IEICE TRANS. INF. & SYST., VOL.E96–D, NO.9 SEPTEMBER 2013

LETTER *Special Section on Dependable Computing*

Improving Robustness via Disjunctive Statements in Imperative Programming

Keehang KWON[†], **Sungwoo HUR**[†a)], ***Nonmembers***, ***and*** **Mi-Young PARK**[†], ***Member***

SUMMARY To deal with failures as simply as possible, we propose a new foundation for the core (untyped) C^{++}, which is based on a new logic called task logic or imperative logic. We then introduce a sequential-disjunctive statement of the form $S : R$. This statement has the following semantics: execute S and R sequentially. It is considered a success if at least one of S, R is a success. This statement is useful for dealing with inessential errors without explicitly catching them.

key words: *imperative programming, exceptions, task logic, failures*

1. Introduction

Imperative programming is an important modern programming paradigm. Successful languages in this paradigm includes C^{++} and Java. Despite much attractiveness, imperative languages have traditionally lacked fundamental notion of success/failure for indicating whether a statement can be successfully completed or not. Lacking such a notion, imperative programming relies on nonlogical, awkward devices such as exception handling to deal with failures. One major problem with exception handling is that the resulting language becomes complicated and not easy to use.

To deal with failures as simply as possible, we propose a new foundation for the core (untyped) C^{++}, which is based on a new logic called task logic [1], [2] or imperative logic. 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. All the operators of the core C^{++} can be interpreted in this way. For example, $S; R$ can be interpreted as sequential conjunction of S and R, as shown in Definition 1. Furthermore, an exception can be interpreted as failure.

The premature exit of a statement due to failures can be problematic. To avoid this, we adopt "all-or-nothing" semantics discussed in [3] to guarantee atomicity. Thus, if a failure occurs in the couse of executing a statement, we assume that the machine rolls back partial updates.

The use of task logic makes it possible to extend this "logic-based" C^{++} with other new and useful logical operations, thus allowing versatile executions of programs. To improve robustness, it is useful to divide errors into essential errors and inessential errors. Essential errors are required to be caught by an exception handler, while inessential errors are non-serious errors that can be ignored.

We list some basic design guidelines for handling inessential errors.

- It is desirable to spend no time and no efforts dealing with inessential errors.
- It is desirable to erase all the inessential errors raised so that none of these exceptions can have further interactions with the environment.

To deal with inessential errors, we introduce a sequential-disjunctive statement of the form $S : R$. Here, to avoid complications, we assume that S and R are *independent* of each other, *i.e.*, no variables appear in both S and R. This statement has the following semantics: execute S and R sequentially. It is considered a success if at least one of S, R is a success. This statement generates less exceptions, is easier to succeed, and hence is more robust than other statements such as $S; R$. This statement has the effect of reducing the number of exceptions to be dealt with without catching them. It is useful for dealing with inessential errors that can be ignored. For example, the statement $S : true$ has the effect of both ignoring and erasing all the possible exceptions raised in the course of executing S so that none of these exceptions can have further interactions with the environment. Thus, the statement $S : true$ satisfies the above design guidelines. There is another advantage: this statement can make expressions (and, hence, executions) more versatile and often simpler via logical equivalence. For example, the execution $S; (R : true)$ can be converted to a simpler $S : R$, assuming executing S is a success and S, R are independent of each other.

To deal with essential errors, we introduce a choice-disjunctive statement of the form $S\ else\ R$ which is a logical version of the *try* S *catch* R statement. This statement has the following semantics: execute S. If it is a success, then do nothing. If it fails, execute its exception handler R.

The remainder of this paper is structured as follows. We describe the new language C_L^{++} in the next section. In Sect. 3, we present some examples. Section 4 concludes the paper.

2. The Language

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

$$G ::= \ t \mid f \mid A \mid x = E \mid G; G \mid G : G \mid G\ else\ G$$

Manuscript received December 28, 2012.
Manuscript revised April 11, 2013.
[†]The authors are with Computer Eng., DongA Univ., Korea.
a) E-mail: swhur@dau.ac.kr (Corresponding author)
DOI: 10.1587/transinf.E96.D.2036

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

In the rules above, A represents an atomic procedure definition of the form $p(t_1, \ldots, t_n)$. A D-formula is called a procedure definition. f denotes $false$ which correponds to a user-thrown exception.

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. The rules are formalized by means of what it means to execute the main task G from a program $\mathcal{P}$. These rules in fact depend on the top-level constructor in the expression, a property known as uniform provability [5]. 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 goal-reduction phase (one without a distinguished clause) 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. The notation *not*() denotes a failure.

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

(1) $ex(\mathcal{P}, t, \mathcal{P})$. % True is always a success.

(2) $ex((A = G_1); \mathcal{P}, A)$ if $ex(\mathcal{P}, G_1)$ and $ex(D; \mathcal{P}, A)$.

(3) $ex(\forall x D; \mathcal{P}, A)$ if $ex([t/x]D; \mathcal{P}, A)$. % argument passing

(4) $ex(\mathcal{P}, A)$ if $D \in \mathcal{P}$ and $ex(D; \mathcal{P}, A)$. % a procedure call

(5) $ex(\mathcal{P}, x = E, \mathcal{P} \uplus \{\langle x, E'\rangle\})$ if $eval(\mathcal{P}, E, E')$. % $\uplus$ denotes a set union but $\langle x, V\rangle$ in $\mathcal{P}$ will be replaced by $\langle x, E'\rangle$.

(6) $ex(\mathcal{P}, G_1; G_2, \mathcal{P}_2)$ if $ex(\mathcal{P}, G_1, \mathcal{P}_1)$ sand $ex(\mathcal{P}_1, G_2, \mathcal{P}_2)$.

(7) $ex(\mathcal{P}, G_1 : G_2, \mathcal{P}_2)$ if $ex(\mathcal{P}, G_1, \mathcal{P}_1)$ sand $ex(\mathcal{P}, G_2, \mathcal{P}_2)$. % both G_1 and G_2 succeed.

(8) $ex(\mathcal{P}, G_1 : G_2, \mathcal{P}_2)$ if $not(ex(\mathcal{P}, G_1, \mathcal{P}_1))$ sand $ex(\mathcal{P}, G_2, \mathcal{P}_2)$. % only G_2 succeeds.

(9) $ex(\mathcal{P}, G_1 : G_2, \mathcal{P}_1)$ if $ex(\mathcal{P}, G_1, \mathcal{P}_1)$ sand $not(ex(\mathcal{P}, G_2, \mathcal{P}_2))$. % only G_1 succeeds.

(10) $ex(\mathcal{P}, G_1 \ else \ G_2, \mathcal{P}_1)$ if $ex(\mathcal{P}, G_1, \mathcal{P}_1)$

(11) $ex(\mathcal{P}, G_1 \ else \ G_2, \mathcal{P}_2)$ if $not(ex(\mathcal{P}, G_1, \mathcal{P}_1))$ sand $ex(\mathcal{P}, G_2, \mathcal{P}_2))$.

If $ex(\mathcal{P}, G, \mathcal{P}_1)$ has no derivation, then the machine returns F, the failure. For example, $ex(\mathcal{P}, f, \mathcal{P}_1)$ is a failure because it has no derivation.

3. Examples

So far, we have considered only one kind of failures. In reality, there are many kinds of failures in imperative programming. Thus, we need to expand f to include $f(e)$ for a user-thrown exception e. The notion of exception trees [4] is then useful to organize failures, similar to a file system in Unix and similar to an exception class in Java. Below we assume that the machine returns an exception tree stored in $Failtree$ rather than just F. We also assume that $/F$ is the root directory of $Failtree$ and $/F/usr$ is the directory for user-thrown failures. An exception can be derived from the parent exception. Exception trees allow the programmer to select to deal with failures at varying degrees of specificity. An example of the use of this construct is provided by the following program which contains some basic file-handling rules.

$main$
$openfile(); readfile()$
$else$
  $case \ Failtree \ of$
    $/F/sys : \ldots$
    $/F/usr/EOF : \ldots;$
$x = factorial(4)$
$readfile() = (read() \neq -1); \ldots \ else \ f(EOF)$

Our language makes it possible to simplify the program if some statements are inessential. For example, the following program explicitly tells the machine that the statement $openfile(); readfile()$ is inessential and optional and thus it is OK not to perform the statement if it fails.

$main$
$(openfile(); readfile()) :$
$x = factorial(4)$
$readfile() = (read() \neq -1); \ldots \ else \ f(EOF)$

4. Conclusion

In this paper, we have considered an extension to the core C^{++} with disjunctive statements. This extension allows statements of the form $S \ : \ R$ where S, R are statements. These statements are particularly useful for dealing with inessential errors.

Acknowledgements

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

References

[1] G. Japaridze, "Introduction to computability logic," Annals of Pure

and Applied Logic, vol.123, pp.1–99, 2003.
[2] G. Japaridze, "Sequential operators in computability logic," Information and Computation, vol.206, no.12, pp.1443–1475, 2008.
[3] C. Fetzer and P. Felber, "Improving program correctness with atomic exception handling," J. Universal Computer Science, vol.13, no.8, pp.1047–1072, 2007.
[4] P. Buhr and W. Bok, "Advanced exception handling mechanisms," IEEE Trans. Softw. Eng., vol.26, no.9, pp.1–15, 2000.
[5] 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.