Variables and Substitution

2015/2016 1st Semester

CSIS0259 / COMP3259
Principles of Programming Languages

Resources

Lecture covers:

- Chapter 3 of "Anatomy of Programming Languages"

http://www.cs.utexas.edu/~wcook/anatomy/anatomy.htm

Variable Discussion

What is meant by a Variable?

- Discussing variables in programming languages can be confusing. **What's the difference between?**

π

math (or Haskell)

f(x) = x * x

Java or C

int x = 0; x = x+1;

What is meant by a Variable?

- π is a constant

Math/FP
- f(x) = x * x   here x is a (immutable) variable

Java/C
- int x = 0; x = x+1;   here x is a (mutable) variable

What is meant by a Variable?

- constant: The value of a constant is always the same in any context.

- (immutable) variable: In a particular definition the value of the variable never changes. However the value of the variable can be vary in different contexts.

  - Example: different calls f(2), f(3) use different values for the argument x of f, but x never changes in the definition of f.

- (mutable) variable: The value of a variable can change even in a particular definition.

Immutable variables in Java

- Although many languages have mutable variables by default, some languages allow immutable variables as well:

```
int f(final int x) {
  return x * x;
}

int g(final int x) {
  x = x * x;
  return x;
}
```

This is a compile-time error!

Meaning of variable the course

- By default, when referring to variable we mean (immutable) variable

- We will use the term mutable variable later in the course, when we talk about imperative programming.

Arithmetic Expressions with Variables

Arithmetic Expressions with Variables

- A simple extension to the language of arithmetic is to allow variables:

  $x + x$ where $x = 5$

- This can be useful to avoid repetition and reuse expressions

Bindings

- We call the pair constituted of a variable and the associated value a binding.

binding

x + x where x = 5

- We will use the following notation to denote binding:

  variable $\mapsto$ value   (Example: $x \mapsto 5$)

- Bindings can be represented in Haskell as a pair. For example ("x",5).

Substitution

- Substitution replaces a variable in with a value in an expression. For example:
  - substitute $x \mapsto 5$ in $x + 2 \longrightarrow 5 + 2$
  - substitute $x \mapsto 5$ in $2 \longrightarrow 2$
  - substitute $x \mapsto 5$ in $x \longrightarrow 5$
  - substitute $x \mapsto 5$ in $x * x + x \longrightarrow 5 * 5 + 5$
  - substitute $x \mapsto 5$ in $x + y \longrightarrow 5 + y$

Implementing Substitution

Arithmetic Expressions with Variables

- To allow this extension we first need to change the abstract syntax:

$$
\begin{aligned}
\textbf{data}\ Exp = {} & Number\ Int \\
& \mid Add \qquad\qquad\quad Exp\ Exp \\
& \mid Subtract\ Exp\ Exp \\
& \mid Multiply\ Exp\ Exp \\
& \mid Divide \qquad\quad\ \ Exp\ Exp \\
& \mid Variable\ String{-}-added \\
& \textbf{deriving}\ (Eq)
\end{aligned}
$$

Demo:
Implementing Substitution

Substitution

- The substitution operation can be defined in Haskell as:

$$
\begin{array}{l}
\mathit{substitute1} :: (\mathit{String}, \mathit{Int}) \to \mathit{Exp} \to \mathit{Exp} \\
\mathit{substitute1}\ (\mathit{var}, \mathit{val})\ \mathit{exp} = \mathit{subst}\ \mathit{exp}\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{subst}\ (\mathit{Number}\ i) & = \mathit{Number}\ i \\
\mathit{subst}\ (\mathit{Add}\ a\ b) & = \mathit{Add}\ (\mathit{subst}\ a)\ (\mathit{subst}\ b) \\
\mathit{subst}\ (\mathit{Subtract}\ a\ b) & = \mathit{Subtract}\ (\mathit{subst}\ a)\ (\mathit{subst}\ b) \\
\mathit{subst}\ (\mathit{Multiply}\ a\ b) & = \mathit{Multiply}\ (\mathit{subst}\ a)\ (\mathit{subst}\ b) \\
\mathit{subst}\ (\mathit{Divide}\ a\ b) & = \mathit{Divide}\ (\mathit{subst}\ a)\ (\mathit{subst}\ b)
\end{array} \\
\quad \mathit{subst}\ (\mathit{Variable}\ \mathit{name}) = \textbf{if}\ \mathit{var} \equiv \mathit{name} \\
\qquad \textbf{then}\ \mathit{Number}\ \mathit{val} \\
\qquad \textbf{else}\ \mathit{Variable}\ \mathit{name}
\end{array}
$$

Using Substitution

- The substitution function can be used as follows

$$substitute\ (\texttt{"x"}, 5)\ [x + 2]$$
$$==> [5 + 2]$$

pseudo-code. Real code is:
Add (Variable “x”) (Number 2)

$$substitute\ (\texttt{"x"}, 5)\ [32]$$
$$==> [32]$$
$$substitute\ (\texttt{"x"}, 5)\ [x]$$
$$==> [5]$$
$$substitute\ (\texttt{"x"}, 5)\ [x * x + x]$$
$$==> [5 * 5 + 5]$$
$$substitute\ (\texttt{"x"}, 5)\ [x + 2 * y + z]$$
$$==> [5 + 2 * y + z]$$

Using Substitution

The substitution function can be used as follows

```
substitute ("x",4) (parseExp "x+5")
```

or even

```
substituteExp ("x",4) "x+5"
```

Real code!

Multiple Substitution

Environments

- Arithmetic expressions can have multiple variables. For example:

  2 * x + y where x = 3 and y = -2

- A collection of bindings is called an environment.

- In Haskell we can represent an environment as a list of bindings.

$$\mathbf{type}\ Env = [(String, Int)]$$

Some Preliminaries

An important operation on environments is variable lookup:

lookup “x” [(“y”,5),(“x”,6)] ===> 6

lookup is an operation that given a variable name and an environment returns the value bound to that variable in the environment.

Some Preliminaries

lookup can fail if the variable being looked up does not exist in the environment:

lookup “x” [(“y”,5),(“z”,6)] ===> ????

Thus lookup needs to account for this possibility of failure.

It is important that we know when lookup fails. How can we do that?

Lookup in Haskell

In Haskell there's already a lookup function:

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

The Maybe datatype is used when some exceptional value is needed:

data Maybe a = Nothing | Just a

When lookup succeeds finding a value v in the list/environment, it will return "Just v". When it fails, an exceptional value is needed, so "Nothing" is returned:

lookup "x" [("y",5),("x",6)] ===> Just 6

lookup "x" [("y",5),("z",6)] ===> Nothing

Demo:
Implementing Multiple Substitution

Multiple Substitution

- Multiple substitution allows us to simultaneously replace many variables at once using an environment.

$$
\begin{array}{l}
substitute :: Env \to Exp \to Exp \\
substitute\ env\ exp = subst\ exp\ \mathbf{where} \\
\quad subst\ (Number\ i) \qquad = Number\ i \\
\quad subst\ (Add\ a\ b) \qquad = Add\ (subst\ a)\ (subst\ b) \\
\quad subst\ (Subtract\ a\ b) = Subtract\ (subst\ a)\ (subst\ b) \\
\quad subst\ (Multiply\ a\ b) = Multiply\ (subst\ a)\ (subst\ b) \\
\quad subst\ (Divide\ a\ b) \quad = Divide\ (subst\ a)\ (subst\ b) \\
\quad subst\ (Variable\ name) = \\
\qquad \mathbf{case}\ lookup\ name\ env\ \mathbf{of} \\
\qquad\quad Just\ val \to Number\ val \\
\qquad\quad Nothing \to Variable\ name
\end{array}
$$

lookup the variable called name in the environment

Multiple Substitution

- In multiple substitution the interesting case happens when we deal with variables.

  - If the variable being looked up exists in the environment then we should substitute it by the corresponding value.

substitute [(“y”,6),(“x”,5)] (Variable “x”) ===> Number 5

  - Otherwise we should just return the same expression

substitute [(“y”,6),(“x”,5)] (Variable “z”) ===> Variable “z”

Multiple Substitution

- Using multiple substitution

$$e1 = [(\texttt{"x"}, 3), (\texttt{"y"}, -1)]$$

$$substitute\ e1\ [x + 2]$$
$$==> [3 + 2]$$
$$substitute\ e1\ [32]$$
$$==> [32]$$
$$substitute\ e1\ [x]$$
$$==> [3]$$
$$substitute\ e1\ [x * x + x]$$
$$==> [3 * 3 + 3]$$
$$substitute\ e1\ [x + 2 * y + z]$$
$$==> [3 + 2 * -1 + z]$$

Evaluation using Environments

Evaluation

- To deal with variables, evaluation can be modified to take an environment.

```
eval :: Env -> Exp -> Int
```

- This way it is possible to lookup the value of variables and deal with the new Variable case.

Demo:
Implementing Evaluation with Environments

Next Extension:
Local Variables

Local Variables

- So far variables are defined outside the language.

- Local variables allow the definition of variables inside the language.

```
int x = 3;
return 2 * x + 5;
```

C or Java code

```
var x = 3;
return 2 * x + 5;
```

JavaScript

$$\mathbf{let}\ x = 3\ \mathbf{in}\ 2 * x + 5$$

Haskell

Multiple Local Variables

- There can be multiple local variables.

```
int x = 3;
int y = x * 2;
return x + y;
```

C or Java code

$$\textbf{let } x = 3 \textbf{ in let } y = x * 2 \textbf{ in } x + y$$

Haskell

Shadowing

- Multiple variables introduce an interesting issue: there can be multiple local variables with the same name!

- What should happen in this situation?

Shadowing in C

What is the output of the following C programs? Is the output the same?

```c
#include <stdio.h>

int main(void)
{
  int x = 0;

  if (x == 0) {
    int x = 1;
    printf("Inside if x is: %d\n",x);
  }

  printf("Here x is: %d\n", x);
  return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
  int x = 0;

  if (x == 0) {
    x = 1;
    printf("Inside if x is: %d\n",x);
  }

  printf("Here x is: %d\n", x);
  return 0;
}
```

Shadowing in C

This program declares two variables called x. Each having different values.

```c
#include <stdio.h>

int main(void)
{
  int x = 0;

  if (x == 0) {
    int x = 1;
    printf("Inside if x is: %d\n",x);
  }

  printf("Here x is: %d\n", x);
  return 0;
}
```

This program declares one variable called x and it mutates its value.

```c
#include <stdio.h>

int main(void)
{
  int x = 0;

  if (x == 0) {
    x = 1;
    printf("Inside if x is: %d\n",x);
  }

  printf("Here x is: %d\n", x);
  return 0;
}
```

Shadowing in C

```c
#include <stdio.h>

int main(void)
{
  int x = 0;

  if (x == 0) {
    int x = 1;
    printf("Inside if x is: %d\n",x);
  }

  printf("Here x is: %d\n", x);
  return 0;
}
```

this variable declaration shadows the previous definition of x inside the if block

here the value of the most local variable called x is used

Shadowing in Haskell

- We can create a similar program in Haskell to illustrate variable shadowing

```
Prelude> let x = 0 in (let x = 1 in x, x)
(1,0)
```

In the past students asked me whether let expressions in Haskell are not the same as mutation. The answer is no. What is happening here is shadowing, not mutation!

Shadowing in General

- Nearly every language has some form of shadowing
- It is important for programmers to be aware of shadowing and its semantics as it can often give rise to subtle bugs
- Some languages forbid certain types of shadowing (though usually not all)
- Generally speaking it is better to avoid shadowing (although there are some use cases for it)