Bottom in the Imperative World

Apostolos Syropoulos *
Alexandros Karakos†

Abstract

Bottom is used in functional programming languages to denote abnormal situations. We propose the incorporation of this value in any existing imperative language. We show how this can be done and finally what are the benefits of the bottom incorporation in any imperative language.

1 Introduction

The value $\perp$ (pronounced *bottom*) refers to situations where something abnormal happens, i.e. $\perp = \frac{1}{0}$. [1] Bottom has been successfully incorporated in some existing functional programming languages as an alternative exception handling mechanism. In LML [AJ91] and Haskell [HPJ+91] bottom is implemented as a polymorphic function with type $String \rightarrow \alpha$, which terminates program execution and prints onto the output medium its argument (in LML this function is called **fail** and in Haskell **error**). In SISAL 2.0 [BCFO92] bottom is denoted by the keyword **error** and some type specifier attached to it. The function **is_error** determines if some expression evaluates to bottom. A SISAL 2.0 implementation may choose either to allow value propagation or to terminate program execution when such a value is detected. To the extend we do know bottom has not been incorporated in any existing imperative programming language.

In this work we describe the general properties of bottom and show how it is possible to incorporated it in any existing imperative language. Finally we demonstrate how bottom may be used in an imperative language.

2 General Properties

Let $D_\perp$ denote a data type that includes the bottom value and $\sqsubseteq$ a partial ordering on $D_\perp$ then,

$$\forall\, d{:}D_\perp,\ \perp \sqsubseteq d$$

Consequently it is possible to compare a data value against $\perp$.

Suppose that $\oplus$ is a binary operator with type $A_\perp \times B_\perp \rightarrow C_\perp$ and that θ is an unary operator with type $A_\perp \rightarrow B_\perp$ then,[2] the following holds:

$$\forall\, a{:}A_\perp, b{:}B_\perp : a \oplus \perp_B = \perp_A \oplus b = \perp_C \text{ and } \theta\perp_A = \perp_B$$

The above property holds for operators with arity greater than 2, but not for any data structuring operator.

Especially for the data type T={$\perp$, *ff*, *tt*} bottom denotes the fact that we are not sure if something holds or if it does not. Consequently the truth tables are augmented with the following cases:

*366, 28th October str., GR-671 00 XANTHI, GREECE
†Democritus University of Thrace, Department of Electrical and Computer Engineering, GR-671 00 XANTHI, GREECE, email: karakos@xanthi.cc.duth.gr

[1] Bottom is used in domain theory to describe recursion in a purely mathematical way.

[2] Note that $A_\perp$, $B_\perp$, and $C_\perp$ are not necessarily different

$$tt \wedge \perp = \perp \wedge tt = \perp$$
$$ff \wedge \perp = \perp \wedge ff = ff$$
$$tt \vee \perp = \perp \vee tt = tt$$
$$ff \vee \perp = \perp \vee ff = \perp$$
$$\neg \perp = \perp$$

where $\wedge$ denotes logical conjunction, $\vee$ logical disjunction, and $\neg$ logical negation.

3 Bottom in an Imperative Language

Before making any attempt to incorporate the bottom value in an existing language we must answer the following two questions:

1. Should every data type have its own distinguished bottom value or not?
2. Which control constructs should be mainly modified in order to allow proper use of the bottom value?

We believe that every data type should have a distinguished bottom value. This approach prevents us from type errors and consequently from any improper use of this value. On the other hand, a "typeless" approach enables someone to write meaningless things, e.g. $\frac{1}{0}$=**nil**. Certainly these bottom values must have the general properties described in section 2. So, for example, a function that has an argument which evaluates to bottom, evaluates itself to bottom. The control constructs that should be mainly modified are those that involve logical tests, i.e., branching and iteration constructs. We start our discussion by considering the **if**-instruction, the main branching instruction. We propose that the **if**-instruction should be augmented by a new branch that would handle the bottom case. The bottom case should be optional and it could be present only if the traditional "else" branch is present or in BNF:

if-instruction ::= if-part [elsef-part [elseb-part]] "end"
if-part ::= "if" test "then" I_1
elsef-part ::= "elsef" I_2
elseb-part ::= "elseb" I_3

where I_i denotes an instruction sequence. It is a straight-forward exercise to provide an operational semantics for this new instruction. Let I denote the instruction **if b then** I_1 **elsef** I_2 **elseb** I_3 **end** and let ρ denote the store then, the following must hold:

$$\frac{\langle b, \rho\rangle \longrightarrow \langle tt, \rho\rangle}{\langle I, \rho\rangle \longrightarrow \langle I_1\, \rho\rangle}$$

$$\frac{\langle b, \rho\rangle \longrightarrow \langle ff, \rho\rangle}{\langle I, \rho\rangle \longrightarrow \langle I_2\, \rho\rangle}$$

$$\frac{\langle b, \rho\rangle \longrightarrow \langle \perp, \rho\rangle}{\langle I, \rho\rangle \longrightarrow \langle I_3\, \rho\rangle}$$

Accordingly the **case**-instruction should have an **elseb** branch to catch abnormal situation, the way the extended **if**-instruction does.

It seems that iterative constructs need a $\perp$-catcher too. Here we will consider only the traditional **while**-instruction. We propose an augmented **while**-instruction that can deal with abnormal situations. We feel that the following form is quite reasonable:

```
while b do
 except I |
  IS
end
```

where `b` is a boolean expression, `I` denotes what should be done when `b` evaluates to $\perp$, and finally `IS` is the normal instruction sequence. When an abnormal situation is encountered `I` is executed and then the loop aborts execution. Some may object to this, but we feel that attempting another loop iteration may lead to a "vicious circle". Similar modifications should be done to the various other iteration constructs to allow the proper use of the bottom value.

A final issue that must be settled is the behavior of functions and procedures when they have arguments that evaluate to $\perp$ (pass-by-value), or have arguments that are references to storage cells that contain the bottom value (pass-by-reference). Any function returns $\perp$ as soon as it detects that an actual argument either evaluates to bottom or is the address of some bottom value. Accordingly a procedure terminates its execution without affecting the global state of the entire program.

4 Using bottom

Bottom is not of theoretical interest only but it can be used in practical ways. Here we outline two possible uses of bottom.

An acceptable compiler is one that detects as many as possible syntactic or type errors in a source program. Accordingly, we feel that an executable file should be able to detect as many as possible run-time errors. It is possible to achieve such a behavior by making a run-time error a legal value (viz., bottom). Furthermore this approach to run-time errors, may lead to sophisticated debugging tools.

System prototyping is another area where bottom may find a use. Suppose that in some program, P, the value of the variable `Factor` is updated by means of the following instruction:

```
Factor:=max(a/b, c/d)
```

In a conventional language we must check that both `b` and `d` are not equal to zero in order to prevent a run-time error. On the other hand, being able to compare `Factor` against bottom makes code more compact and leaves us space to thing about important issues and to ignore, for the moment, these "minor" issues.

References

[AJ91] Lennart Augustsson and Thomas Johnsson. Lazy ML user's manual. Technical report, Chalmers University of Technology and University of Gothenburg, 1991.

[BCFO92] A. P. W. Bohm, David C. Cann, John T. Feo, and Rodney R. Oldehoeft. SISAL reference manual. Technical report, Colorado State University and Lawrence Livermore National Laboratory, 1992.

[HPJ+91] Paul Hudak, Simon Peyton Jones, et al. Report on the programming language Haskell. Technical report, Yale University and University of Glasgow, 1991.