Typed Program Analysis without Encodings

Barry Jay barry.jay8@gmail.com

December 12, 2024

Abstract

Programs are viewed as both functions to be executed and data structures to be analysed, but this has always required encoding, e.g. of a lambda-term to a syntax tree, so that a self-interpreter could not be applied to itself directly, but only to its code. Further, the code of a typed program should have a distinctive type. In a tree calculus, however, program analysis can be supported directly, without encodings, including the self-application of a breadth-first self-interpreter of type

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

1 Introduction

Programs are viewed as both functions to be executed and data structures to be analysed. For example, an *interpreter* [25] may be given by a program **int** that is a function of programs viewed as data structures, including **int** itself [26, 36, 1, 32, 30, 33, 4, 38, 8, 35, 22, 5, 6]. Actually, interpreters are canonical examples of program analyses that: exercise the whole language; stand in for program compilation; and try to fold the meta-theory of computing into its theory, e.g. [24].

Unfortunately, traditional models of computation support only one of these two viewpoints. For example, Turing's machines are programs-as-functions, which must be encoded as tapes [41]. Conversely, if the tape is primary, as in a *partial combinatory alebra* [12], then decoding is required to expose the machinery. Again, a program p in a *lambda-calculus* [9, 2] or *combinatory logic* [37, 10, 15] is a function whose structure is revealed by encoding it as a syntax tree $\lceil p \rceil$. In that setting, self-interpretation is given by **int** $\lceil \mathbf{int} \rceil$. Since the encodings use meta-theory that is not definable within the calculi, they are incomplete for program analysis. Thus, program analysis is usually static: source code is parsed into syntax trees, which are analysed before being converted to, say lambda abstractions and machine code. Dynamic program analysis is then equivalent to analysing the tape of a Turing machine, which is not compositional. A more subtle approach is to use *staged compilation*, e.g. [34, 29] but still there is an irreversible flow from data structures to functions.

Even when the syntax tree is available, it has been hard to reason about, or even to give it a type without resorting to extensions of System Fω or higher kinds [8, 35, 5, 6]. If p is a program of type $U \to V$ then its encoding $\lceil p \rceil$ will require its own *expression type* $\mathbf{Exp}\{U \to V\}$ which adds an extra layer of complexity. Second, there are two or three natural choices of type for the interpreter [22]. The type $\mathbf{Exp}\{U \to V\} \to U \to V$ is that of a *self-recogniser* that decodes an expression back to its original program. The type

$$\mathbf{Exp}\{U \to V\} \to \mathbf{Exp}\{U\} \to \mathbf{Exp}\{V\}$$

is that of a *self enactor* which produces a function of expressions. The latter is more interesting, as it allows one to impose an evaluation strategy on reduction. Third, since encodings can be nested, one is obliged to consider expression types of expression types in an unbounded hierarchy. All of these difficulties are driven by the use of encodings.

These difficulties have seemed unavoidable. Not only are the traditional calculi *extensional*, unable to distinguish different implementations of the same function, but the recursive functions have been represented by terms that do *not* have normal forms. However, it turns out that combinatory logic can indeed represent recursive functions by normal forms [17], which may then be identified with the programs. That is, the notions of program and computation, which have been identified in lambda-calculus, can now be separated. All of the μ-recursive arithmetic functions [27] required for Turing completeness can be represented by programs in combinatory logic. In this sense, combinatory logic is complete for numerical analysis. However, it is not complete for program analysis, being unable to analyse its own programs, its own normal forms, even to separate different representations of the identity function.

Program analysis requires the *intensional computation* [18] of, say, a *factorisation calculus* (such as SF-calculus) [21] or that of [22]) or a *tree calculus* [19]. Among these, tree calculus is more natural, in the following sense. The computations in SF-calculus are represented by syntax trees with nodes labelled by S or F. Since S and F both act when given three arguments, i.e. are ternary operators, the programs are given by binary trees labelled by S or F. However, the choice of labels is artificial, not natural like the numbers, which makes for poor semantics. By contrast, a tree calculus has only one operator $\triangle$ called node, so that labels are redundant, and the syntax trees become *natural trees*, i.e. without labels. Since $\triangle$ is a ternary operator, the programs become the binary natural trees. The original tree calculus has three reduction rules, for the application of $\triangle$ to a leaf, stem or fork. Presumably the results in this paper can be developed there, but it is easier to work with a variant called *triage calculus* suggested by Johannes Bader in conversation. Its five reduction rules can be summarised by the mnemonic

$$\begin{array}{rcl} \triangle\triangle &=& K \\ \triangle(\triangle x) &=& Sx \\ \triangle(\triangle wx)y &=& \mathbf{triage}\{w,x,y\}\ . \end{array}$$

Extensional functions are supported by the use of K and S, while triage supports intensionality: the $\mathbf{triage}\{w,x,y\}$ specialises to w, x or y according to whether its next argument z is a leaf, stem or fork. This can be used to decide the size of a program, or equality of programs. Arguably, tree calculus is complete for program analysis. Certainly, it supports self-interpreters such as the *breadth-first* interpreter **bf** which can be applied to itself directly, as **bf bf**.

This paper reprises previous work on tree calculus (which has yet to survive peer review) and then goes on to develop a type theory for it. A high point of this is a type

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

for a self-interpreter.

Any type system for tree calculus must allow a single program to take both a data type and a function type. The principal type of each program is a *tree type* such as the type $\mathbf{F}\ (\mathbf{S}\ (\mathbf{S}\ \mathbf{L}))\ (\mathbf{S}\ \mathbf{L})$ built from type forms for leaves ($\mathbf{L}$) and stems ($\mathbf{S}$) and forks ($\mathbf{F}$). Any function type for a program is then given by *subtyping* of the tree type. For example, consider the reduction

$$Kuv \longrightarrow u\ .$$

Here $K = \triangle\triangle$ so that $Ku = \triangle\triangle u : \mathbf{F}\ \mathbf{L}\ U$ if $u : U$. To apply Ku to $v : V$ requires that $\mathbf{F}\ \mathbf{L}\ U$ be a subtype of some type $V \to T$. Further, for the reduction rule above to preserve typing, T must be U. So we have the subtyping axiom

$$\mathbf{F}\ \mathbf{L}\ U < V \to U\ .$$

In this manner, each of the reduction rules of the calculus inspires a subtyping axiom.

Once subtyping is involved, it can be made to do all of the interesting work, in that two type derivation axioms will suffice to type all programs, with one rule for applications and one that can type a leaf by any subtype of $\mathbf{L}$ (see Figure 2). Thus, the expressive power of the type system is determined by that of subtyping, which will grow throughout this paper. The general flavour of the approach is exemplified by the treatment of recursive functions.

In lambda calculus, the Y combinator is defined to be $\omega\omega$ where

$$\omega = \lambda x.\lambda f.f(xxf)\ .$$

This supports the reduction $Yf \longrightarrow f(\omega\omega f) = f(Yf)$ expected of a fixpoint combinator. However, this combinator Y does not have a normal form, as it reduces to $\lambda f.f(Yf)$ which contains itself as a subexpression. Thus Y can reduce to ever larger terms. It follows that programs in lambda calculus are unstable, and that program analysis is forced to work with encodings of programs as syntax trees.

According to Church's Thesis, combinatory logic is equivalent to lambda calculus, so one might expect the same problems to arise there, as when Y is defined to be the self-application of the translation $S(K(SI))(SII)$ of ω. However, combinatory logic supports finer control over evaluation than lambda calculus. By a process akin to η-expansion of f to $\lambda x.fx$, one can replace an application MN by $\mathbf{wait}\{M, N\}$ given by

$$S(S(KM)(KN))I$$

since $\mathbf{wait}\{M, N\}x$ reduces to MNx. Now $\mathbf{wait}\{M, N\}$ is a normal form if M and N are, so that every combinator is functionally equivalent to a normal form! In particular, we can define the combinator Y to be the normal form

$$Y = \mathbf{wait}\{\omega, \omega\} \; .$$

Unfortunately, this does not mean that Yf is a normal form when f is, since $Yf \longrightarrow f(Yf)$ as before.

Rather, given a binary function f we can construct a *fixpoint function* $Z\{f\}$ that is normal if f is, yet satisfies

$$Z\{f\}\; x \longrightarrow f\; Z\{f\}\; x \; .$$

It is defined to be of the form $\mathbf{wait2}\{\omega_2, \omega_2, f\}$ where **wait2** is analogous to **wait** and ω_2 is a normal form analogous to ω. In order to show $f : (U \to V) \to (U \to V)$ implies $Z\{f\} : U \to V$ it is enough to show that $\omega_2\omega_2 : \mathbf{Fix}\; (U \to V)$ where $\mathbf{Fix}\, T = (T \to T) \to T$. Supposing that there is a canonical type Ω_2 of ω_2, it is enough to add the subtyping axiom

$$\Omega_2 < \Omega_2 \to \mathbf{Fix}\; (U \to V) \; .$$

Thus, even in traditional combinatory logic, all of the μ-recursive arithmetic functions necessary for Turing completeness can be expressed as normal forms, and be simply-subtyped using type constants **Bool** and **Nat** and Ω_2 for booleans, natural numbers and the type of ω_2.

This subtyping rule is not inspired by the deduction rules of some logic, or the reduction rules of the calculus. Rather, it captures the correctness of the implementation of Z. That is, the proof that reduction preserves typing will recapitulate the proof that the $Z\{f\}$ meets its specification. As the sophistication of the correctness proofs increases, so too will that of the subtyping rules. For example, generic queries will require their own subtyping rule, to type examples such as

$$\begin{array}{rcl} \mathbf{size} & : & \forall X.X \to \mathbf{Nat} \\ \mathbf{equal} & : & \forall X.\forall Y.X \to Y \to \mathbf{Bool} \; . \end{array}$$

In turn, this will not be enough to type a self-evaluator **bf** that requires its first argument to be of function type. Instead we will show it has type $\forall Z.Z \to \mathbf{A}\; Z$ where $\mathbf{A}\; Z$ records the expectation that Z will become a function type. By specialising Z to $X \to Y$, the expectation is met and we can prove

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

Note that we now have a mixture of subtyping and the parametric polymorphism of universal quantifiers $\forall X$. Such mixtures are rare, but not unheard of [7, 16, 39]. Here, the instantiation of quantified types will be characterised by a subtyping rule.

It follows that there are many type systems for tree calculus, parametrised by the choice of types and subtyping relation. Each extension considered here will be introduced in its own section of the paper, with theorems asserting that the examples are well typed. Note however, that although the theorems are true for the type system then in play, the formal verification in **Coq** [3] (soon to be known as **Rocq**) will be given only once, for the richest type system, as defined in Section 9. For ease of concordance, the theorem names in the paper are the same as in the **Coq** verification.

The structure of the paper is as follows. Section 1 is the introduction. Section 2 shows how to type traditional combinatory logic so that all μ-recursive functions are given by typed normal forms. Section 3 introduces the variant of tree calculus used in this paper and shows some basic properties of its reduction system that will be used later to show correctness of **bf**. Sections 4 – 8 introduce examples of programs, prove that they meet their specifications, and give an informal account of their typing. In particular: Section 4 introduces lambda-abstractions; Section 5 introduces the booleans, the natural numbers and tagging; Section 6 introduces fixpoint functions; Section 7 introduces generic queries, exemplified by **size** and **equal**; and Section 8 introduces a self-interpreter. Section 9 introduces the richest type system we will consider, able to handle all of the earlier examples. Section 10 proves that the type system is sound, in that reduction preserves typing. Section 11 draws conclusions and considers future work.

2 Combinatory Logic

This section will show how traditional combinatory logic is able to represent all of the μ-recursive arithmetic functions, i.e. the functions required for Turing completeness, as normal forms, and give them types in a simply-typed system, equipped with subtyping.

The *combinations* of SK*-calculus*, also here called *traditional combinatory logic*, are

$$M, N ::= S \mid K \mid MN\ .$$

The *factorable forms* are the combinations of the form S, SM, SMN, K and KM. The *programs* are the factorable forms whose sub-expressions are also programs. Its two reduction rules are

$$\begin{array}{rcl} SMNP & \longrightarrow & MP(NP) \\ KMN & \longrightarrow & M\ . \end{array}$$

The reduction relation is the congruence generated by the reduction rules. That is, it is closed under application and is reflexive and transitive.

Some examples of programs and program constructions are

$$\begin{array}{rcl} I & = & SKK \\ \mathbf{wait}\{M,N\} & = & S(S(KM)(KN))I \\ \mathbf{wait2}\{M,N,P\} & = & S(S(S(KM)(KN))(KP))I \\ \omega_2 & = & S(K(S(SKK)))(S(S(KS) \\ & & (S(K(S(KS))) \\ & & (S(S(KS)(S(KK)(S(KS) \\ & & (S(S(KS)K)K))))(KK)))) \\ & & (K(K(SKK)))) \\ Z\{f\} & = & \mathbf{wait2}\{\omega_2,\omega_2,f\}\ . \end{array}$$

For example, we have

$$\begin{array}{rcl} \mathbf{wait}\{M,N\}x & = & S(S(KM)(KN))Ix \\ & \longrightarrow & S(KM)(KN)x(Ix) \\ & \longrightarrow & (KMx)(KNx)x \\ & \longrightarrow & MNx\ . \end{array}$$

Thus, $\mathbf{wait}\{M,N\}$ is a factorable form that has the same functional behaviour as MN and so is a program if M and N are. Similarly, we have $\mathbf{wait2}\{M,N,P\}x \longrightarrow MNPx$ for any terms M,N,P and x.

The combination ω_2 can be thought of as a representation of the lambda-abstraction

$$\lambda w.\lambda f.f\ \mathbf{wait2}\{w,w,f\}\ .$$

Thus, $Z\{f\}$ is a program if f is, and

$$Z\{f\}x \longrightarrow \omega_2\omega_2 fx \longrightarrow f(\omega_2\omega_2 f)x = f\ Z\{f\}\ x$$

shows that $Z\{f\}$ is a fixpoint function for f.

The representation of booleans and natural numbers is traditional, with

$$\begin{array}{rcl} \mathbf{tt} & = & K \\ \mathbf{ff} & = & KI \\ \mathbf{zero} & = & KI \\ \mathbf{succ} & = & S(S(KS)K) \end{array}$$

where the successor **succ** may be thought of as representing $\lambda n.\lambda f.\lambda x.f(nfx)$. Then pairing, conditionals, composition, primitive recursion and minimisation can all be defined in the usual manner, on the understanding that minimisation uses Z to ensure that all the μ-recursive functions are represented by programs. Now let us see how to type them.

The *simple types* are here given by

$$T, U, V ::= \mathbf{Bool} \mid \mathbf{Nat} \mid \Omega_2 \mid U \to T \ .$$

That is they are function types built from the type constants **Bool** (for booleans), **Nat** (for natural numbers) and Ω_2 (for ω_2). The intended semantics of the type constants is captured by the *subtyping relation*, whose rules are given by

$$\begin{array}{rcl} \mathbf{Bool} & < & U \to U \to U \\ \mathbf{Nat} & < & (U \to U) \to (U \to U) \\ \Omega_2 & < & \Omega_2 \to \mathbf{Fix}\{U \to V\} \end{array}$$

where $\mathbf{Fix}\{T\} = (T \to T) \to T$. The subtyping of **Bool** supports conditionals. The subtyping of **Nat** supports iterators. The subtyping of Ω_2 supports fixpoint functions. The subtyping relation is the reflexive, transitive relation generated by these rules and closed under the implication

$$\frac{U_2 < U_1 \quad V_1 < V_2}{U_1 \to V_1 < U_2 \to V_2} \ .$$

Type derivation is given by some axioms for typing **tt** etc plus a rule for typing applications. Here are some special cases of the axioms for any types U, V, W and T:

$$\begin{array}{rcl} S & : & (U \to V \to W) \to (U \to V) \to (U \to W) \\ K & : & U \to V \to U \\ \mathbf{tt} & : & \mathbf{Bool} \\ \mathbf{ff} & : & \mathbf{Bool} \\ \mathbf{zero} & : & \mathbf{Nat} \\ \mathbf{succ} & : & \mathbf{Nat} \to \mathbf{Nat} \\ \omega_2 & : & \Omega_2 \ . \end{array}$$

The general versions of the axioms may replace the type by any supertype of it. For example, the general rule for **tt** is

$$\mathbf{tt} : T \quad \text{if } \mathbf{Bool} < T \ .$$

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

Figure 1: Reduction Rules of Triage Calculus

The rule for typing applications is

$$\frac{M : U \to V \quad N : U}{MN : V} \ .$$

It follows that if $M : U$ is any type derivation and $U < V$ then $M : V$.

Theorem 1 (SK_reduction_preserves_typing) *For all combinations M and N and type T, if $M : T$ and $M \longrightarrow N$ then $N : T$.*

3 Tree Calculus

The *combinations* of a tree calculus are given by

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

The operator $\triangle$ (pronounced "node") is a ternary operator. When unapplied, $\triangle$ is a *leaf* while combinations of the form $\triangle M$ are *stems* and of the form $\triangle MN$ are *forks*. Together these constitute the *factorable forms*. The *programs* are the factorable forms whose sub-expressions (branches) are also programs.

There is some flexibility in the choice of reduction rules. The original tree calculus [19] uses just three rules but they are hard to reason about, or to type. This paper introduces a variant tree calculus called *triage calculus* that better separates concerns by using the five reduction rules in Figure 1.

In shorthand, they become

$$\begin{array}{rcl} \triangle\triangle & = & K \\ \triangle(\triangle\ x) & = & Sx \\ \triangle(\triangle\ w\ x)\ y & = & \mathbf{triage}\{w, x, y\} \end{array}$$

on the understanding that K and S behave as in combinatory logic, while *triage* is the novelty, that performs case analysis on its argument, with cases for leaves, stems and forks. Indeed, these equations can be reversed to define K and triage but some care is required with S (defined below) since Sx merely reduces to $\triangle(\triangle x)$.

Some examples of program constructions are

$$\begin{array}{rcl}
K &=& \triangle\triangle \\
S_1\{p\} &=& \triangle(\triangle p) \\
S &=& S_1\{K\triangle\}\triangle \\
I &=& S_1\{K\}\ K \\
\mathbf{triage}\{w,x,y\} &=& \triangle(\triangle\ w\ x)\ y \\
\mathbf{swap}\{f\} &=& S_1\{K(S_1\{f\})\}K \\
\mathbf{wait}\{M,N\} &=& S_1\{S_1\{KM\}(KN)\}I \\
\mathbf{wait2}\{M,N,P\} &=& S_1\{S_1\{S_1\{KM\}(KN)\}(KP)\}I\ .
\end{array}$$

Note that, unlike $S_1\{p\}$ the combination $S\ p$ is *not* a program since it is the application of fork. Rather $S\ p$ reduces to $S_1\{p\}$.

Since there is only one operator, the programs and computations can be thought of as *natural trees*, i.e. trees without any labels on the nodes. Each program p can be represented as a numeral **num** $\{p\}$ in ternary arithmetic, using Polish notation, where a node is represented by the number of its arguments. For example, we have

$$\begin{array}{rcl}
\mathbf{num}\{\triangle\} &=& 0 \\
\mathbf{num}\{K\} &=& 10 \\
\mathbf{num}\{S_1\{K\}\} &=& 1110 \\
\mathbf{num}\{S\} &=& 212000 \\
\mathbf{num}\{I\} &=& 211010
\end{array}$$

As well as suggesting approaches to implementation based on ternary arithmetic [44], this notation serves as a reminder that all programs, no matter their surface syntax, are binary trees.

This section ends with a couple of conventional theorems. Confluence [40] shows that terms related by reduction have a common reduct; this yields a good equational theory.

Theorem 2 (confluence_tree_calculus) *Reduction of triage calculus is confluent.*

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

Figure 2: Type Derivation for Combinators

Head reduction will be used to show the correctness of the breadth-first evaluator in Section 8. The *head reduction relation* $t \longrightarrow_h u$ is weaker than the general reduction relation. Although it also supports the reduction rules, and is reflexive and transitive, reduction of applications is restricted to the following three possibilities: if $t_1 \longrightarrow_h t_2$ then

$$\begin{array}{rcl}
t_1\ u &\longrightarrow_h& t_2\ u \\
\triangle\ t_1\ u\ v &\longrightarrow_h& \triangle\ t_2\ \ u\ v \\
\triangle\ (\triangle\ w\ u)\ v\ t_1 &\longrightarrow_h& \triangle\ (\triangle\ w\ u)\ v\ t_2\ .
\end{array}$$

The first possibility allows reduction at the head of an application: the other two are required to support the intensional reduction rules that must inspect their arguments.

Theorem 3 (head_reduction_to_factorable_form) *Reduction to factorable form can always begin with head reduction. That is, for all terms t and v, if $t \longrightarrow v$ and v is factorable then there is a term u such that $t \longrightarrow_h u$ and $u \longrightarrow v$.*

Type derivation for the combinations is given by the two rules in Figure 2. These rules are parametrised by the choice of types and their subtyping relation, which determine expressive power. A minimal collection of types for tree calculus is given by

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

Here $\mathbf{L}$ is the type of a leaf and $\mathbf{S}\ U$ is the type of a stem whose branch is of type U and $\mathbf{F}\ U\ V$ is the type of a fork whose branches have types U and V respectively and $U \to V$ is the type of functions from U to V.

A minimal collection of subtyping axioms is given in Figure 3. The subtyping relation is generated from the axioms by closing under the type formation rules

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

Figure 3: Minimal Subtyping Axioms

(function types are contravariant in their arguments) and closing under reflexivity and transitivity.

For example, if $M : U$ and $N : V$ then $\triangle M : \mathbf{S}\ U$ and $\triangle MN : \mathbf{F}\ U\ V$ by applying the first two axioms. Similarly, $K : \mathbf{S}\ \mathbf{L}$ implies $KM : \mathbf{F}\ \mathbf{L}\ U < V \to U$ by the third axiom. Again, I of type $\mathbf{F}\ (\mathbf{S}\ \mathbf{L})\ (\mathbf{S}\ \mathbf{L})$ is also of type $U \to U$ for any type U. Also, for any types U, V and T the combinator S can be shown to have type $(U \to V \to T) \to (U \to V) \to U \to T$.

It may seem strange to limit subtyping to leaf types in Figure 2 but this makes proofs easier without compromising expressivity, since the general subsumption rule

$$\frac{M : U \quad U < V}{M : V}$$

follows by induction on the structure of M, as in Theorem 14 in Section 9.

Theorem 4 (derive_basic) *For every type U, V and T we have*

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

and

$$\frac{f : U \to V \to T}{\mathbf{swap}\{f\} : V \to U \to T}$$

and

$$\frac{M : U \to V \to T \quad N : U}{\mathbf{wait}\{M, N\} : V \to T} .$$

and

$$\frac{M : U \to V \to W \to T \quad N : U \quad P : V}{\mathbf{wait2}\{M, N, P\} : W \to T} .$$

Each program p has a canonical type $\mathbf{progty}(p)$ defined by

$$\begin{array}{rcl}
\mathbf{progty}\{\triangle\} & = & \mathbf{L} \\
\mathbf{progty}\{\triangle q\} & = & \mathbf{S}\ \mathbf{progty}\{q\} \\
\mathbf{progty}\{\triangle qr\} & = & \mathbf{F}\ \mathbf{progty}\{q\}\ \mathbf{progty}\{r\} .
\end{array}$$

Theorem 5 (programs_have_types)
$p : \mathbf{progty}(p)$ for each program p.

4 Lambda-Abstraction

In practice, it is convenient to support lambda abstraction $\lambda x.t$ of a term t by a term variable x. This is achieved by adding term variables to the combinators to get the *terms*

$$t, u ::= x \ \mid\ \triangle \ \mid\ t\ u$$

where x is from a collection of term variables $x, y, z \ldots$ that support decidable equality. Since variables are not bound in the BNF, it is trivial to decide if a variable is free in a term or not, and to define the *substitution* $\{u/x\}t$ of a term u for a variable x in a term t. Then we can *define* lambda-abstraction by

$$\begin{array}{rcl}
\lambda x.x & = & I \\
\lambda x.y & = & K\ y \quad (y \neq x) \\
\lambda x.\triangle & = & K\ \triangle \\
\lambda x.tu & = & K\ (tu) \quad \text{(if } x \text{ is not free in } t\ u) \\
\lambda x.tu & = & \triangle(\triangle(\lambda x.t))(\lambda x.u) \quad \text{(if } x \text{ is free in } tu).
\end{array}$$

Note that the fourth line above introduces some light optimisation to the definition. Note that the right-hand side of the last line is a reduct of $S(\lambda x.t)(\lambda x.u)$.

Theorem 6 (star_beta) *For all terms t and u and term variable x*

$$(\lambda x.t)u \longrightarrow \{u/x\}t .$$

$$\frac{\Gamma(x) < T}{\Gamma \vdash x : T} \qquad \frac{\mathbf{Leaf} < T}{\Gamma \vdash \triangle : T}$$

$$\frac{\Gamma \vdash t : U \to V \quad \Gamma \vdash u : U}{\Gamma \vdash t\ u : V} \ .$$

Figure 4: Type Derivation with Contexts

Now type derivation requires a context Γ that maps term variables to types. Given such a type context and a term variable x and a type U then $\Gamma, x : U$ is the context that maps x to U and maps any other variable y to $\Gamma(y)$. The corresponding type derivation axioms are given in Figure 4. These rules are enough to type abstractions with respect to the minimal set of subtyping rules as follows.

Theorem 7 (derive_star) *For all type contexts* Γ *and term variables* x *and terms* t

$$\frac{\Gamma, x : U \vdash t : T}{\Gamma \vdash \lambda x.t : U \to T} \ .$$

We will assume the presence of term variables and type contexts from now on, even when they are not strictly necessary.

5 Booleans, Natural Numbers and Tagging

Quantified types, of the form $\forall X.T$ where X is a type variable and T is a type, will be used to type some polymorphic examples. Quantification by some sequence of type variables $\vec{X}$ may be written as $\forall \vec{X}.T$. The free type variables of T, and the substitution $\{U/X\}T$ of a type U for X in T is defined in the usual way, to avoid variable capture. The resulting subtyping rules are

$$\begin{array}{rcl} \forall X.T & < & \{U/X\}T \\ T & < & \forall X.T \quad \text{(if } X \text{ is not free in } T\text{).} \end{array}$$

In the **Coq** verification, type variables are represented by *deBruijn indices*. That is, type variables are named by natural numbers, and the quantified type "Quant T" or $\forall T$ binds the variable with index 0. Then the second subtyping rule above becomes $T < \forall(\mathbf{lift}\ 1\ T)$ where **lift** 1 raises all indices by one, to avoid variable capture. However, in this paper we will worked with named bindings, as in $\forall X.T$.

Types for booleans and natural numbers could be introduced as type constants, as in Section 2, but it is more convenient to represent them as quantified function types, as in System F [13], by

$$\begin{array}{rcl} \mathbf{Bool} & = & \forall X.X \to X \to X \\ \mathbf{Nat} & = & \forall X.(X \to X) \to (X \to X) \end{array}$$

so that we can infer

$$\begin{array}{rcl} \mathbf{true} & = & K : \mathbf{Bool} \\ \mathbf{false} & = & KI : \mathbf{Bool} \\ \mathbf{zero} & = & KI : \mathbf{Nat} \\ \mathbf{succ} & = & S_1\{S_1\{K\triangle\}(S_1\{K\triangle\}K)\} : \mathbf{Nat} \to \mathbf{Nat} \end{array}$$

As well as supporting all the earlier theorems of this paper, this is enough to define primitive recursion and all of the other μ-recursive functions [27] except for minimisation, which requires general recursion.

Note that **false** and **zero** are both represented by KI which may lead to confusion. This can be avoided by *tagging* terms with their types, or indeed any other information of interest, without changing the functional behaviour of the term.

Given a term t and another term i which is to be its tag, define

$$\mathbf{tag}\{t, i\} = \mathbf{wait}\{Kt, i\} \ .$$

It has the same functional behaviour as t since

$$\mathbf{wait}\{Kt, i\}x \longrightarrow Ktix \longrightarrow tx$$

and i can be recovered by applying **untag** given by

$$\mathbf{triage}\{\triangle, \triangle, \mathbf{triage}\{\triangle, \mathbf{triage}\{\triangle, \triangle, \mathbf{triage}\{\triangle, KI, \triangle\}\}, \triangle\}\} \ .$$

untag can be thought of as pattern matching on the structure of $\mathbf{tag}\{t, i\} = S_1\{S_1\{K^2t\}(Ki)\}I$ where $K^2t = K(Kt)$. It reduces as follows:

$$\mathbf{untag}\ \mathbf{tag}\{t, i\} = KI\ (K^2t)\ (Ki)\ I = (Ki)\ I = i \ .$$

In this manner, typed tree calculus could be embedded within untyped tree calculus. A thorough development of this approach is beyond the scope of this paper.

6 Recursive Programs

Recursive functions will be of the form $Z\{f\}$ as defined in Section 2.

Theorem 8 (Z_red) *For all terms f and x,*

$$Z\{f\}\ x \longrightarrow f\ Z\{f\}\ x\ .$$

The typing of $Z\{f\}$ proceeds much as before, except that now Ω_2 is defined to be $\mathbf{progty}\{\omega_2\}$ and we will generalise from $U \to V$ to $\forall \vec{X}.U \to V$ to get the subtyping axiom

$$\Omega_2 < \Omega_2 \to \mathbf{Fix}\ (\forall \vec{X}.U \to V)$$

for any sequence $\vec{X}$ of type variables and types U and V. This implies

Theorem 9 (derive_Z) *For all type context Γ and sequence of type variables $\vec{X}$ and types U and V and term f, we can prove*

$$\frac{\Gamma \vdash f : (\forall \vec{X}.U \to V) \to (\forall \vec{X}.U \to V)}{\Gamma \vdash Z\{f\} : \forall \vec{X}.U \to V}\ .$$

It follows that tree calculus, like traditional combinatory logic, supports all of the μ-recursive functions as typed programs, i.e. as typed normal forms.

Note that there are several ways of defining fixpoint functions in combinatory logic, and each choice will lead to different subtyping rules. For example, the implementation of Z could be made smaller by using

$$\mathbf{Z\_alt}\{f\} = \mathbf{wait2}\{\mathbf{self\_apply}, \omega_2, f\}$$

where $\mathbf{self\_apply} = \lambda x.xx = S_1\{I\}I$. Then the relevant subtyping rule would act on the program type of **self_apply** instead of that of ω_2. Similarly, we could begin with $Y_2\{f\}$ that satisfies $Y_2\{f\}\ x \longrightarrow f\ x\ Y_2\{f\}$ but instead we will define it by

$$Y_2\{f\} = Z\{\mathbf{swap}\{f\}\}\ .$$

$$\begin{array}{l}
\mathbf{equal} = \\
\quad Y_2\ (\mathbf{triage}\{ \\
\quad\quad \lambda e.\mathbf{triage}\{\mathbf{true}, K\ \mathbf{false}, K^2\ \mathbf{false}\}, \\
\quad\quad \lambda x.\lambda e.\mathbf{triage}\{\mathbf{false}, e\ x, K^2\mathbf{false}\}, \\
\quad\quad \lambda x_1.\lambda x_2.\lambda e.\mathbf{triage}\{\mathbf{false}, K\ \mathbf{false}, \\
\quad\quad\quad\quad\quad\quad \lambda y_1.\lambda y_2.ex_1y_1(ex_2y_2)\ \mathbf{false}\})
\end{array}$$

Figure 5: Generic Equality

7 Generic Queries

Generic queries have types such as $\forall X.X \to \mathbf{Bool}$ which produces a boolean for any argument. Typically, they are given by triage. For example, the tests for being a leaf, stem or fork are given by

$$\begin{array}{rcl}
\mathbf{isLeaf} & = & \mathbf{triage}\{\mathbf{true}, K\ \mathbf{false}, K^2\ \mathbf{false}\} \\
\mathbf{isStem} & = & \mathbf{triage}\{\mathbf{false}, K\ \mathbf{true}, K^2\mathbf{false}\} \\
\mathbf{isFork} & = & \mathbf{triage}\{\mathbf{false}, K\ \mathbf{false}, K^2\ \mathbf{true}\}
\end{array}$$

These fundamental queries will all have type $\forall X.X \to \mathsf{Bool}$. In each case, the natural type of the triage is

$$\begin{array}{c}
\mathbf{F}\ (\mathbf{F}\ \mathbf{Bool}\ (\forall X.X \to \mathbf{Bool})) \\
(\forall X.\forall Y.X \to Y \to \mathbf{Bool})
\end{array}$$

so for these examples it would be enough to make this type a subtype of $\forall Z.Z \to \mathbf{Bool}$.

Similarly, we can define a program **size** that computes the size of a program. The required subtyping replaces **Bool** above with **Nat** to prove:

Theorem 10 (derive_size) $\mathbf{size} : \forall X.X \to \mathbf{Nat}$.

However, some queries take more than one argument. For example, consider the generic equality in Figure 5. It is a recursive function that performs triage on each of its arguments in turn, for a total of nine cases. The program **equal** has 780 nodes: its ternary representation is given in Figure 6.

Theorem 11 (equality_of_programs) *For all programs M and N, if $M = N$ then* **equal** *M N reduces to* **tt** *else it reduces to* **ff**.

```
212121202120112110102121200212002120112002120
```

Figure 6: The Ternary Representation of **equal**

Theorem 12 (derive_equal) **equal** : $\forall X.\forall Y.X \to Y \to \mathbf{Bool}$.

The usual type $\forall X.X \to X \to \mathbf{Bool}$ for equal can be generalised to $\forall X.\forall Y.X \to Y \to \mathbf{Bool}$. Indeed, this generality is necessary for the type derivation. To see this, consider the equality of two forks $\triangle u_1 u_2$ and $\triangle v_1 v_2$ whose common type $X \to Z$ is given by the subtypings:

$$\begin{array}{rcl} \mathbf{F}\ \mathbf{L}\ Z & < & X \to Z \\ \mathbf{F}\ (\mathbf{S}\ (X \to Y \to Z))\ (X \to Y) & < & X \to Z\ . \end{array}$$

The reduction of **equal** $(\triangle u_1 u_2)$ $(\triangle v_1 v_2)$ to **equal** u_1 v_1 (**equal** u_2 v_2) **false** invokes **equal** u_1 v_1 where $u_1 : \mathbf{L}$ and $v_1 : \mathbf{S}\ (X \to Y \to Z)$ have incompatible types. A side-effect of this generous typing is that arguments may have different types and yet be equal. For example, **false** : **Bool** and **zero** : **Nat** are both equal to K. Hence, the type of the result of applying **equal** to an argument is not **Bool** but $Y \to \mathbf{Bool}$.

Typing of such examples can be handled by generalising from **Bool** to any type T in which X is not free, to get the subtyping rule

$$\begin{array}{l} \mathbf{F}\ (\mathbf{F}\ T\ (\forall X.X \to T))\ (\forall X.\forall Y.X \to Y \to T) \\ \quad < \forall Z.Z \to T \end{array}$$

if X, Y and Z are not free in T.

Updates, however, require even more generality. For example, a generic database update should have type $\forall X.X \to X$ where the argument type X appears positively in the result type. Further, a generic update of programs may have type $\forall X, \forall Y.(X \to Y) \to (X \to Y)$ in which triage is applied to the second argument, of type X. To exploit this, swap the arguments, to get the type

$$\forall X, \forall Y.X \to (X \to Y)\ \to Y)$$

where the argument type X is still in a positive position in $(X \to Y) \to Y$ because it has been "doubly-negated." Thus, the general form of the subtyping rule for triage yields

$$\begin{array}{l} \mathbf{F}\ (\mathbf{F}\ (\{\mathbf{L}/Z\}T)\ (\forall X.X \to \{\mathbf{S}\ X/Z\})) \\ (\forall X, Y.X \to Y \to \{\mathbf{F}\ X\ Y/Z\}T) < \forall Z.Z \to T \end{array}$$

where the type variable Z appears *covariantly* in the type T. Covariance is defined in the traditional manner; details are in the **Coq** implementation. Since the notation in this rule is quite heavy, let us compress it by defining

$$\begin{array}{rcl} \mathbf{AsLf}\{Z,T\} & = & \{\mathbf{L}/Z\}T \\ \mathbf{AsSm}\{Z,T\} & = & \forall X.X \to \{\mathbf{S}\ X/Z\}T \\ \mathbf{AsFk}\{Z,T\} & = & \forall X.\forall Y.X \to Y \to \{\mathbf{F}\ X\ Y/Z\}T \\ & & \text{(if } X, Y \text{ not free in } T). \end{array}$$

Then the subtyping rule becomes

$$\begin{array}{l} \mathbf{F}\ (\mathbf{F}\ \mathbf{AsLf}\ \{\mathbf{Z},\mathbf{T}\}\mathbf{AsSm}\{\mathbf{Z},\mathbf{T}\})\ \mathbf{AsFk}\{\mathbf{Z},\mathbf{T}\} \\ \quad < \forall Z.Z \to T \end{array}$$

if Z is covariant in T.

For example, the breadth-first evaluator will use a program **eager** : $(X \to Y) \to (X \to Y)$ to enforce eager evaluation of the second argument, defined by **swap**{**eager_s**} where **eager_s** is

$$\mathbf{triage}\{\lambda f.f\triangle, \lambda x.\lambda f.f(\triangle x), \lambda w.\lambda x.\lambda f.f(\triangle wx)\}.$$

That is, **eager_s** $x\ f$ reduces to $f\ v$ if x reduces to some v which is a leaf, stem or fork. If x does not have such a value then f is never applied.

These examples implement algorithms that do not try to exploit type information but further work will be required to support generic functions such as **map** and **fold** that may act over lists or tuples, etc. Current practice may use encodings [28] or staging [43] to specialise the algorithm to each type. In a tree calculus, however, data structures can be represented as tuples that have been tagged by information about the constructor and its arguments, so that specialisation is not required.

$$\frac{}{\triangle, y \Downarrow \triangle y} \qquad \frac{}{\triangle y, z \Downarrow \triangle y z}$$

$$\frac{}{\triangle\triangle y, z \Downarrow y} \qquad \frac{x, z \Downarrow u \quad y, z \Downarrow v \quad u, v \Downarrow w}{\triangle(\triangle x)y, z \Downarrow w}$$

$$\frac{}{\triangle(\triangle wx)y, \triangle \Downarrow w} \qquad \frac{x, z \Downarrow v}{\triangle(\triangle wx)y, \triangle z \Downarrow v}$$

$$\frac{y, z_1 \Downarrow v_1 \quad v_1, z_2 \Downarrow v}{\triangle(\triangle wx)y, \triangle z_1 z_2 \Downarrow v}$$

Figure 7: Breadth-First Evaluation

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

Figure 8: The Breadth-First Evaluator

8 Self-Interpretation

Traditionally, there have been two sorts of self-interpreters that act on encodings [22]. A *self-recogniser* of type

$$\mathbf{Exp}\{U \to V\} \to U \to V$$

performs decoding, while a *self-enactor* of type

$$\mathbf{Exp}\{U \to V\} \to \mathbf{Exp}\{U\} \to \mathbf{Exp}\{V\}$$

must actually describe how evaluation is to act on expression, by applying a strategy. Since we are not using encodings at all, self-recognisers are redundant, so the focus turns to the enactors.

The strategy we will implement is breadth-first: when evaluating a fork, both branches will be evaluated before applying any redex, much as in an eager evaluation strategy. The evaluation rules for this binary relation $\Downarrow$ on programs given in Figure 7. By contrast, a depth-first strategy would always reduce the head if possible, only evaluating branches when forced to do so.

The breadth-first evaluation strategy is internalised by the program **bf** defined in Figure 8. The fixpoint operator Z causes it to traverse its first argument. If it is a leaf or stem then the program halts. If it is a fork $\triangle f y$ then **bff** performs triage on f. If f is a leaf then halt else if it is a stem then apply **bffs** else apply **bfff**. In turn, **bffs** $x\ y\ z$ reduces to **eager** (**bf** (**bf** $x\ z$)) (**bf** $y\ z$) which forces evaluation of **bf** $y\ z$ as well as **bf** $x\ z$. Finally, **bfff** $w\ x\ y$ reduces to **bf** (**bf** $y\ w$) x. The program **bf** has 877 nodes; its ternary representation is given in Figure 9. This concrete representation reminds us that despite the use of high-level constructions such as fixpoints and triage, all programs are actually natural binary trees. Further, it is remarkable that self-interpretation requires so little; that an evaluation strategy can be expressed by a tree with less than one thousand nodes.

Theorem 13 (branch_first_eval_iff_bf) *For all programs t, u and p we have $t, u \Downarrow p$ if and only if* **bf** $t\ u \longrightarrow p$.

Typing of **bf** introduces significant challenges. First, consider the reduction

$$\mathbf{bf}\ (\triangle uv)\ z \longrightarrow \mathbf{bff}\ u\ v\ z\ .$$

If $u : U$ and $v : V$ and $z : Z$ then the typing of the left-hand side by some type T requires that $\mathbf{F}\ U\ V < Z \to$

```
21212120212011211101021212002120021201120021 20
11200212120021200212010212002120021212002120 0
```

212121202120112111010212120021200212011200212 0

Figure 9: The Ternary Representation of **bf**

T. So we must show that **bff** $: U \to V \to Z \to T$ whenever $\mathbf{F}\ U\ V < Z \to T$. However, conditional typing is awkward at best.

Second, consider the reduction

$$\begin{array}{ll} \mathbf{bf}\ (\triangle(\triangle u)\ v)\ z & \\ \quad\longrightarrow & \mathbf{bffs}\ u\ v\ z \\ \quad\longrightarrow & \mathbf{eager}\ (\mathbf{bf}\ (\mathbf{bf}\ u\ z))\ (\mathbf{bf}\ v\ z)\ . \end{array}$$

The left-hand side has type T if $u : Z \to Y \to T$ and $v : Z \to Y$ and $z : Z$. In this case, the right-hand side will also have type T. However, this information is *not* available at the time that **bffs** is being typed. Similar remarks apply to the typing of **bfff**.

The solution is to introduce a new type form $\mathbf{A}\ T$ (pronounced "as fun T") which records the obligation for T to become a function type. The corresponding subtyping rules are

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

Now our goal is to prove **bf** $: \forall Z.Z \to \mathbf{A}\ Z$. Once that is done then instantiating Z to a function type allows the tagging to be removed, so that we have

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

as desired.

Returning to our challenges, there remains a gap in the typing of **bfff** and **bffs**. The intended result type for **bfff** is of the form $\mathbf{A}\ (\mathbf{F}\ (\mathbf{F}\ U\ V)\ W)$ but the derived type is

$$\mathbf{F}\ (\mathbf{F}\ U\ (\mathbf{A}\ V))\ (\mathbf{bfffa}\ W)$$

where

$$\mathbf{bfffa}\ U = \mathbf{F}\ (\mathbf{S}\ (\mathbf{F}\ \mathbf{L}\ (\forall X.X \to \mathbf{A}\ X)))\ (\mathbf{A}\ U)$$

so add a subtyping rule

$$\mathbf{F}\ (\mathbf{F}\ U\ (\mathbf{A}\ V))\ (\mathbf{bfffa}\ W) < \mathbf{A}\ (\mathbf{F}\ (\mathbf{F}\ U\ V)\ W)\ .$$

Similarly, the intended result type for **bffs** is of the form $\mathbf{A}\ (\mathbf{F}\ (\mathbf{S}\ U)\ V)$ but the derived type is $\mathbf{F}\ (\mathbf{S}\ (\mathbf{bffsa}\ U))\ (\mathbf{A}\ V)$ where $\mathbf{bffsa}\ U$ is

$$\mathbf{F}\,(\mathbf{S}\,(\mathbf{F}\,\mathbf{L}\,(\forall X.\forall Y.(X \to Y) \to (X \to Y))))\,(\mathbf{bfffa}\,U)$$

so add a subtyping rule

$$\mathbf{F}\ (\mathbf{S}\ (\mathbf{bffsa}\ U))\ (\mathbf{A}\ V) < \mathbf{A}\ (\mathbf{F}\ (\mathbf{S}\ U)\ V)\ .$$

With these additions, **bf** can be shown to have the desired type.

9 The Richest Type System

This section will derive types for all of the examples in the paper while using all of the subtyping rules that have been considered.

The *terms* and *types* are given by

$$\begin{array}{rcl} t, u, v & ::= & x \ \mid\ \triangle \ \mid\ t\ u \\ T, U, V & ::= & \mathbf{L} \ \mid\ \mathbf{S}\, U \ \mid\ \mathbf{F}\, U\, V \ \mid\ U \to V \ \mid \\ & & X \ \mid\ \forall X.T \ \mid\ \mathbf{A}\, T\ . \end{array}$$

The term reduction rules are given in Figure 1.

Define

$$\begin{array}{rcl} \Omega_2 & = & \mathbf{progty}(\omega_2) \\ \mathbf{Fix}\, T & = & (T \to T) \to T \\ \mathbf{bfffa}\, U & = & \mathbf{F}\ (\mathbf{S}\ (\mathbf{F}\ \mathbf{L}\ (\forall X. X \to \mathbf{A}\ X)))\ (\mathbf{A}\ U) \\ \mathbf{bffsa}\, U & = & \mathbf{F}\ (\mathbf{S}\ (\mathbf{F}\ \mathbf{L} \\ & & \quad (\forall X.\forall Y.(X \to Y) \to (X \to Y)))) \\ & & \quad (\mathbf{bfffa}\, U)\ . \end{array}$$

The subtyping rules are generated from the subtyping axioms in Figure 10 and closed under the type formation operations (argument types are contravariant in function types) and used to generate a reflexive, transitive order. Most of the axioms have been discussed earlier. The remaining four allow quantifiers to distribute over stems, forks, function types and **A** types.

The type derivation rules are given in Figure 4. When the context Γ is empty, we may write $t : T$ for $\Gamma \vdash t : T$. Although subtyping appears in the derivations for term variables and nodes only, any type derivation can be subtyped, as follows.

Theorem 14 (derive_subtype) *For each type context Γ and term t and types T_1 and T_2, if $\Gamma \vdash t : T_1$ and $T_1 < T_2$ then $\Gamma \vdash t : T_2$.*

All of the theorems about tree calculus in earlier sections can be proved in this system, as shown in the attached **Coq** proofs.

10 Reduction Preserves Typing

It remains to prove that reduction preserves typing by examining the reduction rules. For example, consider the reduction $Kuv \longrightarrow u$. where $u : U$ and $v : V$. If $\mathbf{F}\ \mathbf{L}\ U < V \to T$ then $Kuv : T$ so we must show that $u : T$ i.e. that $U < T$. The next five theorems handle all of the different cases. Then the final theorem shows that reduction preserves typing.

Theorem 15 (subtype_from_fork_of_leaf_to_fun)

$\mathbf{F}\ \mathbf{L}\ V < Z \to T$ implies $V < T$ for all types V, Z and T.

Theorem 16 (subtype_from_fork_of_stem_to_funty) *For all types U, V, Z and T if $\mathbf{F}\ (\mathbf{S}\ U)\ V < Z \to T$ then there is a type Y such that $U < Z \to Y \to T$ and $V < Z \to Y$.*

Theorem 17 (subtype_from_fork_of_fork_of_leaf)

For all types W, U, V and T, if $\mathbf{F}\ (\mathbf{F}\ W\ U)\ V < \mathbf{L} \to T$ then $W < T$.

Theorem 18 (subtype_from_fork_of_fork_of_stem)

For all types W, U, V, Z and T, if $\mathbf{F}\ (\mathbf{F}\ W\ U)\ V < \mathbf{S}\ Z \to T$ then $U < Z \to T$.

Theorem 19 (subtype_from_fork_of_fork_of_fork)

For all types $W, U, V, Z_1 Z_2$ and T, if $\mathbf{F}\ (\mathbf{F}\ W\ U)\ V < \mathbf{F}\ Z\ {}_1 Z_2 \to T$ then $V < Z_1 \to Z_2 \to T$.

Theorem 20 (reduction_preserves_typing) *For all type context Γ and terms t_1 and t_2 and type T, if $\Gamma \vdash t_1 : T$ and $t_1 \longrightarrow t_2$ then $\Gamma \vdash t_2 : T$.*

It is important to note that none of the proofs of the theorems above is straightforward. Rather, they follow from a complete characterisation of the subtyping relation which must address several sources of complexity. First, quantifiers can be introduced and eliminated at any point: these can be combined into type substitutions which are passed as parameters. Second, **A** can be introduced and eliminated at any point. Third, the special subtyping rules for recursion, generic queries and **bf** must be shown to respect typing. For example, if the given subtyping in Theorem 16 is $\Omega_2 < \Omega_2 \to \mathbf{Fix}\, T$ then the proof of the theorem mirrors the proof that $Z\{-\}$ matches its specification. Finally, the main theorem relies on a relatively routine characterisation of type derivation. The details of the development are too verbose for inclusion in the main body of the paper but can be found in the related **Coq** code.

$$
\begin{array}{rcl}
\mathbf{L} & < & U \to \mathbf{S}\; U \\
\mathbf{S}\; U & < & V \to \mathbf{F}\; U\; V \\
\mathbf{F}\; \mathbf{L}\; U & < & V \to U \\
\mathbf{F}\; (\mathbf{S}\; (U \to V \to T))\; (U \to V) & < & U \to T \\
\mathbf{F}\; (\mathbf{F}\; U\; V)\; W & < & \mathbf{L} \to U \\
\mathbf{F}\; (\mathbf{F}\; U\; (V_1 \to V_2))\; W & < & \mathbf{S}\; V_1 \to V_2 \\
\mathbf{F}\; (\mathbf{F}\; U\; V)\; (W_1 \to W_2 \to W_3) & < & \mathbf{F}\; W_1\; W_2 \to W_3 \\
\forall X.\mathbf{S}\; U & < & \mathbf{S}\; (\forall X.U) \\
\forall X.\mathbf{F}\; U\; V & < & \mathbf{F}\; (\forall X.U)\; (\forall X.V) \\
\forall X.\mathbf{A}\; U & < & \mathbf{A}\; (\forall X.U) \\
\forall X.U \to V & < & (\forall X.U) \to (\forall X.V) \\
\forall X.T & < & \{U/X\}T \\
T & < & \forall X.T \quad (X \notin T) \\
\Omega_2 & < & \Omega_2 \to \mathbf{Fix}\; (\forall \vec{X}.U \to V) \\
\mathbf{F} \quad (\mathbf{F} \quad \mathbf{AsLf}\{\mathbf{Z},\mathbf{T}\}\; \mathbf{AsSm}\{\mathbf{Z},\mathbf{T}\}) & & \\
\mathbf{AsFk}\{Z,T\} & < & U \to \{U/Z\}T \\
 & & \text{(if } Z \text{ is covariant in } T\text{)} \\
T & < & \mathbf{A}\; T \\
\mathbf{A}\; (U \to V) & < & U \to V \\
\mathbf{F}\; (\mathbf{S}\; (\mathbf{bffsa}\; U))\; (\mathbf{A}\; V) & < & \mathbf{A}\; (\mathbf{F}\; (\mathbf{S}\; U)\; V) \\
\mathbf{F}\; (\mathbf{F}\; U\; (\mathbf{A}\; V))\; (\mathbf{bfffa}\; (\mathbf{A}\; W)) & < & \mathbf{A}\; (\mathbf{F}\; (\mathbf{F}\; U\; V)\; W)\; .
\end{array}
$$

Figure 10: Subtyping Axioms

11 Conclusions and Future Work

Unlike traditional combinatory logic, tree calculus supports program analysis as well as numerical analysis, without invoking any of the encodings or meta-theory that is usually required. This is because the programs are exactly the (closed) normal forms, given as natural binary trees, whose internal structure can be explored using generic queries, such as equality, or manipulated by, say, a self-interpreter. Rather, the inherent tension between the views of programs as data structures or as functions re-emerges in the type theory. Although two derivation axioms are enough to type all programs, they are parametrised by a subtyping relation that is used to convert tree types to functions types. More precisely, any cleverness that is used to show correctness of a program must be captured by the subtyping rules, as was demonstrated for recursive functions and self-interpreters.

Much remains to be done. It will be interesting to see how systematically partial evaluation [21] and other program analyses, e.g. [11, 42] can be developed and typed without encodings. Theoretically, this account of typed tree calculus is still using meta-theory to represent the types and the type-level computations. However, it should be possible to represent types, typed terms and type-level computations within tree calculus by using tags. Among other things, they can be used to demarcate the boundaries between shapes and the data they hold [23, 31] to support generic mapping and folding, etc. *Dependent types* and *dynamic types* [14] should be equally supported in this setting.

Indeed, perhaps we can even support self-inference, as given by self-application of a program **infer** such that

$$\textbf{infer}\ p\ U = \textbf{Some}\ V$$

implies $p : U \to V$.

In another direction, program analysis becomes complexity theory, which estimates the execution time of programs, relative to the size of their inputs. These sizes are easily computed in the Turing model, but are unstable in lambda calculus, since the input to a higher-order function may be a program that does not have a normal form. In tree calculus, however, we have defined the program **size** and so could develop programs to compute complexity of higher-order programs.

12 Data-Availability Statement

The proofs of all theorems have been verified in Coq [20].

References

[1] Henk Barendregt. Self-interpretations in lambda calculus. *J. Functional. Programming*, 1(2):229–233, 1991.

[2] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North Holland, 1981.

[3] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant Reference Manual : Version 6.1. Research Report RT-0203, INRIA, May 1997. Projet COQ.

[4] Alessandro Berarducci and Corrado Böhm. *A self-interpreter of lambda calculus having a normal form*, pages 85–99. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[5] Matt Brown and Jens Palsberg. Self-representation in girard's system u. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 471–484, New York, NY, USA, 2015. ACM.

[6] Matt Brown and Jens Palsberg. Breaking through the normalization barrier: A self-interpreter for f-omega. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*,

POPL '16, pages 5–17, New York, NY, USA, 2016. ACM.

[7] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. 17(4):471–523, December 1985.

[8] Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19:509–543, 9 2009.

[9] A. Church. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, New Jersey, 1941.

[10] H.B. Curry and R. Feys. *Combinatory Logic*, volume 1 of *Combinatory Logic.* North-Holland Publishing Company, 1958.

[11] Paul Downen, Philip Johnson-Freyd, and Zena M Ariola. Abstracting models of strong normalization for classical calculi. *Journal of Logical and Algebraic Methods in Programming*, 111:100512, 2020.

[12] Solomon Feferman. A language and axioms for explicit mathematics. In John Newsome Crossley, editor, *Algebra and Logic*, pages 87–139, Berlin, Heidelberg, 1975. Springer Berlin Heidelberg.

[13] J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[14] Fritz Henglein. Dynamic typing. In Bernd Krieg-Brückner, editor, *ESOP '92*, pages 233–253, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[15] R. Hindley and J.P. Seldin. *Introduction to Combinators and Lambda-calculus.* Cambridge University Press, 1986.

[16] Barry Jay. *Pattern Calculus: Computing with Functions and Structures.* Springer, 2009.

[17] Barry Jay. Recursive programs in normal form (short paper). In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '18, pages 67–73. ACM, 2018.

[18] Barry Jay. Intensional computation with higher-order functions. *Theoretical Computer Science*, 768:76–90, 2019.

[19] Barry Jay. *Reflective Programs in Tree Calculus.* 2021. including an appendix with Jose Vergara.

[20] Barry Jay. Typedprogramanalysis: repository of proofs in Coq, December 2024.

[21] Barry Jay and Thomas Given-Wilson. A combinatory account of internal structure. *Journal of Symbolic Logic*, 76(3):807–826, 2011.

[22] Barry Jay and Jens Palsberg. Typed self-interpretation by pattern matching. In *Proceedings of the 2011 ACM Sigplan International Conference on Functional Programming*, pages 247–58, 2011.

[23] C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[24] Ende Jin, Nada Amin, and Yizhou Zhang. Extensible metatheory mechanization via family polymorphism. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1608–1632, 2023.

[25] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice Hall International, 1993.

[26] S. C. Kleene. λ -definability and recursiveness. *Duke Math. J.*, 2(2):340–353, 06 1936.

[27] Stephen C. Kleene. λ-definability and recursiveness. *Duke Mathematical Journal*, 2:340–353, 1936.

[28] Hsiang-Shang Ko, Liang-Ting Chen, and Tzu-Chi Lin. Datatype-generic programming meets elaborator reflection. *Proceedings of the ACM on Programming Languages*, 6(ICFP):225–253, 2022.

[29] András Kovács. Staged compilation with two-level type theory. *Proc. ACM Program. Lang.*, 6(ICFP), August 2022.

[30] Konstantin Läufer and Martin Odersky. Self-interpretation and reflection in a statically typed language. In *OOPSLA/ECOOP workshop on object-oriented reflection and metalevel architectures*. Citeseer, 1993.

[31] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Parallel skeletons for manipulating general trees. *Parallel Computing*, 32(7-8):590–603, 2006.

[32] Torben Æ. Mogensen. Efficient self-interpretations in lambda calculus. *Journal of Functional Programming*, 2(3):345–363, 1992. See also DIKU Report D–128, Sep 2, 1994.

[33] Torben Æ. Mogensen. Linear-time self-interpretation of the pure lambda calculus. *Higher-Order and Symbolic Computation*, 13(3):217–237, 2000.

[34] Flemming Nielson and Hanne Riis Nielson. *Two-level functional languages*. Cambridge University Press, USA, 1992.

[35] Tillmann Rendel, Klaus Ostermann, and Christian Hofer. Typed self-representation. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 293–303, New York, NY, USA, 2009. Association for Computing Machinery.

[36] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740. ACM Press, 1972. The paper later appeared in Higher-Order and Symbolic Computation.

[37] Moses Schönfinkel. Über die bausteine der mathematischen logik. *Mathematische Annalen*, 92:305–316, 1924.

[38] Fangmin Song, Yongsen Xu, and Yuechen Qian. The self-reduction in lambda calculus. *Theoretical Computer Science*, 235(1):171 – 181, 2000.

[39] Wenhao Tang, Daniel Hillerström, James McKinna, Michel Steuwer, Ornela Dardha, Rongxiao Fu, and Sam Lindley. Structural subtyping as parametric polymorphism. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):1093–1121, 2023.

[40] Terese. *Term Rewriting Systems*, volume 53 of *Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[41] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2 42:230–65, 1936.

[42] Guannan Wei, Oliver Bračevac, Shangyin Tan, and Tiark Rompf. Compiling symbolic execution with staging and algebraic effects. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–33, 2020.

[43] Jeremy Yallop. Staged generic programming. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–29, 2017.

[44] Furqan Zahoor, Ramzi A. Jaber, Usman Bature Isyaku, Trapti Sharma, Faisal Bashir, Haider Abbas, Ali S. Alzahrani, Shagun Gupta, and Mehwish Hanif. Design implementations of ternary logic systems: A critical review. *Results in Engineering*, 23:102761, 2024.