Typed Program Analysis Without Encodings

Barry Jay
barry.jay8@gmail.com

21st January, 2025

Abstract

In a tree calculus, the *natural* trees, without labels, are reduced to binary trees, which are both the values and the programs. Thus, the size, equality and evaluation of programs are easily defined. The two type derivation rules, for nodes and applications, are parametrised by a subtyping relation that can be tuned to support the examples. No encodings are required.

Programs have a Dual Nature

Programs are functions from inputs to outputs.

Programs are also data, e.g. numbers or syntax trees.

Usually, syntax trees are recovered by a meta-theoretic *encoding* of functions but it is better to make the syntax trees fundamental and use algebra to describe functionality. For example, in combinatory logic (CL) the reduction rule of

$$Sxyz \longrightarrow xz(yz)$$

maps a ternary tree with root S to a tree in which z and yz are attached to the right of the root of x. However,

- Syntax is artificial (why not add I, B or C?).
- CL can't analyse programs, e.g. separate SKK from SKS

Natural Trees Support Program Analysis

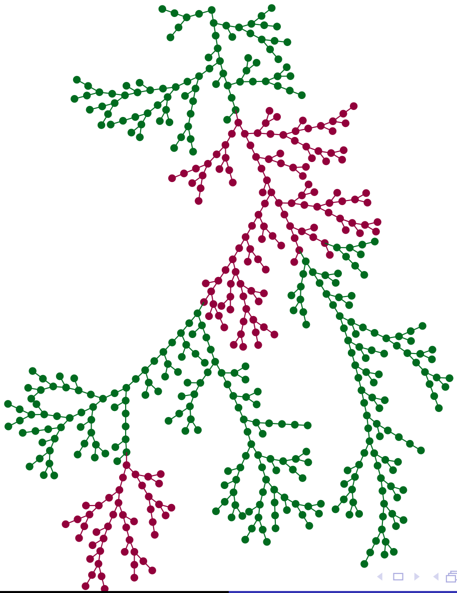
Replace each operator *S*, *K* etc. by a small tree so that there are no labels: the trees are now natural. That is, admit one operator only, called "node" written $\triangle$.

Make the binary trees be the values. That is $\triangle$ is a ternary operator.

Choose reduction rules to support both extensionality (like *S* and *K*) and intensionality (for program analysis).

It is a choice. The original tree calculus in my book Reflective Programs in Tree Calculus has three reduction rules. It was used to define the self-interpreter below. Note how the tree structure reflects the program structure.

A Breadth-First Self-Interpreter, **bf** as in the Book



Triage Calculus

Here, we use triage calculus (suggested by Johannes Bader) as it is easier to understand.

$$M, N := \triangle \mid MN\ .$$

$$\begin{array}{lcl}
\triangle\triangle\ yz & \longrightarrow & y \\
\triangle(\triangle x)\ yz & \longrightarrow & xz(yz) \\
\triangle(\triangle wx)y\triangle & \longrightarrow & w \\
\triangle(\triangle wx)y(\triangle\ u) & \longrightarrow & x\ u \\
\triangle(\triangle wx)y(\triangle\ u\ v) & \longrightarrow & y\ u\ v\ .
\end{array}$$

Now combinatory logic is covered by

$$\begin{array}{rcl}
K & = & \triangle\triangle \\
S_1\{x\} & = & \triangle(\triangle x)
\end{array}$$

while leaves, stems and forks are separated by

$$\textbf{triage}\{w, x, y\} = \triangle(\triangle wx)y\ .$$

Type Derivation

$$T, U, V := \mathbf{L} \mid \mathbf{S}\ U \mid \mathbf{F}\ U\ V \mid U \to V \mid \ldots$$

The *tree types* **L** and **S** U and **F** U V are types for leaves, stems and forks. Every program has a tree type that exactly describes its structure. To be a useful program, this tree type must be a subtype of a function type $U \to V$. Since all computations are either a node or an application, two type derivation rules suffice

$$\frac{\mathbf{Leaf} < T}{\triangle : T} \qquad \frac{M : U \to V \quad N : U}{MN : V} .$$

Expressive power is determined by *subtyping* $U < V$.

Typing Programs

$$\begin{array}{rcl}\mathbf{L} & < & U \to \mathbf{S}\ U \\ \mathbf{S}\ U & < & V \to \mathbf{F}\ U\ V\end{array}$$

is enough to type programs by leaf, stem or fork types.

$$U_2 < U_1 \text{ and } V_1 < V_2 \text{ implies } U_1 \to V_1 < U_2 \to V_2$$

is enough to support subsumption

$$\frac{M : T_1}{M : T_2}\ T_1 < T_2$$

Typing Redexes

Each reduction rule suggests a subtyping rule

$$\begin{array}{rcl}
\mathbf{F}\,\mathbf{L}\,T & < & V \to T \\
\mathbf{F}\,(\mathbf{S}\,(U \to V \to T))\,(U \to V) & < & U \to T \\
\mathbf{F}\,(\mathbf{F}\,T\,V)\,W & < & \mathbf{L} \to T \\
\mathbf{F}\,(\mathbf{F}\,U\,(V \to T))\,W & < & \mathbf{S}\,V \to T \\
\mathbf{F}\,(\mathbf{F}\,U\,V)\,(W_1 \to W_2 \to T) & < & \mathbf{F}\,W_1\,W_2 \to T
\end{array}$$

These imply

$$\begin{array}{rcl}
K & : & U \to V \to U \\
S & : & (U \to V \to T) \to (U \to V) \to (U \to T) \\
I & : & T \to T
\end{array}$$

Fixpoint Functions

We can define the Y combinator in the usual manner by

$$\begin{aligned} \textbf{omega} &= \lambda w.\lambda f.f(wwf) = S(K(SI))(SII) \\ Yf &= \textbf{omega omega } f \end{aligned}$$

but this is unstable since $Yf \longrightarrow f(Yf)$ so use

$$Z\{f\} = \textbf{wait2}\{\textbf{omega2}, \textbf{omega2}, f\}$$

where **wait2**$\{f, g, h\}$ is stable and
wait2$\{f, g, h\}x \longrightarrow fghx$ and
omega2 adds waiting to **omega.**
Now $Z\{f\}$ is the fixpoint function of f and
$Z\{f\}$ a normal form (a binary tree) if f is.

Typing Fixpoint Functions

Let **Omega2** be the tree type of **omega2.**
Define $\textbf{Fix}\{T\} = (T \to T) \to T$.
The subtyping rule

$$\textbf{Omega2} < \textbf{Omega2} \to \textbf{Fix}\{U \to V\}$$

is enough to type arithmetic but not polymorphism, so introduce quantified types, $\forall \vec{X}.T$ and the subtyping rule

$$\textbf{Omega2} < \textbf{Omega2} \to \textbf{Fix}\{\forall \vec{X}.(U \to V)\}\ .$$

Program Size

The pattern-matching pseudo-code

```
$\textbf{size}\ p =$
    $\textbf{match}\ p\ \textbf{with}$
     $\mid \triangle \Rightarrow 1$
     $\mid \triangle\ q \Rightarrow 1 + \textbf{size}\ q$
     $\mid \triangle\ q\ r \Rightarrow 1 + \textbf{size}\ q + \textbf{size}\ r$
    $\textbf{end}$
```

is captured by triage

$$\begin{aligned}\textbf{size} = Z\{\lambda s.\textbf{triage}\{&\textbf{one},\\ &\lambda q.\textbf{succ}\ (s\ q),\\ &\lambda q.\lambda r.\textbf{succ}\ (\textbf{plus}\ (s\ q)\ (s\ r))\}.\end{aligned}$$

Typing Triage

The natural type of the triage in **size** is

$$\mathbf{F}\ (\mathbf{F}\ \mathbf{Nat}\ (\forall X.X \to \mathbf{Nat}))\ (\forall X.\forall Y.X \to Y \to \mathbf{Nat})$$

so make this a subtype of the intended type $\forall Z.Z \to$ **Nat.**

We can generalise from **Nat** to any type T that is *covariant* in Z but the notation is heavy.

Breadth-First Evaluation in Pseudo-Code

```
**bf** $f\ a =$
    **match** $f$ **with**
     $| \triangle \Rightarrow \triangle a$
     $| \triangle\ z \Rightarrow \triangle\ z\ a$
     $| \triangle\ z\ y \Rightarrow$
        **match** $z$ **with**
         $| \triangle \Rightarrow y$
         $| \triangle\ x \Rightarrow$ **eager** (**bf** (**bf** $x\ a$)) (**bf** $y\ a$)
         $| \triangle\ w\ x \Rightarrow$
            **match** $a$ **with**
             $| \triangle \Rightarrow w$
             $| \triangle\ b \Rightarrow$ **bf** $x\ b$
             $| \triangle\ b\ c \Rightarrow$ **bf** (**bf** $y\ b$) $c$
    **end end end**
```

The breadth-first evaluation of $(xa)(ya)$ is enforced by **eager**.

Breadth-First Evaluation by Triage

$$\begin{aligned}
\mathbf{bf} &= Z\{\lambda e.\mathbf{triage}\{\triangle, \triangle, \mathbf{bff}\}\} \\
\mathbf{bff} &= \mathbf{triage}\{K, \mathbf{bffs}, \mathbf{bfff}\} \\
\mathbf{bffs} &= \lambda x.\lambda y.S_1\{S_1\{K\ \mathbf{eager}\}(S_1\{Ke\}(ex))\}(ey) \\
\mathbf{bfff} &= \lambda w.\lambda x.\lambda y.\mathbf{triage}\{w, ex, \mathbf{bfff\_fork}\ y\} \\
\mathbf{bfff\_fork} &= \lambda y.S_1\{Ke\}(ey)
\end{aligned}$$

Unfortunately, the existing rule for typing triage would require $\mathbf{bff} : U \to V \to \mathbf{F}\ U\ V$ but all we have is

$$\text{if } \mathbf{F}\ U\ V < Z \to T \text{ then } \mathbf{bff} : U \to V \to Z \to T.$$

Hacking the Type System

So capture the conditional in a new type form $\mathbf{A}\ T$ with the rules

$$\begin{array}{rcl} T & < & \mathbf{A}\ T \\ \mathbf{A}\ (U \to V) & < & U \to V \end{array}$$

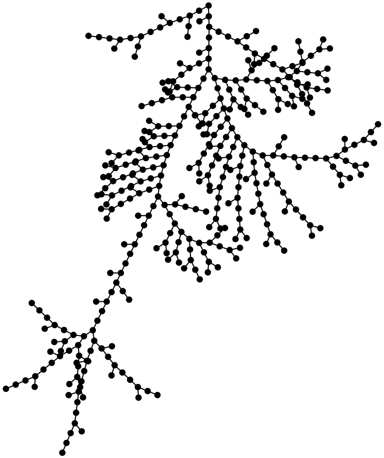
then hack some subtyping rules for sub-terms of **bf** to derive

$$\mathbf{bf} : \forall Z.Z \to \mathbf{A}\ Z$$

Specialising Z and subtyping $\mathbf{A}\ (X \to Y)$ to $X \to Y$ yields

$$\mathbf{bf} : \forall X.\forall Y.(X \to Y) \to (X \to Y)$$

Shrinking **bf** requires more type hacking



This typed version of **bf** has 392 nodes. Johannes Bader's untyped version has 352. How small can it go?

Kleene's Argument and Church's Thesis (1936)

Kleene used encodings to *compare* models of computation

$$\lambda \longleftrightarrow \mathit{CL} \longleftrightarrow \mu\text{-rec} \longleftrightarrow \mathit{TM}$$

Since they preserve *arithmetic*, the models were deemed *equivalent*.
Church's Thesis was that these models are *complete* for *computation*.
Refutation The double translation $\lambda \longrightarrow \mu$-rec $\longrightarrow \lambda$ of *normal forms* is not definable, so λ-calculus is incomplete!
Conjecture 2 Normal forms are not *values* (no recursion).
Refutation 2 The normal forms of CL represent all μ-recursive functions but the double translation is still not definable.
Conjecture 3 Combinators in normal form are not values since syntax trees are artificial, not natural like the numbers.
Refutation 3 Replace syntax trees with *natural trees* (no labels) in a *tree calculus*.

Simpler, yet more Expressive

- No encodings
- No expression types, or type hierarchy
- No meta-theory
- algebraic
- two typing rules
- mediated by a subtyping relation
- program manipulation is programming
- compilation is program execution

Key Ideas

- Encodings are patches.
- Theorems trump theses.
- Programs are normal forms.
- Programs are trees not numbers.
- Tree types become function types.
- Tricky programs require type hacking.

Links

My Blog
Reflective Programs in Tree Calculus
Tree Calculus Implementations

Conclusions

- Traditional models of computation use encodings to do program analysis as meta-theory, so are incomplete.
- In tree calculus, programs and values are binary natural trees, which can be analysed without encodings, e.g.
- the breadth-first evaluator **bf** can be self-applied as **bf bf**.
- Tree calculus should support many (all?) program analyses, e.g partial evaluation, complexity, type inference.
- Typing is **not** from logic, category theory or inheritance.
- Programs have tree types, subtypes of function types.
- Subtyping rules capture properties of programs such as fixpoints, queries or evaluators, to support, e.g.
- **bf** : $\forall X.\forall Y.(X \to Y) \to X \to Y$.