Lecture Notes on Continuations

15-317: Constructive Logic
Frank Pfenning

Lecture 12
Thursday, February 23, 2023

1 Introduction

We have discussed classical logic in the two principle calculi we have considered so far. In natural deduction, Gentzen [1935] added the law of the excluded middle as an axiom $A \vee \neg A$ for any proposition A. In the sequent calculus, he allowed sequents with multiple conclusions, $\Gamma \Longrightarrow \Delta$, which may be read as the *conjunction* of the propositions in Γ entails the *disjunction* of the propositions in Δ.

Another important classical principle is *proof by contradiction*, that is,

$$\dfrac{\begin{array}{c}\overline{\neg A\ \mathit{true}^*}\ u\\ \vdots\\ \bot\ \mathit{true}^*\end{array}}{A\ \mathit{true}^*}\ \mathsf{PbC}^u$$

where we wrote true^* to remind ourselves that this is a judgment of *classical* truth. It is an interesting exercise to show that the different forms of obtaining classical logic are equivalent (from the point of view of provability).

There is a slight variant of proof by contraction which we call CC that avoids the use of $\bot$, namely

$$\dfrac{\begin{array}{c}\overline{\neg A\ \mathit{true}^*}\ u\\ \vdots\\ A\ \mathit{true}^*\end{array}}{A\ \mathit{true}^*}\ \mathsf{CC}^u$$

In the next section we will prove that CC and PbC are equivalent. However, the CC rule is interesting because it allows a computational interpretation as the so-called *call-with-current-continuation* construct that is available in the (dynamically typed) programming language Scheme. The relation to classical logic was established by Griffin [1990], echoing previous discoveries such as the Gödel-Gentzen translation [Gödel, 1933, Gentzen, 1936] from Peano (classical) to Heyting (intuitionistic) arithmetic, and Friedman's [1978] translation.

2 Equivalence of PbC and CC

Let's assume the rule of proof by contradiction. We have to show that CC is derivable. The following in such a derivation. It shows that if we have a hypothetical proof of A *true** from $\neg A$ *true** (labeled u) then we can conclude A *true** using the rule of proof by contradiction.

$$\dfrac{\dfrac{\dfrac{}{\neg A\ \mathit{true}^*}\,u \qquad \begin{array}{c}\dfrac{}{\neg A\ \mathit{true}^*}\,u\\ \vdots\\ A\ \mathit{true}^*\end{array}}{\bot\ \mathit{true}^*}\,{\supset}E}{A\ \mathit{true}^*}\,\mathsf{PbC}^u$$

Next, let's assume we have the rule CC. We have to derive the rule PbC.

$$\dfrac{\dfrac{\begin{array}{c}\dfrac{}{\neg A\ \mathit{true}^*}\,u\\ \vdots\\ \bot\ \mathit{true}^*\end{array}}{A\ \mathit{true}^*}\,\bot E}{A\ \mathit{true}^*}\,\mathsf{CC}^u$$

3 Evaluation Contexts

Our current definition of the dynamics for a functional language is based on two basic judgments: $M \longrightarrow M'$ and M *value*, plus a multi-step relation derived from the first. Unfortunately, the rules defining $M \longrightarrow M'$ are not well-suited to define the computational meaning of CC. Instead we follow the approach of Wright and Felleisen [1994]. Without changing the extent of the judgment $M \longrightarrow M'$, we rewrite its definition by factoring out the congruence rules so it is defined by the following single rule that extends local reduction:

$$\dfrac{M \Longrightarrow_R M'}{C[M] \longrightarrow C[M']}$$

where $C[\,]$ is the notation for an *evaluation context* C with a hole, and $C[M]$ denotes the result of plugging M into this hole.

We transform the collection of congruence rules into rules defining permissible evaluation contexts. Formally, we would have a judgment C *evctx* and rules like the following (shown here for functions):

General Rule

$$\frac{}{[\,]\ \mathit{evctx}}$$

Evaluation Context	**Previous Congruence Rule**
$\dfrac{C\ \mathit{evctx}}{C\,N\ \mathit{evctx}}$	$\dfrac{M \longrightarrow M'}{M\,N \longrightarrow M'\,N}$
$\dfrac{M\ \mathit{value} \quad C\ \mathit{evctx}}{M\,C\ \mathit{evctx}}$	$\dfrac{N \longrightarrow N'}{M\,N \longrightarrow M\,N'}$

We won't prove that the two systems are indeed equivalent, and while it requires some effort it is not particularly difficult. We summarize the admissible evaluation contexts in EBNF form. Here we write V for terms M such that M *value*.

$$\begin{array}{llll}
\text{Evaluation contexts} & C & ::= & [\,] & \text{(general)} \\
& & \mid & \mathbf{fst}\,C \mid \mathbf{snd}\,C & (A \wedge B) \\
& & \mid & C\,N \mid V\,C & (A \supset B) \\
& & \mid & \mathbf{inl}\,C \mid \mathbf{inr}\,C \mid \mathbf{case}(C, u.N, w.P) & (A \vee B) \\
& & \mid & \textit{(none)} & (\top) \\
& & \mid & \mathbf{abort}\,C & (\bot)
\end{array}$$

Let's return to the example from and recast it with this new notation. We defined:

$$\begin{array}{lcll}
\mathsf{bool} & = & 1+1 & (\sim \top \vee \top) \\
\mathsf{true} & = & \mathbf{inl}\,\langle\,\rangle \\
\mathsf{false} & = & \mathbf{inr}\,\langle\,\rangle
\end{array}$$

we might expect

$$\mathbf{snd}\,(\mathbf{fst}\,\langle\,\langle \mathsf{true}, \mathsf{false}\rangle, \mathsf{true}\,\rangle) \longrightarrow^2 \mathsf{false}$$

We show at each step how the term is decomposed and then recomposed.

$$\begin{array}{ll}
& \mathbf{snd}\,(\mathbf{fst}\,\langle\,\langle \mathsf{true}, \mathsf{false}\rangle, \mathsf{true}\,\rangle) \\
= & \mathbf{snd}\,[\mathbf{fst}\,\langle\,\langle \mathsf{true}, \mathsf{false}\rangle, \mathsf{true}\,\rangle] \\
\longrightarrow & \mathbf{snd}\,[\langle \mathsf{true}, \mathsf{false}\rangle] \\
= & \mathbf{snd}\,\langle \mathsf{true}, \mathsf{false}\rangle \\
= & [\mathbf{snd}\,\langle \mathsf{true}, \mathsf{false}\rangle] \\
\longrightarrow & [\mathsf{false}] \\
= & \mathsf{false}
\end{array}$$

4 Call with Current Continuation

Now we are prepared to assign a proof term to the CC rule. In fact, we will do a little more, introducing $\neg A$ as a new connective with the following *two* rules:

$$\frac{\neg A\ \mathit{true}^* \quad A\ \mathit{true}^*}{C\ \mathit{true}^*}\ \neg E \qquad \frac{\begin{array}{c}\overline{\neg A\ \mathit{true}^*}\ k \\ \vdots \\ A\ \mathit{true}^*\end{array}}{A\ \mathit{true}^*}\ \mathsf{CC}^k$$

All other rules remain the same as for intuitionistic logic. Note that there is no introduction rule for $\neg A$, so the only way we can obtain it is as a hypothesis from the CC rule.

Adding proof terms:

$$\frac{M : \neg A \quad N : A}{\mathbf{throw}\, M\, N : C}\ \neg E \qquad \frac{\begin{array}{c}\overline{k : \neg A}\ k \\ \vdots \\ M : A\end{array}}{\mathbf{callcc}\,(k.\, M) : A}\ \mathsf{CC}^k$$

The key idea of the dynamics is that a value of type $\neg A$ is an evaluation context with a hole of type A. The rule for **callcc** captures the evaluation context in which the **callcc** occurs and substitutes it for k. The rule for **throw** essentially returns to the state of computation as it was when **callcc** was invoked.

$$\frac{}{C[\,]\ \mathit{value}}$$

$$\frac{}{C[\mathbf{callcc}\,(k.\, M)] \longrightarrow C[[C[\,]/k]M]} \qquad \frac{V\ \mathit{value}}{C[\mathbf{throw}\, C'[\,]\, V] \longrightarrow C'[V]}$$

Values of the form $C[\,] : \neg A$ are runtime artifacts, captured by uses of **callcc**, and can cannot explicitly appear in an expression before it is evaluated. Allowing this would lead to a failure of type preservation.

A significant aspect of the **throw** rule is that it abandons the current evaluation context C, resurrecting the context C' that was captured at an earlier stage and substituted for a $k : \neg A$.

We also have to extend evaluation contexts to account for the components of **throw** that mirror the ones for function application $M\, N$

$$\begin{array}{llll}\text{Evaluation contexts} & C & ::= & \ldots \\ & & \mid & \mathbf{throw}\, C\, N \mid \mathbf{throw}\, V\, C \quad (\neg A)\end{array}$$

We next review some examples to get an intuition for the behavior of **callcc** and **throw**. To make these more intuitive, we assume have a primitive type int or nat with the usual arithmetic operations.

First, if a continuation is not thrown to we simply return the value of the body of the **callcc**.

$$\mathbf{callcc}(k.\, 1 + 3) \longrightarrow^* 4$$

If we throw to a continuation we abandon the local context.

$$\begin{array}{rl} & 1 + \mathbf{callcc}(k.\, 2 + \mathbf{throw}\, k\, 3) \\ = & 1 + [\mathbf{callcc}(k.\, 2 + \mathbf{throw}\, k\, 3)] \\ \longrightarrow & 1 + (2 + \mathbf{throw}\, (1 + [\,])\, 3) \\ = & 1 + (2 + [\mathbf{throw}\, (1 + [\,])\, 3]) \\ \longrightarrow & 1 + [3] \\ = & 1 + 3 \\ = & [1 + 3] \\ \longrightarrow & [4] \\ = & 4 \end{array}$$

Since the **throw** itself abandons the local context, its result type is arbitrary (as you can also see from the its typing rule). For example, the expression below is well-typed and also evaluates to 4 as the previous example.

$$1 + \mathbf{callcc}(k.\, 2 + \mathbf{fst}(\mathbf{throw}\, k\, 3)) \longrightarrow^* 4$$

5 Using Continuations Like Exceptions

One use for continuations is similar to exceptions in the sense that they can be used for the same purpose: to escape a local computation context. But while continuations are lexically scoped, exceptions are typically dynamically scoped.

We use the form of **callcc** available in Standard ML:

```
  type 'a cont                         (* not A *)
  val callcc : ('a cont -> 'a) -> 'a   (* callcc (fn k => M) *)
  val throw : 'a cont -> 'a -> 'b      (* throw M N *)
```

Examples similar to the ones from the previous section work as expected.

```
  open SMLofNJ.Cont;

  val ex1 = callcc (fn k => throw k 4);          (* = 4 *)
  val ex2 = callcc (fn k => 5);                  (* = 5 *)
  val ex3 = 7 + callcc (fn k => 3 + throw k 2);  (* = 9 *)
```

Here is a standard example where we short-circuit multiplication of elements in a list once we encounter a 0. The key here is not that we avoid multiplication with later elements in the list, but that we bypass multiplication of all the prior elements.

```
open SMLofNJ.Cont;

(* mult : int cont -> int list -> int *)
fun mult k nil = 1
  | mult k (x::xs) = if x = 0 then throw k 0
                     else x * mult k xs
```

Similarly, here is an example of short-circuiting a conjunction if the first boolean argument is `false`. This is somehow tricky to do in a call-by-value language and the usual technique would be to pass in function `unit -> bool`. With continuations, we can pass in a continuation which is the destination for the result of the conjunction. This allows us to escape the local context and avoid evaluating the second boolean argument to `conjoin`.

```
(* conjoin : bool cont -> bool -> bool -> 'a *)
fun conjoin k false = C.throw k false
  | conjoin k true = fn y => C.throw k y

C.callcc (fn k => conjoin k false true);           (* = false *)
C.callcc (fn k => conjoin k true false);           (* = false *)
C.callcc (fn k => conjoin k false (raise Match));  (* = false *)
```

Some of the most advanced uses of first-class continuations involve returning a capture continuation, and possibly storing it in a data structure. Many of these uses also involve mutable store (to save the continuation), so we will not give such examples here. We may come back to this point in a later lecture when we discuss backtracking search.

6 Bidirectional Typing with Continuations

We wrote a bidirectional type-checker in Lecture 6, based on reading of the inference rules as a recipe for proof construction. We had a type `prop` for propositions and `term` for proof terms and the following functions:

```
  val check : term -> prop -> bool
  val synth : term -> prop option
```

They code was extracted from constructive proofs of

$$\begin{array}{ll} \text{check} & \forall M.\, \forall A.\, (M \Leftarrow A) \lor \neg(M \Leftarrow A) \\ \text{synth} & \forall M.\, (\exists A.\, M \Rightarrow A) \lor \neg(\exists A.\, M \Rightarrow A) \end{array}$$

One pragmatic downside of this approach is that there no room for error messages. If we change the return types `bool` and `prop option` to account for error messages, the code because unpleasantly complicated.

Instead, we can pass in a continuation that takes an error message (a string) as an argument and even simplify the return type.

```
  val check : string cont -> term -> prop -> bool
  val synth : string cont -> term -> prop
```

We could even simplify further and replace `bool` by `unit`, in which case a normal return indicates that the term checks.

With these simplified types, the code can also be simplified because, for example, we no longer need to check if synthesis succeeds. If not, it will discard the local context and throw and error message to the continuation. We show a few lines of this code; our complete live-coded implementation can be find in tcheck-cont.sml.

```
fun check k (Pair(M,N)) (And(A,B)) = check k M A andalso check k N B
  | check k (Pair(M,N)) C = throw k "pair/not-and"
  | check k (Lam(F)) (Imp(A,B)) = check k (F (Hyp A)) B
  | check k (Lam(F)) C = throw k "lam/not-imp"
  | check k (Inl(M)) (Or(A,B)) = check k M A
  | check k (Inl(M)) C = throw k "inl/not-or"
  | check k (Inr(M)) (Or(A,B)) = check k M B
  | check k (Inr(M)) C = throw k "inr/not-or"
  | check k (Case(M,F,G)) C =
    (case synth k M
      of Or(A,B) => check k (F (Hyp A)) C andalso check k (G (Hyp B)) C
       | _ => throw k "case/not-or")
  | ...

and synth k (Hyp(A)) = A
  | synth k (Fst(M)) =
    (case synth k M
      of And(A,B) => A
       | _ => throw k "fst/not-pair")
  | synth k (Snd(M)) =
    (case synth k M
      of And(A,B) => B
       | _ => throw k "snd/not-pair")
  | synth k (App(M,N)) =
    (case synth k M
      of Imp(A,B) => if check k N A then B
                     else throw k "app/not-match"
       | _ => throw k "app/not-implies")
  | synth k M = throw k "does␣not␣synthesize"

fun check_tp M A = callcc (fn k => if check k M A then "OK" else "
    Impossible")
```

7 The Dark Underside

callcc has been called "the 'goto' of functional languages". Despite the fact that the continuation type $\neg A$ connects to classical propositional logic, its departure from the verificationist explanation of connectives and propositions means that it has its limitations when one also includes quantifiers. It has to be considered a "control effect" and as such is not a pure functional construct [Harper et al., 1993].

As one example of the strangeness of **callcc** we consider how it is used to prove the law of excluded middle, and what that means computationally. First, the logical proof. We avoid here overloading negation because of its new, classical meaning, and just write $A \vee (A \supset \bot)$ for the law of excluded middle.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg(A \vee (A \supset \bot))}\,k \qquad \dfrac{\dfrac{}{A}\,u}{A \vee (A \supset \bot)}\,\vee I_1}{\bot}\,\neg E}{A \supset \bot}\,\supset I^u}{A \vee (A \supset \bot)}\,\vee I_2}{A \vee (A \supset \bot)}\,\mathsf{CC}^k$$

The right way to understand this proof is to construct it using our usual hybrid bottom-up/top-down strategy, or look at the meaning of the proof term which comes next.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\overline{k : \neg(A \vee (A \supset \bot))}\;k \quad \dfrac{\overline{u : A}\;u}{\mathbf{inl}\, u : A \vee (A \supset \bot)}\;\vee I_1}{\mathbf{throw}\, k\, (\mathbf{inl}\, u) : \bot}\;\neg E}{\lambda u.\, \mathbf{throw}\, k\, (\mathbf{inl}\, u) : A \supset \bot}\;{\supset} I^u}{\mathbf{inr}\, (\lambda u.\, \mathbf{throw}\, k\, (\mathbf{inl}\, u)) : A \vee (A \supset \bot)}\;\vee I_2}{\mathbf{callcc}\, (k.\, \mathbf{inr}\, (\lambda u.\, \mathbf{throw}\, k\, (\mathbf{inl}\, u))) : A \vee (A \supset \bot)}\;\mathsf{CC}^k
$$

We define

$$
\begin{array}{lcl}
\mathsf{exm}_A & : & A \vee (A \supset \bot) \\
 & = & \mathbf{callcc}\, (k.\, \mathbf{inr}\, (\lambda u.\, \mathbf{throw}\, k\, (\mathbf{inl}\, u)))
\end{array}
$$

When we use this as the subject of a case statment

$$C[\mathbf{case}([\mathsf{exm}_A], x.\, N, y.\, P)]$$

the **callcc** will capture the current evaluation context $C'[\,] = C[\mathbf{case}([\,], x.\, N, y.\, P)]$ as k and then proceed with $\mathbf{inr}\, (\lambda u.\, \mathbf{throw}\, C'[\,]\, (\mathbf{inl}\, u))$. This is a value, so in essence exm_A claims that $A \supset \bot$ holds! So

$$
\begin{array}{ll}
 & C[\mathbf{case}(\mathsf{exm}_A, x.\, N, y.\, P)] \\
= & C[\mathbf{case}(\mathbf{callcc}\, (k.\, \mathbf{inr}\, (\lambda u.\, \mathbf{throw}\, k\, (\mathbf{inl}\, u))), x.\, N, y.\, P)] \\
\longrightarrow & C[\mathbf{case}(\mathbf{inr}\, (\lambda u.\, \mathbf{throw}\, C'[\,]\, (\mathbf{inl}\, u)), x.\, N, y.\, P)] \quad \text{for } C'[\,] = C[\mathbf{case}([\,], x.\, N, y.\, P)] \\
\longrightarrow & C[[(\lambda u.\, \mathbf{throw}\, C'[\,]\, (\mathbf{inl}\, u))/y]P]
\end{array}
$$

Now P could finish normally (without using y), resulting in some value W, and overall we would obtain $C[W]$ which then further reduces to a final answer.

But how could P use y? Since $y : A \supset \bot$, it can only use it by applying $y\, M$ where $M : A$. After M has been reduced to a value V, this then becomes $\mathbf{throw}\, C'[\,]\, (\mathbf{inl}\, V)$. But this is just

$$C[\mathbf{case}(\mathbf{inl}\, V, x.\, N, y.\, P)] \longrightarrow C[[V/x]N]$$

In other words, we have returned to the **case** and now claim that A holds. The evidence V for that was provided by P itself. This means this original assertion that $A \supset \bot$ was a lie!

This fib is an example of what makes the whole enterprise of a computational interpretation of classical logic brittle and ultimately unsatisfactory. For example, it is inconsistent with an extension to a dependent type theory.

References

Harvey Friedman. Classically and intuitionistically provably recursive functions. In D.S. Scott and G.H. Muller, editors, *Higher Set Theory*, pages 21–27. Springer-Verlag LNM 699, 1978.

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

Gerhard Gentzen. Die Widerspruchsfreiheit der reinen Zahlentheorie. *Mathematische Zeitschrift*, 112:493–565, 1936. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, 1969.

Kurt Gödel. Zur intuitionistischen arithmetik und zahlentheorie. In *Ergebnisses eines mathematischen Kolloquiums*, volume 4, pages 34–38, 1933. English translation "*On intuitionistic arithmetic and number theory*", M. Davis, ed., *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems, and Computable Functions*, pp. 75–81, Dover Publications, 1965.

Timothy Griffin. A formulae-as-types notion of control. In F. E. Allen, editor, *17th Symposium on Principle of Programming Languages*, pages 47–58, San Francisco, California, jan 1990. ACM Press.

Robert Harper, Bruce F. Duba, and David B. MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, 1993.

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.