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:
  ```oz
  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:
  ```oz
  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 1d]` 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:
  ```oz
  declare fun [ShiftRight L] 0 L end
  [Browse [ShiftRight 4]] % unintended misuse
  [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:
  ```oz
  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:
  ```oz
  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:
  - Faster prototyping (partial, incomplete programs can be tested)
  - Separate compilation—individually 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
  ```oz
  fun {Sum Xs A}
  case Xs
  | 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
  ```oz
  Xs = [1 2 3 4]
  A = 0
  {Sum Xs A}
  ```
  ```oz
  Xs = [1 2 3 4]
  A = 0
  {Sum Xs A}
  ```

- The following program:
  ```oz
  fun {Sum Xs A}
  case Xs
  | [Xr] then {Sum Xr A+X}
  | [] nil then A
  end
  end
  {Browse (Sum [1 2 3 4] 0)}
  ```
What is explicit state?

• X = \{NewCell 1\}
  – Creates a cell with initial value 1
  – 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 10\}
  Y = \{NewCell 10\}
• X \rightarrow Y \% returns false
  Because X and Y refer to different cells, with different identities
• \{Access X\} \rightarrow \{Access Y\} returns true

The model extended with cells

Semantic stack

\[w = f(x)\]
\[z = \text{person}(a,y)\]
\[y = \alpha_1\]
\[u = \alpha_2\]
\[
\text{single assignment store}
\]
\[
\text{mutable store}
\]
The stateful model

\[ \{s\} ::= \text{skip} \quad \text{empty statement} \]
\[ \{s_1, s_2\} \quad \text{statement sequence} \]
\[ \{\text{NewCell}(\alpha)\} \quad \text{cell creation} \]
\[ \{\text{Exchange}(\alpha)(\beta)\} \quad \text{cell exchange} \]

Exchange: bind \(\alpha\) to the old content of \(\beta\) and set the content of the cell \(\alpha\) to \(\beta\)

---

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 \(\text{NewStack}\) nil end
  - fun \(\text{Push}\) \(S\ E\) \(E|S\) end
  - fun \(\text{Pop}\) \(S\) \((E \mapsto S)\) end
  - fun \(\text{isEmpty}\) \(S\) \(\text{nil}\) 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 \text{Wrap Unwrap}
    in
    \(\text{NewWrapper} \text{ Wrap Unwrap}\)
    fun \(\text{NewStack}\) \(\text{Wrap} \text{ nil}\) end
    fun \(\text{Push}\) \(S\ E\) \(\text{Wrap} E\) \(\text{Unwrap} S\) end
    fun \(\text{Pop}\) \(S\) \(\text{case}\) \(\text{Unwrap}\) \(\text{S}\) \(\text{of}\) \(X|S\) \(\text{then}\) \(E\mapsto X\) \(\text{Wrap} S\) end
    fun \(\text{isEmpty}\) \(S\) \(\text{nil}\) 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 \text{Wrap Unwrap}
    in
    \(\text{NewWrapper} \text{ Wrap Unwrap}\)
    fun \(\text{NewStack}\) \(\text{Wrap} \text{NewCell nil}\) end
    fun \(\text{Push}\) \(W\) \(X\) \(X\mapsto W\) \(X\) \(\text{in}\) \(\text{Assign}\) \(\text{C}\) \(\text{X}\) \(\text{Access}\) \(\text{C}\) \(\text{end}\)
    fun \(\text{Pop}\) \(W\) \(X\) \(\text{in}\) \(\text{Assign}\) \(\text{C}\) \(\text{X}\) \(\text{Access}\) \(\text{C}\) \(\text{end}\)

- 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:

```plaintext
  C = NewCell nil
in
  proc (Push X) [Assign C X | Access C] end
  fun (Pop) case (Access C) of X then [Assign C $] X end end
  fun (IsEmpty) [Isnil Access C] 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

```plaintext
procedure sq(i:integer, var b:integer);
begin
  b:=a*a
end
var i:integer;
i:=25;
sq(i);
writeln(i);
```

- 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

```plaintext
procedure sq(var a:integer);
begin
  a:=a+1;
  writeln(a*a)
end
sq(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

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

• A modification of call by variable. Variable argument can be modified.
• There are two mutable variables: one inside `B` (namely `i`) 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

```pascal
procedure callbyname(i:integer);
begin
  var
    A := @D
  end
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.
• Thanks were originally invented for Algol 60.

Call by need

```pascal
procedure callbyneed(i:integer);
begin
  var
    a:=a*a
  end
end
```

• A modification of call by name. The thank 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 calls and procedure values.
  • Arguably, call by value (along with calls 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 functional languages with lazy evaluation (e.g., Haskell and Miranda.)
  • For parallel programs, call by 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., Or).
• 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 `RemoteInterface`
  • 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, Or, Emerald), so call by move is not possible.

Exercises

51. [VR Exercise 6.10.2 (page 482)]
52. 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.)
53. Create a program in which call by name and call by need parameter passing styles result in different outputs.
54. *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?