# Continuations

Continuations allow for lots of things. Intuitively, we can think of continuations as “functions that never come back.” That is, continuations transfer control to some other part of your program. In a way, continuations are like a much nicer version of `goto`.

We have two operations when working with continuations: `letcc` and `throw`.

- `letcc` is the intro form. It duplicates the current control stack, effectively “marking” the code as some place we could come back to.
- `throw` is the elim form. It destroys the current control stack, and replaces it with the stack we’re throwing to.

Verify this in the rules below:

```
30.5a:
____________________________________________
 k ▷ letcc{τ}(x.e)  -->  k ▷ [cont(k) / x]e

30.5e:
___________________________________________
 k; throw{τ}(v; -) ◁ cont(k')  -->  k' ◁ v
```

# Uses for Continuations

Continuations can be hard to get a feel for. With this in mind, here are some places where continuations can be or are used.

## 1. Short circuit evaluation

With continuations, we can report "the answer" to the next piece of code immediately, without having to walk back up the stack.

We can use this to implement short circuiting logical "and" and "or".

This is also useful when we don't want to have to evaluate all the pieces (maybe expensive) to report an answer we already know. Think about the regex homework from 15-150: we used continuations to encapsulate backtracking decisions. When we arrived at the answer, we could call the continuation immediately and jump directly to the answer.[1].

## 2. Compilation

Continuations make control flow explicit. If the analogy is that continuations are `goto`, then they're really not much more than an assembly `jump` instruction.

When writing a compiler, at some point we have to convert high-level control flow logic into jumps. Continuations are a nice way to model this[2].

## 3. Giving insight into logic

Continuations model negation if we treat programs and types as proofs and propositions. There are a lot of interesting analogies and insights here, and we'll discuss them next.

[1]This is a somewhat broken analogy, because we used callback functions—not continuations—in 150. This meant that we did in fact have to walk all the way back up the stack.

[2]For more on how this works, take 15-417 HOT Compilation, or look up the book *Compiling with Continuations*.

## Understanding `throw`

`throw` is basically deriving a contradiction. If our assumptions allow us to derive a contradiction, we can prove anything. Let's look at the type of `throw` (in curried form):

```
throw{τ} : σ → σ cont → τ
```

If we can come up with both `σ` and `σ cont`, this is like having $P$ and also $\neg P$. In logic, this allows us to conclude anything. Thus, `throw{τ}` *can* introduce any type we want (namely, `τ`).

## Understanding `letcc`

`letcc` is basically a short-hand version of proof by contradiction. This form is commonly begun with, "Assume for sake of contradiction…" If we look at `letcc`'s type (again, as a curried function):

```
letcc{τ} : (τ cont → τ) → τ
```

So in the body of the function we pass to `letcc`, we're assuming `¬ τ`. From here, there are two options:

1. We don't use our assumption, but still "prove" (or "return") a `τ`. This is the same as not assuming anything in the first place.
2. We use our assumption, thus `throw`ing to it. Recall that we already equated `throw`ing with deriving a contradiction and using it.

When we use a contradiction, we jump back to right before we AFSOC'd, because that assumption must have been false. The types witness this by jumping from `¬τ`, then implicitly to `¬¬τ` (we assumed `¬τ`, but it was wrong), and finally to `τ`. This implicit jump is discussed extensively in the next section.

## Proof by Contradiction

Compare the constructs `throw` and `letcc` with a "proof by contradiction," that Bob spoke at length about on Piazza:

> What *is* a proof by contradiction is deriving $P$ from $\neg\neg P$. In short-hand form you would assume $\neg P$, derive a contradiction, thereby proving $\neg\neg P$, obviously. However, then you dramatically conclude "therefore $P$," for which you have **no justification**.

Let's explore the code for a "proof by contradiction" (using the above interpretation of the term). Logically, it'd be $\neg\neg P \implies P$. Let's write a function with this type using continuations:

```
f : τ cont cont → τ
f == λ(x : τ cont cont).
       letcc{τ} k in
         throw{τ} k to x
```

It relies on the fact that when we `throw`, we "never come back" to the current execution context, so we get to choose any type we want for the result.

This program gives us about as much constructive information as an proof by contradiction. Look at `x`. It's a `τ cont cont`, which means that it gets thrown a `τ cont` that it can use to perform it's job.

But look at what `f` does: if `x` ever tries to throw something back to us using `k`, nothing happens! `f` in no way uses or does anything interesting with the result that might be thrown to `k`. Thus, the continuation `x` gets no useful behavior by throwing to `k`.

This is the same as with a proof by contradiction. Sure, we can prove that $\exists a, b \in \mathbb{R} \setminus \mathbb{Q}.\ a^b \in \mathbb{Q}$. If programs are constructive proofs, we should be able to evaluate the program and get back the answer of which numbers witness the proof. But if we prove it by a contradiction, it doesn't work this way.

## Continuations as Functions

This whole time, continuations have been a standalone type, with dedicated intro and elim forms. We can actually view continuations as a mode of use of functions and `void`. That is, `τ cont` is isomorphic to `τ → 0`:

```
f : τ cont → (τ → 0)
f == λ(k : τ cont).
       λ(x : τ).
         throw{void} x to k
```

If we have `τ` and `¬τ`, we have a contradiction and can prove anything. In particular, we can prove "false" or `0`. This might be concerning: There shouldn't be any inhabitants of `0` (or `void`).

As a result, that means the call stack that our `τ → 0` function is defined in will never be fully evaluated. Instead, if someone tries to call this function, control immediately transfers to the call stack `k`, where `¬ τ` is true.

Going the other way:

```
g : (τ → 0) → τ cont
g == λ(f : τ → 0)
       letcc{τ cont} x in
         abort{τ cont}(
           f (letcc{τ} y in
             throw{τ} y to x))
```

If we have an `f : τ → 0`, then if we can somehow conjure a `τ` we can get back any type by calling `f` and `abort`ing on the result. The trick we use to get a `τ` is to `letcc` *outside* the `abort` call. This means that `x : τ cont cont`. Then we use `letcc` again *inside* the `abort`, so that `y` is a `τ cont`.

With all these continuations, we use `throw` to get a `τ` which we can use to call `f` and abort on the result. The cost is that the `throw` actually transfers control back outside the `abort`! So we never actually call `f`, and control proceeds with `g` returning a continuation to inside the `abort` (namely, `y`). If anyone ever threw to `y` we'd have a problem. But the fact that `f` is a function from `τ → 0` means that no one can conjure a real `τ` in the first place.