The \(\lambda\)-calculus was intended from its inception as a model of computation. It was used by Church to explore computability in the 1930s — scooping Turing by about six months.\(^1\) Once high-level programming languages were needed, language designers were naturally drawn to the \(\lambda\)-calculus, starting with McCarthy’s use of LAMBDA in Lisp.

An important issue in reasoning about the \(\lambda\)-calculus is which \(\beta\)-reduction to do next. This issue was explored by Alonzo Church and Barkley Rosser starting in the 1930s. They proved that the \(\lambda\)-calculus, even the untyped version, is confluent, meaning that if \(E_0 \triangleright E_1\) and \(E_0 \triangleright E_1'\), then there is a term \(E_2\) such that \(E_1 \triangleright E_2\) and \(E_1' \triangleright E_2\).\(^2\) In other words, if you draw a directed graph of every formula you can reach from \(E_0\) by a set of \(\beta\)-reductions, there are no dead ends, nor is there any branch point where reduction \(R\) takes you to one subgraph and \(R'\) takes you to another, entirely disjoint subgraph.\(^3\)

Confluence does \textit{not} mean that there are no infinite loops. In the untyped

---

\(^1\)Which is why the Church-Turing thesis is so called. However, other logicians found Turing’s treatment more satisfying, because he had an explicit model of “resource allocation”: There’s one symbol per tape square, and if the read/write head moves off the end of the tape, you go out and buy more. There’s no limit to what you can record, as our civilization attests — if we don’t run out of paper. In Church’s system, \(\beta\)-reduction requires you to copy arbitrarily large expressions to an arbitrary number of places, coming up with an arbitrary number of new variable names to avoid capturing free variables. As a primitive, it left something to be desired. Once you have the concept of Turing machine, you can easily produce a TM that “evaluates” an arbitrary \(\lambda\)-calculus term, thus grounding Church’s beautiful flights of fancy on actual pieces of paper.

\(^2\)What about all that variable renaming during \(\beta\)-reduction to avoid free-variable capture? Ah yes, that complication requires us to complicate the definition of confluence. There are various ways of doing it. The simplest way is probably to say “There are terms \(E_2\) and \(E_2'\) such that \(E_1 \triangleright E_2\) and \(E_1' \triangleright E_2'\), and \(E_2\) and \(E_2'\) are equivalent modulo bound-variable renaming. I.e., there’s a one-to-one function between their variables that transforms one into the other, and doing so doesn’t change which variable occurrences are bound where.” This fine print takes all the excitement out of confluence. It makes one want to switch to a formalism that does without variables, which exist, but are harder for human readers to make sense of than even the \(\lambda\)-calculus.

\(^3\)Here’s another way to fix the statement of confluence: Let’s pretend we were talking about equivalence classes of formulas all along. That is, \(E_0, E_1,\) et al. are each an infinite set of formulas equivalent modulo renaming. (See above.) It’s not hard to prove that if a \(\beta\)-reduction is possible for one member of such a set, it’s possible for all, and the formulas you get are all in the same equivalence class. So we can just say \(E \triangleright F\), where \(E\) and \(F\) are equivalence classes. Now the original statement of the confluence property is correct.
\(\lambda\)-calculus, there certainly are. Not only that, but strongly typed programming languages have to have notations for recursion and loops to ensure that they can compute anything computable, and this flexibility comes at the cost of tolerating infinite loops.

In the \(\lambda\)-calculus, a computation comes to an end when a term is reached for which no \(\beta\)-reductions are possible. Recall that a \textit{redex} is an occurrence of a \(\beta\)-convertible subterm in a term. So a term with no redexes is the terminus of a computation, a \textit{quiescent} term.

Another important theorem of Church and Rosser's is\(^4\) that if there exists a series of \(\beta\)-conversions that will yield a quiescent term, then the following rule for picking the next redex will yield a quiescent term: Always pick the leftmost outermost redex. An \textit{outermost} redex is one not contained inside another redex.

Exercise: Prove to yourself that of two outermost redexes, one must lie entirely to the left of the other.

Exercise: Prove to your roommate that confluence guarantees that if a \(\lambda\)-term can be reduced to two quiescent terms, then they are equivalent modulo renaming.

If you write a programming-language interpreter, such as the classic "meta-circular" Lisp interpreter written in Lisp, you’ll discover the ways in which such interpreters differ from Church & Rosser’s scheme. Given a function expression to evaluate and nothing else, as in this snippet of Racket code:

\[
(\text{cons} \ (\text{lambda} \ (x) \ ((\text{lambda} \ (z) \ (\text{list} \ z \ z)) \ x)) \ '())
\]

the result is a list with one element, that function of \(x\).\(^5\) It would be the rare meta-circular interpreter that did anything at all with the body of a \(\lambda\)-expression that wasn’t applied to anything. But the \(\lambda\)-calculus equivalent:

\[
c(\lambda x. (\lambda z. l z z) x) n
\]

\((c=\text{cons}, \ l=\text{list}, \ n = '())\) has a perfectly good redex: \((\lambda z. l z z)x\). \(\beta\)-converting it yields the quiescent term \(c(\lambda x. l x x) n\).

\(^4\)Insert careful bibliographic trace here.

\(^5\)If you don’t know Lisp syntax, \((\text{cons} \ e \ '())\) makes a list of one element, the value of \(e\). The point is that in the case at hand \(e\) is a \texttt{lambda}-expression, which is Racket syntax for a constant function. The Scala equivalent is \((x) => ((z) => \text{List}(z, z))(x)) :: Nil\). I chose to use Racket here because it’s so easy to write a meta-circular interpreter. If you’ve never done it, you should find time to try.
Even more important is the fact that interpreters don’t usually substitute the arguments of a function into the text of the function. They keep track of variable-binding environments and closures. The details aren’t important.

Most programming languages violate the Church/Rosser redex-choice rule in other ways. Given a function and its arguments, they evaluate the arguments first, then pass the values to the function. The name for this rule is “call by value.” The Church/Rosser rule would pass an argument in, and evaluate it when the parameter the argument was bound to was needed. What does it mean to pass an unevaluated argument?

Think of it this way: If you allow functions of zero arguments, as most programming languages do, then you can get closer to the Church/Rosser rule by replacing every argument expression $E$ with a zero-argument function $(\cdot \mapsto E)$, and replacing every occurrence of parameter $p$ with $p()$. Take the classic example of trying to define “ifnot $c$ then $e_1$ else $e_2$” as a subroutine (it’s supposed to evaluates $e_1$ only if $c$ is false; if it’s true, it evaluates $e_2$). That’s because in calling ifnot$(c, e_1, e_2)$, you’ll evaluate all three expressions no matter what.

We’d like to define ifnot as

$$(c, e_1, e_2) \mapsto \text{if } (c) \text{ e}_2 \text{ else } e_1$$

If instead we define it as

```python
def ifnot(c, e1, e2) = if (c()) e2() else e1()
```

(duing types out of it) and call it by writing (e.g.)

```python
ifnot(((() => n==0), (() => k/n), (() => 0))
```

then we’ve approximated the Church/Rosser rule in a pure call-by-value language. This name for this behavior is call by name. (Presumably because every time the name of the parameter is encountered the function is called again, which amounts to evaluating the argument again.)

This was the way arguments were passed in Algol 60, although it’s hard to understand why. A much better idea is to evaluate the argument the first time the parameter is referred to, and save the result; on subsequent

---

6The only exceptions I know of are macro processors like \text{T\LaTeX}.  
7Of course, many languages are implemented using compilers rather than interpreters; for those languages, when I say “language does $X$,” I mean “the compiler translates the program into code that does $X$.”  
8Or even how. It was 1960, for crying out loud! [[Insert careful historical investigation here.]]
references the value is just retrieved. This is called *call by need*, or *lazy evaluation*.9

Exercise: Much as we did for call by name, show how call by need could be implemented using functions and auxiliary variables in a call-by-value language.

In languages with call by name or by need, you *can* write \texttt{ifnot} as a function. That’s because every function is compiled as if its arguments were implicit functions. Programming-language researchers call a function *strict in its $n$th argument* or *in parameter $p$* if it always evaluates its $n$th argument, or the argument $p$ is bound to. It’s *strict* (period) if it’s strict in all its arguments. In a pure call-by-value language, every function is strict in all its arguments.

Rather than insist that every function be strict or that no function be strict in any argument, languages often supply mechanisms for overriding the default. Haskell is lazy, but has notations indicating that a parameter is strict (i.e., that the function being defined is strict in that parameter). Scala is strict, but has notations indicating that a parameter is call-by-name, meaning that it will be re-evaluated every time the parameter is accessed, or never if the parameter is never accessed. Simply declare parameter $p$ thus: “$p:=>T$”; The notation looks like a function type with a defective argument type. It’s *not* the same as “$p: \emptyset:=>T$”, which would require writing $p()$, just as in my trick for defining \texttt{ifnot}. You only have to write $p$, which is weirdly appropriate.

Scala has another device, *lazy variable bindings*. If a \texttt{val} is prefixed with \texttt{lazy}, as in

\begin{verbatim}
lazy val x = 1/y
\end{verbatim}

than $1/y$ is not evaluated until the first time $x$ is evaluated. Note that this is true lazy evaluation.

\begin{verbatim}
var y = 2.0
lazy val x = { val r = 1/y; y = y - 2.0; r}
List(x,x)
\end{verbatim}

9The only justification I’ve heard for true call by name is that, using side effects, you could alter the value of the parameter each time it was accessed. Thus you could build a weird kind of functional argument. (In dark corners of Scala, such as the \texttt{Enumeration} class, call-by-name parameters are used for just this purpose.) Passing actual functions makes much more sense, and is easier to keep under control. In a pure functional language, of course, it’s impossible for a call-by-name argument to have a different value on different occasions.
works just fine; its value is \texttt{List(0.5, 0.5)}.