A New Statement for Selection and Exception Handling in Imperative Languages

Keehang Kwon
Dept. of Computer Engineering, DongA University
840 hadan saha, Busan, Korea
khkwon@dau.ac.kr

Abstract: Diverse selection statements – if-then-else, switch and try-catch – are commonly used in modern programming languages. To make things simple, we propose a unifying statement for selection. This statement turns out to have a simple syntax and semantics. Examples will be provided for this statement.

keywords: selection, imperative programming, exceptions, if then else.

1 Introduction

Most programming languages have selection statements to direct execution flow. A selection statement allows the machine to choose one between two or more tasks during computation. Selection statements include if-then-else, switch and try-catch. Unfortunately, these statements were designed on an ad-hoc basis and have several shortcomings.

• if-then-else have the well-known dangling-if problem.

• if-then-else and try-catch are designed as binary connectives. Consequently, they are not easily extended to general $n$-ary cases.

• All three of these, try-catch in particular, have complicated syntaxes and semantics. As a result, they are quite cumbersome to use.

• An exception typically causes a non-local, dynamic lookup of an exception handler. This often violates the declarative reading of the execution sequence of the program.
To overcome these problems, we propose a new foundation for imperative languages, which is based on a new logic called task logic \cite{2, 3}. The task logic expands the traditional t/f (true/false) so as to include T/ F(success/failure). The task logic interprets each statement as T/F, depending on whether it can be successfully completed or not. We often need to extend F to F(Elist) where Elist is a list of exceptions raised during executing a sentence. This new foundation requires little changes to the existing interpreters.

In this setting, we propose a new selection statement. This statement is quite simple and of the form

\[
\text{choose}(G_1, \ldots, G_n)
\]

where each \( G_i \) is a statement. This has the following execution semantics:

\[
ex(P, \text{choose}(G_1, \ldots, G_n)) \text{ if } ex(P, G_i)
\]

where \( i \) is chosen by the machine and \( P \) is a set of procedure (and function) definitions. In the above definition, the machine chooses a successful disjunct \( G_i \) and then proceeds with executing \( G_i \). For simplicity, we assume that the machine chooses the least \( i \) if there are many such \( i \)'s. From now on, we assume that the machine tries these statements sequentially from left to right. In such a setting, if \( G_j \) fails, then we assume that the machine performs the recovery action. In other words, it rolls back partial updates caused by \( G_j \).

It can be easily seen that our new statement subsumes the if-then-else statement. For example, \( if \ cond \ then \ S \ else \ T \) can be converted to

\[
\text{choose}(cond; S, \neg cond; T).
\]

In the above, note that a boolean condition is a legal statement in our language, as we shall see in Section 2.

It is also straightforward to convert the switch statement to our language. For example, the following Java-like code displays the employee’s age.

```java
getAge(emp) {
    switch (emp) {
    case tom: age = 31; break;
    case kim: age = 40; break;
    case sue: age = 22; break;
    default: age = 0;
    }
    return age;
}
```
Note that the above code can be converted to the one below:

```java
getAge(emp) {
  choose(
    emp == tom; age = 31,
    emp == kim; age = 40,
    emp == sue; age = 22,
    t; age = 0);
  return age;
}
```

This program expresses the task of the machine choosing one among three employees. Note that this program is a little more compact and easier to read.

This paper focuses on the minimum core of Java. This is to present the idea as concisely as possible. The remainder of this paper is structured as follows. We describe the core Java in Section 2. In Section 3, we present an example of Java\textsuperscript{chou}. Section 4 concludes the paper.

## 2 The Language

The language is a subset of the core (untyped) Java with some extensions. It is described by $G$- and $D$-formulas given by the syntax rules below:

$$G ::= t | f | A | \text{cond} | \neg \text{cond} | x = E | G; G | \text{choose}(G_1, \ldots, G_n)$$

$$D ::= A = G | \forall x D$$

In the above, $t$ represents a success and $f$ represents a (user-defined) failure/exception. $f$ is often extended to $f(\text{errcode})$. $A$ represents a head of an atomic procedure definition of the form $p(s_1, \ldots, s_n)$. A $D$-formula is called a procedure definition.

In the transition system to be considered, $G$-formulas will function as the main program (or statements), and a set of $D$-formulas enhanced with the machine state (a set of variable-value bindings) will constitute a program.

We will present an operational semantics for this language via a proof theory \cite{1}. This style of semantics has been used in logic languages \cite{2, 4, 7}. The rules are formalized by means of what it means to execute the main task $G$ from a program $\mathcal{P}$. These rules define precisely what is a success and failure. Below the notation $D; \mathcal{P}$ denotes $\{D\} \cup \mathcal{P}$ but with the $D$ formula being distinguished (marked for backchaining). Note that execution
alternates between two phases: the main phase (the phase of executing the main program) and the backchaining phase (one with a distinguished clause). The notation \( S \sand R \) denotes the following: execute \( S \) and execute \( R \) sequentially. It is considered a success if both executions succeed.

**Definition 1.** Let \( G \) be a main task and let \( P \) be a program. Then the notion of executing \( \langle P, G \rangle \) successfully and producing a new program \( P' \) – \( \text{ex}(P, G, P') \) – is defined as follows:

1. \( \text{ex}((A = G_1); P, A) \) if \( \text{ex}(P, G_1) \) and \( \text{ex}(D; P, A) \).
2. \( \text{ex}(\forall x D; P, A) \) if \( \text{ex}([s/x]D; P, A) \) where \( s \) is a term. % argument passing
3. \( \text{ex}(P, A) \) if \( D \in P \) and \( \text{ex}(D; P, A) \). % a procedure call
4. \( \text{ex}(P, t, P) \). % True is always a success.
5. \( \text{ex}(P, \text{cond}, P) \) if \( \text{cond} \) is true. % boolean condition.
6. \( \text{ex}(P, \neg \text{cond}, P) \) if \( \text{cond} \) is false. % boolean condition.
7. \( \text{ex}(P, x = E, P \uplus \{\langle x, E'\rangle\}) \) if \( \text{eval}(P, E, E') \). % the assignment statement. If evaluating \( E \) fails, then the whole statement fails. Here, \( \uplus \) denotes a set union but \( \langle x, V \rangle \) in \( P \) will be replaced by \( \langle x, E' \rangle \).
8. \( \text{ex}(P, G_1; G_2, P_2) \) if \( \text{ex}(P, G_1, P_1) \) and \( \text{ex}(P_1, G_2, P_2) \).
9. \( \text{ex}(P, \text{choose}(G_1, \ldots, G_n), P_1) \) if \( \text{ex}(P, G_i, P_1) \) where \( G_i \) is the first successful disjunct chosen by the machine.

If \( \text{ex}(P, G, P_1) \) has no derivation, then the machine returns the failure. For example, \( \text{ex}(P, f, P_1) \) is a failure because there is no derivation for \( f \).

### 3 Examples

As mentioned in Section 1, exception handling in modern programming languages has several shortcomings. In short, it is difficult to be concise and it is difficult to be right.

Our *choose* statement is well-suited to exception handling.

- Our language naturally gives a *hierarchical* structure to exceptions. Now all the exceptions belong to the Failure set. This considerably simplifies the exception handling.
Our language disallows abrupt, nonlocal, dynamic transfers of control which violates the declarative reading of the execution sequence. This property is essential for program verification.

Our language can easily handle nested exception handling.

Our language gives a logical status to exceptions. This means that other useful logical connectives such as disjunctions can be added. Some progress has been made towards this direction [4].

As an example of exception handling, let us consider the following three tasks.

A: read a set of numbers from a file foo and compute their sum.
B: read a set of numbers from the first backup file foo.1 and compute their sum.
C: read a set of numbers from the second backup file foo.2 and compute their sum.

It is not easy to write robust codes for these tasks in traditional languages. Fortunately, it is rather simple in our setting. For example, the following statement expresses the task of trying A, B and C sequentially.

\[
\text{choose}(A, B, C, \text{print('error'))}
\]

In the above, if all fail, the machine prints a simple error message. Of course, a further analysis of exceptions is possible by inspecting the exceptions raised during execution. These exceptions are stored in a global variable \(\text{Elist}\). In summary, our language considerably reduces many complications.

4 Conclusion

In this paper, we have considered an extension to a core Java with a new selection statement. This extension allows \(\text{choose}(G_1, \ldots, G_n)\) where each \(G_i\) is a statement. This statement makes it possible for the core Java to model decision steps.

5 Acknowledgements

This work was supported by Dong-A University Research Fund.
References

[1] G. Kahn, “Natural Semantics”, In the 4th Annual Symposium on Theoretical Aspects of Computer Science, LNCS vol. 247, 1987.

[2] G. Japaridze, “Introduction to computability logic”, Annals of Pure and Applied Logic, vol.123, pp.1–99, 2003.

[3] G. Japaridze, “Sequential operators in computability logic”, Information and Computation, vol.206, No.12, pp.1443-1475, 2008.

[4] K. Kwon, S. Hur and M. Park, “Improving Robustness via Disjunctive Statements in Imperative Programming”, IEICE Transations on Information and Systems, vol.E96-D, No.9, September, 2013.

[5] J. Hodas and D. Miller, “Logic Programming in a Fragment of Intuitionistic Linear Logic”, Information and Computation, vol.110, No.2, pp.327-365, 1994.

[6] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov, “Uniform proofs as a foundation for logic programming”, Annals of Pure and Applied Logic, vol.51, pp.125–157, 1991.

[7] D. Miller, G. Nadathur, Programming with higher-order logic, Cambridge University Press, 2012.