semantics

Racket example

```
(define-language Λc
  (e (e e ...)
     x
     (λ (x ...) e)
     call/cc
     +
     number)
  (x variable-
```

```
(define red
  (reduction-relation
   Λc/red #:domain e
   (--> (in-hole E (A e))
        e
        "abort")
   (--> (in-hole E (call/cc v))
        (in-hole E (v (λ (x) (A (in-hole E x)))))
        (fresh x)
        "call/cc")
   (--> (in-hole E ((λ (x ..._1) e) v ..._1))
        (in-hole E (subst e (x v) ...))
        "βv")
   (--> (in-hole E (+ number ...))
        (in-hole E (Σ number ...))
        "+")))
```

Related work: K

POPL'12, ACM, pp 533-544. 2012

An Executable Formal Semantics of C with Applications *

Chucky Ellison Grigore Roșu
University of Illinois
{celliso2, grosu}@illinois.edu

Abstract

This paper describes an executable formal semantics of C. Being executable, the semantics has been thoroughly tested against the GCC torture test suite and successfully passes 99.2% of 776 test programs. It is the most complete and thoroughly tested formal definition of C to date. The semantics yields an interpreter, debugger, state space search tool, and model checker "for free". The semantics is shown capable of automatically finding program errors, both statically and at runtime. It is also used to enumerate nondeterministic behavior.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

General Terms Languages, Standardization, Verification.

1. Introduction

C provides just enough abstraction above assembly language for programmers to get their work done without having to worry about the details of the machines on which the programs run. Despite this abstraction, C is also known for the ease in which it allows a version of C that includes every language feature except for `_Complex` and `_Imaginary` types, and that includes only a subset of the standard library. Our semantics is the first arguably complete dynamic semantics of C (see Section 2).

Above all else, our semantics has been motivated by the desire to develop formal, yet practical tools. Our semantics was developed in such a way that the single definition could be used immediately for interpreting, debugging, or analysis (described in Section 6). At the same time, this practicality does not mean that our definition is not formal. Being written in a subset of rewriting logic (RL), it comes with a complete proof system and initial model semantics [18]. Briefly, a rewrite system is a set of rules over terms constructed from a signature. The rewrite rules match and apply everywhere, making RL a simple, uniform, and general formal computational paradigm. This is explained in greater detail in Section 3.

Our C semantics defines 150 C syntactic operators. The definitions of these operators are given by 1,163 semantic rules spread over 5,884 source lines of code (SLOC). However, it takes only 77 of those rules (536 SLOC) to cover the behavior of statements, and another 163 for expressions (748 SLOC). There are 505 rules

K example

MODULE LAMBDA-SYNTAX

SYNTAX $Exp ::= Val$
$\mid ExpExp$ [seqstrict]

SYNTAX $Val ::= \lambda Id.Exp$ [binder]
$\mid Id$

END MODULE

MODULE LAMBDA

IMPORTS LAMBDA-SYNTAX+SUBSTITUTION

CONFIGURATION:

k

$\$PGM$

SYNTAX $KResult ::= Val$

β-substitution

RULE $(\lambda X.E)\,V \Rightarrow E[V\ /\ X]$

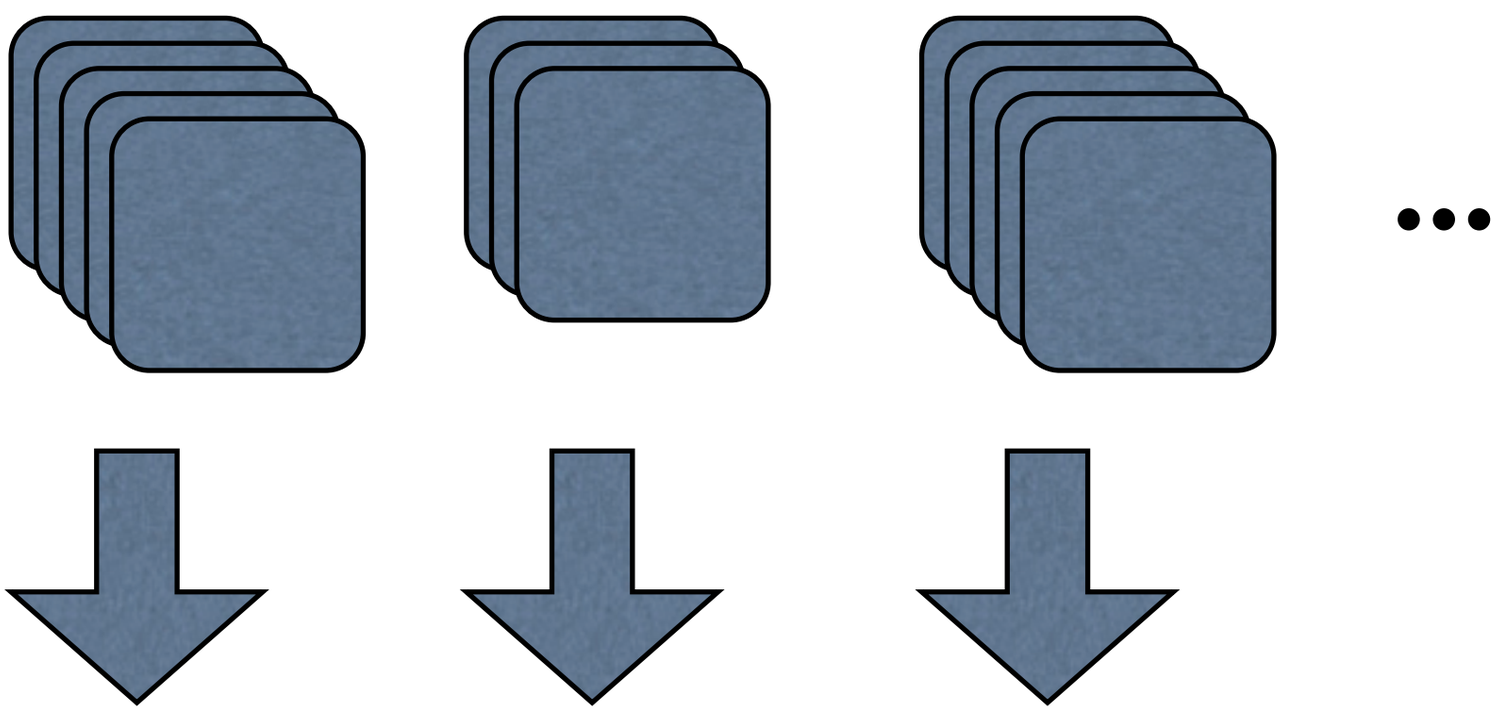
END MODULE

What is component-based semantics?

bb

programming languages

evolving

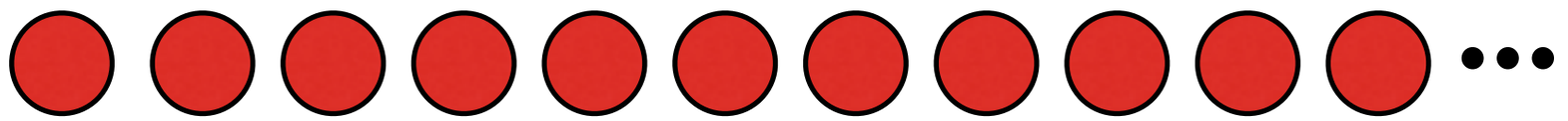


...

translation

stable reusable components

fundamental constructs



...

open-ended repository

What are fundamental constructs?

Computation primitives and combinators

- sequential, if-then-else, while, bind, bound, scope, allocate-initialised-variable, store-value, stored-value, …

Value constants, operations, and types

- booleans, is-less-or-equal, not, integers, integer-add, ( ), environments, map-unite, variables, values, types, …

Values can be implicitly lifted to computations

- e.g.: while(**not**(stored-value(bound(“b”))), …)

CBS

an *external* domain-specific *meta*-language

CBS: component-based specification – *denotational-style translation*

```
stmt ::= block
       |  id '=' aexp ';'
       |  'if' '(' bexp ')' block ('else' block)?
       |  'while' '(' bexp ')' block
       |  stmt stmt
```

abstract syntax

translation functions

```
evaluate[[ _:aexp ]] : =>integers
```

```
execute[[ I '=' AExp ';' ]] =
  store-value(bound(id[[ I ]]), evaluate[[ AExp ]])
```

fundamental constructs

translation equations

Fundamental construct specifications – *using CBS variant of modular SOS*

Entity environment(ρ: environments) ⊢ _ → _

Funcon **scope**(_ : ⇒environments, _ : ⇒*T*) : ⇒*T*

$$\frac{\mathsf{environment}(\rho) \vdash D \to D' \qquad \mathsf{environment}(\rho'/\rho) \vdash X \to X'}{\mathsf{environment}(\rho) \vdash \mathbf{scope}(\rho', D) \to \mathbf{scope}(\rho', D')}$$

$$\mathsf{environment}(\rho) \vdash \mathbf{scope}(\rho, V{:}\mathsf{values}) \to V$$

Tool support

Tool support for CBS: IDE

The Spoofax Language Workbench

Spoofax is a platform for developing textual domain-specific languages with full-featured Eclipse editor plugins.

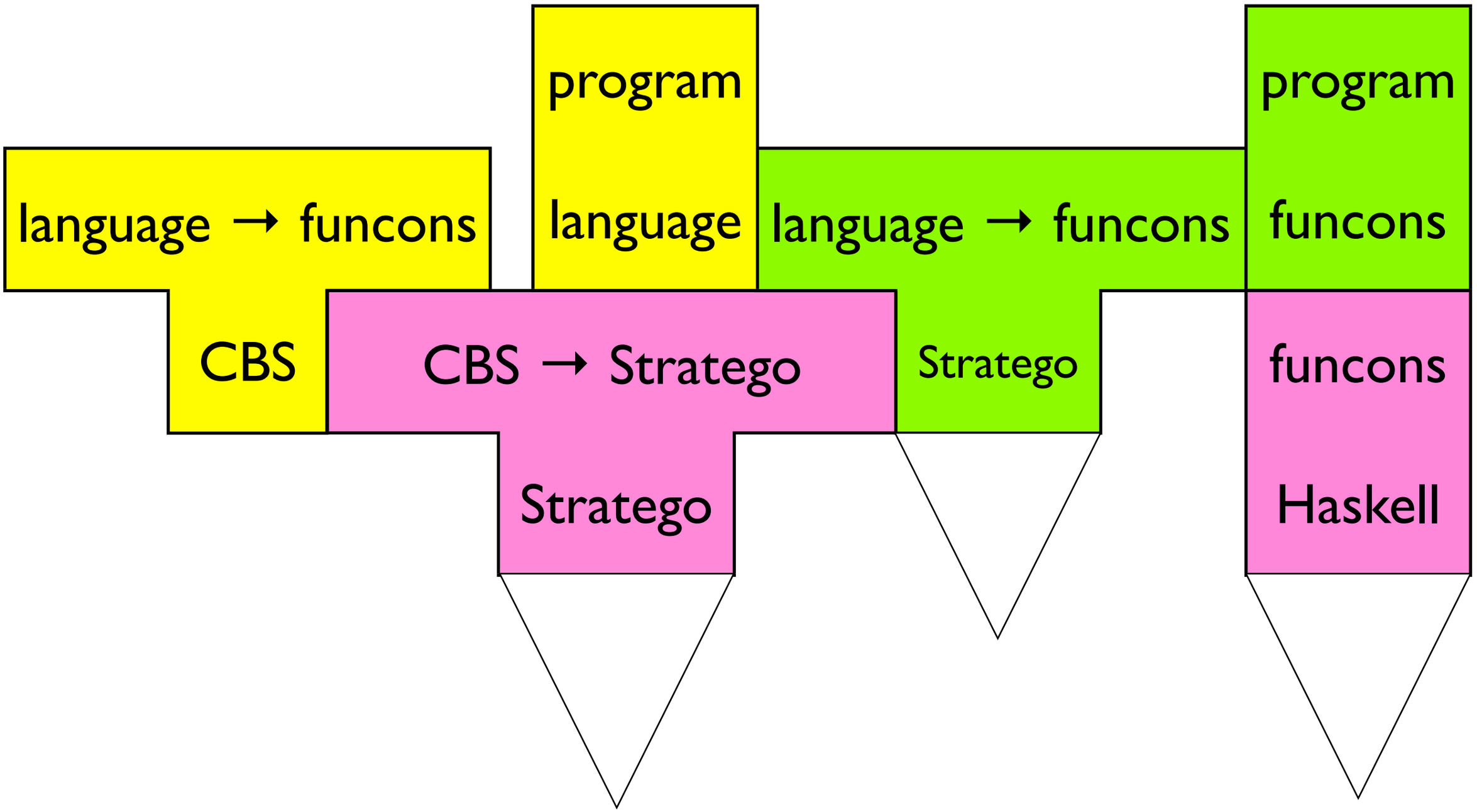
`metaborg.org/spoofax`

Meta Languages

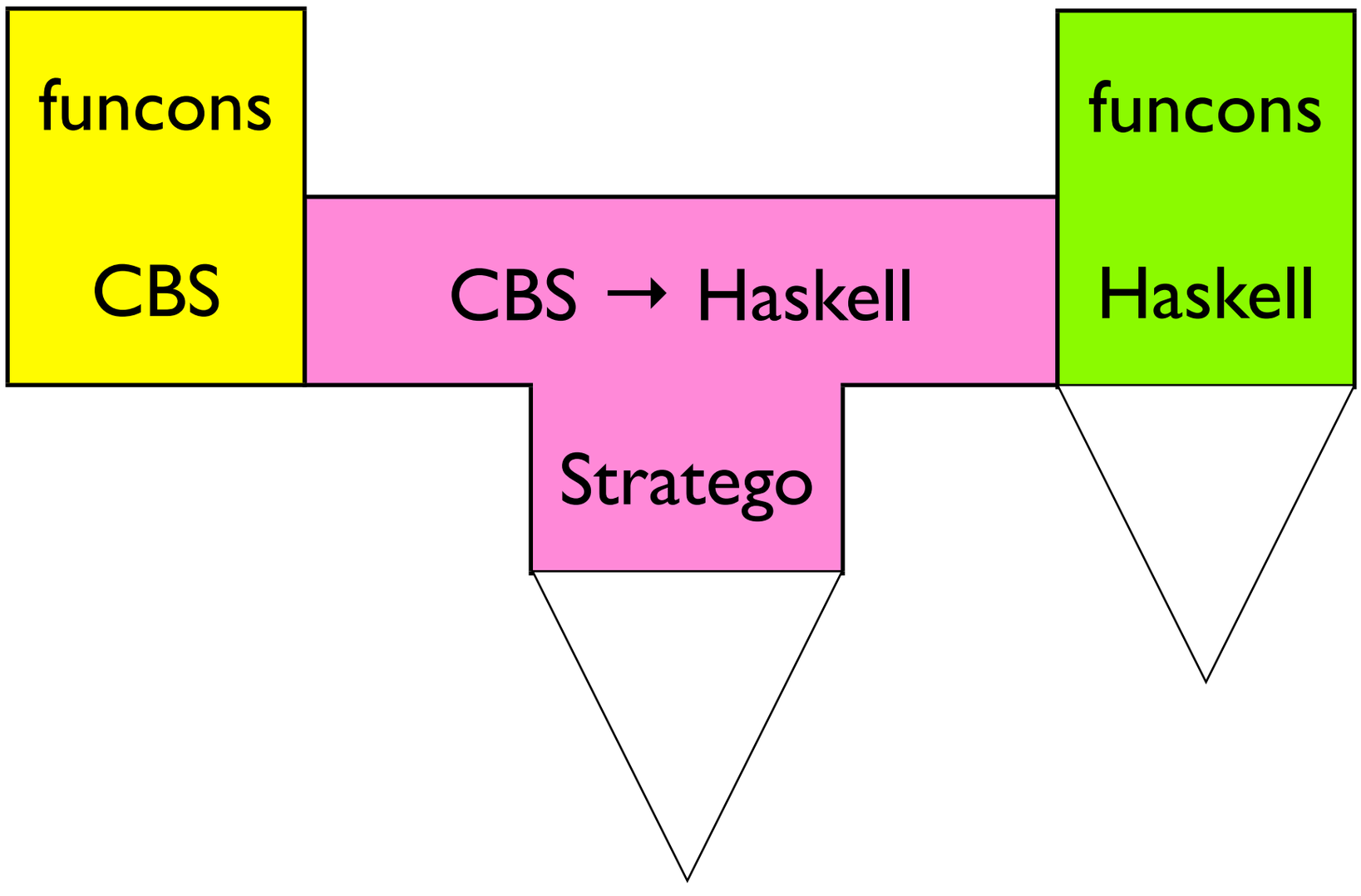
Language definitions in Spoofax are constructed using the following meta-languages:

- The SDF3 syntax definition formalism
- The NaBL name binding language
- The TS type specification language
- The Stratego transformation language

Current tool support:
CBS-based program execution



Future tool support: CBS-based interpreter generation



Demo

- browsing/editing CBS specifications
- translating programs to funcons
- executing funcons
- generating translators

Conclusion

Current version of CBS tools available for download

- `www.plancomps.org/nwpt2015-tsc`
- tested with pre-packaged Spoofax/Eclipse distribution

CBS scales up to larger languages

- CAML LIGHT [*Modularity'14 special issue, Trans. AOSD, 2015*]
- C# [*work in progress*]

Fundamental constructs (funcons) appear to be

highly reusable components