`Cont r` is not a Comonad

Matthew Doty

Outline

Stuff You Can't Program

Stuff You Can't Program — Halting Problem

The most famous negative result in computer science is due to Alan Turing [12]

Proposition

There is no Turing machine that can compute for all programs $\ulcorner T \urcorner$ *if they will halt or not on* x*.*

$\ulcorner T \urcorner$ is the code for Turing machine T given a *universal* Turing machine U

Stuff You Can't Program — Other Impossible Tasks

Gödel	Can't compute if there's a proof in Peano arithmetic for an arbitrary ϕ [2]
Kolmogorov	Can't make the perfect compression algorithm [4]
Soare	Can't compute if a program halts on every input [11]

Stuff You Can't Program — Ever More Impossible I

If you *could* compute for any program if would halt on an input, you could compute for any ϕ if it was provable in arithmetic or not.

However, you could not compute if a program halts *on every input*. This is even more *uncomputable*!

Stuff You Can't Program — Ever More Impossible II

The relation "*if I could compute x, I could compute y*" (called *Turing reducibility*) creates classes of algorithms like complexity theory.

For every problem P, we can make a Q where even if we could solve P then Q would *still* be impossible.

There are an infinite number of seperable classes in higher complexity theory under Turing reducibility.

Stuff You Can't Program — Another *Type* of Impossibility

All the impossibility theorems mentioned involve some kind of *diagonal argument* or another.

Types were invented by Bertrand Russell to avoid the paradox he uncovered in Frege's *Die Grundlagen der Arithmetik* [9].

A century later we can use types as an alternate way of proving programs don't exist.

The Continuation Monad

The Continuation Monad — Definition

```haskell
newtype Cont r a
  = Cont {runCont :: (a → r) → r}

instance Monad (Cont r) where
  return x = Cont ($ x)
  s >>= f =
    Cont $ \c →
      runCont s $ \x →
        runCont (f x) c
```

The Continuation Monad — Example Usage

`Cont r` can be used for a non-local exit

This is called an *escape continuation*; see the `mtl` package for more details

```
whatsYourName :: String → String
whatsYourName name = (`runCont` id) $ do
  response ← callCC $ \exit → do
    when (null name) $
      exit "Must have a name!"
    return $ "Welcome, " ++ name ++ "!"
  return response
```

Combinatory Logic

Combinatory Logic — History

Topic of Haskell Curry's PhD thesis in 1930 [1]

Based on earlier work by Moses Schönfinkel in 1924 [10]

The presentation here has been formalized in Isabelle/HOL

Combinatory Logic — Syntax

```
datatype Var = Var nat ("𝒳")

datatype SKComb =
    Var_Comb Var ("⟨_⟩" [100] 100)
  | S_Comb ("S")
  | K_Comb ("K")
  | Comb_App "SKComb" "SKComb"  (infixl "·" 75)

datatype 'a Simple_Type =
    Atom 'a  ("⦃ _ ⦄" [100] 100)
  | To "'a Simple_Type" "'a Simple_Type" (infixr "⇒" 70)
```

Combinatory Logic — Simple Typing

Simple typing is achieved for the pure S and K combinators using the following inductively defined predicate $(::)$

$$\frac{}{\mathsf{K} \ :: \ \varphi \ \Rightarrow \ \psi \ \Rightarrow \ \varphi} \ \text{K\_TYPE}$$

$$\frac{}{\mathsf{S} \ :: \ (\varphi \ \Rightarrow \ \psi \ \Rightarrow \ \chi) \ \Rightarrow \ (\varphi \ \Rightarrow \ \psi) \ \Rightarrow \ \varphi \ \Rightarrow \ \chi} \ \text{S\_TYPE}$$

$$\frac{\mathsf{E}_1 \ :: \ \varphi \ \Rightarrow \ \psi \qquad \mathsf{E}_2 \ :: \ \varphi}{\mathsf{E}_1 \ \cdot \ \mathsf{E}_2 \ :: \ \psi} \ \text{APPLICATION\_TYPE}$$

Combinatory Logic — Lambda-Abstraction I

The λ-calculus can be embedded in combinatory logic.
The embedding here is due to David Turner [13].

$$
\begin{array}{lcl}
\texttt{free}_{SK}\ (\langle \texttt{x} \rangle) & = & \{\texttt{x}\} \\
\texttt{free}_{SK}\ \texttt{S} & = & \emptyset \\
\texttt{free}_{SK}\ \texttt{K} & = & \emptyset \\
\texttt{free}_{SK}\ (\texttt{E}_1 \cdot \texttt{E}_2) & = & \texttt{free}_{SK}\ \texttt{E}_1 \cup \texttt{free}_{SK}\ \texttt{E}_2
\end{array}
$$

Combinatory Logic — Lambda-Abstraction II

$$
\begin{aligned}
\boldsymbol{\lambda}\mathtt{x.}\ \mathsf{S} &= \mathsf{K} \cdot \mathsf{S} \\
\boldsymbol{\lambda}\mathtt{x.}\ \mathsf{K} &= \mathsf{K} \cdot \mathsf{K} \\
\boldsymbol{\lambda}\mathtt{x.}\ \langle \mathtt{y} \rangle &= \textit{if}\ \mathtt{x} = \mathtt{y}\ \textit{then}\ \mathsf{S} \cdot \mathsf{K} \cdot \mathsf{K} \\
&\ \textit{else}\ \mathsf{K} \cdot \langle \mathtt{y} \rangle \\
\boldsymbol{\lambda}\mathtt{x.}\ (\mathsf{E}_1 \cdot \mathsf{E}_2) &= \textit{if}\ \mathtt{x} \in \mathtt{free}_{SK}\ (\mathsf{E}_1 \cdot \mathsf{E}_2) \\
&\ \textit{then}\ \mathsf{S} \cdot \boldsymbol{\lambda}\mathtt{x.}\ \mathsf{E}_1 \cdot \boldsymbol{\lambda}\mathtt{x.}\ \mathsf{E}_2 \\
&\ \textit{else}\ \mathsf{K} \cdot (\mathsf{E}_1 \cdot \mathsf{E}_2)
\end{aligned}
$$

Kripke Semantics

Kripke Semantics — History I

Kripke Semantics refers to possible world semantics given a *transition* relation.

Credited to the mathematician Saul Kripke, who invented these models for logic while in high school in 1959 [5]

Kripke Semantics — History II

In the 60s & 70s, Hoare [3] and Pratt [8] adapted Kripke models to *Labeled Transition Systems* in order to generalize Kleene's *regular expressions*.

Also in the 70s, Pnueli used Kripke semantics for *Linear Temporal Logic* (LTL), which he proposed for formal verification [7]. This inspired Leslie Lamport to ultimately create *TLA+* [6].

Kripke Semantics — History III

We are going to use Kripke semantics for *intuitionistic logic*. This is the logic of simple types according to the Curry-Howard correspondence.
In a sense, Kripke semantics can be seen as the *dual* to Combinatory logic.

Kripke Semantics — Data Structure

record ('a, 'b) Kripke_Model =
R :: "'a ⇒ 'a ⇒ bool"
V :: "'a ⇒ 'b ⇒ bool"

Kripke Semantics — Model Theory

Define the *Tarski Truth Predicate* $\models$ inductively as follows:

$$\mathfrak{M}\ \mathtt{x} \models \{\!|\ \mathtt{v}\ |\!\}$$
$$=$$
$$\exists \mathtt{w}.\ (\mathtt{R}\ \mathfrak{M})^{**}\ \mathtt{w}\ \mathtt{x} \wedge \mathtt{V}\ \mathfrak{M}\ \mathtt{w}\ \mathtt{v}$$

$$\mathfrak{M}\ \mathtt{x} \models \varphi \Rightarrow \psi$$
$$=$$
$$\forall \mathtt{y}.\ (\mathtt{R}\ \mathfrak{M})^{**}\ \mathtt{x}\ \mathtt{y} \longrightarrow \mathfrak{M}\ \mathtt{y} \models \varphi \longrightarrow \mathfrak{M}\ \mathtt{y} \models \psi$$

Here $(\mathtt{R}\ \mathfrak{M})^{**}$ is the *reflexive, transitive closure* of the relation $\mathtt{R}$ for $\mathfrak{M}$

Kripke Semantics — Properties I

Monotony:

$$\frac{(\mathsf{R}\ \mathfrak{M})^{**}\ \mathsf{x}\ \mathsf{y} \qquad \mathfrak{M}\ \mathsf{x} \models \varphi}{\mathfrak{M}\ \mathsf{y} \models \varphi}$$

Proof Sketch: Use induction on φ while allowing y to be free.

Kripke Semantics — Properties II

Monotony, as well as reflexivity and transitivity of $(\mathtt{R}\ \mathfrak{M})^{**}$ give us three other derived rules:

$$\frac{}{\mathfrak{M}\ \ \mathsf{x}\ \models\ \varphi\ \Rightarrow\ \psi\ \Rightarrow\ \varphi}$$

$$\frac{}{\mathfrak{M}\ \ \mathsf{x}\ \models\ (\varphi\ \Rightarrow\ \psi\ \Rightarrow\ \chi)\ \Rightarrow\ (\varphi\ \Rightarrow\ \psi)\ \Rightarrow\ \varphi\ \Rightarrow\ \chi}$$

$$\frac{\mathfrak{M}\ \ \mathsf{x}\ \models\ \varphi\ \Rightarrow\ \psi \qquad \mathfrak{M}\ \ \mathsf{x}\ \models\ \varphi}{\mathfrak{M}\ \ \mathsf{x}\ \models\ \psi}$$

These reflect the Combinatory logic typing rules K_TYPE, S_TYPE and APPLICATION_TYPE respectively

Kripke Semantics — Soundness

$$\texttt{If } \exists \texttt{X. X} :: \varphi \texttt{ then } \forall \mathfrak{M} \texttt{ x. } \mathfrak{M} \texttt{ x} \models \varphi \texttt{.}$$

The other direction (called *completeness*) also holds

...But the proof is complicated

Kripke Semantics — Comonad Refresher I

```haskell
class Comonad w where
  extract :: w a → a
  duplicate :: w a → w (w a)
```

Kripke Semantics — Comonad Refresher II

Comonads obey the laws:

```
extract        . duplicate = id
fmap extract . duplicate = id

extract . fmap f
  = f . extract

duplicate . duplicate
  = fmap duplicate . duplicate
```

Kripke Semantics — Comonad Refresher III

`Cont r` cannot be a monad, because we will show it is impossible to write

```
extract :: ((a → r) → r) → a
```

Kripke Semantics — `extract` Counter Example I

Lemma

Let $\mathfrak{M}$ *be*

$(\!| \mathtt{R = \lambda x\ y.\ x = a \wedge y = b,\ V = \lambda x\ y.\ x = b \wedge y = p} |\!)$

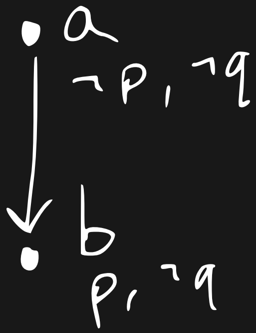
where $\mathtt{a} \neq \mathtt{b}$ *and* $\mathtt{p} \neq \mathtt{q}$

then

$$\neg\ \mathfrak{M}\ \mathtt{a} \models ((\{\!|\ \mathtt{p}\ |\!\} \Rightarrow \{\!|\ \mathtt{q}\ |\!\}) \Rightarrow \{\!|\ \mathtt{q}\ |\!\}) \Rightarrow \{\!|\ \mathtt{p}\ |\!\}$$

Kripke Semantics — `extract` Counter Example II

Here's a diagram of what's going on in this model:



Kripke Semantics — `extract` Counter Example III

Proof.

First observe that $\mathfrak{M}\ \mathtt{b} \models \{\!\!\{\ \mathtt{p}\ \}\!\!\}$ and $\neg\ \mathfrak{M}\ \mathtt{b} \models \{\!\!\{\ \mathtt{q}\ \}\!\!\}$.

Since $(\mathtt{R}\ \mathfrak{M})^{**}\ \mathtt{b}\ \mathtt{y} \equiv \mathtt{b} = \mathtt{y}$, then

$$\neg\ \mathfrak{M}\ \mathtt{b} \models \{\!\!\{\ \mathtt{p}\ \}\!\!\} \Rightarrow \{\!\!\{\ \mathtt{q}\ \}\!\!\}$$

Kripke Semantics — `extract` Counter Example IV

In order to show $\neg\ \mathfrak{M}\ a \models \{\!|\ p\ |\!\} \Rightarrow \{\!|\ q\ |\!\}$, we must find a x such that:

1. $(R\ \mathfrak{M})^{**}\ a\ x$
2. $\mathfrak{M}\ x \models \{\!|\ p\ |\!\}$
3. $\neg\ \mathfrak{M}\ x \models \{\!|\ q\ |\!\}$

We can see that $x = b$ works.

Since all we have is a and b to worry about, we have:

$$\forall x.\ (R\ \mathfrak{M})^{**}\ a\ x \longrightarrow \neg\ \mathfrak{M}\ x \models \{\!|\ p\ |\!\} \Rightarrow \{\!|\ q\ |\!\}$$

Hence $\mathfrak{M}\ a \models (\{\!|\ p\ |\!\} \Rightarrow \{\!|\ q\ |\!\}) \Rightarrow \{\!|\ q\ |\!\}$ vacuously.

Kripke Semantics — `extract` Counter Example V

But since $\neg\ \mathfrak{M}\ \mathsf{a}\ \models\ \{\!|\ \mathsf{p}\ |\!\}$, then by modus ponens we have our result!

$\square$

Kripke Semantics — No Combinator For `extract`

By the soundness result previously established

$$\texttt{If}\ \exists \texttt{X}.\ \texttt{X} :: \varphi\ \texttt{then}\ \forall \mathfrak{M}\ \texttt{x}.\ \mathfrak{M}\ \texttt{x} \models \varphi.$$

And from the lemma we just proved, if $\texttt{p} \neq \texttt{q}$ then

$$\nexists \texttt{X}.\ \texttt{X} :: ((\{\!|\ \texttt{p}\ |\!\} \Rightarrow \{\!|\ \texttt{q}\ |\!\}) \Rightarrow \{\!|\ \texttt{q}\ |\!\}) \Rightarrow \{\!|\ \texttt{p}\ |\!\}$$

Follow Up

Follow Up — ContT Monad Transformer I

```haskell
newtype ContT r m a
  = ContT {runContT :: (a → m r) → m r}

type Cont r = ContT r Identity
```

Follow Up — `ContT` Monad Transformer II

`ContT` is thought to *not be* a functor in the category of monads

Can we prove this?

Follow Up — ContT Monad Transformer III

That is, is there a monad `m` and a `c` such that there is no function:

```
hoist ::
  forall a b.
  (m a → m b) →
  (ContT c m a → ContT c m b)
```

which obeys these laws:

```
hoist (f . g) = hoist f . hoist g
hoist id = id
```

?

Bibliography

[1] H. B. Curry, *Grundlagen der Kombinatorischen Logik*, American Journal of Mathematics, 52 (1930), pp. 509–536.

[2] K. Gödel, *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*, Monatshefte für Mathematik und Physik, 38 (1931), pp. 173–198.

[3] C. A. R. Hoare, *An axiomatic basis for computer programming*, Communications of the ACM, 12 (1969), pp. 576–580.

[4] A. N. Kolmogorov, *Three approaches to the quantitative definition of information*, International Journal of Computer Mathematics, 2 (1968), pp. 157–168.

[5] S. A. Kripke, *A Completeness Theorem in Modal Logic*, The Journal of Symbolic Logic, 24 (1959), pp. 1–14.

[6] L. Lamport, *Specifying Concurrent Systems with TLA+*, Calculational System Design, (1999), pp. 183–247.

[7] A. Pnueli, *The temporal logic of programs*, in 18th Annual Symposium on Foundations of Computer Science (Sfcs 1977), Providence, RI, USA, Sept. 1977, IEEE, pp. 46–57.

[8] V. R. Pratt, *Semantical Considerations on Floyd-Hoare Logic*, 1976.

[9] B. Russel, *Letter To Frege*, in From Frege to Gödel, J. van Heijenoort, ed., Harvard Univ. Pr, Cambridge, Mass, 4. pr ed., 1981, pp. 124–125.
OCLC: 256165246.

[10] M. Schönfinkel, *Über die Bausteine der mathematischen Logik*, Mathematische Annalen, 92 (1924), pp. 305–316.

[11] R. I. Soare, *Turing Computability: Theory and Applications*, Theory and Applications of Computability / in Cooperation with the Association Computability in Europe, Springer, Berlin Heidelberg, 2016.
OCLC: 954167269.

[12] A. M. Turing, *On Computable Numbers, with an Application to the Entscheidungsproblem*, Proceedings of the London Mathematical Society, s2-42 (1937), pp. 230–265.

[13] D. A. Turner, *Another algorithm for bracket abstraction*, Journal of Symbolic Logic, 44 (1979), pp. 267–270.