Extending Functional Languages with High-Level Exception Handling

Keehang KWON†, Member

SUMMARY We extend functional languages with high-level exception handling. To be specific, we allow sequential-choice disjunctive (SCD) expressions of the form $E_0 \triangledown E_1$ where $E_0, E_1$ are expressions. These expressions have the following intended semantics: sequentially choose the first successful $E_i$ and evaluate $E_i$ where $i = 0$ or 1. These expressions thus allow us to specify an expression $E_0$ with the failure-handling (exception handling) routine, i.e., expression $E_1$. The language is a version of the core functional languages with SCD operators. The remainder of this paper is structured as follows. We describe Func$\triangledown$ in the next section. In Section 3, we present some examples of Func$\triangledown$. Section 4 concludes the paper.

2. The Language

The language is a version of the core functional languages – also called primitive recursive functions – with SC expressions. It is described by $E$- and $D$-formulas given by the syntax rules below:

- $E ::= t \mid h(E, \ldots, E) \mid G \triangledown G$
- $D ::= f(x_1, \ldots, x_n) = E \mid D \land D$

In the rules above, $x$ is a variable, and $t$ is a term which is either a variable or a constant. A $D$-formula is called a program.

Following [3],[4], we will present an interpreter for this language as a set of rules. These rules are based on an eager evaluation. Note that execution alternates between two phases: the evaluation phase and the backchaining phase. In the evaluation phase (denoted by $eval(D, E, K)$), the machine tries to evaluate an expression $E$ from the program $D$ to get a value $K$. The rules (6) – (8) are related to this phase. If $E$ is a function call $h$, the machine first evaluates all of its arguments and then looks for a definition of $h$ in the program in the backchaining mode. This is encoded in the rule (5). In the backchaining mode (denoted by $bc(D_1, D, h, K)$), the machine tries to evaluate a function call $h$ by using the function definition in the program $D_1$. The rules (1) – (4) describe these operations.

Definition 1. Let $E$ be an expression and let $D$ be a program. Then the notion of evaluating $(D, E)$ to a value $K - eval(D, E, K)$ – is defined as follows:
(1) \(bc(h(c_1, \ldots, c_n) = E, D, h(c_1, \ldots, c_n), K)\)
if \(eval(D, E, K)\). % switch to evaluation mode.

(2) \(bc(D_1 \land D_2, D, h(c_1, \ldots, c_n), K)\)
if \(bc(D_1, D, h(c_1, \ldots, c_n), K)\). % look for \(h\) in \(D_1\)

(3) \(bc(D_1 \land D_2, D, h(c_1, \ldots, c_n), K)\)
if \(bc(D_2, D, h(c_1, \ldots, c_n), K)\). % look for \(h\) in \(D_2\)

(4) \(bc(h(x_1, \ldots, x_n) = E, D, h(c_1, \ldots, c_n), K)\)
if \(bc(h(c_1/x_1, \ldots, c_n/x_n) = E', D, h(c_1, \ldots, c_n), K)\)
where \(E' = [c_1/x_1, \ldots, c_n/x_n]E\). % argument passing to \(h\) and \(E\).

(5) \(eval(D, h(c_1, \ldots, c_n), K)\)
if \(bc(D, D, h(c_1, \ldots, c_n), K)\). % switch to backchaining. we make a copy of \(D\) for a function call.

(6) \(eval(D, h(E_1, \ldots, E_n), K)\)
if \(eval(D, E_i, c_i)\) and \(eval(D, h(c_1, \ldots, c_n), K)\). % evaluate the arguments first.

(7) \(eval(D, c, c)\) % A success if \(c\) is a constant.

(8) \(eval(D, E_1 \triangleleft \ldots \triangleleft E_n, K)\)
if \(eval(D, E_i, K)\), provided that \(E_i\) is the first successful expression. % exception handling.

In the above rules, only the rule (8) is a novel feature.

3. Examples

As an example, let us consider the well-known \(div\) function. There are many different ways to define this function and below is another one.

\(div(x, y) = (x/y) \triangleleft \text{infinity}\)

In the above, let us consider evaluating \(div(4, 0)\). Using backchaining, it first tries 4/0. Since it leads to a failure, the system next tries a constant \(\text{infinity}\). This leads to a success and the system returns \(\text{infinity}\).

4. Conclusion

In this paper, we have considered an extension to functional languages with SCD expressions. This extension allows expressions of the form \(E_0 \triangleleft E_1\) where \(E_0, E_1\) are expressions. These expressions are particularly useful for specifying exception handling in a high level way.

References

[1] R. Govindarajan, “Exception Handlers in Functional Programming”, IEEE Transactions on Software Engineering, vol.19, No.8, pp.826–834, 1993.

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

[3] E. Komendantskaya and V. Komendantsky, “On uniform proof-theoretical operational semantics for logic programming”, In J.-Y. Beziau and A.Costa-Leite, editors, Perspectives on Universal Logic, pages 379–394. Polimetrica Publisher, 2007.

[4] D. Miller, “A logical analysis of modules in logic programming”, Journal of Logic Programming, vol.6, pp.79–108, 1989.