BCKW Combinator Calculus

An Examination of Haskell Curry's Ph.D. Thesis

Charles Averill

Computer Security Group
The University of Texas at Dallas

April 2023

History

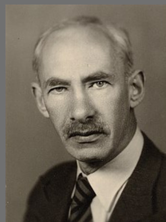
- The year is 1930. Haskell Curry is a Ph.D. student at the University of Göttingen studying under legendary mathematician David Hilbert and working with legendary mathematician Paul Bernays.
- 6 years prior, Moses Schönfinkel publishes a paper called "On the building blocks of mathematical logic," attempting to formalize the description of mathematical propositions, but failing to go into enough detail.
- Hilbert is famous for striving to describe the foundations of mathematics at the deepest level,[1] Bernays is familiar with Schönfinkel's work, and Curry has already written a small preliminary paper expanding on Schönfinkel's discoveries.
- The setting is perfect to finally formalize the construction of mathematical propositions.

[1]Unfortunately, mathematician Kurt Gödel will curtail this effort in 1931 with his famed incompleteness theorem.

People

Curry

Hilbert



Bernays

Church

Turing

Schönfinkel

Gödel

Schönfinkel's Discoveries

- Schönfinkel proposed a set of "combinators" to represent theorems.
- Combinators are like functions — they take in arguments and evaluate to something.
    - Function application would look like $f(x, y, z)$ in normal math. In combinator calculus, it looks like $fxyz$ or $(((fx)y)z)$
- Schönfinkel's combinators are:
    - $Bxyz = x(yz)$
    - $Cxyz = xzy$
    - $Kxy = x$
    - $Wxy = xyy$

Combinator Example

Curry proposes that $BCCx = Ix = x$ (identity function).

$$BCCx$$

$$C(Cx)$$ [from Bxyz = x (y z)]

$$\ldots$$

Hmm... that didn't go the way I had hoped.

Aside: Currying

We need a concept called "currying" to continue (yes, it's named after Haskell Curry).

Exercise: 2 stars, advanced (currying)

The type X → Y → Z can be read as describing functions that take two arguments, one of type X and another of type Y, and return an output of type Z. Strictly speaking, this type is written X → (Y → Z) when fully parenthesized. That is, if we have f : X → Y → Z, and we give f an input of type X, it will give us as output a function of type Y → Z. If we then give that function an input of type Y, it will return an output of type Z. That is, every function in Coq takes only one input, but some functions return a function as output. This is precisely what enables partial application, as we saw above with `plus3`.

By contrast, functions of type X × Y → Z -- which when fully parenthesized is written (X × Y) → Z -- require their single input to be a pair. Both arguments must be given at once; there is no possibility of partial application.

It is possible to convert a function between these two types. Converting from X × Y → Z to X → Y → Z is called *currying*, in honor of the logician Haskell Curry. Converting from X → Y → Z to X × Y → Z is called *uncurrying*.

We can define currying as follows:

```
Definition prod_curry {X Y Z : Type}
  (f : X × Y → Z) (x : X) (y : Y) : Z := f (x, y).
```

`void main(int argc, int *argv[])` has type `int` → `int **` → `void`

Combinator Example

$$BCCx$$
$$C(Cx)$$

From here, we can use currying to think about what the result *would be* like if we gave it two more arguments $-_1$ and $-_2$:

$$C(Cx)\ -_1\ -_2$$
$$(Cx)\ -_2\ -_1 \texttt{[Cxyz = x z y]}$$
$$x\ -_1\ -_2 \texttt{[C rule]}$$
$$x$$

The identity function!

Why it Matters

Schönfinkel came up with these rules in 1924, but he didn't write a true proof that the combinators could actually be used to do anything interesting. Curry actually wrote a fairly scathing review in his thesis:

> *"Because Schönfinkel has in no way shown how the introduction of the other fundamental concepts is to be avoided, and because he cannot define them from others, he has not justified his claim. In fact he has achieved only a new and inconvenient notation."*

If my Ph.D. advisor were David Hilbert, I'd probably be dissatisfied with this too. Hilbert sought a way to formalize the description of mathematics such that we could ensure that it had no inconsistencies or paradoxes.[2]

[2] See Hilbert's Program

Curry's Approach

I'm not going to walk you through all of Curry's thesis for a few reasons:

- It's a Ph.D. thesis
- It is deceptively simple towards the beginning
- You should read it for yourself here: https://www.charles.systems/Combinators

Instead I'm going to broadly summarize his strategy for proving that combinators are Turing-complete.[3]

[3] Fun fact: Turing machines weren't proposed until 1936. Turing-completeness just means that a system can perform any decidable computation!

Chapter 1. General Foundations

Most of Chapter 1 §A, B explain the need for such a formalization, and lay out philosophical groundwork for the rest of the thesis. Some big concepts:

1. A good system of mathematics has few axioms (statements assumed to be true) without forfeiting any generality
2. A good system of mathematics eliminates paradoxes that arise from the "prelogic" that exists underneath mathematics[4]
3. Contradictory concepts (like `P & !P = True`) are not inherently meaningless and cannot be disregarded

§C evaluates Schönfinkel's approach and propose new axioms to build off of. §D proposes and proves theorems regarding the equality of sequences of combinators.

[4] See Russel's Paradox

$$F(\phi) = \text{not } \phi(\phi), \; F(F) = \text{not } F(F)$$

Chapter 2. The Theory of Combinators

Curry starts off this chapter with a bold claim: that anything derivable in the standard frame of mathematics is derivable in the system of combinators.

- The theorems in §C deal with the equivalence of combinators after they're reduced,[5] the idea of "normal" combinations (too complicated to explain here), sequences of variables called "groupings," and combinations with no parentheses called "transformations."
- §D is where it starts to get more interesting (in my opinion). Curry uses the theorems from the previous three sections to prove that combinators can represent a theorem of commutativity. The following theorems and those in the final §E prove more concepts about regularity and the behavior between sequences of regular combinations.

[5] As in $WBKCW \to BKKCW \to K(KC)W \to KC$

And Then What?

Hang on a second. Curry said that anything we could write in ordinary logic could be written with combinators, but he never actually proved it!

Place yourself in the shoes of early 20th century mathematicians. Set theory was proposed by Georg Cantor in 1874, and in 1900 it was proven to be inconsistent due to Russell's paradox. In those 26 years, there was no proof that set theory was complete or decidable or anything like that! Just hopes and dreams and duct tape.

The idea of classifying a model of computation as "complete" or "consistent" or "decidable" was postulated by Hilbert in 1900 but would not be formally proposed until 1936[6].

BCKW **is** Turing-complete because SKI is Turing-complete because lambda calculus is Turing-complete.
Take my word for it or read Turing's Proof.

[6] See the Church-Turing Thesis

Numbers

Why don't we actually put the combinators to use? Here's an encoding of Church numerals[7] with BCKW:

$$0 = K(WK)$$
$$succ(n) = (B(BW)(BBC))^8 Bn$$
$$add(a, b) = af(bfx)$$
$$mul(a, b) = a(bf)x$$

We can encode booleans, arbitrary pairs, lists, data structures, everything!

[7] `zero f x = x, one f x = f x, two f x = f (f x) ...`

[8] $(B(BW)(BBC))$ is actually S from SKI combinator calculus, so this is more concisely described as $succ(n) = SBn$

Materials

- My English transcription of Curry's thesis - https://www.overleaf.com/read/rzhdyjvrzbgy
- Hardcopy of the translated thesis - https://www.amazon.com/dp/1848902026
- My implementation of Curry's thesis in the Coq proof assistant language: https://www.github.com/CharlesAverill/HCLT
- BCKW wiki page - https://en.wikipedia.org/wiki/B,_C,_K,_W_system
- Tetris with combinators - http://dirk.rave.org/combinatris/