A component-based approach to semantics

Peter Mosses
Swansea University, UK

Modularity’15 • March 18, 2015
Fort Collins, Colorado, USA
Modularity

Good to have!

What might be even better?

reusable components

Software development:

“The Unix Philosophy”

Programming language definitions:

component-based semantics
### Programming language definitions

Reference manuals, standards documents

- **syntax:**
  - always *formal*

### 14.12 The `while` Statement

The `while` statement executes an `Expression` and a `Statement` repeatedly until the value of the Expression is `false`.

**WhileStatement:**
```
while ( Expression ) Statement
```
The expression must have type `boolean` or `Boolean`, or a compile-time error occurs.

A `while` statement is executed by first evaluating the expression. If the result is of type `boolean`, it is subject to unboxing conversion (§5.1.8).

If evaluation of the expression or the subsequent unboxing conversion (if any) completes abruptly for some reason, the `while` statement completes abruptly for the same reason. Otherwise, execution continues by making a choice based on the resulting value:

- If the value is `true`, then the contained statement is executed. Then there is a choice:
  - If execution of the statement completes normally, then the entire `while` statement is executed again, beginning by re-evaluating the expression.
  - If execution of the statement completes abruptly, see §14.12.1.
- If the (possibly unboxed) value of the expression is `false`, no further action is taken and the `while` statement completes normally.

If the (possibly unboxed) value of the expression is `false` the first time it is evaluated, then the statement is not executed.

### 14.12.1 Abrupt Completion of `while` Statement

Abrupt completion of the contained statement is handled in the following manner:

- If execution of the statement completes abruptly because of a `break` with no label, no further action is taken and the `while` statement completes normally.
- If execution of the statement completes abruptly because of a `continue` with no label, then the entire `while` statement is executed again.
- If execution of the statement completes abruptly because of a `continue` with label `L`, then there is a choice:
  - If the `while` statement has label `L`, then the entire `while` statement is executed again.
  - If the `while` statement does not have label `L`, the `while` statement completes abruptly because of a `continue` with label `L`.
- If execution of the statement completes abruptly for any other reason, the `while` statement completes abruptly for the same reason.
Formal semantics

Many semantic frameworks:
- operational, denotational, algebraic, axiomatic, ...

Only a few official language definitions use formal semantics:
- Ada, Modula-2, Standard ML, Scheme

Some other languages have unofficial formal semantics:
- Algol 60, C, C#, Java, PL/I, Prolog, ...

Many major languages have no formal semantics:
- C++, Haskell, OCaml, Scala, ...
# Formal semantic frameworks

## Operational
- VDL
- SOS (small- or big-step)
- Reduction semantics, K
- ASM

## Denotational
- Scott–Strachey
- VDM
- Monadic

## Axiomatic
- Hoare logic
- Algebraic

## Hybrid
- Action semantics
- UTP

## Static
- Typing rules
- Abstract interpretation
Programming language evolution

Mother Tongues
Tracing the roots of computer languages through the ages

Just like half of the world's spoken tongues, most of the 2,300-plus computer programming languages are either endangered or extinct. As powerhouses C++, Visual Basic, Cobol, Java and other modern source codes dominate our systems, hundreds of older languages are running out of life.

An ad hoc collection of engineers-electronic lexicographers, if you will—aim to save, or at least document the lingo of classic software. They're combing the globe's 9 million developers in search of coders still fluent in these nearly forgotten lingua francas. Among the most endangered are Ada, APL, B (the predecessor of C), Lisp, Oberon, Smalltalk, and Simula.

Code-raker Grady Booch, Rational Software's chief scientist, is working with the Computer History Museum in Silicon Valley to record and, in some cases, maintain languages by writing new compilers so our ever-changing hardware can grok the code. Why bother? "They tell us about the state of software practice, the minds of their inventors, and the technical, social, and economic forces that shaped history at the time," Booch explains. "They'll provide the raw material for software archaeologists, historians, and developers to learn what worked, what was brilliant, and what was an utter failure." Here's a peek at the strongest branches of programming's family tree. For a nearly exhaustive rundown, check out the Language List at HTTP://www.informatik.uni-freiburg.de/Jara/mics/lang_list.html. - Michael Mendes

Survival of the Fittest
Reasons a language endures, with examples of some classic tongues

Appeals to a wide audience

C (blistered by the popularity of Unix)

Gets a job done

Cobol (designed for business-report writing)

Delivers new functionality

Java runs on any hardware platform

Fills a niche

Mathematica (speeds up complex computations)

Offers a modicum of elegance

Icon (has friendly, line-oriented syntax)

Has a powerful user base or backer

C# (developed by Microsoft for .Net)

Has a charismatic leader

Perl (programmer-author Larry Wall)

Sources: Paul Boutin; Brent Hallgren, associate director of computer science at IBM Research; The Retrocomputing Museum; Todd Proebsting, senior researcher at Microsoft; Gio Wiederhold, computer scientist, Stanford University.
The importance of being formal

Only a *formal* semantics can be

- precise
- concise

and allow

- validation
- reasoning
- prototyping
How to improve?

**Reusable components**

to reduce the *initial* effort

**High modularity**

to reduce the effort of *change*

**Tool support**

to reduce the effort of *getting it right!***
Abstract. Semantic specifications of programming languages typically have poor modularity. This hinders reuse of parts of the semantics of one language when specifying a different language – even when the two languages have many constructs in common – and evolution of a language may require major reformulation of its semantics. Such drawbacks have discouraged language developers from using formal semantics to document their designs.

In the PLanCompS project, we have developed a component-based approach to semantics. Here, we explain its modularity aspects, and present an illustrative case study: a component-based semantics for Caml Light.

We have tested the correctness of the semantics by running programs on an interpreter generated from the semantics, comparing the output with that produced on the standard implementation of the language.

Our approach provides good modularity, facilitates reuse, and should support co-evolution of languages and their formal semantics. It could be particularly useful in connection with domain-specific languages and language-driven software development.

Keywords: modularity, reusability, component-based semantics, fundamental constructs, funcons, modular SOS

1 Introduction

Various programming constructs are common to many languages. For instance, assignment statements, sequencing, conditional branching, loops, and procedure calls are almost ubiquitous among languages that support imperative programming; expressions usually include references to declared variables and constants, arithmetic and logical operations on values, and function calls; and blocks are provided to restrict the scope of local declarations. The details of such constructs often vary between languages, both regarding their syntax and their intended behaviour, but sometimes they are identical.

Many constructs are also ‘independent’, in that their contributions to program behaviour are unaffected by the presence of other constructs in the same language. For instance, consider conditional expressions ‘E₁ ? E₂ : E₃’. How they are evaluated is unaffected by whether expressions involve variable references, side effects, function calls, process synchronisation, etc. In contrast, the behaviour of a loop may depend on whether the language includes break and continue statements.
Reusable components
Reusable software components

**COTS** – ‘Components Off The Shelf’

- typically complex software
  
  - *example*: Windows for driving medical devices

**Libraries and packages**

- greatly enhance productivity

- but upgrades to new versions can be problematic…
The Unix Philosophy

Formulated in the 1980s by Ken Thompson, Dennis Ritchie, Brian Kernighan, Doug McIlroy, Rob Pike, et al.

The design of `cat` is typical of most UNIX programs: it implements one simple but general function that can be used in many different applications (including many not envisioned by the original author). Other commands are used for other functions.

Reusable components of language definitions

- *language* constructs?
- *kernel language* constructs?
- *fundamental* programming constructs!

Component-based semantics

Translation

Language₁  Language₂  Language₃

...
Reusable components

Fundamental constructs (funcons)

- correspond to *individual* programming constructs
  - each funcon is a separate component

- have *(when validated and released)*
  - fixed notation
  - fixed behaviour
  - fixed algebraic properties

specified/proved once and for all!
Modular Structural Operational Semantics

Peter D. Mosses

Implicit Propagation in Structural Operational Semantics

Peter D. Mosses¹  Mark J. New²
Abstract. For structural operational semantics (SOS) of process algebras, various notions of bisimulation have been studied, together with rule formats ensuring that bisimilarity is a congruence. For programming languages, however, SOS generally involves auxiliary entities (e.g. stores) and computed values, and the standard bisimulation and rule formats are not directly applicable.

Here, we first introduce a notion of bisimulation based on the distinction between computations and values, with a corresponding liberal congruence format. We then provide metatheory for a modular variant of SOS (MSOS) which provides a systematic treatment of auxiliary entities. This is based on a higher order form of bisimulation, and we formulate an appropriate congruence format. Finally, we show how algebraic laws can be proved sound for bisimulation with reference only to the (M)SOS rules defining the programming constructs involved in them. Such laws remain sound for languages that involve further constructs.
Fundamental constructs (funcons)

Funcons *normally compute values*

- values compute themselves

Funcon computations may also:

- *terminate abruptly*
  - signalling some value as the reason
  - failure is a special case

- *never terminate*

- *have effects*
Values

Universe

- **primitive** (booleans, numbers, characters, symbols)
- **composite** (sequences, maps, sets, variants)
- **types** (names for sets of values)
- **abstractions** (encapsulating funcons)

*New types of values are defined in terms of old ones*
Funcon ‘aspects’

(Mostly) independent concerns

- control flow
- data flow
- binding
- storing
- interacting

Each funcon has a primary ‘aspect’
Sorts of funcons

Notation

- **commands**
  - $C :$ computes $()$

- **declarations**
  - $D :$ computes environments (mapping ids $I$ to values $V$)

- **expressions**
  - $E :$ computes values

Generic funcons

- $X :$ could be commands, declarations, expressions
Control flow

Normal

- **seq**(\(X_1, \ldots\))
  - left to right sequencing
  - concatenates computed values

- **null** is the empty sequence ( )
  - unit for \(\text{seq}(X_1, X_2)\)
Control flow

**Conditional**

- **if-true-else**($E, X_1, X_2$)
  - $E$ has to be boolean-valued

- **while-true**($E, C$)
  - doesn’t handle break or continue

**Call**

- **enact**($E$)
  - evaluates $E$ to an abstraction value $\text{abs}(X)$
  - executes $X$
Control flow

Alternatives

- **either**($X_1, \ldots$)
  - unordered alternatives
- **else**($X_1, \ldots$)
  - left to right alternatives
- **fail**
  - unit for either($X_1, X_2$) and else($X_1, X_2$)
- **when-true**($E, X$), **check-true**($E$)
  - **fail** when $E$ false
Data flow

Lifting operations

- value operations $F(V_1, \ldots)$ lift to funcons $F(E_1, \ldots)$
  - argument evaluation implicitly *interleaved*
  - $F(\text{seq}(E_1, \ldots))$ ensures *left to right* evaluation

  e.g.: $\text{not} (\text{is-equal} (\text{seq}(E_1, E_2)))$

Discarding values

- $\text{effect}(X)$
  - executes $X$, but computes $()$
Control and data flow

Giving

- **give-val**($E, X$)
  - first evaluates $E$ to a value $V$
  - then executes $X$, with the funcon **given** referring to $V$

- **given**

Application

- **apply**($E_1, E_2$)
  - evaluates $E_1$ to an abstraction **abs**(X), and evaluates $E_2$ to a value $V$
  - then executes $X$, with the funcon **given** referring to $V$
Control and data flow

Exception handling

- **handle-thrown**($X_1, X_2$)
  
  - try to handle abrupt termination of $X_1$ by giving the thrown value to the execution of $X_2$

- **throw-val**($E$)
  
  - terminates abruptly, throwing the value of $E$

Continuations

- see the paper by Neil Sculthorpe et al. at the ETAPS 2015 Workshop on Continuations
Binding

Scopes

- **scope**($D, X$)
  
  - localises the bindings computed by $D$ to $X$

- **bind-val**($l, E$)
  
  - computes the binding of the id $l$ to the value of $E$

- **bound-val**($l$)
  
  - inspects the current binding of the id $l$
Binding

Scopes

- override($D_1, D_2$)
- unite($D_1, D_2$)
- accumulate($D_1, D_2$)
- recursive($lset, D$)

- various ways of composing declarations
Binding

**Scopes in abstractions**

- **close**(E)
  - evaluates E to an abstraction **abs**(X)
  - returns the *closure* incorporating the current bindings

**Patterns**

- **simple**: abstractions **abs**(D)
- **composite**: formed using value *constructors*
  - structure (and any immutable components) required to be identical when matching
Binding

Pattern matching

- **match-val**($E_1, E_2$)
  - evaluates $E_1$ to a pattern $P$ and $E_2$ to a value $V$
  - matching $P$ to $V$ computes bindings

- **case**($E, X$)
  - evaluates $E$ to a pattern $P$, then matches $P$ to a given value
  - the scope of the computed bindings is $X$
  - equivalent to **scope**(match($E$, given), $X$)
Storing

Variables

- **simple**: representing independent storage locations
  - for storing values of a fixed type
  - monolithic update

- **composite**: formed using value *constructors*
  - component variables can be independently updated
  - structure (and any *immutable* components) required to be identical when updating
Storing

Variable allocation

- **alloc**\((E_1, E_2)\)
  - evaluates \(E_1\) to a type \(T\), and \(E_2\) to a value \(V\)
  - allocates a simple or composite variable for storing values of type \(T\)
  - assigns \(V\) to the variable

- **release**\((E)\)
  - evaluates \(E\) to a variable
  - terminates the allocation of the variable
Storing

**General assignment**

- **assign**($E_1, E_2$)
  
  - evaluates $E_1$ to $V_1$, and $E_2$ to $V_2$
  
  - when $V_1$ and $V_2$ have the same structure, updates the stored values of any simple variables in $V_1$ by the corresponding component values of $V_2$

- **current-val**($E$)
  
  - evaluates $E$ to $V$
  
  - gives the value formed by replacing any simple variables in $V$ by their stored values
A component reuse example

Language construct:

\[ stm ::= \text{while}(\text{exp}) \text{stm} \]

Translation to funcons:

\[ \text{exec} \left[ \text{while}(E) S \right] = \]
\[ \text{while-true}(\text{current-val}(\text{eval} \left[ E \right]), \text{exec} \left[ S \right]) \]

For languages with break statements:

\[ \text{exec} \left[ \text{while}(E) S \right] = \]
\[ \text{handle-thrown}(\text{while-true}(\text{current-val}(\text{eval} \left[ E \right]), \text{exec} \left[ S \right]), \]
\[ \text{case}(\text{‘break’, null}) ) \]
High modularity
Component-based semantics

Reusable components of language definitions

- **fundamental** programming constructs

Translation

Flat structure: open-ended

Moderated – no versioning!
Component specification
SOS: Structural operational semantics

\[(X, \rho, \sigma, \ldots) \xrightarrow{\alpha} (X', \rho, \sigma', \ldots) \xrightarrow{\alpha'} (X'', \rho, \sigma'', \ldots)\]

Plotkin (1981)

- **(optionally-)labelled** transition relations
- **states**: include programs \(X\), environments \(\rho\), stores \(\sigma\), …
  - environments preserved by \(\rho \vdash (\ldots) \rightarrow (\ldots)\)
- **labels**: simple synchronisation actions \(\alpha\)
MSOS: Modular SOS

\[ X \xrightarrow{(\rho, \sigma, \sigma', \alpha', \ldots)} X' \xrightarrow{(\rho, \sigma', \sigma'', \alpha'', \ldots)} X'' \]

M (1999)

- *arrow-labelled* transition relations
- **states:** simple programs $X$, computed values $V$
- **labels:** include environments $\rho$, stores $\sigma$, actions $\alpha$, …
  - adjacent labels required to be *composable*
    - fixed environment ($\rho$)
    - store updates ($\sigma, \sigma'$)
Component specification

Notation

\[ \text{if-true-else}(E : \text{computes(booleans)}, X_1, X_2 : \text{computes}(T)) : \text{computes}(T) \]

Static semantics

\[
\begin{align*}
E : \text{booleans} & \quad X_1 : T \\
\hline
\text{if-true-else}(E, X_1, X_2) : T
\end{align*}
\]

Dynamic semantics

\[
\begin{align*}
E \rightarrow E' & \quad \text{if-true-else}(true, X_1, X_2) \rightarrow X_1 \\
\hline
\text{if-true-else}(E, X_1, X_2) \rightarrow \text{if-true-else}(E', X_1, X_2) & \quad \text{if-true-else}(false, X_1, X_2) \rightarrow X_2
\end{align*}
\]
Component specification

Notation

\[ \text{if-true-else}(E : \text{computes(boolean)}, X_1, X_2 : \text{computes}(T)) : \text{computes}(T) \]

Static semantics

\[
\begin{align*}
E : \text{boolean} & \quad X_1 : T & \quad X_2 : T \\
\hline
\text{if-true-else}(E, X_1, X_2) : T
\end{align*}
\]

Dynamic semantics

\[
\begin{align*}
E \rightarrow E' & \quad \text{if-true-else}(\text{true}, X_1, X_2) \rightarrow X_1 \\
\text{if-true-else}(E, X_1, X_2) & \rightarrow \text{if-true-else}(E', X_1, X_2) \\
\text{if-true-else}(\text{false}, X_1, X_2) & \rightarrow X_2
\end{align*}
\]
Component specification

Notation

\texttt{scope}(\text{computes(\text{envs}), computes(T) }): \text{computes(T)}

Static semantics

\[
\begin{align*}
\text{env}(\rho) \vdash D : \rho' \\
\text{env}(\rho'/\rho) \vdash X : T
\end{align*}
\]

\[
\text{env}(\rho) \vdash \text{scope}(D, X) : T
\]

Dynamic semantics

\[
\begin{align*}
D \rightarrow D' \\
\text{scope}(D, X) \rightarrow \text{scope}(D', X)
\end{align*}
\]

\[
\begin{align*}
\text{env}(\rho'/\rho) \vdash X \rightarrow X' \\
\text{env}(\rho) \vdash \text{scope}(\rho', X) \rightarrow \text{scope}(\rho', X')
\end{align*}
\]

\[
\text{scope}(\rho, V) \rightarrow V
\]
Component specification

**Notation**

\[
\text{scope}(\text{computes(\text{envs}), computes}(T)) : \text{computes}(T)
\]

**Static semantics**

\[
\begin{align*}
\text{env}(\rho) & \vdash D : \rho' \\
\text{env}(\rho'/\rho) & \vdash X : T \\
\text{env}(\rho) & \vdash \text{scope}(D, X) : T
\end{align*}
\]

**Dynamic semantics**

\[
\begin{align*}
D & \rightarrow D' \\
\text{scope}(D, X) & \rightarrow \text{scope}(D', X) \\
\text{env}(\rho'/\rho) & \vdash X \rightarrow X' \\
\text{env}(\rho) & \vdash \text{scope}(\rho', X) \rightarrow \text{scope}(\rho', X')
\end{align*}
\]

\[
\text{scope}(\rho, V) \rightarrow V
\]
Tool support
Preliminary tool support

**Spoiler/Eclipse**

- parsing programs (SDF3, disambiguation, AST creation)
- translating ASTs to funcon terms (SDF3, STRATEGO)
- browsing and editing component-based specifications (SDF3, NABL, STRATEGO)

**Prolog**

- translating MSOS rules for funcons to PROLOG
  - currently migrating to STRATEGO
- running funcon terms
Future tool support

ESOP’14:

- refocusing
  - small-step
  - (M)SOS rules

---

Deriving Pretty-Big-Step Semantics from Small-Step Semantics

Casper Bach Poulsen and Peter D. Mosses

Department of Computer Science, Swansea University, Swansea, UK,
cscbp@swansea.ac.uk, p.d.mosses@swansea.ac.uk

Abstract. Big-step semantics for languages with abrupt termination and/or divergence suffer from a serious duplication problem, addressed by the novel ‘pretty-big-step’ style presented by Charguéraud at ESOP’13. Such rules are less concise than corresponding small-step rules, but they have the same advantages as big-step rules for program correctness proofs. Here, we show how to automatically derive pretty-big-step rules directly from small-step rules by ‘refocusing’. This gives the best of both worlds: we only need to write the relatively concise small-step specifications, but our reasoning can be big-step as well as small-step. The use of strictness annotations to derive small-step congruence rules gives further conciseness.
FunKons: Component-Based Semantics in K

Peter D. Mosses and Ferdinand Vesely

Swansea University, Swansea SA2 8PP, UK
{p.d.mosses,csfvesely}@swansea.ac.uk

Abstract. Modularity has been recognised as a problematic issue of programming language semantics, and various semantic frameworks have been designed with it in mind. Reusability is another desirable feature which, although not the same as modularity, can be enabled by it. The K Framework, based on Rewriting Logic, has good modularity support, but reuse of specifications is not as well developed.

The PLanCompS project is developing a framework providing an open-ended collection of reusable components for semantic specification. Each component specifies a single fundamental programming construct, or ‘funcon’. The semantics of concrete programming language constructs is given by translating them to combinations of funcons. In this paper, we show how this component-based approach can be seamlessly integrated with the K Framework. We give a component-based definition of CinK (a small subset of C++), using K to define its translation to funcons as well as the (dynamic) semantics of the funcons themselves.
PLAnComPS project (2011-2015)

Foundations

- component-based semantics [Swansea]
- GLL parsing, disambiguation [RHUL]

Case studies

- CAML LIGHT, C#, JAVA [Swansea]

Tool support

- IDE, funcon interpreter/compiler [RHUL, Swansea]

Digital library

- interface [City], historic documents [Newcastle]
Conclusion

Reusable components

to reduce the initial effort

High modularity

to reduce the effort of change

Tool support

to reduce the effort of getting it right!

Fundamental constructs:
The Unix philosophy for semantics of programming languages