Interpretation and Compilation of Programming Languages
Part 3 - Dynamic and Static Typing

João Costa Seco

March 19, 2014

In this lecture we discuss the fundamental aspect of runtime errors and safety of programming languages as given by type verification mechanisms. We proceed by defining the semantics of a small untyped programming language, and check its runtime error situations. We then study dynamic typing mechanisms that allow runtime detection of erroneous situations, and static typing mechanisms that conservatively identify a larger set of runtime errors.

We define the static typing mechanism by means of a semantic function from programs to types, which guarantees that a set of runtime errors, so-called type errors, do not occur at runtime.

1 Error handling

Consider the abstract syntax for language *CALC* already presented in the previous lecture, here defined in Figure 1 using Haskell, and the corresponding semantic function, in Figure 2. The function `eval` yields a result for all syntactically correct expression, except in the case of a division by zero. This is an unexpected error in our implementation, i.e. the interpreter does not detect the error and crashes. Thus, the Haskell function `eval` defines a partial function from programs to integers, since it is not defined for all the elements of its domain (the set of programs or expressions), given by the data type `CALC`. Also, the target set of the eval function is directly mapped to the `Int` data type, which does not allow us to encode any form of error handling.

Undefined results correspond to what we commonly know as execution errors. A common approach to handling execution errors is to trap them,

```
data CALC =
    Num Int
  | Add CALC CALC
  | Sub CALC CALC
  | Mul CALC CALC
  | Div CALC CALC
  deriving Show
```

Figure 1: Haskell data type for *CALC* expressions.

```
eval :: CALC -> Int
eval (Num n) = n
eval (Add e e') = (eval e) + (eval e')
eval (Sub e e') = (eval e) - (eval e')
eval (Mul e e') = (eval e) * (eval e')
eval (Div e e') = div (eval e) (eval e')
```

Figure 2: Operational semantics for *CALC* expressions.

and treat them within the language domain, by dynamically analysing the operands of each language operation. For instance, in Java, an *array-out-of-bounds* error or a wrong cast operation is detected and handled using the exception mechanism of the language. A simpler approach, adopted by Javascript, is to extend the set of denotations, the target set of function `eval`, to include the special value `Undefined`, and define it by

$$Integer \cup \{\texttt{Undefined}\}.$$

Where the `Undefined` value is the denotation given to the programs for which no integer value can be computed due to an error.

The extended result set can be defined as an abstract data type, and implemented by an `Haskell` inductive data type `Result`, like the one in Figure 3, or by a `Java` class hierarchy, like the one in Figure 4. To extend our interpreter to handle errors, one needs to anticipate all runtime errors and give the appropriate denotation. Check the `Haskell` code in Figure 5. Notice that there is an explicit check to ensure that both operands are indeed valid (`Value n`), otherwise the result is the special value `Undefined`. In the case of the division operation, the division by zero is detected and handled using the value `Undefined`. In the other cases of the language, an error occurring in one of the operand expressions is propagated to the result.

Exercise 1. Are there more special situations where this language semantics is not defined. Can you enumerate them?

```
data Result =
    Integer Int
  | Undefined
  deriving (Eq,Show)
```

Figure 3: Result set for language *VAL* in Haskell

```
interface Result {}

class IntegerValue { int value; }

class Undefined {}
```

Figure 4: Result set for language *VAL* in Java

```
eval :: AST -> Result

eval (Num n) = Value n

eval (Add e e') =
         case (eval e, eval e') of
              (Value n, Value n') -> Value (n+n')
              (_ , _) -> Undefined

eval (Sub e e') =
         case (eval e, eval e') of
              (Value n, Value n') -> Value (n-n')
              (_ , _) -> Undefined

eval (Mul e e') =
         case (eval e, eval e') of
              (Value n, Value n') -> Value (n*n')
              (_ , _) -> Undefined

eval (Div e e') =
         case (eval e, eval e') of
              (Value n, Value 0) -> Undefined
              (Value n, Value n') -> Value (div n n')
              (_ , _) -> Undefined
```

Figure 5: Evaluation function for language *VAL* in Haskell

```
data AST =
    Num Int
  | Add AST AST
  | Sub AST AST
  | Mul AST AST
  | Div AST AST
  | VTrue
  | VFalse
  | IF AST AST AST
  | And AST AST
  | Or AST AST
  | Geq AST AST
  | Gt AST AST
  | Leq AST AST
  | Lt AST AST
  | Eq AST AST
  deriving (Eq,Show)
```

Figure 6: Haskell data type for *VAL* expressions.

2 Dynamic Typing

Runtime errors are usually due to a mismatch between given operands and the valid domain of the operations. For instance, the domain of division operation is the set of pairs that belong to $\mathbb{N} \times \mathbb{N}_{\setminus\{0\}}$ and the target set is $\mathbb{N}$.

When considering a language like *VAL*, with the abstract syntax given in Figure 6, the set of valid results is

$$\mathit{Integer} \cup \{\texttt{true}, \texttt{false}\}.$$

In this language, there are many expressions for which there is not an integer or boolean denotation, which result from a mismatch in the operators domain and the nature of its operands. Adding a boolean value with an integer value makes no sense. These undefined results should be classified as execution errors, and can be trapped, as before, by analysing the operands of each language operation before the operation is erroneously applied. Given the approach above, We can extend the target set of function `eval` to

$$\mathit{Integer} \cup \{\texttt{true}, \texttt{false}\} \cup \{\texttt{Undefined}\}.$$

```
data Result =
    Integer Int
  | Boolean bool
  | Undefined
  deriving (Eq,Show)
```

Figure 7: Result set for language *VAL* in Haskell

```
interface Result {}

class IntegerValue { int value; }

class BooleanValue { int value; }

class Undefined {}
```

Figure 8: Result set for language *VAL* in Java

Hence, the result set can be defined by the `Haskell` data type in Figure 7 or by the corresponding `Java` class hierarchy in Figure 8.

We can now extend our interpreter definition to detect all possible execution errors and yield an `Undefined` result, as in the OCaml code fragment in Figure 9, or the Haskell code fragment in Figure 10 This strategy can be found in many interpreted languages, whose interpreters do not crash on invalid programs, but instead yield some distinguishable invalid result (e.g. Javascript). An alternative would be to generate some trapped exception that can be dealt by language mechanisms. This is a form of dynamic typing, where the interpreter code is able to detect the nature of the operand values, and act accordingly. Possible actions of the interpreter can be to convert some value from boolean to integer, or integer to a real number, or they can be to yield some undefined result.

Dynamic typing is a technique that demands a typed dynamic representation of values at runtime It allows for the design of more flexible languages, but it can be a factor of significant impact in the efficiency of the interpreter. An alternative to representing and checking type information at runtime, prior to each operation, is to use static typing.

```
let rec eval e =
  match e with
    Number n -> Integer n
  | Add(l,r) ->
      (match eval l, eval r with
       | Integer n, Integer m -> Integer m+n
       | _, _ -> Undefined)
...
```

Figure 9: Dynamically typed interpreter for *VAL* (OCaml).

```
...

eval VTrue = Boolean True

eval VFalse = Boolean False

eval (IF e e' e'') =
         case eval e of
              Boolean b -> if b then eval e' else eval e''
              _ -> Undefined

eval (And e e') =
          case (eval e, eval e') of
              (Boolean b, Boolean b') -> Boolean (b && b')
              (_ , _) -> Undefined

...

eval (Lt e e') =
          case (eval e, eval e') of
              (Integer n, Integer m) -> Boolean (n < m)
              (_ , _) -> Undefined

eval (Eq e e') =
          case (eval e, eval e') of
              (Integer n, Integer m) -> Boolean (n == m)
              (Boolean b, Boolean b') -> Boolean (b == b')
              (_ , _) -> Undefined
```

Figure 10: Dynamically typed interpreter for *VAL* (Haskell).

3 Type Systems

Static typing works by analysing the source code, and conservatively consider all the possible runtime scenarios. Static typing consists in interpreting a program using representatives for the runtime values, i.e. types. Unlike computations over runtime values, computation over types is (on most type systems) decidable, and can be computed efficiently at development time.

The main goal of static typing is to avoid runtime errors, by rejecting, *à priori*, a program that does not conform to a type specification. Ill-typed programs may crash due to type errors at runtime. Not all execution errors can be easily predicted, and there is a wide range of type systems, to avoid an equally wide range of runtime errors.

The use of static typing also allows for a more efficient use of resources (memory and time), by totally (or partially) erasing the type information from the runtime representation of values. For instance, the runtime information, in languages like C, is totally erased from the runtime, while in Java, the produced bytecode is typed with basic and object types, but generic type variables are erased and replaced by the `Object` type. Also in Java, non-essencial information about identifiers used by the developer are simply erased from the compiled code. Static typing also allows for the generation of more efficient target code operations, by choosing in advance the operations suitable for a given value representation. Dynamic type information is usually used to choose, at runtime, the actual operations to apply, or which method to call (in OO languages), based on the actual nature of the operands (or arguments).

Type systems ensure basic properties like the declaration and initialisation of all used local variables' identifiers, or the implementation of all methods declared in some class interface. Type systems impose a programming discipline on the developer which is not enforced in the more flexible, and dynamically typed languages like Ruby, Python, or JavaScript.

A type system is a semantic function, and can be implemented by an interpreter working at a different abstraction level. We will use a technique similar to the definition of the operational semantics studied in the previous lecture. Denotations for programs, in this context, belong to the set of possible types of values. In the case of language *VAL* the set is:

$$\{Integer, Boolean, None\}$$

where type *Integer* is given to all programs always yielding integer values, *Boolean* is given to all programs always yielding boolean values, and *None* is given to all programs for which it is not possible to determine the nature of their result values.

```
data Type =
    IntType
  | BoolType
  | None
  deriving (Eq,Show)
```

Figure 11: Abstract data type for types in language *VAL*.

```
typecheck :: AST -> Type

typecheck (Num n) = IntType

typecheck (Add e e') =
         case (typecheck e, typecheck e') of
              (IntType, IntType) -> IntType
              (_ , _) -> None
...
typecheck (Eq e e') =
          case (typecheck e, typecheck e') of
              (IntType, IntType) -> BoolType
              (BoolType, BoolType) -> BoolType
              (_ , _) -> None
```

Figure 12: Interpretation function for types in language *VAL*.

The signature of the type checking semantic function on programs of language *VAL*, called *typecheck*, is the following:

$$typecheck : VAL \rightarrow \{Integer, Boolean, None\}$$

Once again the target set can be represented by the abstract data type defined in Figure 11, and the abstraction function is given by the Haskell code in Figure 12.

Using the `Java` language, we can express the typecheck function in the way used to expressed the operational semantics. Types are expressed using a class hierarchy (see Figure 13) and the typing function is expressed as a new method of interface `ASTNode` and implemented in all classes of the AST (see Figure 14). Notice that, for the sake of simplicity, type classes are here represented as singleton classes, which makes comparisons be implemented by the comparator of base values.

```
interface Type {}

class IntType implements Type {
  static public final IntType value = new IntType();

  private IntType() {}
}

class BoolType implements Type {
  static public final BoolType value = new BoolType();

  private BoolType() {}
}
```

Figure 13: Types for language *VAL* in `Java`.

A discussion will follow about the guaranties that are associated to well typed programs, which will appended to these notes afterwards.

In this lecture we've studied two different ways of trapping execution errors in programs. The dynamic type verification avoids that execution erros cause the interpreter to crash, and also provides a structured way of dealing with errors (exception mechanisms, undefined values, etc). The static type verification allows for a more conservative approach, by detecting execution errors without actually executing the code. One may inspect the definitions of Figures 10 and 12 and conclude that, all type verifications in the operational semantics function will always succeed.

4 Exercises

Exercise 2. Does the type system avoid all execution errors? Enumerate 5 examples of errors that are not usually trapped by a type system, and 5 execution errors that are.

5 Well typed programs do not go wrong

Write subject reduction proof.

```
class ASTNum implements ASTNode {
  ...
  @Override
  public Type typecheck() throws TypeException {
    return IntType.value;
  }
}

class ASTAdd implements ASTNode {
  public final ASTNode left;
  public final ASTNode right;
  ...
  @Override
  public Type typecheck() throws TypeException {
    if( left.typecheck() == IntType.value &&
        right.typecheck() == IntType.value )
      return IntType.value;
    throw new TypeException();
  }
}
```

Figure 14: Typechecking fragment for *VAL* in `Java`.