Reusable Components for Programming Language Design

Neil Sculthorpe

Department of Computing and Technology
Nottingham Trent University
neil.sculthorpe@ntu.ac.uk

Nottingham, England
5th October 2016

Defining Programming Languages

- Who decides the meaning of a programming language?
- How do they convey that meaning to others?

Defining Programming Languages

- Who decides the meaning of a programming language?
    - The language creator
- How do they convey that meaning to others?

Defining Programming Languages

- Who decides the meaning of a programming language?
  - The language creator
  - A standards committee

- How do they convey that meaning to others?

Defining Programming Languages

- Who decides the meaning of a programming language?
  - The language creator
  - A standards committee
  - A compiler writer

- How do they convey that meaning to others?

Defining Programming Languages

- Who decides the meaning of a programming language?
  - The language creator
  - A standards committee
  - A compiler writer

- How do they convey that meaning to others?
  - Tutorials/examples

Defining Programming Languages

- Who decides the meaning of a programming language?
  - The language creator
  - A standards committee
  - A compiler writer

- How do they convey that meaning to others?
  - Tutorials/examples
  - A reference implementation

Defining Programming Languages

- Who decides the meaning of a programming language?
  - The language creator
  - A standards committee
  - A compiler writer

- How do they convey that meaning to others?
  - Tutorials/examples
  - A reference implementation
  - A reference manual

Defining Programming Languages

- Who decides the meaning of a programming language?
  - The language creator
  - A standards committee
  - A compiler writer
- How do they convey that meaning to others?
  - Tutorials/examples
  - A reference implementation
  - A reference manual
  - A mathematical definition

Tutorials/Examples

- Good for explaining the basic idea

Tutorials/Examples

- Good for explaining the basic idea
- E.g. after a tutorial explains

  ```
  int i=10, j=3, k=0;
  k = -i;
  k = i-j;
  k = --i;
  ```

  you'll understand most uses of '-'

Tutorials/Examples

- Good for explaining the basic idea
- E.g. after a tutorial explains

```
int i=10, j=3, k=0;
k = -i;
k = i-j;
k = --i;
```

  you'll understand most uses of '-'

- But what about these?

```
k = -INT_MIN;
k = -i---j--;
```

Implementations

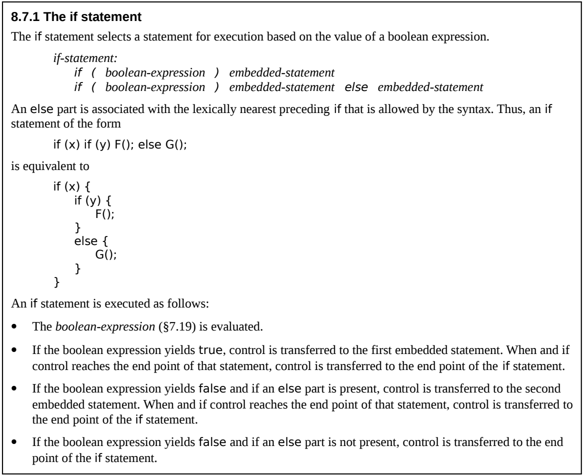
- Easy to find out what one particular bit of code does
- Good for testing against
- Poor for understanding language features in full generality

Reference Manuals

- Typically written in natural language
- Good for reading by humans
- Not executable, cannot be mechanically checked/tested
- Can contain omissions, contradictions, ambiguities
- Often verbose

Reference Manuals

- Example: C♯ conditionals[1]



8.7.1 The if statement

The if statement selects a statement for execution based on the value of a boolean expression.

```
if-statement:
    if ( boolean-expression ) embedded-statement
    if ( boolean-expression ) embedded-statement else embedded-statement
```

An else part is associated with the lexically nearest preceding if that is allowed by the syntax. Thus, an if statement of the form

```
if (x) if (y) F(); else G();
```

is equivalent to

```
if (x) {
    if (y) {
        F();
    }
    else {
        G();
    }
}
```

An if statement is executed as follows:

- The *boolean-expression* (§7.19) is evaluated.
- If the boolean expression yields true, control is transferred to the first embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the if statement.
- If the boolean expression yields false and if an else part is present, control is transferred to the second embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the if statement.
- If the boolean expression yields false and if an else part is not present, control is transferred to the end point of the if statement.

[1] C♯ Language Specification, v3.03, Microsoft, 2007.

Formal Mathematics

- Precise, concise, unambiguous

Formal Mathematics

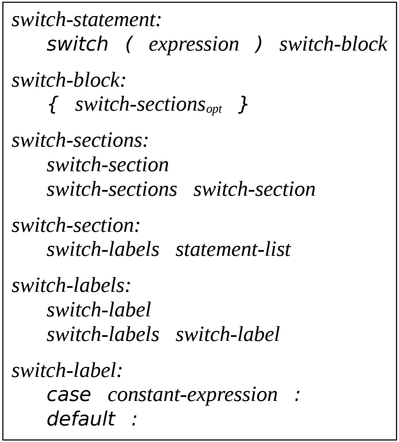
- Precise, concise, unambiguous
- Can be executable

Formal Mathematics

- Precise, concise, unambiguous
- Can be executable
- Commonly used to define syntax (e.g. context-free grammars)

Formal Mathematics

- Example: C$^\sharp$ switches[2]



```
switch-statement:
    switch ( expression ) switch-block

switch-block:
    { switch-sections_opt }

switch-sections:
    switch-section
    switch-sections switch-section

switch-section:
    switch-labels statement-list

switch-labels:
    switch-label
    switch-labels switch-label

switch-label:
    case constant-expression :
    default :
```

[2] C$^\sharp$ Language Specification, v3.03, Microsoft, 2007.

Formal Mathematics

- Precise, concise, unambiguous
- Can be executable
- Commonly used to define syntax (e.g. context-free grammars)

Formal Mathematics

- Precise, concise, unambiguous
- Can be executable
- Commonly used to define syntax (e.g. context-free grammars)
- Rarely used to define semantics (for major languages)

Formal Mathematics

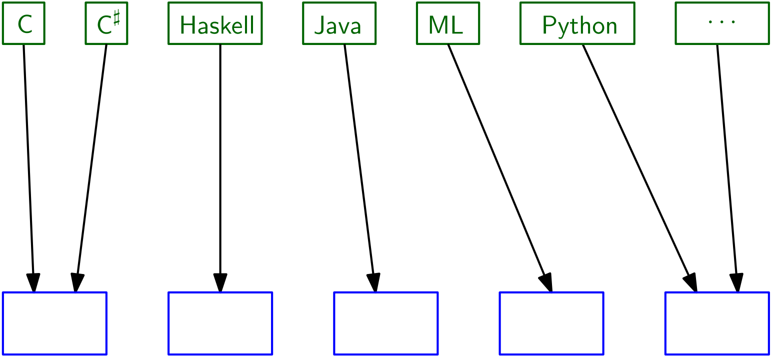
- Precise, concise, unambiguous
- Can be executable
- Commonly used to define syntax (e.g. context-free grammars)
- Rarely used to define semantics (for major languages)
- Formal semantics often considered too much effort to specify/maintain

Component-based Semantics

Component-based Semantics

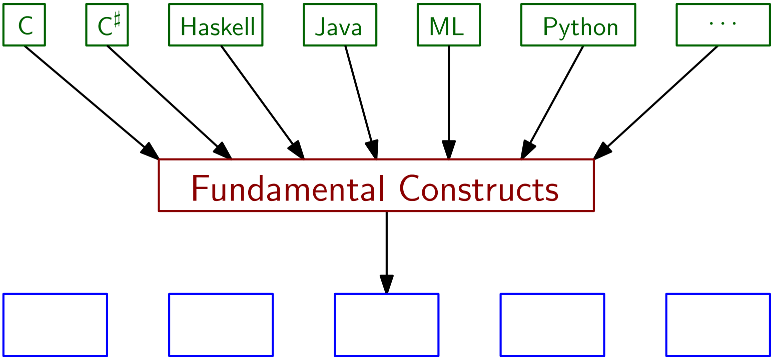
- Aim: Making formal semantics easier to specify and update
- Approach: A component-based framework of fundamental constructs

Component-based Semantics



Frameworks for Formal Semantics

Component-based Semantics



C

C$^\sharp$

Haskell

Java

ML

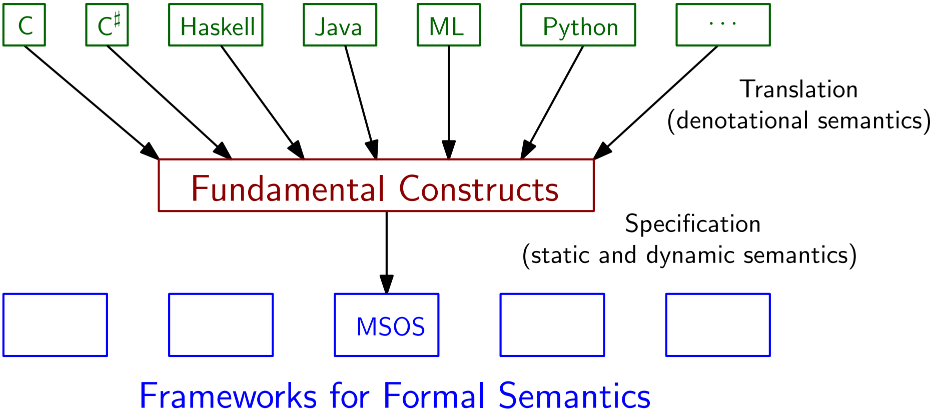
Python

$\cdots$

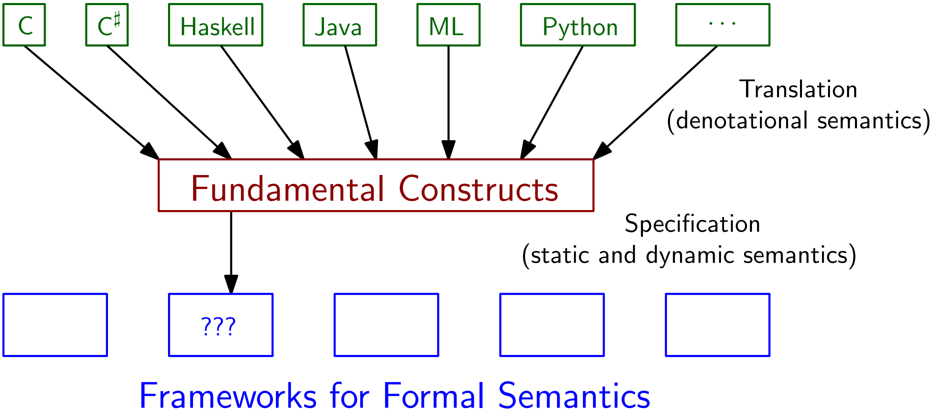
Fundamental Constructs

Frameworks for Formal Semantics

Component-based Semantics



Component-based Semantics



Fundamental Constructs (funcons)

- Each funcon expresses a programming concept, e.g.
  - variable assignment
  - function application
  - command sequencing
  - declaration scoping
  - conditional branching

- Funcons are similar to existing programming constructs to facilitate translation . . .

- . . . but general enough to be reusable for many languages.

An Open Collection of Reusable Funcons

. . . allocate apply assign bind booleans

call-cc catch closure curry deallocate

identifiers if-then-else is-equal integers lambda

lists pattern-match not null pointers print

records references scope sequential throw

types variables variants vectors while . . .

An Open Collection of Reusable Funcons

- The funcon framework is an open collection
- Each funcon:
  - is modular
  - has fixed syntax and semantics
- New funcons can be added, but existing funcons cannot be modified
- If a programming language changes, the translation to funcons changes

Funcon Formal Semantics

- Our framework: Modular Structural Operational Semantics
- Based on relations specified by inference rules

Funcon Formal Semantics

- Our framework: Modular Structural Operational Semantics
- Based on relations specified by inference rules
- E.g. specifying **if-then-else**:
  - Type Checking:

$$\frac{B : \textbf{booleans} \qquad X : T \qquad Y : T}{\textbf{if-then-else}(B, X, Y) : T}$$

  - Operational Semantics:

$$\frac{B \longrightarrow B'}{\textbf{if-then-else}(B, X, Y) \longrightarrow \textbf{if-then-else}(B', X, Y)}$$

$$\textbf{if-then-else}(\textbf{true}, X, Y) \longrightarrow X$$

$$\textbf{if-then-else}(\textbf{false}, X, Y) \longrightarrow Y$$

Example Translation (1): ML-like conditional expressions

- Source language syntax:

$$\mathbf{expr} ::= \textbf{if } \mathbf{expr} \textbf{ then } \mathbf{expr} \textbf{ else } \mathbf{expr}$$
$$\mid \ \ldots$$

Example Translation (1): ML-like conditional expressions

- Source language syntax:

$$\begin{array}{rcl} \mathsf{expr} & ::= & \textbf{if}\ \mathsf{expr}\ \textbf{then}\ \mathsf{expr}\ \textbf{else}\ \mathsf{expr} \\ & \mid & \ldots \end{array}$$

- Translation to funcons:

$$\begin{array}{l} \mathit{eval}[\![\textbf{if}\ E_1\ \textbf{then}\ E_2\ \textbf{else}\ E_3]\!] = \\ \qquad \textbf{if-then-else}(\mathit{eval}[\![E_1]\!], \mathit{eval}[\![E_2]\!], \mathit{eval}[\![E_3]\!]) \end{array}$$

Example Translation (2): C-like conditional statements

- Source language syntax:

$$\begin{aligned} \textbf{stmt} &::= \textbf{if} \ (\ \textbf{expr} \ ) \ \textbf{stmt} \ \textbf{else} \ \textbf{stmt} \\ &\mid \quad \ldots \end{aligned}$$

Example Translation (2): C-like conditional statements

- Source language syntax:

$$\begin{aligned}\textbf{stmt} &::= \textbf{if}\ (\ \textbf{expr}\ )\ \textbf{stmt}\ \textbf{else}\ \textbf{stmt} \\ &\mid\ \ldots\end{aligned}$$

- Translation to funcons:

$$\begin{aligned}&\mathit{exec}[\![\textbf{if}\ (\ E\ )\ S_1\ \textbf{else}\ S_2]\!] = \\ &\qquad \textbf{if-then-else}(\textbf{not}(\textbf{is-equal}(\mathit{eval}[\![E]\!], \textbf{0})), \mathit{exec}[\![S_1]\!], \mathit{exec}[\![S_2]\!])\end{aligned}$$

Case Studies

- IMP [vSM16]
- SIMPLE (under review)
- Caml Light [CMST15]
- C$^\sharp$ (work in progress)
- Control Operators [STM16]

Tool Support

- IDE as an Eclipse Plugin (using Spoofax)
- Translations from object languages to funcons are executable (using term rewriting)
- Funcon specifications compiled to produce a reference interpreter (using Haskell)

Summary

- Funcons are reusable semantic components
- Translation to funcons is simple and direct
- The funcon framework is open and modular
- Goal: provide a practical framework for formally specifying real-world programming languages

Publications

Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini.
Reusable components of semantic specifications.
In *Transactions on Aspect-Oriented Software Development XII*, volume 8989 of *Lecture Notes in Computer Science*, pages 132–179. Springer, 2015.

Peter D. Mosses and Ferdinand Vesely.
FunKons: Component-based semantics in K.
In *International Workshop on Rewriting Logic and its Applications*, volume 8663 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2014.

Neil Sculthorpe, Paolo Torrini, and Peter D. Mosses.
A modular structural operational semantics for delimited continuations.
In *Workshop on Continuations*, volume 212 of *Electronic Proceedings in Theoretical Computer Science*, pages 63–80. Open Publishing Association, 2016.

L. Thomas van Binsbergen, Neil Sculthorpe, and Peter D. Mosses.
Tool support for component-based semantics.
In *Companion Proceedings of the 15th International Conference on Modularity*, pages 8–11. ACM, 2016.