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, November 2015
Related work: Redex

POPL 2012:

Run Your Research
On the Effectiveness of Lightweight Mechanization

Casey Klein¹ John Clements² Christos Dimoulas³ Carl Eastlund³ Matthias Felleisen³
Matthew Flatt⁴ Jay A. McCarthy⁵ Jon Rafkind⁴ Sam Tobin-Hochstadt³ Robert Bruce Findler¹
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
Redex toolkit covers a variety of tasks related to executing semantic models in Racket. Redex programmers inherit the DrRacket IDE, a large suite of tools for programming and debugging. The Redex language allows for the specification of operational semantics, and Redex programs can be used to verify properties of semantic models.

The next section reviews the Redex modeling language and tool support. Redex supports randomized testing à la QuickCheck (Claessen and Hughes 2000); spectors for reduction graphs; a unit testing framework; a tool for checking formal and informal claims into testable conjectures; and checked proofs for their validity. In the process, we found mistakes in all of the papers, including one whose essential result had been verified in Coq.

Research papers. Specifically, the authors encoded nine ICFP 2009 implementations with a language model. The second shows that the authors have used the Redex language for other tasks, such as a PDF snippet.
Related work: K

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  \[ Exp ::= Val \\
         | ExpExp [seqstrict] \]

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

END MODULE

MODULE LAMBDA

IMPORTS LAMBDA-SYNTAX + SUBSTITUTION

CONFIGURATION:

\[ \text{\$PGM} \]

SYNTAX  \[ KResult ::= Val \]

\[ \beta\text{-substitution} \]

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

END MODULE
What is component-based semantics?

bb
programming languages

translation

fundamental constructs

stable reusable components

open-ended repository

evolving
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(\textcolor{red}{\textbf{not}}(\text{stored-value}(\text{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

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

execute[[ I '!=' AExp ';
] =
store-value(bound(id[[ I ]]), evaluate[[ AExp ]])
Fundamental construct specifications – using CBS variant of modular SOS

**Entity** environment\( (\rho: \text{environments}) \vdash _\_ \rightarrow _\_ \)**

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

\[
\text{environment}(\rho) \vdash D \rightarrow D'
\]

\[
\text{environment}(\rho) \vdash \text{scope}(D, X) \rightarrow \text{scope}(D', X)
\]

\[
\text{environment}(\rho'/\rho) \vdash X \rightarrow X'
\]

\[
\text{environment}(\rho) \vdash \text{scope}(\rho', X) \rightarrow \text{scope}(\rho', X')
\]

\[
\text{environment}(\rho) \vdash \text{scope}(\rho, V: \text{values}) \rightarrow V
\]
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.

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

- CBS-based program execution
- CBS → Stratego
- Stratego → funcons
- Funcons → Haskell

Program language → funcons language → funcons language → funcons

CBS
**Future tool support:**

**CBS-based interpreter generation**

- **CBS** → **Haskell**
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