A User-Friendly Programming Language for a Logical Abstract Machine

Paul Downen

Eastern Canada Logic and Programming Seminar — Saturday, November 14, 2026

A Little Context

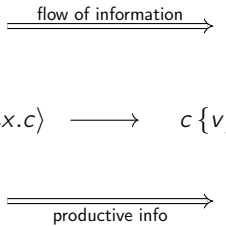
Sequent calculus: a symmetric language

Commands c

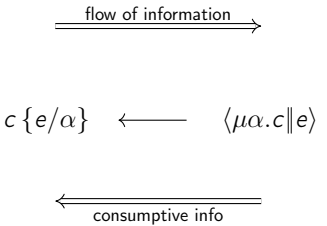
$\langle v \| e \rangle$

Producers v	Consumers e (contexts)
Function abstraction	Function call (call stack)
$\lambda x.v$	$v \cdot e$
Input variable	Output variable (co-variable)
x	α
Output abstraction	Input abstraction (let binding)
$\mu\alpha.c$	$\tilde{\mu}x.c$
...	...

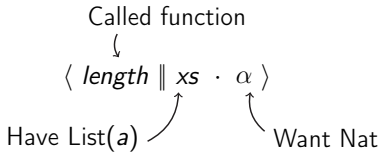
Flow of information



Flow of information

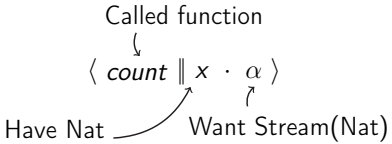


Recursion on data structures



$$\langle \mathit{length} \| \mathsf{Nil} \cdot \alpha \rangle = \langle \mathsf{Z} \| \alpha \rangle$$
$$\langle \mathit{length} \| \mathsf{Cons}(x, xs) \cdot \alpha \rangle = \langle \mathit{length} \| xs \cdot \tilde{\mu} y . \langle \mathsf{S}(y) \| \alpha \rangle \rangle$$

Recursion on co-data structures



$$\langle count \| x \cdot \mathsf{Head}[\alpha] \rangle = \langle x \| \alpha \rangle$$
$$\langle count \| x \cdot \mathsf{Tail}[\alpha] \rangle = \langle count \| \mathsf{S}(x) \cdot \alpha \rangle$$

Cool Story

Me: “This duality stuff is really neat!”

You: “Oh? That’s kind of interesting. How can I actually use it?”

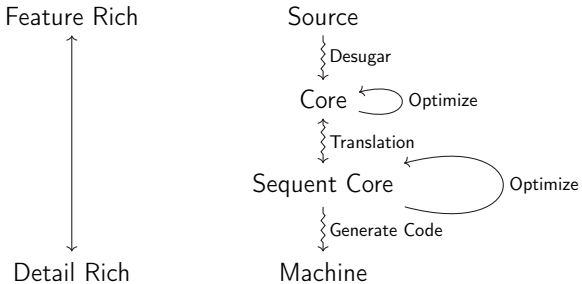
Cool Story

Me: “This duality stuff is really neat!”

You: “Oh? That’s kind of interesting. How can I actually use it?”

Me: ¯_(ツ)_/¯

Sequent-Style Intermediate Language



- Two-Way translation doesn't care about evaluation strategy
- Sequent Core = sequent calculus counterpart to Core

Duality in Compilers!

Case-of-Case and Friends

In Sequent Core (using Core syntax):

$$
\begin{array}{l}
\textbf{let}\ j\ x\ y = \mathit{big} \\
\textbf{in}\ \mathit{not}(\textbf{case}\ z\ \textbf{of}\ A\ x\ y \to j\ x\ y \\
\qquad\qquad\qquad\qquad\quad B \qquad \to \mathit{False}) \\
\qquad\qquad\qquad\qquad \Downarrow \\
\textbf{let}\ j\ x\ y = \mathit{not}\ \mathit{big} \\
\textbf{in}\ \textbf{case}\ z\ \textbf{of}\ A\ x\ y \to j\ x\ y \\
\qquad\qquad\qquad\qquad\quad B \qquad \to \mathit{not}\ \mathit{False}
\end{array}
$$

This is much better! The join point is preserved!

Getting Warmer

Me: “This duality stuff actually runs now! It can optimize your program!” ◡̈

You: “Oh? Cool! How can I write one of those programs for myself?”

Getting Warmer

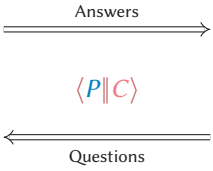
Me: “This duality stuff actually runs now! It can optimize your program!” ◡̈

You: “Oh? Cool! How can I write one of those programs for myself?”

Me: “Uh…well…It’s more just something for compilers…I don’t recommend actually reading or writing the code…” ⌒̈

Still Banging That Drum...

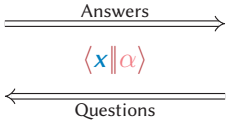
Dualities of Computation



Producer	Consumer
Answers	Questions
Program	Context

Dualities of Computation

Embodying the Context



A producer $x : A$ gives an answer of type A

A consumer $\alpha \div A$ asks a question of type A

A command $\langle x \| \alpha \rangle$ is an interaction at a type

$$\frac{\Gamma \vdash x : A \quad \Gamma \vdash \alpha \div A}{\Gamma \vdash \langle x \| \alpha \rangle}\ \mathit{Cut}$$

Smuggling Small Treasures in the Back Door

Tremendously useful for compiler internals

- Sequent Core
- Join points — In both Haskell (GHC) and OCaml
- Calling Conventions are dual to Unboxed Data Representations

Smuggling Small Treasures in the Back Door

- Tremendously useful for compiler internals
  - Sequent Core
  - Join points — In both Haskell (GHC) and OCaml
  - Calling Conventions are dual to Unboxed Data Representations
- Copatterns are a nice retro-fit for a lot of languages
  - Functional — Scheme
  - Object-Oriented — Python

Smuggling Small Treasures in the Back Door

Tremendously useful for compiler internals

- Sequent Core
- Join points — In both Haskell (GHC) and OCaml
- Calling Conventions are dual to Unboxed Data Representations

Copatterns are a nice retro-fit for a lot of languages

- Functional — Scheme
- Object-Oriented — Python

But still, these languages don't let me express myself…

And Now?

What would it really be like to use this as a programming language?

Would it be useful?

Would it be easy?

Would it be fun?

…

And Now?

What would it really be like to use this as a programming language?

Would it be useful?

Would it be easy?

Would it be fun?

…

Let's find out!

Sequent Calculus as Abstract Machine

Dual Core Language ≈ Sequent Calculus

Based on the $\mu\tilde{\mu}$-family of sequent calculi (equating terms and coterms)

$$
\begin{aligned}
\mathit{Command} \ni \quad & J ::= \langle M \parallel N \rangle \\
(\mathit{Co})\mathit{Term} \ni \ & M, N ::= x \mid \mu x.\, J \mid P[M/x...] \mid \mu\{P.\, J \mid ...\} \\
(\mathit{Co})\mathit{Pattern} \ni \ & P, Q ::= \dots
\end{aligned}
$$

Dual Core Language ≈ Sequent Calculus

Based on the $\mu\tilde{\mu}$-family of sequent calculi (equating terms and coterms)

$$
\begin{aligned}
\mathit{Command} \ni \quad & J ::= \langle M \parallel N \rangle \\
(\mathit{Co})\mathit{Term} \ni \ & M, N ::= x \mid \mu x.\, J \mid P[M/x...] \mid \mu\{P.\, J \mid ...\} \\
(\mathit{Co})\mathit{Pattern} \ni \ & P, Q ::= \dots
\end{aligned}
$$

But I can't type that on my keyboard…

Dual Core Language ≈ Sequent Calculus

Based on the $\mu\tilde{\mu}$-family of sequent calculi (equating terms and coterms)

$$
\begin{aligned}
\mathit{Command} \ni \quad & J ::= \langle M \parallel N \rangle \\
(\mathit{Co})\mathit{Term} \ni \ & M, N ::= x \mid \mu x.\, J \mid P[M/x...] \mid \mu\{P.\, J \mid ...\} \\
(\mathit{Co})\mathit{Pattern} \ni \ & P, Q ::= \ldots
\end{aligned}
$$

But I can't type that on my keyboard…

$$
\begin{aligned}
\mathit{Command} \ni \quad & J ::= \mathbf{go}\ M\ \mathbf{to}\ N \\
(\mathit{Co})\mathit{Term} \ni \ & M, N ::= x \mid \mathbf{do}\ x \Rightarrow J \mid P[M/x...] \mid \mathbf{do}\ \{\ P \Rightarrow J \mid ...\ \} \\
(\mathit{Co})\mathit{Pattern} \ni \ & P, Q ::= \ldots
\end{aligned}
$$

On Statements versus Expressions

And running versus returning

Same divide as in imperative languages:

- Expressions (i.e., terms) return something
- Statements (i.e., commands) just run for their effect

In pure sequent calculus, there is only one command: *Cut*

For an abstract machine, a *Cut* means jump of some form

On Statements versus Expressions

And running versus returning

Same divide as in imperative languages:

- Expressions (i.e., terms) return something
- Statements (i.e., commands) just run for their effect

In pure sequent calculus, there is only one command: *Cut*

For an abstract machine, a *Cut* means jump of some form

$\mathbf{go}\,()\,\mathbf{to}\,\mathit{label} = \mathbf{go}\,\mathbf{to}\,\mathit{label}$ simple jump

On Statements versus Expressions

And running versus returning

Same divide as in imperative languages:

- Expressions (i.e., terms) return something
- Statements (i.e., commands) just run for their effect

In pure sequent calculus, there is only one command: *Cut*

For an abstract machine, a *Cut* means jump of some form

- **go** () **to** *label* = **go to** *label* — simple jump
- **go** (*arg*,...) **to** *label* — jump with arguments

On Statements versus Expressions

And running versus returning

Same divide as in imperative languages:

- Expressions (i.e., terms) return something
- Statements (i.e., commands) just run for their effect

In pure sequent calculus, there is only one command: *Cut*

For an abstract machine, a *Cut* means jump of some form

go () **to** *label* = **go to** *label*	simple jump
go (*arg*,...) **to** *label*	jump with arguments
go *f* **to** (*arg*,..., *ret*)	function call

On Statements versus Expressions

And running versus returning

Same divide as in imperative languages:

- Expressions (i.e., terms) return something
- Statements (i.e., commands) just run for their effect

In pure sequent calculus, there is only one command: *Cut*

For an abstract machine, a *Cut* means jump of some form

go () **to** *label* = **go to** *label*	simple jump
go (*arg*,...) **to** *label*	jump with arguments
go *f* **to** (*arg*,..., *ret*)	function call

Other statements are sugar on top of **go to**

val $x = M; J$	is	**go** M	**to do** $x \Rightarrow J$
fun $f\ x... = \ldots; J$	is	**go do** $\{ (x, ..., ret) \Rightarrow \ldots \}$	**to do** $f \Rightarrow J$

On Statements AS Expressions

Same divide as in many functional languages:

- Sequence together effectful operations
- Capture their final result as a whole expression

This looks similar to **do**-notation popular in FP…

$$
\begin{aligned}
&\textbf{do}\ k \Rightarrow \\
&\quad \textbf{val}\ x = f\ 1; \\
&\quad \textbf{val}\ y = g\ x\ 2; \\
&\quad \textbf{go}\ x + y\ \textbf{to}\ k
\end{aligned}
$$

On Statements AS Expressions

Same divide as in many functional languages:

- Sequence together effectful operations
- Capture their final result as a whole expression

This looks similar to **do**-notation popular in FP…

$$
\begin{aligned}
&\textbf{do} \quad \Rightarrow \\
&\quad \textbf{val}\ x = f\ 1; \\
&\quad \textbf{val}\ y = g\ x\ 2; \\
&\quad \textbf{go}\ x + y
\end{aligned}
$$

…especially if we leave the label implicit!

An Uncompromising Commitment to Duality

$$\textbf{go } M \textbf{ to } N = \textbf{go } N \textbf{ to } M$$

Abstract Machine as Source Language

Problem: No Function Application

Seriously?? Yes...

Consider the distance function

$$dist((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Problem: No Function Application

Seriously?? Yes...

Consider the distance function

$$dist((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

In a functional language (e.g., SML), this is just

```
fun dist (x1,y1) (x2,y2) = sqrt(square(x1 - x2) + square(y1 - y2));
```

Problem: No Function Application

Seriously?? Yes...

Consider the distance function

$$dist((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

In a functional language (e.g., SML), this is just

```
fun dist (x1,y1) (x2,y2) = sqrt(square(x1 - x2) + square(y1 - y2));
```

Writing this directly in Dual Core, it is (my deepest condolences)

```
fun dist = do {| ((x1,y1), (x2,y2), k) =>
  go do j1 => go square to ((do j2 => go (-) to (x1, x2, {j2})), {j1})
  to do x' => go do j3 => go square to ((do j4 => go (-) to (y1, y2, {j4})), {j3})
              to do y' => go sqrt   to ((do j5 => go (+) to (x', y', {j5})), k) };
```

Solution: Support Both Styles

Allow both “natural” (elimination) and sequent (left) styles

To reduce real redundancy, they must be equal in all contexts

The rules for equality are already known from

- The correspondence between NJ and LJ
- The rules from abstract machines

Solution: Support Both Styles

Allow both “natural” (elimination) and sequent (left) styles

To reduce real redundancy, they must be equal in all contexts

The rules for equality are already known from

- The correspondence between NJ and LJ
- The rules from abstract machines

$$\mathbf{go}\, f(x)\, \mathbf{to}\, k = \mathbf{go}\, f\, \mathbf{to}\, (x, k) \qquad \langle f(x) \parallel k \rangle \mapsto \langle f \parallel x, k \rangle$$

$$\mathbf{go}\, f.Proj\, \mathbf{to}\, k = \mathbf{go}\, f\, \mathbf{to}\, Proj[k] \qquad \langle f.Proj \parallel k \rangle \mapsto \langle f \parallel Proj[k] \rangle$$

Problem: Determining Evaluation Order

Let's keep things simple: (M, N) is eager like call-by-value

- First evaluate $M = V$
- Second, evaluate $N = W$
- Then the result is (V, W)

Problem: Determining Evaluation Order

Let's keep things simple: (M, N) is eager like call-by-value

First evaluate $M = V$

Second, evaluate $N = W$

Then the result is (V, W)

Congratulations! I hope you like Call-By-Push-Value!

Remember, function application = pushing the call stack

$$\mathbf{go}\, f(M)\, \mathbf{to}\, N = \mathbf{go}\, f\, \mathbf{to}\, (M, N)$$

Also have an unbreakable commitment to duality

$$\mathbf{go}\, f\, \mathbf{to}\, (M, N) = \mathbf{go}\, (M, N)\, \mathbf{to}\, f$$

We've now run out of choices!

Solution 1: Manual Shifts

What should $f(g(x))$ do?

CBV: $f(g(x))$ evaluates $g(x)$ first, passes its value.

CBN: $f(g(x))$ passes the suspended code $g(x)$ as a value.

Solution 1: Manual Shifts

What should $f(g(x))$ do?

CBV: $f(g(x))$ evaluates $g(x)$ first, passes its value. ✓ default

CBN: $f\{g(x)\}$ passes the suspended code $\{g(x)\}$ as a value.

Solution 1: Manual Shifts

What should $f(g(x))$ do?

CBV: $f(g(x))$ evaluates $g(x)$ first, passes its value. ✓ default

CBN: $f\{g(x)\}$ passes the suspended code $\{g(x)\}$ as a value.

That means higher-order calls are also suspended, like

$$map\ \{add\ 1\} \qquad\qquad xs.Each\ \{\mathbf{fn}\ x \Rightarrow print(x * x)\}$$

Solution 1: Manual Shifts

What should $f(g(x))$ do?

CBV: $f(g(x))$ evaluates $g(x)$ first, passes its value. ✓ default

CBN: $f\{g(x)\}$ passes the suspended code $\{g(x)\}$ as a value.

That means higher-order calls are also suspended, like

$$map\ \{add\ 1\} \qquad\qquad xs.Each\ \{\mathbf{fn}\ x \Rightarrow print(x * x)\}$$

Matching on suspenders { } lets us define shifts

$$\mathbf{fun}\ force\ \{x\} = x$$
$$\mathbf{fun}\ return\ x\ = \mathbf{do}\ \{k\} \Rightarrow \mathbf{go}\ x\ \mathbf{to}\ k$$
$$\mathbf{fun\ go}\ force\ \{x\}\ \mathbf{to}\ k\ = \mathbf{go}\ x\ \mathbf{to}\ k$$
$$\mathbf{fun\ go}\ return\ x\ \ \mathbf{to}\ \{k\} = \mathbf{go}\ x\ \mathbf{to}\ k$$

Solution 2: Polarity Correction

Why manually shift when you can have automatic drive?

Only need to manually shift parameters

All other shifts can be inferred on the context!

The simplest definition of *return* is

$$\textbf{fun}\ \mathit{return}\ x = x$$

Even $f(g(x))$ has an implicit shift,

$$f(g(x)) = f\ !\ g(x)$$

Problem(?): Resolving the Duality of Types?

To support duality, need involutive negation $A^{\perp}$ where $A^{\perp\perp} = A$

$$\textbf{go}\ M\ \textbf{as}\ A\ \textbf{to}\ N = \textbf{go}\ N\ \textbf{as}\ A^{\perp}\ \textbf{to}\ N = \textbf{go}\ M\ \textbf{as}\ A^{\perp\perp}\ \textbf{to}\ N$$

Problem(?): Resolving the Duality of Types?

To support duality, need involutive negation $A^{\perp}$ where $A^{\perp\perp} = A$

$$\textbf{go}\ M\ \textbf{as}\ A\ \textbf{to}\ N = \textbf{go}\ N\ \textbf{as}\ A^{\perp}\ \textbf{to}\ N = \textbf{go}\ M\ \textbf{as}\ A^{\perp\perp}\ \textbf{to}\ N$$

Must we duplicate every type?

Are connectives restricted by polarities?

How do we automatically infer which one?

Solution: Factoring Out Polarity

Doesn't matter!

Just factor out the polarity direction from the type!

A : **val** produces an output of shape A

A : **fun** consumes an input of shape A

$(A : \textbf{val})^{\perp} = A : (\textbf{val}^{\perp}) = A : \textbf{fun}$ and vice versa

A cut **go** M **to** N must agree on a shape A and connect opposites (**val** vs **fun**)

Reduced number of connectives; no polarity restrictions

$()$ $\quad$ (A, B) $\quad$ $[Tag_i : A_i \mid \ldots]$ $\quad$ $\{A\}$

Solution: Factoring Out Polarity

Doesn't matter!

Just factor out the polarity direction from the type!

- A : **val** produces an output of shape A
- A : **fun** consumes an input of shape A
- $(A : \textbf{val})^{\perp} = A : (\textbf{val}^{\perp}) = A : \textbf{fun}$ and vice versa

A cut **go** M **to** N must agree on a shape A and connect opposites (**val** vs **fun**)

Reduced number of connectives; no polarity restrictions

$$() \qquad (A, B) \qquad [Tag_i : A_i \mid \ldots] \qquad \{A\}$$

Note lack of arrows; they are derived

$$A \to B = A \rightsquigarrow \{B\} \qquad A \rightsquigarrow B = (A, B)$$

Let's Try it Out

Programming With Unrestricted Duality

New forms of structured control!

Many new ways to write programs with non-functional control flow

Calculating with and on the evaluation context

Multiple methods of error handling

- Pattern-matching bind with recovery
- Multiple (handler) continuations
- First-class pattern-matching alternatives

Recursive **match** statements

Coo: A friendly little language to talk about co-things



github.com/pdownen/coo