Continuations, aka Contradictions, aka Contexts, aka staCks*

Robert Harper

Spring 2024

1 Introduction

Computational effects can be roughly classified into two kinds, *control effects* and *storage effects*. Control effects concern deviations from the usual flow of control to effect transfers to other parts of a program. Storage effects concern in-place alterations to data structures that characterize imperative programming. Effects require that the order of execution be made precise, so that it is clear when and where a transfer of control, or a mutation of storage, occurs. One might say, if you don't know where you are, you can't know where you are going.

This note gives a *lax* formulation of control effects. The statics is structured similarly to the formulation given in Harper (2022), distinguishing *pure* expressions from *impure* computations, with the two levels being linked by the *lax modality*. The dynamics is given by a *stack machine* that makes the execution state of a program explicit as a data structure that can be "reified" as a value of a type of *continuations*.[1] Continuations are the "master" control effect in that capture precisely the execution state, which may be later restored by a control transfer operation.

The main result of this note is a proof of termination for the extension of the typed λ-calculus with continuations. Seeing as how these provide a form of "goto" in a language, it might be supposed that they can be used to implement loops—even infinite ones! However, this is not the case. The proof of this fact makes use of an extension of Tait's method to account for reified control. Curiously, as will become apparent from the type system, coninuations provide a computational interpretation of *classical* logic, which is ordinarily conceived as Boolean (two-valued, true or false). But as Griffin discovered (Griffin, 1989), they may also be understood as providing the computational meaning of classical proofs—rendering them constructive after all.

[1]As the title suggests, there are many synonyms for the word "continuation," some of which are called out there.

LAX-I
$$\frac{\Gamma \vdash E \mathrel{\dot\sim} A}{\Gamma \vdash \mathsf{comp}(E) : \mathsf{comp}(A)}$$

RET
$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{ret}(M) \mathrel{\dot\sim} A}$$

LAX-E
$$\frac{\Gamma \vdash M : \mathsf{comp}(A) \qquad \Gamma, x : A \vdash E \mathrel{\dot\sim} B}{\Gamma \vdash \mathsf{bnd}(M;x.E) \mathrel{\dot\sim} B}$$

CONT-E
$$\frac{\Gamma \vdash M_1 : \mathsf{cont}(A_2) \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \mathsf{throw}(M_1;M_2) \mathrel{\dot\sim} B}$$

CONT-I
$$\frac{\Gamma, x : \mathsf{cont}(A) \vdash E \mathrel{\dot\sim} A}{\Gamma \vdash \mathsf{letcc}(x.E) \mathrel{\dot\sim} A}$$

EMP
$$\frac{}{\bullet \div \mathsf{ans}}$$

FRAME
$$\frac{x : A \vdash E \mathrel{\dot\sim} B \qquad K \div B}{K \circ x.E \div A}$$

CONT
$$\frac{K \div A}{\Gamma \vdash \mathsf{cont}(K) : \mathsf{cont}(A)}$$

Figure 1: Statics (Selected Rules)

2 Types for Continuations

The main ideas are well-illustrated for a language with product, (partial) function, continuation, and modal types.

$$\begin{aligned}
A &::= \mathsf{ans} \mid \mathsf{unit} \mid A_1 \times A_2 \mid A_1 \to A_2 \mid \mathsf{cont}(A) \mid \mathsf{comp}(A) \\
M &::= x \mid \mathsf{yes} \mid \mathsf{no} \mid \langle\rangle \mid \langle M_1, M_2\rangle \mid M \cdot 1 \mid M \cdot 2 \mid \lambda(x.M_2) \mid \mathsf{ap}(M_1;M_2) \mid \mathsf{cont}(K) \mid \mathsf{comp}(E) \\
E &::= \mathsf{ret}(M) \mid \mathsf{bnd}(M;x.E) \mid \mathsf{letcc}(x.E) \mid \mathsf{throw}(M_1;M_2) \\
K &::= \bullet \mid K \circ x.E
\end{aligned}$$

The last category is that of *stacks*, which take on a particularly simple form in this setting. The terms are augmented with a value, $\mathsf{cont}(K)$, representing a stack as a form of value.[2]

The statics defines these judgment forms:

1. Expression typing: $\Gamma \vdash M : A$.
2. Computation typing: $\Gamma \vdash E \mathrel{\dot\sim} A$.
3. Continuation typing: $K \div A$.

Notice that well-formed continuations have no free variables. Selected rules of the statics defining these judgments are given in Figure 1.

3 Dynamics of Continuations

Eager evaluation of terms, written $M \Downarrow V$, where V val, is defined as in Harper (2016), either in terms of a structural operational semantics, or directly by rules.

[2]Traditionally these are called “first-class” continuations, or stacks, using long-obsolete terminology signalling their use.

$$\frac{}{\mathsf{cont}(K)\ \mathsf{val}}\ \textsc{cont-val} \qquad \frac{}{\mathsf{comp}(E)\ \mathsf{val}}\ \textsc{comp-val}$$

$$\frac{M \Downarrow V}{K \triangleright \mathsf{ret}(M) \longmapsto K \triangleleft V}\ \textsc{ret} \qquad \frac{M \Downarrow \mathsf{comp}(E)}{K \triangleright \mathsf{bnd}(M; x.E') \longmapsto K \circ x.E' \triangleright E}\ \textsc{bnd}$$

$$\frac{M_1 \Downarrow \mathsf{cont}(K_1)}{K \triangleright \mathsf{throw}(M_1; M_2) \longmapsto K_1 \triangleright \mathsf{ret}(M_2)}\ \textsc{throw} \qquad \frac{}{K \triangleright \mathsf{letcc}(x.E) \longmapsto K \triangleright [\mathsf{cont}(K)/x]E}\ \textsc{letcc}$$

$$\frac{}{\bullet \triangleright E\ \mathsf{initial}}\ \textsc{init} \qquad \frac{V\ \mathsf{val}}{\bullet \triangleleft V\ \mathsf{final}}\ \textsc{final} \qquad \frac{V\ \mathsf{val}}{K \circ x.E \triangleleft V \longmapsto K \triangleright [V/x]E}\ \textsc{pop}$$

Figure 2: Dynamics (Selected Rules)

Expression evaluation, $M \Downarrow V$, is defined for V val to mean $M \longmapsto^{*} V$ according to the usual dynamics. The dynamics of computations, which may have control effects, is given in terms of a *stack machine* with two forms of state:

1. Evaluate E on stack K, written $K \triangleright E$, and
2. Return value V to stack K, written $K \triangleleft V$.

Selected rules of the stack dynamics is given in Figure 2. Plainly, bnd is the only source of sequencing; it pushes its continuation onto the stack and evaluates the encapsulated computation. The dynamics of throw is akin to that of ret, except that it passes control to the given continuation, passing the given value. Finally, letcc reifies the stack as a value, which is passed by substitution to its body.

Exercise 1. *Extend the statics and dynamics with nullary and binary sums. Pay careful attention to extending, if necessary, the forms of stack.*

4 Termination

Figure 3 defines the following Tait computability predicates:

1. On expressions: $\mathsf{HT}_A(M)$, hereditary termination of M at type A.
2. On computations: $\widetilde{\mathsf{HT}}_A(E)$, hereditary termination of E at type A.

$$
\begin{aligned}
\mathsf{HT}_{\mathsf{ans}}(M) &\text{ iff } M \Downarrow \mathsf{yes}, \text{ or } M \Downarrow \mathsf{no} \\
\mathsf{HT}_{\mathsf{unit}}(M) &\text{ iff } M \Downarrow \langle\rangle \\
\mathsf{HT}_{A_1 \times A_2}(M) &\text{ iff } M \Downarrow \langle M_1, M_2\rangle \text{ with } \mathsf{HT}_{A_1}(M_1) \text{ and } \mathsf{HT}_{A_2}(M_2) \\
\mathsf{HT}_{A_1 \to A_2}(M) &\text{ iff } M \Downarrow \lambda(x.M_2) \text{ and if } \mathsf{HT}_{A_1}(M_1) \text{ then } \mathsf{HT}_{A_2}([M_1/x]M_2) \\
\mathsf{HT}_{\mathsf{cont}(A)}(M) &\text{ iff } M \Downarrow \mathsf{cont}(K) \text{ and } \overline{\mathsf{HT}}_A(K) \\
\mathsf{HT}_{\mathsf{comp}(A)}(M) &\text{ iff } M \Downarrow \mathsf{comp}(E) \text{ and } \widetilde{\mathsf{HT}}_A(E)
\end{aligned}
$$

$$\widetilde{\mathsf{HT}}_A(E) \text{ iff } \overline{\mathsf{HT}}_A(K) \text{ implies } K \rhd E \Downarrow$$

$$\overline{\mathsf{HT}}_A(K) \text{ iff } \mathsf{HT}_A(V) \text{ implies } K \lhd V \Downarrow$$

Figure 3: Hereditary Termination Predicates

3. On control stacks: $\overline{\mathsf{HT}}_A(K)$, hereditary termination of K at type A.

Computablity of terms is defined by induction on the stucture their type, making use of of the other two predicates at subsidiary types. Computability of computations is defined in terms of stacks, namely that when executed on a computable stack, the computation terminates. Computability of stacks is defined to mean that, when passed a computable value of its type, the computation terminates.

Lemma 1 (Head Expansion for Expressions). *If $HT_A(M)$ and $M' \longmapsto M$, then $HT_A(M')$.*

Lemma 2 (Termination for Expressions). *If $HT_A(M)$, then $M \Downarrow$.*

Define $\Gamma \gg M \in A$ to mean that if $\mathsf{HT}_\Gamma(\gamma)$, then $\mathsf{HT}_A(\hat{\gamma}(M))$, and similarly define $\Gamma \gg E \in \tilde{A}$ to mean that if $\mathsf{HT}_\Gamma(\gamma)$, then $\widetilde{\mathsf{HT}}_A(E)$, and write $K \in \overline{A}$ to mean $\overline{\mathsf{HT}}_A(K)$.

Theorem 3. *Formally typable terms, expressions, and stacks are computable at their classifying type:*

1. *If $\Gamma \vdash M : A$, then $\Gamma \gg M \in A$.*

2. *If $\Gamma \vdash E \mathrel{\dot{\sim}} A$, then $\Gamma \gg E \in \tilde{A}$.*

3. *If $K \div A$, then $K \in \overline{A}$.*

Proof. By induction on typing. Here are some representative cases of the proof. Throughout the notation $\widehat{M}$ is short for $\hat{\gamma}(M)$ when γ is clear from context.

Application: $\Gamma \vdash \mathsf{ap}(M;M_2) : A$ because $\Gamma \vdash M : A_2 \to A$ and $\Gamma \vdash M_2 \mathrel{\dot{\sim}} A_2$. Suppose that $\mathsf{HT}_\Gamma(\gamma)$; the goal is to show that $\mathsf{HT}_A(\mathsf{ap}(\widehat{M};\widehat{M_2}))$. By inductive assumptions $\mathsf{HT}_{A_2 \to A}(\widehat{M})$ and $\mathsf{HT}_{A_2}(\widehat{M_2})$, from which the result follows immediately.

Ret: $\Gamma \vdash \mathsf{ret}(M) \mathrel{\dot{\sim}} A$ because $\Gamma \vdash M : A$. Fix $\mathsf{HT}_\Gamma(\gamma)$, so that by induction $\mathsf{HT}_A(\widehat{M})$, and hence by Lemma 2 $\widehat{M} \Downarrow V$ for some V. Suppose that $\overline{\mathsf{HT}}_A(K)$; it suffices to show that $K \rhd \widehat{M} \Downarrow$. But $K \rhd \widehat{M} \longmapsto K \lhd V$, by definition of transition, and $K \lhd V \Downarrow$ by assumption on K.

Bind: $\Gamma \vdash \mathsf{bnd}(M;x.E') \mathrel{\dot\sim} B$ because $\Gamma \vdash M : \mathsf{comp}(A)$ and $\Gamma, x : A \vdash E' \mathrel{\dot\sim} B$. Fix $\mathsf{HT}_\Gamma(\gamma)$, and suppose that $\overline{\mathsf{HT}}_B(K)$, with the goal to show that $K \rhd \mathsf{bnd}(\widehat{M};x.\widehat{E}') \Downarrow$. By the first inductive hypothesis $\mathsf{HT}_{\mathsf{comp}(A)}(\widehat{M})$, so $\widehat{M} \Downarrow \mathsf{comp}(E)$ for some $E \mathrel{\dot\sim} A$ such that $\widetilde{\mathsf{HT}}_A(E)$. But then $K \rhd \mathsf{bnd}(\widehat{M};x.\widehat{E}') \longmapsto K \circ x.\widehat{E}' \rhd E$. By the second inductive hypothesis if $\mathsf{HT}_A(V)$, then $\widetilde{\mathsf{HT}}_B([V/x]\widehat{E}')$, and so $\overline{\mathsf{HT}}_A(K \circ x.\widehat{E}')$, using also the assumption on K. But then $K \circ x.\widehat{E}' \rhd E \Downarrow$, as desired.

Letcc: $\Gamma \vdash \mathsf{letcc}(x.E) \mathrel{\dot\sim} A$ because $\Gamma, x : \mathsf{cont}(A) \vdash E \mathrel{\dot\sim} A$. Suppose that $\mathsf{HT}_\Gamma(\gamma)$, and $\overline{\mathsf{HT}}_A(K)$, with the goal to show that $K \rhd \mathsf{letcc}(x.\widehat{E}) \Downarrow$. By the inductive hypothesis $\widetilde{\mathsf{HT}}_A([\mathsf{cont}(K)/x]\widehat{E})$, so that $K \rhd \mathsf{letcc}(x.\widehat{E}) \longmapsto K \rhd [\mathsf{cont}(K)/x]\widehat{E}' \Downarrow$, as desired.

Throw: $\Gamma \vdash \mathsf{throw}(M_1;M_2) \mathrel{\dot\sim} B$ because $\Gamma \vdash M_1 : \mathsf{cont}(A)$ and $\Gamma \vdash M_2 : A$. Fix $\mathsf{HT}_\Gamma(\gamma)$ and suppose that $\overline{\mathsf{HT}}_B(K)$, with the goal to show that $K \rhd \mathsf{throw}(\widehat{M_1};\widehat{M_2}) \Downarrow$. By the first inductive assumption $\mathsf{HT}_{\mathsf{cont}(A)}(\widehat{M_1})$, so $\widehat{M_1} \Downarrow \mathsf{cont}(K')$ with $\overline{\mathsf{HT}}_A(K')$. By the second inductive hypothesis $\mathsf{HT}_A(\widehat{M_2})$. But $K \rhd \mathsf{throw}(\widehat{M_1};\widehat{M_2}) \longmapsto K' \rhd \widehat{M_2}$, which terminates, as desired.

□

Exercise 2. *Do the cases of the proof of the fundamental theorem for pairing and first projection.*

Corollary 4. *If $E \mathrel{\dot\sim} \mathit{ans}$, then $\bullet \rhd E \Downarrow$.*

Proof. The empty stack is computable, $\overline{\mathsf{HT}}_{\mathsf{ans}}(\bullet)$, so $\bullet \rhd E \longmapsto^* \bullet \lhd V$. □

5 Equational Laws

The equational laws governing continuations are given in Figure 4. Rule LETCC-RET abandons a letcc when the body is returning a value that is well-formed in the surrounding context. (If, say, M were a λ-abstraction, it could contain references to k, and cannot therefore be moved out of the scope of k.) Rule LETCC-THROW states that when a throw reaches its target binding, it can be considered to return the thrown value to the surrounding context, within the limitations expressed by Rule LETCC-RET. Rule LETCC-FUSE collapses two consecutive continuations into one, on the grounds that at run-time they will be bound to the same stack. Rule BND-THROW specifies that if the first command in a sequence is a throw, then the second is abandoned before it is executed. Consecutive uses of this rule amount to the propagation of a continuation upward through the frames of the run-time stack.

Rule LETCC-BND characterizes the interaction between letcc and bnd in the case that the continuation is only used within the body of the bnd: in that case, the declaration of the continuation may be moved within the body of the bnd, it being the same continuation in any case, namely that of the surrounding context. Rule BND-LETCC characterizes the interaction between letcc and bnd when the continuation may be used within the encapsulated computation: because the continuation is known to be $x.E_2$, the effect of a throw to k_1 is to pass the thrown value through the body and to the surrounding context. (The indicated composition of a continuation with an abstractor is a standard construct whose definition is given after the rule.) Notice that if E_1 does not throw to k_1, then its returned value is passed to E_2 and then to the surrounding context, exactly as is would be if that value were thrown to k_1. For this reason the replication of E_2 appears to be unavoidable.[3]

[3]To be sure, the computation E_1' could be wrapped with a throw to k_1, but that, too, would involve replicating E_2.

BND-RET

$$\frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash E \div B}{\Gamma \vdash \mathsf{bnd}(\mathsf{comp}(\mathsf{ret}(M)); x.E) \equiv [M/x]E \div B}$$

RET-BND

$$\frac{\Gamma \vdash E \div A}{\Gamma \vdash E \equiv \mathsf{bnd}(\mathsf{comp}(E); x.\, \mathsf{ret}(x)) \div A}$$

BND-BND

$$\frac{\Gamma \vdash M_1 : \mathsf{comp}(A_1) \qquad \Gamma, x_1 : A_1 \vdash E_2 \div A_2 \qquad \Gamma, x_2 : A_2 \vdash E_3 \div A_3}{\Gamma \vdash \mathsf{bnd}(\mathsf{comp}(\mathsf{bnd}(M_1; x_1.E_2)); x_2.E_3) \equiv \mathsf{bnd}(M_1; x_1.\, \mathsf{bnd}(\mathsf{comp}(E_2); x_2.E_3)) \div A_3}$$

LETCC-RET

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{letcc}(k.\, \mathsf{ret}(M)) \equiv \mathsf{ret}(M) \div A}$$

LETCC-THROW

$$\frac{\Gamma, k : \mathsf{cont}(A) \vdash M : A}{\Gamma \vdash \mathsf{letcc}(k.\, \mathsf{throw}(k; M)) \equiv \mathsf{letcc}(k.\, \mathsf{ret}(M)) \div A}$$

LETCC-FUSE

$$\frac{\Gamma, k_1 : \mathsf{cont}(A), k_2 : \mathsf{cont}(A) \vdash E \div A}{\Gamma \vdash \mathsf{letcc}(k_1.\, \mathsf{letcc}(k_2.E)) \equiv \mathsf{letcc}(k.[k, k/k_1, k_2]E) \div A}$$

BND-THROW

$$\frac{\Gamma, k_2 : \mathsf{cont}(A_2) \vdash M : A_2 \qquad \Gamma, k_2 : \mathsf{cont}(A_2), x_1 : A_1 \vdash E_2 \div A_2}{\Gamma, k_2 : \mathsf{cont}(A_2) \vdash \mathsf{bnd}(\mathsf{comp}(\mathsf{throw}(k_2; M)); x_1.E_2) \equiv \mathsf{throw}(k_2; M) \div A_2}$$

LETCC-BND

$$\frac{\Gamma \vdash M_1 : \mathsf{comp}(A_1) \qquad \Gamma, k_2 : \mathsf{cont}(A_2), x_1 : A_1 \vdash E_2 \div A_2}{\Gamma \vdash \mathsf{letcc}(k_2.\, \mathsf{bnd}(M_1; x_1.E_2)) \equiv \mathsf{bnd}(M_1; x_1.\, \mathsf{letcc}(k_2.E_2)) \div A_2}$$

BND-LETCC

$$\frac{\begin{array}{c}\Gamma, k_1 : \mathsf{cont}(A_1) \vdash E_1 \div A_1 \qquad \Gamma, x_1 : A_1 \vdash E_2 \div A_2 \\ E_1' \overset{\text{def}}{=} \mathsf{bnd}(\mathsf{comp}(k_2 \circ x_1.E_2); k_1.E_1) \\ k_2 \circ x_1.E_2 \overset{\text{def}}{=} \mathsf{letcc}(r.\, \mathsf{bnd}(\mathsf{comp}(\mathsf{letcc}(k_1.\, \mathsf{throw}(r; k_1))); x_1.\, \mathsf{bnd}(\mathsf{comp}(E_2); x_2.\, \mathsf{throw}(k_2; x_2))))\end{array}}{\Gamma \vdash \mathsf{bnd}(\mathsf{comp}(\mathsf{letcc}(k_1.E_1)); x_1.E_2) \equiv \mathsf{letcc}(k_2.\, \mathsf{bnd}(\mathsf{comp}(E_1'); x_1.E_2)) \div A_2}$$

Figure 4: Selected Equational Laws

$$
\begin{aligned}
M \doteq M' \in \mathsf{unit} &\text{ iff } M, M' \Downarrow \langle\rangle \\
M \doteq M' \in \mathsf{ans} &\text{ iff } M, M' \Downarrow \mathsf{yes},\ \text{or } M, M' \Downarrow \mathsf{no} \\
M \doteq M' \in A_1 \times A_2 &\text{ iff } M \Downarrow \langle M_1, M_2\rangle,\ M' \Downarrow \langle M_1', M_2'\rangle,\ M_1 \doteq M_1' \in A_1 \text{ and } M_2 \doteq M_2' \in A_2 \\
M \doteq M' \in A_1 \to A_2 &\text{ iff } M \Downarrow \lambda(x.M_2),\ M' \Downarrow \lambda(x.M_2'), \\
&\qquad M_1 \doteq M_1' \in A_1 \text{ implies } \mathsf{ap}(M;M_1) \doteq \mathsf{ap}(M';M_1') \in A_2 \\
M \doteq M' \in \mathsf{cont}(A) &\text{ iff } M \Downarrow \mathsf{cont}(K),\ M' \Downarrow \mathsf{cont}(K'), \text{and } K \doteq K' \in \overline{A} \\
M \doteq M' \in \mathsf{comp}(A) &\text{ iff } M \Downarrow \mathsf{comp}(E),\ M' \Downarrow \mathsf{comp}(E'),\ \text{and } E \doteq E' \in \tilde{A}
\end{aligned}
$$

$$E \doteq E' \in \tilde{A} \text{ iff } K \doteq K' \in \overline{A} \text{ implies } K \rhd E \downarrow K' \rhd E'$$

$$K \doteq K' \in \overline{A} \text{ iff } V \doteq V' \in A \text{ implies } K \lhd V \downarrow K' \lhd V'$$

Figure 5: Exact Equality for Continuations

To verify the validity of these equations requires the definition of exact equality of expressions, computations, and continuations, following along the lines of the definition of hereditary termination given in the previous section. The definitions are given in Figure 5. The relation $s \downarrow s'$ for states s and s' means that they have the same outcome, returning the same answer to the empty stack.

Define $\gamma \doteq \gamma' \in \Gamma$ to mean that $\gamma(x) \doteq \gamma'(x) \in A$ for each variable $\Gamma \vdash x : A$. Then $\Gamma \gg M \doteq M' \in A$ means $\hat{\gamma}(M) \doteq \gamma(\hat{M}') \in A$ for every $\gamma \doteq \gamma' \in \Gamma$, and similarly for computations and continuations.

Theorem 5 (Reflexivity). *1. If $\Gamma \vdash M : A$, then $\Gamma \gg M \in A$.*

2. If $\Gamma \vdash E \mathrel{\dot{\div}} A$, then $\Gamma \gg E \in \tilde{A}$.

3. If $K \div A$, then $K \in \overline{A}$.

Exercise 3. *Prove Theorem 5 by induction on the statics, making use of the definitions given in Figure 5. You need only consider the rules for computations, taking those for expressions as given by previous work.*

With this in hand one may prove that the derivable equations given in Figure 4 are valid as exact equations in this sense.

Theorem 6 (Equational Validity). *1. If $\Gamma \vdash M \equiv M' : A$, then $\Gamma \gg M \doteq M' \in A$.*

2. If $\Gamma \vdash E \equiv E' \mathrel{\dot{\div}} A$, then $\Gamma \gg E \doteq E' \in \tilde{A}$.

Proof. Let us consider here rule BND-THROW, leaving the others as exercises, following along similar lines. Suppose that $\gamma \doteq \gamma' \in \Gamma$, and that $K_2 \doteq K_2' \in \overline{A_2}$, with the goal to show that

$$\mathsf{letcc}(k_2.\,\mathsf{bnd}(\mathsf{comp}(\mathsf{throw}(k_2;\hat{\gamma}(M)));x_1.\hat{\gamma}(E_2))) \doteq \mathsf{letcc}(k_2.\,\mathsf{throw}(k_2;\widehat{\gamma'}(M))) \in \tilde{A}_2.$$

Let the left-hand side be denoted by E, and the right-hand side by E'. It suffices to show that

$$K_2 \rhd E \downarrow K_2' \rhd E'$$

under the assumption governing their respective control stacks. Execution of the left-hand state proceeds as follows:

$$
\begin{aligned}
K_2 \rhd E &\longmapsto \mathsf{bnd}(\mathsf{comp}(\mathsf{throw}(\mathsf{cont}(K_2); [\mathsf{cont}(K_2)/k_2]\hat{\gamma}(M))); x_1.[\mathsf{cont}(K_2)/k_2]\hat{\gamma}(E_2)) \\
&\longmapsto K_2 \circ x_1.[\mathsf{cont}(K_2)/k_2]\hat{\gamma}(E_2) \rhd \mathsf{throw}(\mathsf{cont}(K_2); [\mathsf{cont}(K_2)/k_2]\hat{\gamma}(M)) \\
&\longmapsto K_2 \rhd [\mathsf{cont}(K_2)/k_2]\hat{\gamma}(M)
\end{aligned}
$$

Similarly, the right-hand state executes as follows:

$$
\begin{aligned}
K_2' \rhd E' &\longmapsto \mathsf{throw}(\mathsf{comp}(K_2'); [\mathsf{cont}(K_2')/k_2]\widehat{\gamma'}(M)) \\
&\longmapsto K_2' \rhd [\mathsf{cont}(K_2')/k_2]\widehat{\gamma'}(M)
\end{aligned}
$$

But then by the inductive assumptions given by the premises of the rule the result follows immediately. □

Exercise 4. *Prove Theorem 6 by induction on the derivation of the equations, making use of the definitions given in Figure 5.*

6 Connection to Classical Logic

In a celebrated result of Griffin's the foregoing computational interpretation of continuations as stacks, or control contexts, may be seen as providing a computational meaning for classical logic. To see the connection, write $\neg A$ for the type $\mathsf{cont}(A)$, to be thought of as the type of *refutations* of the type A viewed as a proposition. This interpretation arises from the following observations about the types of the continuation primitives, written in propositional form:

- $\mathsf{throw}_B : (\neg A \wedge A) \supset B$.
- $\mathsf{letcc}_A : (\neg A \supset A) \supset A$.

The first is simply negation elimination: from a proof of $\neg A$ and a proof of A one may conclude anything, it being a contradictory situation. The second is called *Peirce's Law*, which is akin to the classical principal of double-negation elimination. It says that if the assumption of $\neg A$ is contradictory (because A can be derived from it), then A must be true. Indeed, under the usual bivalent intepretation of classical logic in terms of truth values, Peirce's Law is a tautology.

What is fascinating, though, is that this principal has *computational content*! To be sure, its content is not "direct" in the usual sense of constructive logic, but is "indirect" in the sense that continuations provide a means of *not returning to the point at which they are invoked*, and hence can be regarded as proving falsity. Thus, operationally, the meaning of letcc is that it provides its body with a proof of $\neg A$ in the form of a continuation. The body may simply return a value of type A, and that is the overall result. But it may also invoke the provided continuation, which it can do *only by possessing a proof of A to throw to it*. Thus, regardless of the control path, direct or indirect, the upshot is that the letcc evaluates to a proof of A. Notice that, implicitly, the law of the excluded middle is lurking here in that a computation can either return normally, or invoke a contnuation to avoid doing so. This is guaranteed by the termination property proved above: the body of the letcc must return, either directly or via a throw to the continuation at the point of invocation.

The "trick" of the above proof is the duality between *proofs* and *refutations* in classical logic. Whereas the role of a proof is to *affirm* the truth of a proposition, the role of a refutation is to *deny* it. Put another way, the proof is an *implementation* of a proposition, and a refutation is its *client*.[4] By having access to the client—in the form of a continuation or stack—a classical proof can "use the client against itself," evading an otherwise impossible obligation in a purely constructive setting.

A telling example is classical proof—viewed as a program—of the law of the excluded middle, stating that $\neg A \vee A$ is true, regardless of what is A. First, the proof blithely asserts that $\neg A$ is true by indicating the left summand and providing a carefully constructed continuation (built using letcc) as evidence for it. A client of the proof may simply fold, never inquiring any further, and the prover succeeds with its bluff. However, the client might respond by raising the prover and *passing a proof of A* to it (using throw). But that means that the client itself knows that A is true! The aforementioned refutation of A then *backtracks* to the point of the bluff, which is of course not allowed in poker, and instead asserts that A is true, providing the proof that the client gave it. The client is none the wiser, because in a purely functional language it has no way to know that the prover has retracted its earlier bet, and instead placed a sure-fire winner instead. It is thus a pusillanimous proof, one that "changes its mind" to avoid the embarrassment of being called with schmaltz for a hand.

Returning to the types-as-propositions of throw and letcc, it is essential, in light of the foregoing, to interpret implication, $A \supset B$, as $A \rightharpoonup B \overset{\text{def}}{=} A \to \mathsf{comp}(B)$, to allow for (classical) proofs that use the client against itself, the essence of indirect proof.

Exercise 5.
1. *Spell out in detail the above-sketched proof of excluded middle, and trace its execution on the stack machine, noting carefully the use of "time travel" to ensure that the bluff cannot fail.*
2. *Give a* direct *proof of $\neg\neg(\neg A \vee A)$, one that does not use letcc (Peirce's Law or its equivalents). The doubly negated form is thereby seen to be constructively valid, meaning that* no instance of excluded middle is refuted by constructive logic.
3. *Give a* direct *proof of $\neg(A \wedge \neg A)$, which is also constructively valid.*
4. *Given an* indirect *proof of $\neg(\neg A \wedge \neg B) \supset (A \vee B)$. Explain why no direct proof is possible, by reducing it to the law of the excluded middle.*

References

Timothy G Griffin. A formulae-as-type notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 47–58, 1989.

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, Cambridge, England, Second edition, 2016.

Robert Harper. Tait computability for sums. Unpublished lecture note., February 2022. URL `https://www.cs.cmu.edu/~rwh/courses/atpl/pdfs/tait-sums.pdf`.

[4]But why is the client a refutation? In classical logic there is no loss of generality, because any proof of a fact can be turned into a refutation of its negation.

Hayo Thielecke. Control effects as a modality. *Journal of Functional Programming*, 19:17–26, 1 2009. ISSN 09567968. doi: 10.1017/S0956796808006734.