1 Introduction

You will undertake a few simple problems in order to develop basic skills with control operators and reduction semantics.

2 Description

- (Adapted from Exercises 4 and 5 of Chapter 3 from Programming Languages: Build, Prove, and Compare (pp. 299–300).)

Control operators combine in tricky ways, both with each other and with regular operators. See what you make of these combinations.

- Is there a simpler expression that behaves the same as \((\text{return} \ (\text{return} \ 1))\) in every context?
- Is there a simpler expression that behaves the same as \((\text{return} \ (\text{continue}))\) in every context?
- Is there a simpler expression that behaves the same as \((\text{return} \ (\text{break}))\) in every context?
- Is there a simpler expression that behaves the same as \((\text{throw} \ (\text{return} \ 3))\) in every context?
- Is there a simpler expression that behaves the same as \((\text{throw} \ (\text{throw} \ 4))\) in every context?
- Is it true that expression \((\text{begin} \ e_1 \ e_2 \ \cdots \ (\text{throw} \ \text{exn}) \ \cdots \ e_n)\) can always be replaced by \((\text{begin} \ e_1 \ e_2 \ \cdots \ (\text{throw} \ \text{exn}))\) without changing the behavior of the program?
- Can the function application \((e \ e_1 \ e_2 \ \cdots \ (\text{throw} \ \text{exn}) \ \cdots \ e_n)\) be simplified without changing the behavior of the program? If so, what is the simpler version? If not, why not?

Combinations of try-catch with control operators are a bit less tricky. What outcome do you expect from these expressions?

- (try-catch
  (try-catch (throw 'inner) (lambda (exn) (list2 'caught exn)))
  (throw 'outer))
- (try-catch
  (throw (try-catch 'exception (lambda (exn) (list2 'inner-caught exn))))
  (lambda (exn) (list2 'outer-caught exn)))
A number of languages (for example, the POSIX shell language) extend break and continue to exit or resume from within a nest of several loops.

Suppose that we extend (simplified) µScheme+ with two new expression forms:

Concrete Syntax | Abstract Syntax
--- | ---

\[
\text{Concrete Syntax: } \exp \ e = \cdots \\
\quad | \ (\text{break} \ e_k) \\
\quad | \ (\text{continue} \ e_k)
\]

\[
\text{Abstract Syntax: } \text{Exp} \ e = \cdots \\
\quad | \ \text{BREAKK} (e_k) \\
\quad | \ \text{CONTINUEK} (e_k)
\]

Informally, the \((\text{break} \ e_k)\) expression evaluates \(e_k\) to a number \(k\) greater than or equal to 1 and then exits the \(k^{th}\) enclosing loop. Similarly, the \((\text{continue} \ e_k)\) expression evaluates \(e_k\) to a number \(k\) greater than or equal to 1 and then resumes the \(k^{th}\) enclosing loop. Thus, \((\text{break} 1)\) behaves like \((\text{break})\) and \((\text{continue} 1)\) behaves like \((\text{continue})\).

Extend the reduction semantics for µScheme+ to support these new expression forms. You will need to add one or more frame forms as well as one or more inference rules; you should not need to modify or remove any of the existing inference rules.

Based on the experience of the previous problem, it would be a simple exercise to extend µScheme+ with a new expression form \((\text{return} \ e \ e_k)\) that evaluates \(e\) to value and then evaluates \(e_k\) to a number \(k\) greater than or equal to 1 and then returns \(v\) to the \(k^{th}\) enclosing function call. Thus, \((\text{return} \ 1)\) behaves like \((\text{return})\).

However, I know of no language that provides a construct to return a value to anything other than the immediately enclosing function call, although games with global mutable state and try-catch/throw, call/cc, or prompt/control may be played to provide similar functionality. Also, see the Multi-return function call work of Olin Shivers and David Fisher (ICFP’04, JFP 16:4–5).

Do you think that this would be a useful feature? Why or why not? Do your answer and/or reasons change if considering a dynamically-typed language or a statically-typed language?

3 Requirements and Submission

You may use the reference interpreter (see Appendix A), but there may only be one active laptop in each group.

At the end of class, submit the group’s solutions as hard-copy; be sure to include the names of all group members in the submission.

A Interpreter

A reference µScheme+ interpreter is available on the CS Department Linux systems (e.g., glados.cs.rit.edu and queeg.cs.rit.edu and ICLs 1 and 2) at:

\[
/usr/local/pub/mtf/plc/bin/uschemeplus
\]

Use the reference interpreter to check your code.
Abstract Syntax of (simplified) µScheme+

Exp  $e$ = NUME($n$)  Val  $v$ = NUMV($n$)
| BOOLE($b$)  | BOOLV($b$)
| VAR($x$)  | CLOSURE($x, e, \rho$)
| SET($x, e_x$)  |  
| IF($e_c, e_t, e_f$)  | Ctrl  $c$ = BREAKING
| WHILE($e_c, e_b$)  | CONTINUING
| BREAK  | RETURNING($v$)
| CONTINUE  | THROWSING($v$)
| BEGIN($e_1, e_2$)  |  
| LAMBDA($x, e$)  |  
| APPLY($e_f, e_a$)  |  
| RETURN($e$)  |  
| LET($x, e, e_b$)  |  
| LETREC($x, e, e_b$)  |  
| TRY−CATCH($e_b, e_h$)  |  
| THROW($e$)  |  

Frame  $F$ = SET($x, e_x$)  
| IF($e_c, e_t, e_f$)  
| WHILE($e_c, e_b$)  
| WHILE($e_c, e_b$)  
| BEGIN($e_1, e_2$)  
| BEGIN($e_1, e_2$)  
| APPLY($e_f, e_a$)  
| APPLY($e_f, e_a$)  
| CALLENV($\rho$)  
| RETURN($e$)  
| LET($x, e_x, e_b$)  
| LETREC($x, e_x, e_b$)  
| LETENV($\rho$)  
| TRY−CATCH($e_b, e_h$)  
| TRY−CATCH($e_b, v_h$)  
| THROW($e$)  

3
C Reduction Semantics of (simplified) $\mu$Scheme+

\[
\langle e, \rho, \sigma, S \rangle \rightarrow \langle e'/v'/c'/\rho'/\sigma'/S' \rangle
\]

\[
\langle \text{NUME}(n), \rho, \sigma, S \rangle \rightarrow \langle \text{NUMV}(n), \rho, \sigma, S \rangle
\]

\[
x \in \text{dom } \rho \quad \rho(x) \in \text{dom } \sigma \quad v = \sigma(\rho(x))
\]

\[
\langle \text{VAR}(x), \rho, \sigma, S \rangle \rightarrow \langle v, \rho, \sigma, S \rangle
\]

\[
\langle \text{IF}(e_c, e_t, e_f), \rho, \sigma, S \rangle \rightarrow \langle e_c, \rho, \sigma, \text{IF}(e_c, e_t, e_f) :: S \rangle
\]

\[
\langle \text{BREAK}, \rho, \sigma, S \rangle \rightarrow \langle \text{BREAKING}, \rho, \sigma, S \rangle
\]

\[
\langle \text{APPLY}(e_f, e_a), \rho, \sigma, S \rangle \rightarrow \langle e_f, \rho, \sigma, \text{APPLY}(e_f, e_a) :: S \rangle
\]

\[
\ell \notin \text{dom } \sigma
\]

\[
\langle \text{LETREC}(x, e_x, e_h), \rho, \sigma, S \rangle \rightarrow \langle e_x, \rho \{ x \mapsto \ell \}, \sigma \{ \ell \mapsto \text{unspec} \}, \text{LETREC}(x, e_x, e_h) :: \text{LETVN}(\rho) :: S \rangle
\]

\[
\langle \text{TRY}-\text{CATCH}(e_b, e_h), \rho, \sigma, S \rangle \rightarrow \langle e_h, \rho, \sigma, \text{TRY}-\text{CATCH}(e_b, e_h) :: S \rangle
\]

\[
\langle \text{THROW}(e), \rho, \sigma, S \rangle \rightarrow \langle e, \rho, \sigma, \text{THROW}(e) :: S \rangle
\]
\[
\langle v, \rho, \sigma, S \rangle \to \langle e'/v'/\ell', \rho', \sigma', S' \rangle
\]

\[
x \in \text{dom } \rho \quad \rho(x) \in \text{dom } \sigma
\]
\[
\langle v_x, \rho, \sigma, \text{SET}(x, (e_x) :: S) \rangle \to \langle v_x, \rho, \sigma\{\rho(x) \mapsto v_x\}, S \rangle
\]

\[
v_c \neq \text{BOOLV}(\#f)
\]
\[
\langle v_c, \rho, \sigma, \text{IF}(e_c, e_t, e_f) :: S \rangle \to \langle e_t, \rho, \sigma, S \rangle
\]
\[
v_c = \text{BOOLV}(\#f)
\]
\[
\langle v_c, \rho, \sigma, \text{IF}(e_c, e_t, e_f) :: S \rangle \to \langle e_f, \rho, \sigma, S \rangle
\]

\[
x \in \text{dom } \rho \quad \rho(x) \in \text{dom } \sigma
\]
\[
\langle v_x, \rho, \sigma, \text{SET}(x, (e_x) :: S) \rangle \to \langle v_x, \rho, \sigma\{\rho(x) \mapsto v_x\}, S \rangle
\]

\[
v_c = \text{BOOLV}(\#f)
\]
\[
\langle v_c = \text{BOOLV}(\#f), \rho, \sigma, S \rangle \to \langle \ell, \rho, \sigma, S \rangle
\]

\[
v_c = \text{BOOLV}(\#f)
\]
\[
\langle v_c = \text{BOOLV}(\#f), \rho, \sigma, S \rangle \to \langle \ell, \rho, \sigma, S \rangle
\]

\[
v_c = \text{BOOLV}(\#f)
\]
\[
\langle v_c = \text{BOOLV}(\#f), \rho, \sigma, S \rangle \to \langle \ell, \rho, \sigma, S \rangle
\]

\[
v_c = \text{BOOLV}(\#f)
\]
\[
\langle v_c = \text{BOOLV}(\#f), \rho, \sigma, S \rangle \to \langle \ell, \rho, \sigma, S \rangle
\]
\[ \langle c, \rho, \sigma, S \rangle \rightarrow \langle c'/v'/c', \rho', \sigma', S' \rangle \]

\[ \langle \text{BREAKING}, \rho, \sigma, \text{WHILE}(e_c; (e_b)) :: S \rangle \rightarrow \langle \text{BOOLV}(#f), \rho, \sigma, S \rangle \]

\[ \langle \text{CONTINUING}, \rho, \sigma, \text{WHILE}(e_c; (e_b)) :: S \rangle \rightarrow \langle e_c, \rho, \sigma, \text{WHILE}(e_c; e_b) :: S \rangle \]

\[ \langle \text{BREAKING}, \rho, \sigma, \text{LETENV}(\rho') :: S \rangle \rightarrow \langle \text{BREAKING}, \rho', \sigma, S \rangle \]

\[ \langle \text{CONTINUING}, \rho, \sigma, \text{LETENV}(\rho') :: S \rangle \rightarrow \langle \text{CONTINUING}, \rho', \sigma, S \rangle \]

\[ F \neq \text{WHILE}(\_, \_) \quad F \neq \text{LETENV}(\_) \quad F \neq \text{CALLENV}(\_) \]
\[ \langle \text{BREAKING}, \rho, \sigma, F :: S \rangle \rightarrow \langle \text{BREAKING}, \rho, \sigma, S \rangle \]

\[ F \neq \text{WHILE}(\_, \_) \quad F \neq \text{LETENV}(\_) \quad F \neq \text{CALLENV}(\_) \]
\[ \langle \text{CONTINUING}, \rho, \sigma, F :: S \rangle \rightarrow \langle \text{CONTINUING}, \rho, \sigma, S \rangle \]

\[ \langle \text{RETURNING}(v), \rho, \sigma, \text{CALLENV}(\rho') :: S \rangle \rightarrow \langle v, \rho', \sigma, S \rangle \]

\[ \langle \text{RETURNING}(v), \rho, \sigma, \text{LETENV}(\rho') :: S \rangle \rightarrow \langle \text{RETURNING}(v), \rho', \sigma, S \rangle \]

\[ F \neq \text{LETENV}(\_) \quad F \neq \text{CALLENV}(\_) \]
\[ \langle \text{RETURNING}(v), \rho, \sigma, F :: S \rangle \rightarrow \langle \text{RETURNING}(v), \rho, \sigma, S \rangle \]

\[ \langle \text{THROWING}(v), \rho, \sigma, \text{TRY} - \text{CATCH}(e_b; v_h) :: S \rangle \rightarrow \langle v, \rho, \sigma, \text{APPLY}(v_h, \_) :: S \rangle \]

\[ \langle \text{THROWING}(v), \rho, \sigma, \text{LETENV}(\rho') :: S \rangle \rightarrow \langle \text{THROWING}(v), \rho', \sigma, S \rangle \]

\[ \langle \text{THROWING}(v), \rho, \sigma, \text{CALLENV}(\rho') :: S \rangle \rightarrow \langle \text{THROWING}(v), \rho', \sigma, S \rangle \]

\[ F \neq \text{TRY} - \text{CATCH}(\_, \_) \quad F \neq \text{LETENV}(\_) \quad F \neq \text{CALLENV}(\_) \]
\[ \langle \text{THROWING}(v), \rho, \sigma, F :: S \rangle \rightarrow \langle \text{THROWING}(v), \rho, \sigma, S \rangle \]