LETTER

Bounded-Choice Statements for User Interaction in Imperative Programming

Keehang KWON[†a)], ***Member***, **Jeongyoon SEO**[†], ***and*** **Daeseong KANG**[††], ***Nonmembers***

SUMMARY Adding versatile interactions to imperative programming – C, Java and Android – is an essential task. Unfortunately, existing languages provide only limited constructs for user interaction. These constructs are usually in the form of *unbounded* quantification. For example, existing languages can take the keyboard input from the user only via the $read(x)/scan(x)$ statement. Note that the value of x is unbounded in the sense that x can have any value. This statement is thus not useful for applications with bounded inputs. To support bounded choices, we propose new bounded-choice statements for user interation. Each input device (keyboard, mouse, touchpad, ...) naturally requires a new bounded-choice statement. To make things simple, however, we focus on a bounded-choice statement for keyboard – *kchoose* – to allow for more controlled and more guided participation from the user. We illustrate our idea via C^{BI}, an extension of the core C with a new bounded-choice statement for the keyboard.

key words: *interactions, bounded choices, read*

1. Introduction

Adding versatile interactions to imperative programming – C, Java, Android, etc. – has become an essential task. Unfortunately, existing languages provide only limited constructs for user interaction. These constructs are usually in the form of *unbounded* quantification. For instance, the keyboard input statement that has been used in Java-like languages is restricted to the $read/scan$ statement. The *read* statement is of the form $read(x); G$, where G is a statement and x can have any value. Hence, it is a form of an *unbounded* quantified statement. However, in many situations, the system requires a form of *bounded-choice* interactions; the user is expected to choose one among many alternatives. Examples include most interactive systems such as airline ticketing systems.

The use of bounded-choice interactions is thus essential in representing most interactive systems. For this purpose, this paper proposes a bounded-choice approach to user interaction. Each input device naturally requires a new bounded-choice statement. To make things simple, however, we focus only on the keyboard device. It is straightforward to adjust our idea to other input devices such as mouse and touchpad.

Toward this end, we propose a new bounded keyboard input statement $kchoose(G_1, \ldots, G_n)$, where each G_i is a statement. This has the following execution semantics:

$$ex(\mathcal{P}, kchoose(G_1, \ldots, G_n)) \leftarrow ex(\mathcal{P}, G_i),$$

where i is chosen (*i.e.*, a keyboard input) by the user and $\mathcal{P}$ is a set of procedure definitions. The notation $S \leftarrow R$ denotes the reverse implication, *i.e.*, $R \rightarrow S$. In the above definition, the system requests the user to choose i via the keyboard and then proceeds with executing G_i. If i is not among $\{1, \ldots, n\}$, then we assume that the system does nothing. It can be easily seen that our new statement has many applications in representing most interactive systems.

The following C-like code example reads a variable named *emp* from the keyboard, whose value represents an employee's name.

```
read(emp);
switch (emp) {
    case tom: age = 31; break;
    case kim: age = 40; break;
    case sue: age = 22; break;
    default: age = 0;
}
```

In the above, the system requests the user to type in a particular employee. Note that the above code is based on unbounded quantification and is thus very awkward. It is also error-prone because the user may type in an invalid value.

The above application obviously requires a bounded-choice interaction rather than one based on unbounded quantification. Our *kchoose* statement provides such a bounded-choice interaction for keyboard and is useful to avoid this kind of human error. Hence, instead of the above code, consider the statement

```
print("Enter 1 for tom, 2 for kim and 3 for sue:");
kchoose(
    emp = tom; age = 31,
    emp = kim; age = 40,
    emp = sue; age = 22 );
```

This program expresses the task of the user choosing one among three employees. Note that this program is much easier and safer to use. The system now requests the user to select one (by typing 1, 2, 3) among three employees. After it is selected, the system sets his age as well.

Generally speaking, the *kchoose* statement is designed to directly encode most interactive objects which require the user to choose one among several possible tasks. Hence,

Manuscript received June 22, 2015.
Manuscript revised November 4, 2015.
Manuscript publicized November 27, 2015.

[†]The authors are with Dept. of Computer Eng., Dong-A University, Korea.
[††]The author is with Dept. of Electronics Eng., Dong-A University, Korea.
a) E-mail: khkwon@dau.ac.kr

DOI: 10.1587/transinf.2015EDL8141

there is a rich realm of applications for this statement. For example, as we will see later in Sect. 3, the ATM machine requires the user to select one among 1) balance checking, 2) cash withdrawal, and 3) cash deposit. Hence, it can be directly encoded via the *kchoose* statement.

It is easy to observe that *kchoose* statement can be built from the *read-switch* combination. For example, the above example can be rewritten in the following way.

```
print("Enter 1 for tom, 2 for kim and 3 for sue:");
read(n);
switch (n) {
    case 1: emp = tom; age = 31; break;
    case 2: emp = kim; age = 40; break;
    case 3: emp = sue; age = 22; break;
    default:
}
```

It is then tempting to conclude that the *kchoose* construct is not needed because it can be built from the *read-switch* combination. However, this view is quite misleading. Without it, the resulting codes would be low-level for the following reasons:

- The programmer must manually allocate a variable for the *read* construct.
- The programmer must specify the numbering sequence in the *switch* statement.
- The programmer must specify the default part.

As a consequence, these codes are cumbersome, error-prone, difficult to read, and reason about.

The *kchoose* construct should rather be seen as a well-designed, high-level abstraction for bounded-choice interaction and the *read-switch* combination should be seen as its low-level implementation. The advantage of the use of this construct becomes evident when an application has a long sequence of interactions with the user. Therefore, the need for this construct is clear. To our knowledge, this kind of construct has never been proposed before in imperative languages. This is quite surprising, given the ubiquity of bounded-choice interaction in interactive applications.

The *kchoose* construct can be implemented in many ways. One way to implement the *kchoose* construct is via preprocessing, *i.e.*, via transformation to plain C-like code. That is, $kchoose(G_1, \ldots, G_n)$ is transformed to the following:

```
int k;
read(k);
switch (k) {
    case 1: G'_1; break;
    case 2: G'_2; break;
    :
    case n: G'_n; break;
    default:
}
```

Here, k is a new, local storage, and $G'_1, \ldots, G'_n$ are obtained from $G_1, \ldots, G_n$ via the same transformation.

This paper focuses on the minimum core of C. This is to present the idea as concisely as possible. The remainder of this paper is structured as follows. We describe C^{BI}, an extension of core C with a new bounded-choice statement for the keyboard in Sect. 2. In Sect. 3, we present an example of C^{BI}. Section 5 concludes the paper.

2. The Language

The language is core C with procedure definitions. It is described by G- and D-formulas given by the syntax rules below:

$$\begin{aligned} G &::= true \mid A \mid x = E \mid G; G \mid read(x); G \mid \\ &\quad\; kchoose(G_1, \ldots, G_n) \\ D &::= A = G \mid \forall x\, D \end{aligned}$$

In the above, A in D represents a head of an atomic procedure definition of the form $p(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are parameters. A in G represents a procedure call of the form $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are actual arguments. A D-formula is called a procedure definition. In the transition system to be considered, G-formulas will function as the main statement, and a set of D-formulas enhanced with the machine state (a set of variable-value bindings) will constitute a program. Thus, a program is a union of two disjoint sets, *i.e.*, $\{D_1, \ldots, D_n\} \cup \theta$ where each D_i is a D-formula and θ represents the machine state. Note that θ is initially set to an empty set and will be updated dynamically during execution via the assignment statements.

We will present an interpreter via a proof theory [1], [5]–[7]. Note that this interpreter alternates between the execution phase and the backchaining phase. In the execution phase (denoted by $ex(\mathcal{P}, G, \mathcal{P}')$) it tries to execute a main statement G with respect to a program $\mathcal{P}$ and produce a new program $\mathcal{P}'$ by reducing G to simpler forms until G becomes an assignment statement or a procedure call. The rules (6), (7), (8) and (9) deal with this phase. If G becomes a procedure call, the interpreter switches to the backchaining mode. This is encoded in the rule (3). In the backchaining mode (denoted by $bc(D, \mathcal{P}, A, \mathcal{P}')$), the interpreter tries to solve a procedure call A and produce a new program $\mathcal{P}'$ by first reducing a procedure definition D in a program $\mathcal{P}$ to its instance (via rule (2)) and then backchaining on the resulting definition (via rule (1)). To be specific, the rule (2) basically deals with argument passing: it eliminates the universal quantifier x in $\forall x D$ by picking a value t for x so that the resulting instantiation, written as $[t/x]D$, matches the procedure call A. The notation S seqand R denotes the sequential execution of two tasks. To be precise, it denotes the following: execute S and execute R sequentially. It is considered a success if both executions succeed. Similarly, the notation S parand R denotes the parallel execution of two tasks. To be precise, it denotes the following: execute S and execute R in any order. Thus, the execution order is not important here. It is considered a success if both executions succeed. The notation S choose R denotes the selection between two

tasks. To be precise, it denotes the following: the machine selects and executes one between S and R. It is considered a success if the selected one succeeds.

As mentioned in Sect. 1, the notation $S \leftarrow R$ denotes reverse implication, *i.e.*, $R \rightarrow S$.

Definition 1. Let G be a main statement 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) $bc((A = G_1), \mathcal{P}, A, \mathcal{P}_1) \leftarrow ex(\mathcal{P}, G_1, \mathcal{P}_1)$. % A matching procedure for A is found.

(2) $bc(\forall x D, \mathcal{P}, A, \mathcal{P}_1) \leftarrow bc([t/x]D, \mathcal{P}, A, \mathcal{P}_1)$. % argument passing

(3) $ex(\mathcal{P}, A, \mathcal{P}_1) \leftarrow (D \in \mathcal{P}$ parand $bc(D, \mathcal{P}, A, \mathcal{P}_1))$. % a procedure call

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

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

(6) $ex(\mathcal{P}, G_1; G_2, \mathcal{P}_2) \leftarrow (ex(\mathcal{P}, G_1, \mathcal{P}_1)$ seqand $ex(\mathcal{P}_1, G_2, \mathcal{P}_2))$. % sequential composition

(7) $ex(\mathcal{P}, read(x); G, \mathcal{P}_1) \leftarrow ex(\mathcal{P} \uplus \{\langle x, kbd \rangle\}, G, \mathcal{P}_1)$. where kbd is the keyboard input and $\uplus$ denotes a set union but $\langle x, V \rangle$ in $\mathcal{P}$ will be replaced by $\langle x, kbd \rangle$.

(8) $ex(\mathcal{P}, kchoose(G_1, \ldots, G_n), \mathcal{P}_1) \leftarrow$ ((read the keyboard input i) seqand
($i \in \{1, \ldots, n\}$ seqand $ex(\mathcal{P}, G_i, \mathcal{P}_1)$) choose ($i \notin \{1, \ldots, n\}$ seqand $(\mathcal{P}_1 == \mathcal{P})$))

If $ex(\mathcal{P}, G, \mathcal{P}_1)$ has no derivation, then the machine returns the failure.

The rule (8) deals with bounded-choice interaction. To execute $kchoose(G_1, \ldots, G_n)$ successfully, the machine does the following:

(1) It reads and saves the keyboard input value i in some temporary storage.

(2) Then it tries the first branch of the form $i \in \{1, \ldots, n\}$ seqand $ex(\mathcal{P}, G_i, \mathcal{P}_1)$. That is, it first checks whether i is legal, *i.e.*, among $\{1, \ldots, n\}$. The machine then executes G_i.

(3) If the first branch fails, the machine tries the second branch of the form $i \notin \{1, \ldots, n\}$ seqand $(\mathcal{P}_1 == \mathcal{P})$. That is, it first checks whether i is illegal, *i.e.*, not among $\{1, \ldots, n\}$. If it is illegal, then it means that it is the user, not the machine, who failed to do his job. Therefore, the machine sets $\mathcal{P}_1$ to $\mathcal{P}$ and returns the success.

As an example of our language, the following G-formula

```
kchoose(
    emp = tom; age = 31,
    emp = kim; age = 40,
    emp = sue; age = 22 );
```

expresses the task of the user choosing one among three employees. More examples are shown in Sect. 3.

As mentioned earlier, the *kchoose* construct is a well-designed, high-level abstraction for bounded-choice interaction which is quite common to user interaction. As for its implementation, it can be bolted into the language as a basic statement or it can be supported via preprocessing. C++ macro code for some initial implementation of *kchoose* is available under
`http://www.researchgate.net/publication/282331184`[†].

3. Examples

As an example, consider the following statement that performs ATM transaction. The types of ATM transaction are 1) balance checking, 2) cash withdrawal, and 3) cash deposition. An example of this class is provided by the following code where the program $\mathcal{P}$ is of the form:

```
deposit() =
print("type 1 for $1 and 2 for $5:");
kchoose(amount = $1,amount = $5); ...
withdraw() =
print("type 1 for $1 and 2 for $5:");
kchoose(amount = $1,amount = $5); ...
balance() = ...
```

and the goal G is of the form:

```
print("type 1 for balance,2 for withdraw,3 for deposit");
kchoose(balance(), withdraw(), deposit());
```

In the above, the execution basically proceeds as follows: the machine asks the user to choose one among three procedures. If the user choose the withdrawal by typing 2, then the machine will ask the user again to choose the amount of the withdrawal. Then the execution will go on. Note that our code is very concise compared to the traditional one.

As a second example, our language makes it possible to customize the amount for tuition via interaction with the user.

The following C-like code displays the amount of the tuition, based on the user's field of study.

```
read(major);
switch (major) {
    case english: tuition = $2,000; break;
    case medical: tuition = $4,000; break;
    case liberal: tuition = $2,200; break;
```

[†]Unfortunately, C++ has little support for variadic macros such as *kchoose*. For this reason, the current implementation supports only a limited number of arguments (up to 5, to be precise). We plan to improve this implementaton in the future.

```
    default: tuition = 0; }
print(tuition);
```

The above code obviously requires a form of bounded-choice interaction rather than unbounded quantification and can thus be greatly simplified using the kchoose statement. This is shown below:

```
print("type 1 for english,2 for medical,3 for liberal:");
kchoose(
    major = english; tuition = $2,000,
    major= medical; tuition = $4,000,
    major = liberal; tuition = $2,200);
print(tuition);
```

This program expresses the task of the user choosing one among three majors. Note that this program is definitely better than the above: it is concise, much easier to read/write/use, and less error-prone. The system now requests the user to select one (by typing 1, 2 or 3) among three majors. After it is selected, the system displays the amount of the tuition.

4. Empirical Study

This section provides some empirical study comparing two languages, namely C and C^{BI}.

It has the following features:

- The same program is considered for each language. A typical ATM machine in Korea has a sequence of 3 interactions for cash deposit, 4 for cash withdrawal, and 2 for checking balance. The program we require is an implementation of this ATM machine using seven major procedures (deposit, withdrawal, balance, password processing, etc). Overall, there are five occurrences of bounded-choice interactions in the program.
- For each language, we analyze five best implementations of the program by Computer Science undergraduate students in our Software Engineering classes.
- Two different aspects are investigated, namely program length and programming effort.

Program length

The following table shows the numbers of lines of five programs containing a statement, a declaration, or a delimiter such as a closing brace.

	program lines	average line
C	(127, 130, 135, 142, 154)	137.6
C^{BI}	(113, 115, 123, 129, 132)	122.4

We see that C codes are typically 10% longer than C^{BI}.

Work time and productivity

The following table shows the total work time for designing, writing, and testing the program as measured by us in the classes.

	programming hours	average hour
C	(1.6, 1.8, 2.4, 2.5, 2.8)	2.2
C^{BI}	(1.2, 1.4, 1.5, 1.6, 2.1)	1.5

As we see, C^{BI} takes less than 70% as long as C.

5. Conclusion

In this paper, we have extended core C by adding a bounded-choice statement. This extension allows $kchoose(G_1, \ldots, G_n)$, where each G_i is a statement. This statement makes it possible for the core C to model decision steps from the user.

The $kchoose(G_1, \ldots, G_n)$ construct allows only a simple form of user input, *i.e.*, natural numbers. A more flexible form of user input can be obtained using a parameterized *kchoose* statement of the form $kchoose(c_1 : G_1, \ldots, c_n : G_n)$, where $c_1, \ldots, c_n$ are (pairwise disjoint) strings. The semantics is that if some string c_i is typed, then G_i will be executed. Thus, the latter allows the user to type more symbolic names rather than just numbers. We plan to investigate this possibility in the future.

Although we focused on the keyboard input, it is straightforward to extend our idea to the mouse input, which plays a central role in smartphone applications. For example, the statement $mchoose(button_1 : G_1, \ldots, button_n : G_n)$ where each button is a graphic component located at some area can be adopted. The idea is that if $button_i$ is clicked, then G_i will be executed. It can be easily seen that this statement will greatly simplify smartphone programming.

We plan to compare our construct to another popular approach: the monad construct in functional languages. We also plan to connect our execution model to Japaridze's elegant Computability Logic [2], [3], which has many interesting applications (for example, see [4]) in information technology.

Acknowledgements

We thank the anonymous reviewer for several helpful comments including the parameterized *kchoose* statement. This work was supported by Dong-A University Research Fund.

References

[1] G. Kahn, "Natural semantics," Proc. 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, no.1-3, 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.-Y. Park, "Improving robustness via disjunctive statements in imperative programming," IEICE Trans. Inf. & Syst., vol.E96-D, no.9, pp.2036–2038, Sept. 2013.
[5] J.S. 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, no.1-2, pp.125–157, 1991.

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