Typing, State, Parameter Passing

Dynamic and Static Typing (EPL 4.1-4, VRH 2.8.3)
Explicit State and Parameter Passing(VRH 6.1-6.4.4)

Carlos Varela
RPI
October 16, 2006

Partially adapted with permission from:
Seif Haridi
KTH
Peter Van Roy
UCL

Data types

- A datatype is a set of values and an associated set of operations
- An abstract datatype is described by a set of operations
- These operations are the only thing that a user of the abstraction can assume
- Examples:
  - Numbers, Records, Lists,… (Oz basic data types)
  - Stacks, Dictionaries,… (user-defined secure data types)

Types of typing

- Languages can be ***weakly typed***
  - Internal representation of types can be manipulated by a program
    - e.g., a string in C is an array of characters ending in '\0'.
- ***Strongly typed*** programming languages can be further subdivided into:
  - ***Dynamically typed*** languages
    - Variables can be bound to entities of any type, so in general the type is only known at **run-time**, e.g., Oz, SALSA.
  - ***Statically typed*** languages
    - Variable types are known at **compile-time**, e.g., C++, Java.

Type Checking and Inference

- *Type checking* is the process of ensuring a program is well-typed.
  - One strategy often used is *abstract interpretation:*
    - The principle of getting partial information about the answers from partial information about the inputs
    - Programmer supplies types of variables and type-checker deduces types of other expressions for consistency
- *Type inference* frees programmers from annotating variable types: types are inferred from variable usage, e.g. ML.

Example: The identity function

- In a dynamically typed language, e.g., Oz, it is possible to write a generic function, such as the identity combinator:

```
fun {Id X} X end
```

- In a statically typed language, it is necessary to assign types to variables, e.g. in a statically typed variant of Oz you would write:

```
fun {Id X:integer}:integer X end
```

These types are checked at compile-time to ensure the function is only passed proper arguments. `{Id 5}` is valid, while `{Id Id}` is not.

Example: Improper Operations

- In a dynamically typed language, it is possible to write an improper operation, such as passing a non-list as a parameter, e.g. in Oz:

```
declare fun {ShiftRight L}  0|L end
{Browse {ShiftRight 4}}        % unintended missuse
{Browse {ShiftRight [4]}}      % proper use
```

- In a statically typed language, the same code would produce a type error, e.g. in a statically typed variant of Oz you would write:

```
declare fun {ShiftRight L:List}:List  0|L end
{Browse {ShiftRight 4}}        % compiler error!!
{Browse {ShiftRight [4]}}      % proper use
```

Example: Type Inference

- In a statically typed language with type inference (e.g., ML), it is possible to write code without type annotations, e.g. using Oz syntax:

```
declare fun {Increment N} N+1 end
{Browse {Increment [4]}}      % compiler error!!
{Browse {Increment 4}}        % proper use
```

- The type inference system knows the type of **'+'** to be:

  <number> X **<number>** → **<number>**

  Therefore, **Increment** must always receive an argument of type **<number>** and it always returns a value of type **<number>.**

Static Typing Advantages

- Static typing restricts valid programs (i.e., reduces language's expressiveness) in return for:
  - Improving error-catching ability
  - Efficiency
  - Security
  - Partial program verification

Dynamic Typing Advantages

- Dynamic typing allows all syntactically legal programs to execute, providing for:
  - Faster prototyping (partial, incomplete programs can be tested)
  - Separate compilation---independently written modules can more easily interact--- which enables open software development
  - More expressiveness in language

Combining static and dynamic typing

- Programming language designers do not have to make an *all-or-nothing* decision on static vs dynamic typing.
  - e.g, Java has a root **Object** class which enables *polymorphism*
    - A variable declared to be an **Object** can hold an instance of any (non-primitive) class.
    - To enable static type-checking, programmers need to annotate expressions using these variables with *casting* operations, i.e., they instruct the type checker to pretend the type of the variable is different (more specific) than declared.
    - Run-time errors/exceptions can then occur if type conversion (casting) fails.
- Alice (Saarland U.) is a statically-typed variant of Oz.

What is state?

- State is a sequence of values in time that contains the intermediate results of a desired computation
- Declarative programs can also have state according to this definition
- Consider the following program

```
fun {Sum Xs A}
  case Xs
  of X|Xr then {Sum Xr A+X}
  [] nil then A
  end
end

{Browse {Sum [1 2 3 4] 0}}
```


What is implicit state?

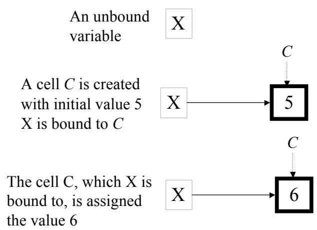
The two arguments Xs and A represent an implicit state

Xs	A
[1 2 3 4]	0
[2 3 4]	1
[3 4]	3
[4]	6
nil	10

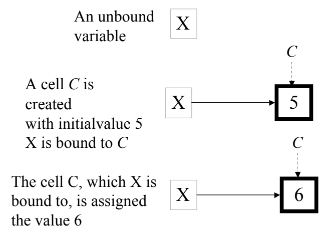
```
fun {Sum Xs A}
  case Xs
  of X|Xr then {Sum Xr A+X}
  [] nil then A
  end
end

{Browse {Sum [1 2 3 4] 0}}
```

What is explicit state: Example?



What is explicit state: Example?



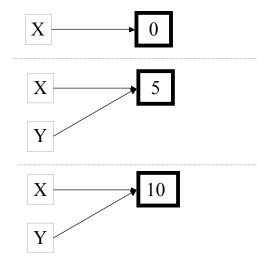
- The cell is a value container with a unique **identity**
- X is really bound to the **identity** of the cell
- When the cell is assigned, X does not change

What is explicit state?

- **X = {NewCell I}**
  - Creates a cell with initial value I
  - Binds X to the identity of the cell
- Example: X = {NewCell 0}
- **{Assign X J}**
  - Assumes X is bound to a cell C (otherwise exception)
  - Changes the content of C to become J
- **Y = {Access X}**
  - Assumes X is bound to a cell C (otherwise exception)
  - Binds Y to the value contained in C

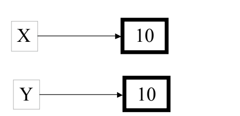
Examples

- X = {NewCell 0}
- {Assign X 5}
- Y = X
- {Assign Y 10}
- {Access X} == 10 % returns true
- X == Y % returns true

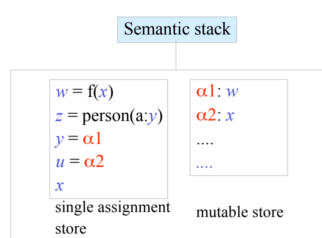


Examples

- X = {NewCell 10}
  Y = {NewCell 10}
- X == Y % returns false
- Because X and Y refer to different cells, with different identities
- {Access X} == {Access Y} returns true



The model extended with cells



The stateful model

⟨s⟩::= **skip** — *empty statement*
| ⟨s₁⟩ ⟨s₂⟩ — *statement sequence*
| ...
| {NewCell ⟨x⟩ ⟨c⟩} — ***cell creation***
| {Exchange ⟨c⟩ ⟨x⟩ ⟨y⟩} — ***cell exchange***

Exchange: bind ⟨x⟩ to the old content of ⟨c⟩ and set the content of the cell ⟨c⟩ to ⟨y⟩

The stateful model

| {NewCell ⟨x⟩ ⟨c⟩} — ***cell creation***
| {Exchange ⟨c⟩ ⟨x⟩ ⟨y⟩} — ***cell exchange***

Exchange: bind ⟨x⟩ to the old content of ⟨c⟩ and set the content of the cell ⟨c⟩ to ⟨y⟩

```
proc {Assign C X} {Exchange C _ X} end
fun {Access C} X in{Exchange C X X}X end
```

C := X is syntactic sugar for **{Assign C X}**
@C is syntactic sugar for **{Access C}**
X=C:=Y is syntactic sugar for **{Exchange C X Y}**

Abstract data types (revisited)

- For a given functionality, there are many ways to package the ADT. We distinguish three axes.
- Open vs. secure ADT: is the internal representation visible to the program or hidden?
- Declarative vs. stateful ADT: does the ADT have encapsulated state or not?
- Bundled vs. unbundled ADT: is the data kept together from the operations or is it separable?
- Let us see what our stack ADT looks like with some of these possibilities

Stack: Open, declarative, and unbundled

- Here is the basic stack, as we saw it before:

```
fun {NewStack} nil end
fun {Push S E} E|S end
fun {Pop S E} case S of X|S1 then E=X  S1 end end
fun {IsEmpty S} S==nil end
```

- This is completely unprotected. Where is it useful? Primarily, in small programs in which expressiveness is more important than security.

Stack: *Secure*, declarative, and unbundled

- We can make the declarative stack secure by using a wrapper:

```
local Wrap Unwrap
in
    {NewWrapper Wrap Unwrap}
    fun {NewStack} {Wrap nil} end
    fun {Push S E} {Wrap E|{Unwrap S}} end
    fun {Pop S E} case {Unwrap S} of X|S1 then E=X {Wrap S1} end end
    fun {IsEmpty S} {Unwrap S} ==nil end
end
```

- Where is this useful? In large programs where we want to protect the implementation of a declarative component.

Stack: Secure, stateful, and *unbundled*

- Let us combine the wrapper with state:

```
local Wrap Unwrap
in
    {NewWrapper Wrap Unwrap}
    fun {NewStack} {Wrap {NewCell nil}} end
    proc {Push W X} C={Unwrap W} in {Assign C X|{Access C}} end
    fun {Pop W} C={Unwrap W} in
        case {Access C} of X|S then {Assign C S} X end
    end
    fun {IsEmpty S} {Access {Unwrap W}}==nil end
end
```

- This version is stateful but lets us store the stack separate from the operations. The same operations work on all stacks.

Stack: Secure, *stateful*, and bundled

- This is the simplest way to make a secure stateful stack:

```
proc {NewStack ?Push ?Pop ?IsEmpty}
     C={NewCell nil}
in
     proc {Push X} {Assign C X|{Access C}} end
     fun {Pop} case {Access C} of X|S then {Assign C S} X end end
     fun {IsEmpty} {Access C} ==nil end
end
```

- Compare the declarative with the stateful versions: the declarative version needs two arguments per operation, the stateful version uses higher-order programming (instantiation)
- With some syntactic support, this is *object-based programming*

Four ways to package a stack

- Open, declarative, and unbundled: the usual declarative style, e.g., in Prolog and Scheme
- Secure, declarative, and unbundled: use wrappers to make the declarative style secure
- Secure, stateful, and unbundled: an interesting variation on the usual object-oriented style
- Secure, stateful, and bundled: the usual object-oriented style, e.g., in Smalltalk and Java
- Other possibilities: there are four more possibilities! **Exercise**: Try to write all of them.

Parameter Passing Mechanisms

- Operations on data types have arguments and results. Many mechanisms exist to pass these arguments and results between calling programs and abstractions, e.g.:
  - Call by reference
  - Call by variable
  - Call by value
  - Call by value-result
  - Call by name
  - Call by need
- We will show examples in Pascal-like syntax, with semantics given in Oz language.

Call by reference

```
procedure sqr(a:integer, var b:integer);
begin
   b:=a*a
end

var i:integer;
sqr(25, i);
writeln(i);
```

```
proc {Sqr A ?B}
   B=A*A
end

local I in
   {Sqr 25 I}
   {Browse I}
end
```

- The variable passed as an argument can be changed inside the procedure with visible effects outside after the call.
- The **B** inside **Sqr** is a synonym (an *alias*) of the **I** outside.
- The default mechanism in Oz is *call by reference*.

Call by variable

```
procedure sqr(var a:integer);
begin
   a:=a*a
end

var i:integer;
i:=25;
sqr(i);
writeln(i);
```

```
proc {Sqr A}
   A:=@A*@A
end

local I = {NewCell 0} in
   I := 25
   {Sqr I}
   {Browse @I}
end
```

- Special case of *call by reference*.
- The identity of the cell is passed to the procedure.
- The **A** inside **Sqr** is a synonym (an alias) of the **I** outside.

Call by value

```
procedure sqr(a:integer);
begin
   a:=a+1;
   writeln(a*a)
end

sqr(25);
```

```
proc {Sqr A}
   C = {NewCell A}
in
   C := @C + 1
   {Browse @C*@C}
end

{Sqr 25}
```

- A value is passed to the procedure. Any changes to the value inside the procedure are purely local, and therefore, **not** visible outside.
- The local cell **C** is initialized with the argument **A** of **Sqr**.
- Java uses call by value for both primitive values and object references.
- SALSA uses call by value in both local and remote message sending.

Call by value-result

```
procedure sqr(inout a:integer);
begin
   a:=a*a
end

var i:integer;
i:=25;
sqr(i);
writeln(i);
```

```
proc {Sqr A}
   D = {NewCell @A}
in
   D := @D * @D
   A := @D
end
local C = {NewCell 0} in
   C := 25
   {Sqr C}
   {Browse @C}
end
```

- A modification of call by variable. Variable argument can be modified.
- There are two mutable variables: one inside **Sqr** (namely **D**) and one outside (namely **C**). Any *intermediate* changes to the variable inside the procedure are purely local, and therefore, **not** visible outside.
- inout is ADA terminology.

Call by name

```
procedure sqr(callbyname a:integer);
begin
   a:=a*a
end

var i:integer;
i:=25;
sqr(i);
writeln(i);
```

```
proc {Sqr A}
   {A} := @{A} * @{A}
end
local C = {NewCell 0} in
   C := 25
   {Sqr fun {$} C end}
   {Browse @C}
end
```

- Call by name creates a function for each argument (a *thunk*). Calling the function evaluates and returns the argument. Each time the argument is needed inside the procedure, the thunk is called.
- Thunks were originally invented for Algol 60.

Call by need

```
procedure sqr(callbyneed a:integer);
begin
   a:=a*a
end

var i:integer;
i:=25;
sqr(i);
writeln(i);
```

```
proc {Sqr A}
   B = {A}  % only if argument used!!
in
   B := @B * @B
end

local C = {NewCell 0} in
   C := 25
   {Sqr fun {$} C end}
   {Browse @C}
end
```

- A modification of *call by name*. The thunk is evaluated **at most** once. The result is stored and used for subsequent evaluations.
- *Call by need* is the same as lazy evaluation. Haskell uses lazy evaluation.
- *Call by name* is lazy evaluation without memoization.

Which one is *right* or *best*?

- It can be argued that *call by reference* is the most primitive.
  - Indeed, we have coded different parameter passing styles using *call by reference* and a combination of cells and procedure values.
  - Arguably, *call by value* (along with cells and procedure values) is just as general. E.g., the example given for *call by variable* would also work in a *call by value* primitive mode. Exercise: Why?
- When designing a language, the question is: for which mechanism(s) to provide linguistic abstractions?
  - It largely depends on intended language use, e.g., *call by name* and *call by need* are integral to programming languages with lazy evaluation (e.g., Haskell and Miranda.)
  - For concurrent programs, *call by value-result* can be very useful (e.g. Ada.)
  - For distributed programs, *call by value* is best due to state encapsulation (e.g., SALSA).

More parameter passing styles

- Some languages for distributed computing have support for ***call-by-move***.
  - Arguments to remote procedure calls are temporarily migrated to the remote location for the time of the remote procedure execution (e.g., Emerald).
  - A dual approach is to migrate the object whose method is to be invoked to the client side before method invocation (e.g., Oz).
- Java Remote Method Invocation (RMI) dynamically determines mechanism to use depending on argument types:
  - It uses *call by reference* in remote procedure calls, if and only if, arguments implement a special (**Remote**) interface
  - Otherwise, arguments are passed using *call by value*.
    - => Semantics of method invocation is different for local and remote method invocations!!
  - There is no language support for object migration in Java (as there is in other languages, e.g., SALSA, Oz, Emerald), so *call by move* is not possible.

Exercises

51. VRH Exercise 6.10.2 (page 482).
52. Explain why the *call by variable* example given would also work over a *call by value* primitive parameter passing mechanism. Give an example for which this is not the case.
53. Explain why *call by need* cannot always be encoded as shown in the given example by producing a counter-example. (Hint: recall the difference between normal order evaluation and applicative order evaluation in termination of lambda calculus expression evaluations.)
54. Create a program in which *call by name* and *call by need* parameter passing styles result in different outputs.
55. *Can type inference always deduce the type of an expression?
    - If not, give a counter-example. How would you design a language to help it statically infer types for non-trivial expressions?