Continuations and Logic

15-814: Types and Programming Languages
Fall 2015
Evan Cavallo (ecavallo@cs.cmu.edu)

Part of the interest in continuations in type theory comes from the deep connections which logic. A few of the questions on the homework hint at these connections, and since we didn't discuss it in class, I am going to go into a bit here. (Incidentally, this might help you get your head around parts of HW5.)

Prof. Harper has mentioned a few times the *propositions-as-types* correspondence, which guides the connections between type theory and logic. The basic idea is that *types* can be thought of as *logical statements* (i.e. *propositions*), with *terms of a type* serving as *proofs of a proposition*. For example, a logical system might contain the following inference rules for logical conjunction $A \wedge B$:

$$\frac{A \text{ true} \quad B \text{ true}}{A \wedge B \text{ true}} \qquad \frac{A \wedge B \text{ true}}{A \text{ true}} \qquad \frac{A \wedge B \text{ true}}{B \text{ true}}$$

These look very similar to the rules for product types $\tau_1 \times \tau_2$, but without terms. From the logical perspective, a term (generally called a *proof term* in this setting) serves as a representation of a derivation. For example, say we add terms to the above rules like so:

$$\frac{M : A \quad N : B}{\langle M, N \rangle : A \wedge B} \qquad \frac{M : A \wedge B}{M \cdot \mathtt{L} : A} \qquad \frac{M : A \wedge B}{M \cdot \mathtt{R} : B}$$

In each rule, the term in the conclusion is an operator applied to the terms in the premises, and each rule corresponds to a distinct operator. Thus, if we know a term $M : A$ is well-typed, then we can reconstruct the typing derivation; to check that M is well-typed is to check that it corresponds to a well-formed derivation. Hence the name *proof term*: a term $M : A$ corresponds to a proof (i.e., derivation) of A.

Each of the basic type formers we've talked about in class corresponds to a logical connective:

types	logic
products ($\tau_1 \times \tau_2$)	conjunction ($A \wedge B$)
sums ($\tau_1 + \tau_2$)	disjunction ($A \vee B$)
functions ($\tau_1 \to \tau_2$)	implication ($A \to B$)
unit	true ($\top$)
void	false ($\bot$)

(The situation is more complicated with non-termination – the logic associated with a language like PCF would be inconsistent.) A logical system with these connectives, defined with rules like those we've given for the type-theoretic equivalents, is *constructive*, in the sense that there is a terminating dynamics (typically called a *normalization* process) in which the values correspond to introductory forms. This means, for example, that $A \vee B$ can be only be proven if one of A and B is provable, because any proof $M : A \vee B$ can be reduced to a value which is either $\mathtt{L} \cdot N$ with $N : A$ or $\mathtt{R} \cdot P$ with $P : B$.

Classical mathematics is based on set theory, which is itself based on *classical (first-order predicate) logic*. Unlike the sort of logic discussed above, classical logic is *not* constructive, primarily because it asserts the following axiom, called the *principle (or law) of the excluded middle:*

$$\frac{}{A \vee (A \to \bot) \text{ true}}$$

We can think of this as asserting that every proposition is true or false (implies $\bot$). This principle is not constructive because it is not possible for *every statement* A to prove either A true or $(A \to \bot)$ true (at least, not in a sufficiently complex logic, and certainly not without inspecting

the structure of A). For one thing, any logic in which Gödel's incompleteness theorem applies contains statements which can be neither proved nor disproved. In addition, a constructive principle of the excluded middle would require a computable decision procedure producing a proof or refutation of any statement – in particular, statements of the form "the Turing machine with Gödel number n halts." (While the type systems we have worked with aren't sophisticated enough to express this kind of proposition, type systems which are do exist!)

A second, equivalent axiom is the principle of *double-negation elimination*:

$$\frac{(A \to \bot) \to \bot \text{ true}}{A \text{ true}}$$

This rule states that any statement which is *not not true* is in fact true. Another way to think of it is as *proof by contradiction*: if assuming A is false leads to a contradiction (that is, if $A \to \bot$ implies $\bot$), then A must in fact be true. This fails to be constructive because there is no general way of extracting a proof of A from a proof of $(A \to \bot) \to \bot$.

Despite all this, there is a way to assign some kind of computational meaning to proofs in classical logic, and this is where continuations come in. The chart extends like so:

types	logic
continuations (τ `cont`)	classical negation ($\neg A$)

The rule for `letcc` corresponds to proof by contradiction (and thus double-negation elimination), while `throw` derives a contradiction (in the form of an arbitrary conclusion) from a statement and its negation:

$$\frac{\neg A \vdash A}{A}\ (\textsc{Letcc}) \qquad \frac{\neg A \quad A}{B}\ (\textsc{Throw})$$

(Until now, I haven't mentioned hypothetical judgments with $\vdash$ in the logical context, but they play the same role as in type theory.) With these rules, classical theorems like $A \vee \neg A$, $\neg\neg A \to A$, and $(A \to B) \to (\neg A \vee B)$ are derivable (and are, as it happens, the three programs asked for in Task 3), and we have a computational interpretation through continuations.

It is important to distinguish the computational behavior of $A \to \bot$ and $\neg A$. There are no values of type $\bot$ in the empty context, so a closed function $A \to \bot$ can never be called – it is dead code. A continuation of type $\neg A$ in the empty context, however, *can* conceivably be thrown to: it makes no guarantee that there are no closed terms of type A, as $A \to \bot$ does, but simply provides an "escape hatch" if one does encounter an A.

Section 2.2 of HW5 describes a translation of types and terms from a language with continuations into a language without them. This setup originates in the *Gödel-Gentzen double-negation translation* from classical into constructive logic. The main idea is that, while a classical theorem may not be provable in constructive logic, there is a modified version of the theorem which is constructively true. For example (using $\neg_I A$ to mean $A \to \bot$), while $A \vee \neg_I A$ is not constructively valid, the weaker $\neg_I \neg_I (A \vee \neg_I A)$ is. In general, the translation works by strategically inserting double-negations. For example, the translation might have $|A \to B| \triangleq |A| \to \neg_I \neg_I |B|$ (there exist several variations). In the translation on the homework, we don't add $(- \to \bot) \to \bot$ but rather $(- \to B) \to B$ where B is the "result type;" it turns out that the correctness theorem for the double-negation translation uses none of the properties of $\bot$, so it can be replaced with an arbitrary proposition. Remarkably, this translation, suitably adapted, is actually useful in the compilation of functional languages!