Principles of Programming Languages
Lecture 3: Semantics

Andrei Arusoaie[1]

[1] Department of Computer Science

October 17, 2017

Outline

Semantics: motivation

C

```
-$ cat test.c
int main()
{
  int x;
  return (x=1) + (x=2);
}
-$ gcc test.c
-$ ./a.out ; echo $?
 4
```

Java

```
-$ cat File.java
public class File {
  ... void main(...) {
    int x = 0;
    println((x=1) + (x=2));
  }
}
-$ javac File.java
-$ java File
 3
```

Semantics: motivation

C

```
-$ cat test.c
int main()
{
 int x;
 return (x=1) + (x=2);
}
-$ gcc test.c
-$ ./a.out ; echo $?
 4
```

Java

```
-$ cat File.java
public class File {
 ... void main(...) {
    int x = 0;
    println((x=1) + (x=2));
  }
}
-$ javac File.java
-$ java File
 3
```

Semantics: motivation

C

```
-$ cat test.c
int main()
{
 int x;
 return (x=1) + (x=2);
}
-$ gcc test.c
-$ ./a.out ; echo $?
 4
```

Java

```
-$ cat File.java
public class File {
 ... void main(...) {
    int x = 0;
    println((x=1) + (x=2));
   }
}
-$ javac File.java
-$ java File
 3
```

Semantics: motivation

GCC: 5.4.0-6 ubuntu

```
~$ cat test.c
int main()
{
 int x;
 return (x=1) + (x=2);
}
~$ gcc test.c
~$ ./a.out ; echo $?
 4
```

GCC: LLVM version 9.0.0 - clang

```
~$ cat test.c
int main()
{
 int x;
 return (x=1) + (x=2);
}
~$ gcc test.c
~$ ./a.out ; echo $?
 3
```

Semantics: motivation

GCC: 5.4.0-6 ubuntu

```
~$ cat test.c
int main()
{
 int x;
 return (x=1) + (x=2);
}
~$ gcc test.c
~$ ./a.out ; echo $?
 4
```

GCC: LLVM version 9.0.0 - clang

```
~$ cat test.c
int main()
{
 int x;
 return (x=1) + (x=2);
}
~$ gcc test.c
~$ ./a.out ; echo $?
 3
```

Semantics: motivation

GCC: 5.4.0-6 ubuntu

```
-$ cat test.c
int main()
{
 int x;
 return (x=1) + (x=2);
}
-$ gcc test.c
-$ ./a.out ; echo $?
 4
```

GCC: LLVM version 9.0.0 - clang

```
-$ cat test.c
int main()
{
 int x;
 return (x=1) + (x=2);
}
-$ gcc test.c
-$ ./a.out ; echo $?
 3
```

Semantics: motivation

GCC: 5.4.0-6 ubuntu

```
-$ cat test.c
int main()
{
 int x;
 return (x=1) + (x=2);
}
-$ gcc test.c
-$ ./a.out ; echo $?
 4
```

GCC: LLVM version 9.0.0 - clang

```
-$ cat test.c
int main()
{
 int x;
 return (x=1) + (x=2);
}
-$ gcc test.c
-$ ./a.out ; echo $?
 3
```

Demo

- `out-of-lifetime.c`

Semantics

- Semantics is concerned with the meaning of language constructs
- Semantics must be unambiguous
- Semantics must be flexible

Semantic - informal

Informal semantics (examples): natural language

Rationale for the ANSI C Programming Language:

- “Trust the programmer”
- “Don’t prevent the programmer from doing what needs to be done”
- “Keep the language small and simple”
- “Provide only one way to do an operation”
- “Make it fast, even if it is not guaranteed to be portable”

Inexact! – could lead to undefined behavior in programs

Semantic - informal

Informal semantics (examples): natural language

Rationale for the ANSI C Programming Language:

- “Trust the programmer”
- “Don’t prevent the programmer from doing what needs to be done”
- “Keep the language small and simple”
- “Provide only one way to do an operation”
- “Make it fast, even if it is not guaranteed to be portable”

Inexact! – could lead to undefined behavior in programs

Semantic - informal

Informal semantics (examples): natural language

Rationale for the ANSI C Programming Language:

- “Trust the programmer”
- “Don’t prevent the programmer from doing what needs to be done”
- “Keep the language small and simple”
- “Provide only one way to do an operation”
- “Make it fast, even if it is not guaranteed to be portable”

Inexact! – could lead to undefined behavior in programs

Semantic - informal

Informal semantics (examples): natural language

Rationale for the ANSI C Programming Language:

- "Trust the programmer"
- "Don't prevent the programmer from doing what needs to be done"
- "Keep the language small and simple"
- "Provide only one way to do an operation"
- "Make it fast, even if it is not guaranteed to be portable"

Inexact! – could lead to undefined behavior in programs

Semantic - informal

Informal semantics (examples): natural language

Rationale for the ANSI C Programming Language:

- “Trust the programmer”
- “Don’t prevent the programmer from doing what needs to be done”
- “Keep the language small and simple”
- “Provide only one way to do an operation”
- “Make it fast, even if it is not guaranteed to be portable”

Inexact! – could lead to undefined behavior in programs

Semantic - informal

Informal semantics (examples): natural language

Rationale for the ANSI C Programming Language:

- “Trust the programmer”
- “Don’t prevent the programmer from doing what needs to be done”
- “Keep the language small and simple”
- “Provide only one way to do an operation”
- “Make it fast, even if it is not guaranteed to be portable”

Inexact! – could lead to undefined behavior in programs

Semantic - informal

Informal semantics (examples): natural language

Rationale for the ANSI C Programming Language:

- “Trust the programmer”
- “Don’t prevent the programmer from doing what needs to be done”
- “Keep the language small and simple”
- “Provide only one way to do an operation”
- “Make it fast, even if it is not guaranteed to be portable”

Inexact! – could lead to undefined behavior in programs

Semantic styles

Some (formal) semantics styles:

- operational
- denotational
- axiomatic

We will focus more on operational semantics styles: K semantics, Small-step SOS, Big-Step SOS

Semantic styles

Some (formal) semantics styles:

- operational
- denotational
- axiomatic

We will focus more on operational semantics styles: K semantics, Small-step SOS, Big-Step SOS

A framework for defining PL semantics

A framework for defining semantics needs to be:

- expressive
- modular
- executable
- based on some "logic of programs" - enables reasoning

The K "machinery"

- We saw that K can be used to define syntax
- For semantics we have to understand the following key ingredients:
  - Komputations
  - Configurations
  - Rules

Configurations & Komputations & Rules

- We need a way to model program states
- K configurations:
  - structures of cells: <k> 2 + 3 + 5 </k>
- Komputations: units of calculus
  - The <k> cell is special: it contains the $PGM as a list of computations
  - Example: <k> 2 + 3 $\curvearrowright$ $\square$ + 5 <k>
  - $\curvearrowright$ is a separator for a KList
  - $\square$ is placeholder for a computation
- Your first K rule:
  - rule I_1 + I_2 => $I_1 +_{Int} I_2$
- DEMO!

Configurations & Komputations & Rules

- We need a way to model program states
- K configurations:
  - structures of cells: `<k> 2 + 3 + 5 </k>`
- Komputations: units of calculus
  - The `<k>` cell is special: it contains the $PGM as a list of computations
  - Example: `<k> 2 + 3` $\curvearrowright$ $\square$ `+ 5 <k>`
  - $\curvearrowright$ is a separator for a `KList`
  - $\square$ is placeholder for a computation
- Your first K rule:
  - `rule` I_1 `+` I_2 `=>` I_1 $+_{Int}$ I_2
- DEMO!

Configurations & Komputations & Rules

- We need a way to model program states
- K configurations:
  - structures of cells: <k> 2 + 3 + 5 </k>
- Komputations: units of calculus
  - The <k> cell is special: it contains the $PGM as a list of computations
  - Example: <k> 2 + 3 $\curvearrowright$ $\square$ + 5 <k>
  - $\curvearrowright$ is a separator for a KList
  - $\square$ is placeholder for a computation
- Your first K rule:
  - rule I_1 + I_2 => $I_1 +_{Int} I_2$
- DEMO!

Configurations & Komputations & Rules

- We need a way to model program states
- K configurations:
  - structures of cells: `<k> 2 + 3 + 5 </k>`
- Komputations: units of calculus
  - The `<k>` cell is special: it contains the $PGM as a list of computations
  - Example: `<k> 2 + 3` $\curvearrowright$ $\square$ `+ 5 <k>`
  - $\curvearrowright$ is a separator for a `KList`
  - $\square$ is placeholder for a computation
- Your first K rule:
  - `rule` I_1 `+` I_2 `=>` $I_1 +_{Int} I_2$
- DEMO!

Filling context in rules

- K rules establish transitions between configurations
- Here is how the above rule should look like:

$$\begin{array}{l} \texttt{rule}\ \ \texttt{<k>}\ I_1\ +\ I_2 \curvearrowright \texttt{K}\ \texttt{</k>} \\ \qquad\quad \texttt{=>} \\ \qquad\quad \texttt{<k>}\ I_1\ +_{Int}\ I_2 \curvearrowright \texttt{K}\ \texttt{</k>} \end{array}$$

- The K above is a variable and stands for other *komputations*
- The $+_{Int}$ is the mathematical addition over integers
- The K tool completes the context automatically

Filling context in rules

- K rules establish transitions between configurations
- Here is how the above rule should look like:

$$
\begin{aligned}
&\texttt{rule}\ \ \texttt{<k>}\ \ I_1\ +\ I_2\ \curvearrowright\ \texttt{K}\ \ \texttt{</k>}\\
&\qquad \texttt{=>}\\
&\qquad \texttt{<k>}\ \ I_1\ +_{Int}\ I_2\ \curvearrowright\ \texttt{K}\ \ \texttt{</k>}
\end{aligned}
$$

- The `K` above is a variable and stands for other *komputations*
- The $+_{Int}$ is the mathematical addition over integers
- The K tool completes the context automatically

Filling context in rules

- K rules establish transitions between configurations
- Here is how the above rule should look like:

$$
\begin{array}{l}
\texttt{rule}\ \texttt{<k>}\ I_1 \ \texttt{+}\ I_2 \curvearrowright \texttt{K}\ \texttt{</k>} \\
\qquad \texttt{=>} \\
\qquad \texttt{<k>}\ I_1 +_{Int} I_2 \curvearrowright \texttt{K}\ \texttt{</k>}
\end{array}
$$

- The K above is a variable and stands for other *komputations*
- The $+_{Int}$ is the mathematical addition over integers
- The K tool completes the context automatically

Filling context in rules

- K rules establish transitions between configurations
- Here is how the above rule should look like:

```
rule <k> I1 + I2 ↷ K </k>
       =>
     <k> I1 +Int I2 ↷ K </k>
```

- The K above is a variable and stands for other *komputations*
- The $+_{Int}$ is the mathematical addition over integers
- The K tool completes the context automatically

Evaluation. Heating and cooling.

- Consider the program: `2 + 3`
- When we apply `rule` I_1 `+` I_2 `=>` $I_1 +_{Int} I_2$ we get:
  - <k> 5 </k>
- But, for: `2 + 3 + 5`
- When we apply `rule` I_1 `+` I_2 `=>` $I_1 +_{Int} I_2$ we get:
  - <k> 2 + 3 + 5 </k>
- Solution: heating/cooling rules!
  - Explained on the blackboard!
  - strict
  - KResult

Evaluation. Heating and cooling.

- Consider the program: `2 + 3`
- When we apply `rule` I_1 `+` I_2 `=>` $I_1 +_{Int} I_2$ we get:
  - `<k> 5 </k>`
- But, for: `2 + 3 + 5`
- When we apply `rule` I_1 `+` I_2 `=>` $I_1 +_{Int} I_2$ we get:
  - `<k> 2 + 3 + 5 </k>`
- Solution: heating/cooling rules!
  - Explained on the blackboard!
  - strict
  - KResult

Evaluation. Heating and cooling.

- Consider the program: `2 + 3`
- When we apply `rule` I_1 `+` I_2 `=>` $I_1 +_{Int} I_2$ we get:
  - `<k> 5 </k>`
- But, for: `2 + 3 + 5`
- When we apply `rule` I_1 `+` I_2 `=>` $I_1 +_{Int} I_2$ we get:
  - `<k> 2 + 3 + 5 </k>`
- Solution: heating/cooling rules!
  - Explained on the blackboard!
  - strict
  - KResult

More complex configurations: the IMP configuration

- We need a way to model IMP program states. Why?
- Assignments require a state where variables are stored.
- We add a new cell called `<env>` to store variables and their values
- IMP configuration:
  - Example:

```
configuration <T>
                <k> x = 3; <k>
                <env> x |-> 0 <env>
              </T>
```

More complex configurations: the IMP configuration

- We need a way to model IMP program states. Why?
- Assignments require a state where variables are stored.
- We add a new cell called `<env>` to store variables and their values
- IMP configuration:
  - Example:

```
configuration <T>
                <k> x = 3; <k>
                <env> x |-> 0 <env>
              </T>
```

Rule for assignment

```
rule <T>
      <k> X = V; ↷ K </k>
      <env> X |-> _ </env>
     </T>
     =>
     <T>
      <k> K </k>
      <env> X |-> V </env>
     </T>
```

Rule for assignment

```
rule <T>
       <k> X = V; ↷ K </k>
       <env> X |-> _ </env>
     </T>
     =>
     <T>
       <k> K </k>
       <env> X |-> V </env>
     </T>
```

Rules at work

The K rule:

```
rule <T>
      <k> X = V; ↷ K </k>
      <env> X |-> _ </env>
     </T>
      =>
     <T>
      <k> K </k>
      <env> X |-> V </env>
     </T>
```

An example:

```
<T>
  <k> x = 2; ↷ y = 2; <k>
  <env> x |-> 0 <env>
</T>
=>
<T>
  <k> y = 2; <k>
  <env> x |-> 2 <env>
</T>
```

Rules at work

The K rule:

```
rule <T>
      <k> X = V; ↷ K </k>
      <env> X |-> _ </env>
     </T>
      =>
     <T>
      <k> K </k>
      <env> X |-> V </env>
     </T>
```

An example:

```
<T>
  <k> x = 2; ↷ y = 2; <k>
  <env> x |-> 0 <env>
</T>
=>
<T>
  <k> y = 2; <k>
  <env> x |-> 2 <env>
</T>
```

Rules at work

The K rule:

```
rule <T>
       <k> X = V; ↷ K </k>
       <env> X |-> _ </env>
     </T>
      =>
     <T>
       <k> K </k>
       <env> X |-> V </env>
     </T>
```

An example:

```
<T>
  <k> x = 2; ↷ y = 2; <k>
  <env> x |-> 0 <env>
</T>
=>
<T>
  <k> y = 2; <k>
  <env> x |-> 2 <env>
</T>
```

Abstractions

Configuration abstraction: write in rules only what is changing!

The K rule:

```
rule <T>
      <k> X = V; ↷ K </k>
      <env> X |-> _ </env>
     </T>
      =>
     <T>
      <k> K </k>
      <env> X |-> V </env>
     </T>
```

An example:

```
<T>
  <k> x = 2; ↷ y = 2; <k>
  <env> x |-> 0 <env>
</T>
=>
<T>
  <k> y = 2; <k>
  <env> x |-> 2 <env>
</T>
```

Abstractions

Configuration abstraction: write in rules only what is changing!

The K rule:

```
rule <T>
      <k> X = V; ↷ K </k>
      <env> X |-> _ </env>
     </T>
      =>
     <T>
      <k> K </k>
      <env> X |-> V </env>
     </T>
```

An example:

```
<T>
  <k> x = 2; ↷ y = 2; <k>
  <env> x |-> 0 <env>
</T>
=>
<T>
  <k> y = 2; <k>
  <env> x |-> 2 <env>
</T>
```

Abstractions

Configuration abstraction: write in rules only what is changing!

The K rule:

```
rule <T>
      <k> X = V; ↷ K </k>
      <env> X |-> _ </env>
     </T>
      =>
     <T>
      <k> K </k>
      <env> X |-> V </env>
     </T>
```

An example:

```
<T>
  <k> x = 2; ↷ y = 2; <k>
  <env> x |-> 0 <env>
</T>
=>
<T>
  <k> y = 2; <k>
  <env> x |-> 2 <env>
</T>
```

Abstractions

Configuration abstraction: write in rules only what is *changing*!

The K rule:

```
rule
      <k> X = V; ↷ K </k>
      <env> X |-> _ </env>

      =>

      <k> K </k>
      <env> X |-> V </env>
```

An example:

```
<k> x = 2; ↷ y = 2; <k>
<env> x |-> 0 <env>

=>

<k> y = 2; <k>
<env> x |-> 2 <env>
```

The K definition compiler fills the context for us!

Abstractions

Configuration abstraction: write in rules only what is *changing*!

The K rule:

```
rule
      <k> X = V; ↷ K </k>
      <env> X |-> _ </env>

      =>

      <k> K </k>
      <env> X |-> V </env>
```

An example:

```
<k> x = 2; ↷ y = 2; <k>
<env> x |-> 0 <env>

=>

<k> y = 2; <k>
<env> x |-> 2 <env>
```

The K definition compiler fills the context for us!

Abstractions

Local rewrites: *put the rewrite inside the cell!*

```
rule
   <k> X = V; ↷ K </k>
   <env> X |-> _ </env>
   =>
   <k> K </k>
   <env> X |-> V </env>
```

```
rule
   <k> (X = V; => .) ↷ K </k>
   <env> X |-> (_ => V)</env>
```

Abstractions

Local rewrites: *put the rewrite inside the cell!*

```
rule
   <k> X = V; ↷ K </k>
   <env> X |-> _ </env>
   =>
   <k> K </k>
   <env> X |-> V </env>
```

```
rule
   <k> (X = V; => .) ↷ K </k>
   <env> X |-> (_ => V)</env>
```

Abstractions

Local rewrites: *put the rewrite inside the cell!*

```
rule
   <k> X = V; ↷ K </k>
   <env> X |-> _ </env>
   =>
   <k> K </k>
   <env> X |-> V </env>
```

```
rule
   <k> (X = V; => .) ↷ K </k>
   <env> X |-> (_ => V)</env>
```

Abstractions

Local rewrites: *put the rewrite inside the cell!*

```
rule
   <k> X = V; ↷ K </k>
   <env> X |-> _ </env>
   =>
   <k> K </k>
   <env> X |-> V </env>
```

```
rule
   <k> (X = V; => .) ...</k>
   <env> X |-> (_ => V)</env>
```

Heating and cooling rules

- Recall:

```
rule
    <k> (X = V; => .) ...</k>
    <env> X |-> (_ => V)</env>
```

- This rules works fine when V is a result!
- Example: x = 2 + 2;
- If V is not a value (e.g., 2 + 2) then we *evaluate* it! How?
- Heating and cooling rules:
  - syntax Stmt ::= Id "=" Exp ";" [strict(2)]
  - Heating: 2 + 2 $\curvearrowright$ x = $\square$;
  - Compute result: 4 $\curvearrowright$ x = $\square$;
  - Cooling: x = 4;
  - Now we can apply the rule!

Heating and cooling rules

- Recall:

```
rule
   <k> (X = V; => .)  ...</k>
   <env> X |-> (_ => V)</env>
```

- This rules works fine when `V` is a result!
- Example: `x = 2 + 2;`
- If `V` is not a value (e.g., `2 + 2`) then we *evaluate* it! How?
- Heating and cooling rules:
  - `syntax Stmt ::= Id "=" Exp ";" [strict(2)]`
  - Heating: `2 + 2` $\curvearrowright$ `x = □;`
  - Compute result: `4` $\curvearrowright$ `x = □;`
  - Cooling: `x = 4;`
  - Now we can apply the rule!

Heating and cooling rules

- Recall:

```
rule
   <k> (X = V; => .)   ...</k>
   <env> X |-> (_ => V)</env>
```

- This rules works fine when `V` is a result!
- Example: `x = 2 + 2;`
- If `V` is not a value (e.g., `2 + 2`) then we *evaluate* it! How?
- Heating and cooling rules:
  - `syntax Stmt ::= Id "=" Exp ";" [strict(2)]`
  - Heating: `2 + 2` $\curvearrowright$ `x = `$\square$`;`
  - Compute result: `4` $\curvearrowright$ `x = `$\square$`;`
  - Cooling: `x = 4;`
  - Now we can apply the rule!

Heating and cooling rules

- Recall:

```
rule
   <k> (X = V; => .)   ...</k>
   <env> X |-> (_ => V)</env>
```

- This rules works fine when `V` is a result!
- Example: `x = 2 + 2;`
- If `V` is not a value (e.g., `2 + 2`) then we *evaluate* it! How?
- Heating and cooling rules:
  - `syntax Stmt ::= Id "=" Exp ";" [strict(2)]`
  - Heating: `2 + 2` $\curvearrowright$ `x =` $\square$`;`
  - Compute result: `4` $\curvearrowright$ `x =` $\square$`;`
  - Cooling: `x = 4;`
  - Now we can apply the rule!

Heating and cooling rules

- Recall:

```
rule
   <k> (X = V; => .)   ...</k>
   <env> X |-> (_ => V)</env>
```

- This rules works fine when `V` is a result!
- Example: `x = 2 + 2;`
- If `V` is not a value (e.g., `2 + 2`) then we *evaluate* it! How?
- Heating and cooling rules:
  - `syntax Stmt ::= Id "=" Exp ";" [strict(2)]`
  - Heating: `2 + 2` $\curvearrowright$ `x = ` $\square$`;`
  - Compute result: `4` $\curvearrowright$ `x = ` $\square$`;`
  - Cooling: `x = 4;`
  - Now we can apply the rule!

Heating and cooling rules

- Recall:

```
rule
   <k> (X = V; => .)   ...</k>
   <env> X |-> (_ => V)</env>
```

- This rules works fine when `V` is a result!
- Example: `x = 2 + 2;`
- If `V` is not a value (e.g., `2 + 2`) then we *evaluate* it! How?
- Heating and cooling rules:
  - `syntax Stmt ::= Id "=" Exp ";" [strict(2)]`
  - Heating: `2 + 2` $\curvearrowright$ `x = □;`
  - Compute result: `4` $\curvearrowright$ `x = □;`
  - Cooling: `x = 4;`
  - Now we can apply the rule!

Heating and cooling rules

- Recall:

```
rule
    <k> (X = V; => .)   ...</k>
    <env> X |-> (_ => V)</env>
```

- This rules works fine when `V` is a result!
- Example: `x = 2 + 2;`
- If `V` is not a value (e.g., `2 + 2`) then we *evaluate* it! How?
- Heating and cooling rules:
  - `syntax Stmt ::= Id "=" Exp ";" [strict(2)]`
  - Heating: `2 + 2` $\curvearrowright$ `x =` $\square$`;`
  - Compute result: `4` $\curvearrowright$ `x =` $\square$`;`
  - Cooling: `x = 4;`
  - Now we can apply the rule!

Heating and cooling rules

- Recall:

```
rule
   <k> (X = V; => .)  ...</k>
   <env> X |-> (_ => V)</env>
```

- This rules works fine when `V` is a result!
- Example: `x = 2 + 2;`
- If `V` is not a value (e.g., `2 + 2`) then we *evaluate* it! How?
- Heating and cooling rules:
  - `syntax Stmt ::= Id "=" Exp ";" [strict(2)]`
  - Heating: `2 + 2` $\curvearrowright$ `x =` $\square$`;`
  - Compute result: `4` $\curvearrowright$ `x =` $\square$`;`
  - Cooling: `x = 4;`
  - Now we can apply the rule!

Heating and cooling rules

- Recall:

```
rule
    <k> (X = V; => .)   ...</k>
    <env> X |-> (_ => V)</env>
```

- This rules works fine when `V` is a result!
- Example: `x = 2 + 2;`
- If `V` is not a value (e.g., `2 + 2`) then we *evaluate* it! How?
- Heating and cooling rules:
  - `syntax Stmt ::= Id "=" Exp ";" [strict(2)]`
  - Heating: `2 + 2` ↷ `x = □;`
  - Compute result: `4` ↷ `x = □;`
  - Cooling: `x = 4;`
  - Now we can apply the rule!

IMP

- We will define a simple imperative language in K
- Features:
  - Arithmetic and boolean expressions
  - Statements: assignments, decisional statement, loops, blocks, sequences
- DEMO

IMP

- We will define a simple imperative language in K
- Features:
  - Arithmetic and boolean expressions
  - Statements: assignments, decisional statement, loops, blocks, sequences
- `DEMO`

Lab - this week

- Extend IMP with various features

Bibliography

- Sections 2.5 and Chapter 6 from the [Gabbrielli&Martini 2010].