Control as Proof

And the Route to Composition, in Three Parts

Paul Downen

Shonan 203: Effect Handlers & General Purpose Languages

Thursday, September 28, 2023 — Programmer-Facing Aspects

Curry-Howard is a double-barreled name that ensures the existence of other double-barreled names.

— Philip Wadler

Prelude:
The Classics

A Type for `call-with-current-continuation`

A dramatic reenactment

MATTHIAS: I've been working on giving a type to call/cc. It seems to be

$$\text{call/cc} : ((a \to b) \to a) \to a$$

TIMOTHY: That's impossible!

MATTHIAS: What?! How could you know? I just showed you!

TIMOTHY: Because that's Peirce's Law! A classical axiom! Everyone knows that classical logic has no computational interpretation!

MATTHIAS: Well, it looks fine to me. Try to prove it wrong.

TIMOTHY: *work, work, work …*

A Formulae-as-Types Notion of Control

Timothy G. Griffin*
Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

The programming language Scheme contains the control construct `call/cc` that allows access to the current continuation (the current control context). This, in effect, provides Scheme with first-class labels and jumps. We show that the well-known formulae-as-types correspondence, which relates a constructive proof of a formula α to a program of type α, can be extended to a typed Idealized Scheme. What is surprising about this correspondence is that it relates *classical* proofs to typed programs. The existence of computationally interesting "classical programs" — programs of type α, where α holds classically, but not constructively — is illustrated by the definition of conjunctive, disjunctive, and existential types using standard classical definitions. We also prove that all evaluations of typed terms in Idealized Scheme are finite.

in general, classical proofs lack computational content. This paper shows, however, that the formulae-as-types correspondence *can* be extended to classical logic in a computationally interesting way. It is shown that classical proofs posses computational content when the notion of computation is extended to include explicit access to the current control context.

This notion of computation is found in the programming language Scheme [16], which contains the control construct `call/cc`[1] that provides access to the current continuation (the current control context). This, in effect, provides Scheme with first-class labels and jumps, and allows for programs that are more efficient than purely functional programs. The formulae-as-types correspondence presented in this paper is based on a typed version of *Idealized Scheme* — a typed ISWIM containing an operator $\mathcal{C}$ similar to `call/cc` — developed by Felleisen *et al* [3,2,4] for reasoning about Scheme programs.

Control Operators for Classical Axioms

Stepping out of the intuitionistic comfort zone

Peirce's Law:	$((a \to b) \to a) \to a$
Double Negation Elimination:	$((a \to \bot) \to \bot) \to a$
Ex Falso Quodibet	$\bot \to a$

Control Operators for Classical Axioms

Stepping out of the intuitionistic comfort zone

Peirce's Law:	call/cc :	$((a \to b) \to a) \to a$
Double Negation Elimination:	$\mathcal{C}$:	$((a \to \bot) \to \bot) \to a$
Ex Falso Quodibet	abort :	$\bot \to a$

$\lambda\mu$-CALCULUS: AN ALGORITHMIC INTERPRETATION OF CLASSICAL NATURAL DEDUCTION

Michel Parigot
Equipe de logique — CNRS UA 753
45-55 5ème étage, Université Paris 7
2 place jussieu, 75251 PARIS Cedex 05, FRANCE
e-mail: parigot@logique.jussieu.fr

1 INTRODUCTION

This paper presents a way of extending the paradigm "proofs as programs" to classical proofs. The system we use is derived from the general Free Deduction system presented in [3].

Usually when considering proofs as programs, one has only in mind some kind of intuitionistic proofs. There is an obvious reason for that restriction: only intuitionistic proofs are contructive, in the sense that from the proof of an existential statement, one can get a witness of this existential statement. But from the programming point of view, constructivity is only needed for Σ^0_1-statements, for which classical and intuitionistic provability coincide. This means that, classical proofs are also candidates for being programs. In order to use them as programs, one has two tasks to achieve:

(i) to find a system in which one can extract directly a program from a classical proof (and not by means of a translation to intuitionistic logic), and

(ii) to understand the algorithmic meaning of classical constructions.

$\lambda\mu$-Calculus: A Language For Classical Logic

Labels and jumps from a logician's perspective

$$\frac{\Gamma \vdash M : A}{\langle M \| \alpha \rangle \ : \ \Gamma \vdash \alpha : A} \ \textit{Passivate} \qquad \frac{c \ : \ \Gamma \vdash \alpha : A}{\Gamma \vdash \mu\alpha.c : A} \ \textit{Activate}$$

$\lambda\mu$-Calculus: A Language For Classical Logic

Labels and jumps from a logician's perspective

$$\frac{\Gamma \vdash M : A \mid \Delta}{\langle M \| \alpha \rangle \ : \ \Gamma \vdash \alpha : A, \Delta}\ \textit{Passivate} \qquad \frac{c \ : \ \Gamma \vdash \alpha : A, \Delta}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ \textit{Activate}$$

$$\begin{aligned} (\beta) \qquad & (\lambda x.M)\ N \to M\{N/x\} \\ (\textit{relabel}) \qquad & \langle \mu\alpha.c \| \beta \rangle \to c\{\beta/\alpha\} \\ (\textit{capture}_1) \qquad & (\mu\alpha.c)\ M \to \mu\alpha'.c\left\{\langle N\ M \| \alpha' \rangle / \langle N \| \alpha \rangle\right\} \end{aligned}$$

$\lambda\mu$-Calculus: A Language For Classical Logic

Labels and jumps from a logician's perspective

$$\frac{\Gamma \vdash M : A \mid \Delta}{\langle M \| \alpha \rangle \ : \ \Gamma \vdash \alpha : A, \Delta}\ \textit{Passivate} \qquad \frac{c \ : \ \Gamma \vdash \alpha : A, \Delta}{\Gamma \vdash \mu\alpha.c : A \mid \Delta}\ \textit{Activate}$$

$$\begin{aligned} (\beta) \qquad & (\lambda x.M)\ N \to M\{N/x\} \\ (\textit{relabel}) \qquad & \langle \mu\alpha.c \| \beta \rangle \to c\{\beta/\alpha\} \\ (\textit{capture}_1) \qquad & (\mu\alpha.c)\ M \to \mu\alpha'.c\left\{\langle N\ M \| \alpha' \rangle / \langle N \| \alpha \rangle\right\} \end{aligned}$$

$$\langle (\mu\alpha.c)\ M_1\ M_2 \ldots M_n \| \beta \rangle \twoheadrightarrow c\{\langle N\ M_1\ M_2 \ldots M_n \| \beta \rangle / \langle N \| \alpha \rangle\}$$

Modeling Control With μs

A shorthand for classical proofs

$$\text{call/cc}\ (h : (A \to B) \to A) = \mu\alpha{:}A.\ \langle h\ (\lambda x{:}A.\mu\beta{:}B.\ \langle x \| \alpha \rangle) \| \alpha \rangle$$

$$\mathcal{C}\ (h : (A \to \bot) \to \bot) = \mu\alpha{:}A.\ \langle h\ (\lambda x{:}A.\mu\beta{:}\bot.\ \langle x \| \alpha \rangle) \| \quad \rangle$$

$$\text{abort}\ (x : \bot) = \mu\alpha{:}A.\ \langle x \| \quad \rangle$$

Modeling Control With μs

A shorthand for classical proofs

$$\text{call/cc}\ (h : (A \to B) \to A) = \mu\alpha{:}A.\ \langle h\ (\lambda x{:}A.\mu\beta{:}B.\ \langle x \| \alpha \rangle) \| \alpha \rangle$$

$$\mathcal{C}\ (h : (A \to \bot) \to \bot) = \mu\alpha{:}A.\ \langle h\ (\lambda x{:}A.\mu\beta{:}\bot.\ \langle x \| \alpha \rangle) \| \mathbf{tp} \rangle$$

$$\text{abort}\ (x : \bot) = \mu\alpha{:}A.\ \langle x \| \mathbf{tp} \rangle$$

$$\frac{\Gamma \vdash M : \bot \mid \Delta}{\langle M \| \mathbf{tp} \rangle\ :\ \Gamma \vdash \Delta}\ \textit{ExFalso}$$

Correspondence Between Control and Classical Logic

$$M, N ::= x \mid \lambda x.M \mid M\ N \mid \mu\alpha.c \qquad c ::= \langle M \| \alpha \rangle \mid \langle M \| \mathbf{tp} \rangle$$

$$[\![\mu\alpha.c]\!] = \text{call/cc}(\lambda\alpha.[\![c]\!])$$
$$[\![\langle M \| \alpha \rangle]\!] = \alpha\ M$$
$$[\![\langle M \| \mathbf{tp} \rangle]\!] = \text{abort}\ M$$

Equational correspondence between

Classical ($\lambda\mu$) proofs $\equiv$ λ-calculus + call/cc

Classical ($\lambda\mu$) proofs + Ex Falso (**tp**) $\equiv$ λ + call/cc + abort
$\equiv$ λ-calculus + $\mathcal{C}$

Act I:
As Outside, Inside

The Prompt

Practically, all control operators (like call/cc) have to end at the interpreter prompt `scheme # ...[call/cc`

Control & Prompt: What if the programmer could insert their own prompt inside the program? Nicer rewriting semantics:

$$\#V \to V \qquad \#E[\text{control } f] \to \#f\ (\lambda x.E[x])$$

Shift & Reset: Inspired by a compositional CPS semantics

$$\#V \to V \qquad \#E[\text{shift } f] \to \#f\ (\lambda x.\#E[x])$$

Rebinding tp

The prompt # is an internal top-level

What if we just re-bind the top-level label **tp** as $\mu\mathbf{tp}.c$? So

$$\#M = \mu\mathbf{tp}.\langle M \| \mathbf{tp}\rangle \quad \text{shift } f = \mu\alpha.\langle f\ (\lambda x.\mu\mathbf{tp}.\langle x\|\alpha\rangle)\|\mathbf{tp}\rangle$$

Simplest example: identity continuation

$$\#\,\text{shift } f \to \#\,f\ (\lambda x.\ \#\ x) \to \#\,f\ (\lambda x.x)$$

Rewriting into μs and **tp**s…

$$\mu\mathbf{tp}.\langle\mu\alpha.\langle f\ (\lambda x.\mu\mathbf{tp}.\langle x\|\alpha\rangle)\|\mathbf{tp}\rangle\|\mathbf{tp}\rangle$$
$$\to \mu\mathbf{tp}.\langle f\ (\lambda x.\mu\mathbf{tp}.\langle x\|\alpha\rangle)\,\{\mathbf{tp}/\alpha\}\|\mathbf{tp}\rangle \quad (\mathit{relabel}\,\{\mathbf{tp}/\alpha\})$$

The Scope of the Top Level

Static or dynamic?

How to substitute under re-bindings: $(\mu\mathbf{tp}.\langle x \| \alpha \rangle)\,\{\mathbf{tp}/\alpha\} = ?$

Static, capture-avoiding substitution:

$$(\mu\mathbf{tp}.\langle x \| \alpha \rangle)\,\{\mathbf{tp}/\alpha\} = \mu\mathbf{tp}'.\langle x \| \mathbf{tp} \rangle$$

α-renaming = Abort!

Dynamic, capture-allowing substitution:

$$(\mu\mathbf{tp}.\langle x \| \alpha \rangle)\,\{\mathbf{tp}/\alpha\} = \mu\mathbf{tp}.\langle x \| \mathbf{tp} \rangle$$

Capture = Compose!

The Dynamically-Rebindable Top Level

Put a hat on $\widehat{\mathrm{tp}}$ to advertise its dynamic scope

Shift and reset boil down to dynamically-rebinding $\widehat{\mathrm{tp}}$

$$\#V \to V \qquad \#E[\mathsf{shift}\ V] \to \#V\ (\lambda x.\#E[x])$$
$$\mu\widehat{\mathrm{tp}}.\langle V \| \widehat{\mathrm{tp}}\rangle \to V \qquad \text{already done by } \mu\alpha.c$$

Reducing to the identity continuation:

$$\begin{aligned}
&\mu\widehat{\mathrm{tp}}.\langle \mu\alpha.\langle f\ (\lambda x.\mu\widehat{\mathrm{tp}}.\langle x\|\alpha\rangle)\|\widehat{\mathrm{tp}}\rangle\|\widehat{\mathrm{tp}}\rangle \\
&\to \mu\widehat{\mathrm{tp}}.\langle f\ (\lambda x.\mu\widehat{\mathrm{tp}}.\langle x\|\widehat{\mathrm{tp}}\rangle)\|\widehat{\mathrm{tp}}\rangle && (\mathit{relabel}\ \{\widehat{\mathrm{tp}}/\alpha\}) \\
&\to \mu\widehat{\mathrm{tp}}.\langle f\ (\lambda x.x)\|\widehat{\mathrm{tp}}\rangle && (\mathit{lookup}\ \widehat{\mathrm{tp}})
\end{aligned}$$

Continuing the Dynamic Top Level

Dynamic binding metaphor: obvious generalization to many different dynamic continuations, $\hat{\alpha}$, $\hat{\beta}$, $\hat{\gamma}$, ...

Previously on: Multiple Prompts

Most direct analogue to effect handlers (by name)

Expressiveness:

Control + Read-only dynamic environment

$\approx$ Control + Read/write mutable State

$\Rightarrow$ Shift + Reset (Representing Monads, Filinski)

Act II:
The Collapse

$\Lambda\mu$-Calculus: Collapsing $\lambda\mu$'s Artificial Barriers

Being sloppy with syntax, with style

Parigot's original $\lambda\mu$ syntax:

$$M, N ::= x \mid \lambda x.M \mid M\ N \mid \mu\alpha.c \qquad\qquad c ::= \langle M \| \alpha \rangle$$

De Groot's style: it's easier to have one syntactic category

$\Lambda\mu$ separates jumps/commands by type, not by syntax

$$M, N ::= x \mid \lambda x.M \mid M\ N \mid \mu\alpha.M \mid \langle M \| \alpha \rangle$$

$$\frac{\Gamma \vdash M : A \mid \Delta}{\Gamma \vdash \langle M \| \alpha \rangle : \perp\!\!\!\perp \mid \alpha : A, \Delta}\ \textit{Passivate} \qquad \frac{\Gamma \vdash M : \perp\!\!\!\perp \mid \alpha : A, \Delta}{\Gamma \vdash \mu\alpha.M : A \mid \Delta}\ \textit{Activate}$$

For well typed terms, $\lambda\mu$ and $\Lambda\mu$ are the same

$\Lambda\mu$-Calculus: Collapsing $\lambda\mu$'s Artificial Barriers

Being sloppy with syntax, with style

Parigot's original $\lambda\mu$ syntax:

$$M, N ::= x \mid \lambda x.M \mid M\ N \mid \mu\alpha.c \qquad\qquad c ::= \langle M \| \alpha \rangle$$

De Groot's style: it's easier to have one syntactic category

$\Lambda\mu$ separates jumps/commands by type, not by syntax

$$M, N ::= x \mid \lambda x.M \mid M\ N \mid \mu\alpha.M \mid \langle M \| \alpha \rangle$$

$$\frac{\Gamma \vdash M : A \mid \Delta}{\Gamma \vdash \langle M \| \alpha \rangle : \perp\!\!\!\perp \mid \alpha : A, \Delta}\ \textit{Passivate} \qquad \frac{\Gamma \vdash M : \perp\!\!\!\perp \mid \alpha : A, \Delta}{\Gamma \vdash \mu\alpha.M : A \mid \Delta}\ \textit{Activate}$$

For well typed terms, $\lambda\mu$ and $\Lambda\mu$ are the same

…but what if we ignore types?

What's Different In $\Lambda\mu$?

Nothing new, and yet new capabilities

All the same syntactic constructs as before, and yet...

$\Lambda\mu$ satisfies Böhm Separability like λ-calculus, $\lambda\mu$ does not

Property (Böhm Separability)
Given any two normalizing terms M and N, if $M \neq N$ (by $\beta, \eta, \ldots$) then there is a separating context C where $C[M] = x$ and $C[N] = y$.

$\Lambda\mu$ lets you compose continuations, $\lambda\mu$ does not

$$\langle\langle\langle x \| \alpha\rangle \| \beta\rangle \| \gamma\rangle$$

$\Lambda\mu$ lets you dig through continuation stacks, $\lambda\mu$ does not

$$\mu\alpha.\mu\beta.\mu\gamma.M$$

The Surprising Power of $\Lambda\mu$

And the natural emergence of delimited control from classical logic

Every command $\langle M \| \alpha \rangle$ acts as a natural delimiter

$\Lambda\mu > \lambda +$ shift $+\#$

$\Lambda\mu$ expresses an unlimited hierarchy of shifts

The Surprising Power of $\Lambda\mu$

And the natural emergence of delimited control from classical logic

Every command $\langle M \| \alpha \rangle$ acts as a natural delimiter

$\Lambda\mu > \lambda + \mathsf{shift} + \#$

$\Lambda\mu$ expresses an unlimited hierarchy of shifts

$\Lambda\mu = \Lambda\mu\tilde{\mu}$

$$\langle N \| \tilde{\mu} x.M \rangle = \langle N \| \mathbf{let}\ x\ \mathbf{in}\ M \rangle = \langle \mathbf{let}\ x = N\ \mathbf{in}\ \mu\_.M \| \delta \rangle$$

$\Lambda\mu\tilde{\mu} \equiv \lambda + \mathsf{shift}_0 + \#$

$$\mathsf{shift}_0\, h = \mu\alpha.\ h\,(\lambda x.\langle x \| \alpha \rangle)$$

$$\#M = \langle M \| \mathbf{let}\ x\ \mathbf{in}\ x \rangle$$

Continuing $\Lambda\mu$

An account of delimited control using only canonical tools

- All you need is the familiar capture-avoiding substitution
- Expresses delimiters without any notion of dynamic scope
- No dynamic handlers or top-level continuation bindings

Continuation composition stacked like function arguments

- Multiple “prompts” passed/accessed by position, not name
- Simple & fast implementation via de Bruijn indexes

Yet, same expressive power as shift_0

Possible analogue to static effect handlers?

Act III: Addition By Subtraction

The Rub

There's still more work to be done

Despite all this, we don't have a canonical type system for delimited control

- So far, there are a collection of type-and-effect systems
- Non-canonical, because type checking changes depending on evaluation order
- Let alone a logic that logicians would care about
- (In contrast, same type system works for CBV and CBN λ-calculus, even with call/cc, state, ...)
- Can we do better?

Duality of functions

Flipping the usual implication around

$$\frac{A\,\mathbf{true} \vdash \qquad B\,\mathbf{true}}{A \to B\,\mathbf{true}} \to R \qquad \frac{A\,\mathbf{true} \qquad B\,\mathbf{false}}{A \to B\,\mathbf{false}} \to L$$

Duality of functions

Flipping the usual implication around

$$\frac{A\,\mathbf{true} \vdash \qquad B\,\mathbf{true}}{A \to B\,\mathbf{true}} \to R \qquad \frac{A\,\mathbf{true} \qquad B\,\mathbf{false}}{A \to B\,\mathbf{false}} \to L$$

$$\frac{A\,\mathbf{true} \vdash \qquad B\,\mathbf{true}}{A - B\,\mathbf{false}} - L \qquad \frac{A\,\mathbf{true} \qquad B\,\mathbf{false}}{A - B\,\mathbf{true}} - R$$

Duality of functions

Flipping the usual implication around

$$\frac{x : A\,\mathbf{true} \vdash M : B\,\mathbf{true}}{\lambda x.M : A \to B\,\mathbf{true}} \to R \qquad \frac{V : A\,\mathbf{true} \quad E : B\,\mathbf{false}}{V \cdot E : A \to B\,\mathbf{false}} \to L$$

$$\frac{x : A\,\mathbf{true} \vdash M : B\,\mathbf{true}}{\lambda x.M : A - B\,\mathbf{false}} - L \qquad \frac{V : A\,\mathbf{true} \quad E : B\,\mathbf{false}}{V \cdot E : A - B\,\mathbf{true}} - R$$

$M : A\,\mathbf{true}$ means "M returns a value of type A"

$E : A\,\mathbf{false}$ means "E takes a value of type A"

Subtraction as yield + resume

As seen in scripting languages

$$\mathbf{yield}\, x = \mu\alpha.\mu\beta.\langle x \cdot \mathbf{let}\, y\, \mathbf{in}\, \langle\langle y \| \alpha\rangle \| \beta\rangle \| \beta\rangle$$

$$M\, \mathbf{resume}\, x \to N = \langle\langle M \| \mathbf{let}\, y\, \mathbf{in}\, \mu\_.y\rangle \| \lambda x.N\rangle$$

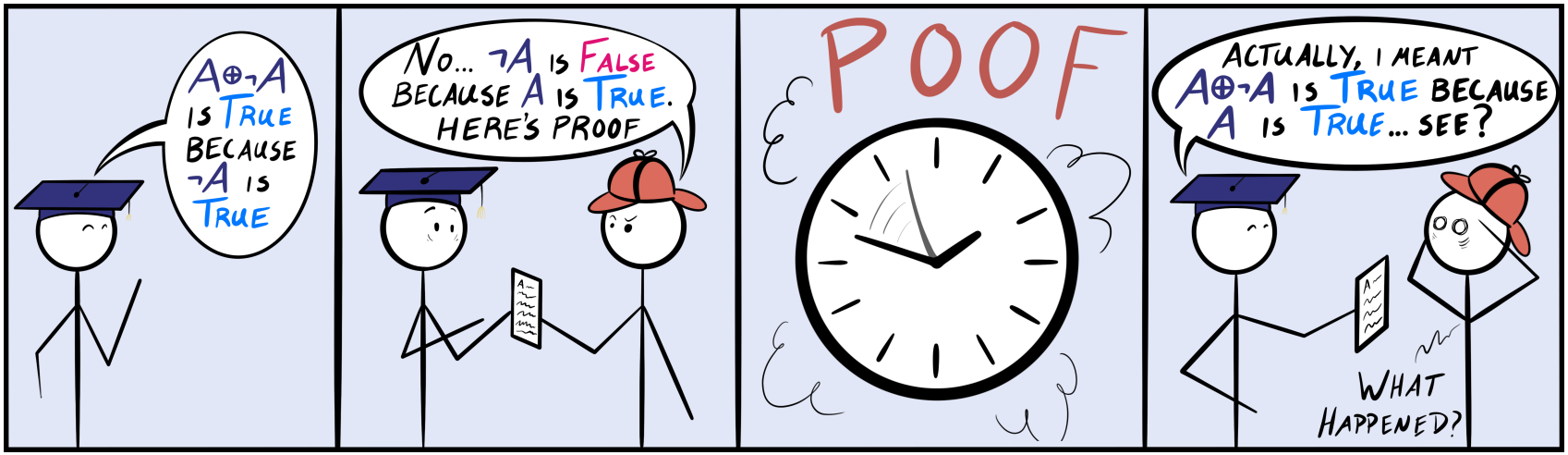
Subtraction as yield + resume

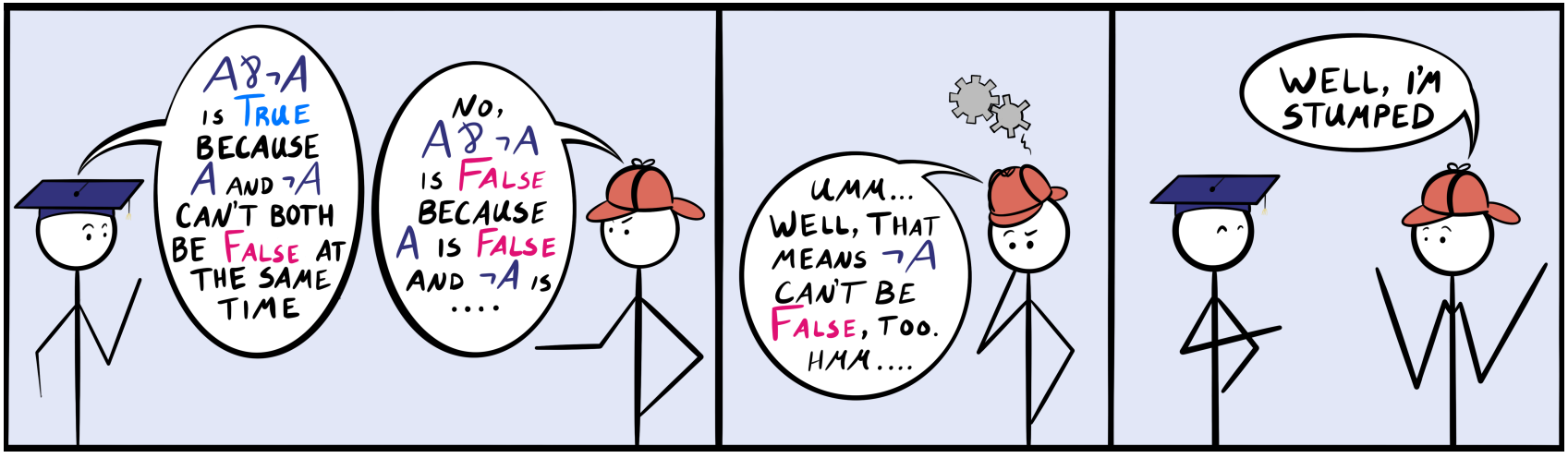
As seen in scripting languages

$$\mathbf{yield}\, x = \mu\alpha.\mu\beta.\langle x \cdot \mathbf{let}\, y\, \mathbf{in}\, \langle\langle y \| \alpha\rangle \| \beta\rangle \| \beta\rangle$$

$$M\, \mathbf{resume}\, x \to N = \langle\langle M \| \mathbf{let}\, y\, \mathbf{in}\, \mu\_.y\rangle \| \lambda x.N\rangle$$

Challenge: Can we assign one type to these sorts of restricted control operators, maybe using exotic types ($A - B$, $A \parr B$, ...) that is safe for both CBN and CBV evaluation?


A⊕¬A IS TRUE BECAUSE ¬A IS TRUE
NO... ¬A IS FALSE BECAUSE A IS TRUE. HERE'S PROOF
POOF
ACTUALLY, I MEANT A⊕¬A IS TRUE BECAUSE A IS TRUE... SEE?
WHAT HAPPENED?


A∨¬A IS TRUE BECAUSE A AND ¬A CAN'T BOTH BE FALSE AT THE SAME TIME
NO, A∨¬A IS FALSE BECAUSE A IS FALSE AND ¬A IS ....
UMM... WELL, THAT MEANS ¬A CAN'T BE FALSE, TOO. HMM....
WELL, I'M STUMPED