Computing with Trees not Numbers
DRAFT

Barry Jay

Centre for Artificial Intelligence
University of Technology Sydney
Barry.Jay@uts.edu.au

Abstract. Numbers provide a representation of both programs and data that is universal. However, the representation is unnatural, as numbers cannot build representations of compound structures from those of their components. A better representation uses unlabeled binary trees. This is not only universal, but also modular, since programs and data all have natural representations as trees. The inputs to a program are represented by additional branches, so that computation must reduce a finitely-branching tree to a binary tree. This approach leads to wave calculus, whose terms are built from a single, ternary operator that satisfies three rules, according to whether the argument is a leaf, stem or fork. As well as being Turing complete, wave calculus is also intensionally complete, in that the internal structure of programs is as easily accessed as that of data. For example, equality of programs, their size, or Goedel number are all definable within the calculus.

Keywords: foundations of computing, intensional computation, wave calculus

1 Introduction

Traditional accounts of computation are founded on arithmetic. While this provides a uniform basis for comparison, it is actually quite awkward to work with for three reasons. First, numerical representations hide the internal structure of data. When computing with, say, lists of numbers or trees of words, i.e. texts, then the structure of the list or tree is encoded in an opaque manner that is expensive to decode. Second, these representations hide the internal structure of programs. When a program represented by the number m

$$m = \circ - \circ - \ldots - \circ$$

is applied to the number n

$$n = \circ - \ldots - \circ$$

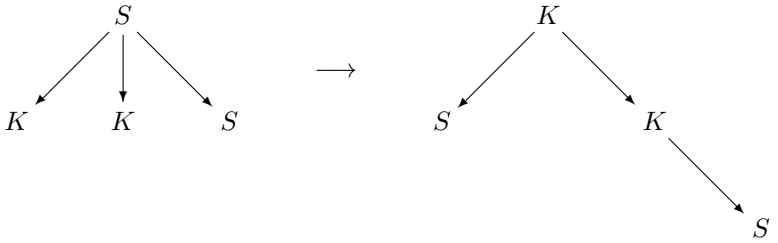
then m must be decoded to some algorithm before it can be applied to n. Third, the algorithms themselves have a complex internal structure, described by, say

Peano arithmetic or μ-recursive algorithms, that in turn require quantifiers, e.g. to nominate the least number.

The core problem is that the linear structure of numbers is too thin. Instead, let us compute with binary trees. Indeed, this approach is implicit in combinatory logic. Its terms are built from the operators S and K by application. Computation is given by the rewriting rules

$$SMNP \longrightarrow MP(NP)$$
$$Ktu \longrightarrow t\ .$$

These terms form finitely-branching trees labeled by S and K. Since S is ternary and K is binary, all normal forms are given by binary trees, and computation reduces finitely-branching trees to binary trees. For example, the reduction $SKKS \longrightarrow KS(KS)$ is given by the tree reduction.



This account of computation avoids the three problems above but introduces two new difficulties. First, although SK-calculus is Turing-complete, in that all Turing-computable numerical functions are definable, it is not *intensionally complete* in that there are effectively calculable functions of its binary trees that are not definable. In particular, neither their equality nor their Gödel numbering is definable. This difficulty has been overcome by adopting a more expressive calculus, SF-calculus, where S is as above and F is given by a pair of rules

$$FOMN \longrightarrow M \qquad (O \text{ is } S \text{ or } F)$$
$$F(PQ)MN \longrightarrow NPQ \quad (PQ \text{ is a compound})$$

where the compounds are of the form SM, SMN, FM or FMN. That is the compounds are the partial applications of S and F. The side-conditions above can be eliminated by expanding out the cases, to get a total of seven rules. As well as being Turing-complete, SF-calculus is also *intensionally complete* in that Gödel numbering is definable, so that program analysis can be reduced to arithmetic.

This leaves one problem, which is that the labels S, K and F appear to be arbitrary when compared to the geometic simpicity of the natural numbers. The solution is replace the labels with small, unlabeled binary trees. Indeed, this trick works with any texts or programs. The corresponding calculus will have a single operator that plays the role of leaf and node. Since terms will consist of nothing but this operator and brackets (and) let us use the horizontal *wave*

symbol $\sim$ to denote the operator. Since binary tree are values, the wave should be a ternary operator, with three reduction rules, according to whether its first argument is a *leaf* $\sim$ or a *stem* $\sim M$ or a *fork* $\sim MN$. Inspired by the calculi above, we propose the following reduction rules for *wave calculus*:

$$\begin{aligned} \sim\sim MN &\longrightarrow M && (K) \\ \sim(\sim N)MP &\longrightarrow MP(NP) && (S) \\ \sim(\sim PQ)MN &\longrightarrow NPQ && (F)\ . \end{aligned}$$

Like SF-calculus, wave calculus is also Turing-complete and intensionally complete, but it is simpler, having only one operator and three unconditional reduction rules, instead of two operators and seven rules. Also, it provides a direct account of computation on binary trees, with the higher-order, modular, quantifier-free account of programs and data that is so lacking in numerical models of computation.

Further, wave calculus can be typed, by introducing type forms for the leaves, stems and forks, and using subtyping to coerce fork types to funciton types.

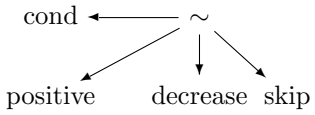
The structure of the paper is as follows. Section 1 is the introduction. Section 2 reduces computation with texts to the study of unlabeled trees. Sections 3 considers various representations of texts by numbers, Turing machines, combinatory logic, λ-calculus, and SF-calculus. Section 4 introduces wave calculus. Section 5 shows how to represent λ-abstraction and the combinators in wave calculus. Section 6 shows that wave calculus is intensionally complete. Section 7 shows how to type wave calculus. Section 8 draws conclusions.

All theorems have been verified in Coq.

2 Trees

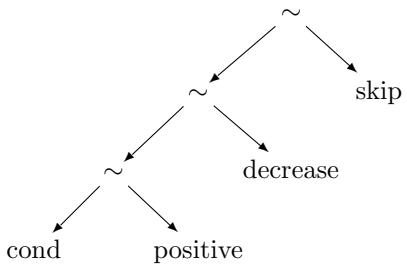
This section shows, by example, how a labeled rose tree can be represented by a binary tree (without labels), and then how a function of such binary trees that is equipped with its arguments can be represented by a rose tree (without labels).

The first step is to push all labels into leaves, so that the example from the introduction becomes



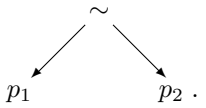
where $\sim$ denotes an unlabeled node.

Further, we can add some "dummy" nodes to ensure that all nodes have at most two branches, as in

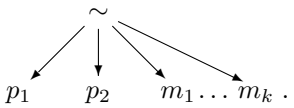


Finally, since the collection of labels is finite, we can represent each one by a designated binary tree, just as one can represent the alphabet of a Turing machine by numbers. Thus all syntax trees can be represented by binary trees.

Now consider how to evaluate a program p applied to some arguments $m_1, \ldots m_k$. Let us suppose that p is represented by a binary tree with two sub-trees p_1 and p_2 so that we may write p as



Then the application of p to its arguments may be represented by the rose tree



with $2 + k$ branches. Similar remarks apply if p has zero or one branches. Hence any model of computation on texts can be represented by a partial function from rose trees to binary trees.

Further, the rose trees can be described by the Backus-Naur Form (BNF)

$$M, N ::= \sim \ | \ (MN)$$

in which every term M is given by a combination of *waves* $\sim$ built by *application*. For example, p above may be written as $((\sim\ p_1)p_2)$. By convention throughout the paper, application is left-associative, and superfluous brackets can be dropped, so that the example above may be written $\sim p_1 p_2$. Let us consider how well the various models of computation can represent the binary trees.

3 Existing Models of Computation

This section considers how to compute with trees using existing models of computation while trying to avoid encodings.

3.1 Natural Numbers

Like any text, binary trees can be encoded as natural numbers. The most direct encoding might be to represent the symbols "(" and "∼" and ")" by the digits $0, 1$ and 2 in base three arithmetic so that, for example, $\sim(\sim\sim)$ becomes 10112 or 357 in decimal notation. However, this bakes in the choice of base, which is quite artificial. A more stable representation uses Gödel numbers, so that $\sim(\sim\sim)$ becomes $2^1 3^0 5^1 7^1 11^2 = 25,910$. However, the cost of encoding and decoding is now comparable to that of encryption and decryption. Since the natural numbers correspond to unary trees, another way to think of the challenge is to represent binary trees as unary trees, which makes it seem unlikely that there is a simple representation.

3.2 Turing Machines

If the alphabet of a Turing machine is binary then the challenge of encoding trees is as before. If the alphabet is "(" and "∼" and ")" there is a natural translation of trees onto the tape. However, encoding of Turing machines onto the tape is not so easy, as the state-transition table must be encoded as a tree. So Turing machines require codings to support higher-order computation.

3.3 Combinatory Logic

Combinatory logic [2] is a higher-order calculus in which every combinator can be either an argument or a function. Its *combinators* are given by the BNF

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

Its *reduction rules* are

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

The *reduction relation*, also written $\longrightarrow$, is the reflexive transitive closure of the *one-step reduction relation*. The latter is the congruence generated by the reduction rules. Often, the combinator I (for identity) is included as a primitive, but here it is defined by SKK since

$$SKKM \longrightarrow KM(KM) \longrightarrow M \ .$$

A combinator is in *normal form* if there are no one-step reductions from it. SK-calculus is *combinatorially complete* [1] because it can represent λ-abstraction, in the sense that will be illustrated in Section 4.

Now let us consider how binary trees might be represented as normal forms of SK-calculus. Assuming that the representation of waves is to preserve application, then the only possible representation of $\sim$ is by S. However, the system

built from S alone is unable to define (the function represented by) K so this interpretation is manifestly inadequate.

The only other approach that suggests itself is to generalise the collection of values from the binary trees to the normal forms of SK-calculus, which form a subset of the binary trees with leaves labelled by S or K. However, there are computable functions of the normal forms of SK-calculus that are not definable as combinators. For example, the equality of normal forms is not decidable, as no combinator can distinguish the two identity functions SKK and SKS [4].

Hence, there is no representation of texts within SK-calculus that can represent the application of functions by the application of combinators.

3.4 λ-Calculus

Using λ-calculus as the target is much the same as using combinatory logic. Any representation of $\sim$ should be a closed normal form, but these correspond to combinators of SK-calculus, which does not look promising.

3.5 *SF*-Calculus

Combinations in SF-calculus are given by the BNF

$$M, N ::= S \mid F \mid MN$$

and reduction rules

$$\begin{array}{rll} SMNP & \longrightarrow MP(NP) & \\ FOMN & \longrightarrow M & (O \text{ is an operator}) \\ F(PQ)MN & \longrightarrow NPQ & (PQ \text{ is a compound}) \end{array}$$

where the *operators* O are S and F and the *compounds* are combinations of the form SP, SPQ, FP and FPQ. It follows that the operators are both ternary, so that the compounds are exactly the partial applications of the operators, and so are never redexes. Of course, the side conditions to reduction are easily eliminated, to yield the following rules:

$$\begin{array}{rl} SMNP & \longrightarrow MP(NP) \\ FSMN & \longrightarrow M \\ FFMN & \longrightarrow M \\ F(SP)MN & \longrightarrow NSP \\ F(FP)MN & \longrightarrow NFP \\ F(SPQ)MN & \longrightarrow N(SP)Q \\ F(FPQ)MN & \longrightarrow N(FP)Q\ . \end{array}$$

The definitions of the reduction relation, normal forms, etc is as above.

Like SK-calculus, SF-calculus is Turing complete, in the sense that it can compute the same numeric functions as a universal Turing machine. Unlike SK-calculus, SF-calculus is also *intensionally complete*, in the sense that it can define

the Gödel function that maps a normal form to its Gödel number. Further, all programs, even recursive ones, can be represented by normal forms [3]. In this manner, any program analysis can be performed by applying the Gödel function, performing the analysis numerically, and then, if desired, inverting the Gödel function. In practice, there will be more direct methods of analysis, but this definition establishes a test for completeness.

Most of the difficulties above concerning SK-calculus continue unchanged in SF-calculus. For example, the only ternary combinations available to represent $\sim$ are S and F but neither operator is sufficiently expressive. The main difference is that we *can* generalize the values to be all normal forms of SF-calculus. For example, the equality of normal forms is definable, as is the Gödel function.

The only non-trivial point is to ensure that the normal forms are able to represent all of the functions of interest. In traditional accounts, recursive functions, even primitive recursive functions such as addition, are given by fixpoint functions that do not have normal forms.

4 Wave Calculus

Instead of trying to represent the trees in other systems, let us try to compute with the trees directly, to give rewriting rules for the wave as an operator. Since binary tree are values, the simplest approach is to make the wave into a ternary operator. If it were a *combinator* like S or K then its action would be given by a single rewriting rule that maps any three arguments M, N and P to some combination of them. However, a single such *extensional* rule is clearly inadequate to support arbitrary computations. Rather, the wave must, like F, be an *intensional* operator, able to query the internal structure of its arguments. If the calculus is able to represent the combinators then the three arguments can be examined in turn, so it is enough that the rules query one argument, say the first one. Similarly, there is no need to query more than the topmost layer of structure, since lower layers can be handled recursively. That is, it is enough to give rules for the following three cases:

$$\begin{aligned} \sim\sim MN &\longrightarrow \dots \\ \sim(\sim N)MP &\longrightarrow \dots \\ \sim(\sim PQ)MN &\longrightarrow \dots \end{aligned}$$

Theorem 1. *No matter how the three rules above are completed, the resulting normal forms are exactly the binary trees.*

Proof. Trivial.

The particular rules chosen for wave calculus are inspired by the operators K and S of combinatory logic, and the operator F of SF-calculus. The rules are

$$\begin{aligned} \sim\sim MN &\longrightarrow M && (K) \\ \sim(\sim N)MP &\longrightarrow MP(NP) && (S) \\ \sim(\sim PQ)MN &\longrightarrow NPQ && (F)\ . \end{aligned}$$

The relationships emerge as follows.

The (K) rule gets its name by identifying $\sim\sim$ with the operator K of combinatory logic. Then we can define I by $\sim KK = \sim (\sim\sim)(\sim\sim)$ since

$$\begin{array}{rcll} Ix & = & \sim (\sim\sim)Kx & \\ & \longrightarrow & Kx(\sim x) & (S) \\ & \longrightarrow & x & (K) \end{array}$$

for any combination x.

The (S) rule gets its name because we can identify the combination SMP from combinatory logic with $\sim (\sim P)M$. An exact representation of the operator S will be given in Section 5.

The rule (F) gets its name from the operator F of SF-calculus since, like F, it can reveal internal structure. An exact account of factorisation will be given in Section 6.

Taken together, the three reduction rules of wave calculus show how to eliminate arguments, like K, duplicate arguments, like S, and how to decompose arguments, like F. The simplest way of showing the expressive power of wave calculus is to consider its relationship to SK-calculus and to SF-calculus.

5 λ-Abstraction and Combinators

Let the *terms* of wave calculus be given by the BNF

$$t, u ::= x \mid \ \sim \ \mid tu$$

where x is from a countable class of *variables*. Now we can define the abstraction $\lambda^* x.t$ of a term t with respect to a variable x in the traditional manner by

$$\begin{aligned} \lambda^* x.x &= I \\ \lambda^* x.y &= Ky \quad (y \neq x) \\ \lambda^* x.\sim &= K \sim \\ \lambda^* x.tu &= \sim (\sim (\lambda^* x.u))(\lambda^* x.t) \ . \end{aligned}$$

Theorem 2. *Wave calculus is combinatorially complete.*

Proof. Define K by $\sim\sim$ and S by $\lambda^* x.\lambda^* y. \sim (\sim y)x$. These definitions support the reduction rules of SK-calculus.

Being combinatorially complete, wave calculus has all of the expressive power of combinatory logic, so can define all of the Turing-computable numeric functions, fixpoints, etc.

Thus, there is a reduction- and application-preserving translation of combinatory logic, or SK-calculus, to the calculus of wave combinations with reduction rules (K) and (S). Since this translation does not require the rule (F) it provides another measure of the limitations of SK-calculus: these two rules cannot do the work of three.

Theorem 3. *The wave cannot be defined in SK-calculus. That is, there is no reduction- and application-preserving translation of wave calculus to SK-calculus.*

Proof. The heart of the proof is to show that $\sim$ is able to distinguish combinations that have the same extensional behaviour, which is impossible in combinatory logic. Here are the details.

If there were such as translation then there would be such as translation $[\![-]\!]$ that also preserves variables, from the terms of wave calculus to λ-calculus with both β-contraction and η-expansion, obtained by composing with the translation of SK-calculus to this λ-calculus. Now any combination of the form

$$I_M = \sim (\sim M)(\sim\sim)$$

is an identity function since $\sim (\sim M)(\sim\sim)x \longrightarrow \sim\sim x(Mx) \longrightarrow x$. Hence, the translation $[\![\sim (\sim M)(\sim\sim)]\!]$ reduces to $\lambda x.x$ by first η-expanding to $\lambda x.[\![\sim (\sim M)(\sim\sim)x]\!]$ then reducing in wave calculus and simplifying the translation.

However, $\sim I_M KK$ reduces to $K(\sim M)K$ and then to $\sim M$. Hence

$$\begin{aligned} \sim (\sim I_M KKK)KK &\longrightarrow \sim (\sim MK)KK \\ &\longrightarrow KMK \\ &\longrightarrow M \end{aligned}$$

recovers M from I_M. Hence, there is a way to recover $[\![M]\!]$ from $[\![I_M]\!]$ or from its reduct $\lambda x.x$. Since this is impossible, there can be no such translation.

6 Intensional Completeness

It remains to consider the relationship between SF-calculus and wave calculus. We begin by showing how to factorise the compounds in wave calculus, and then consider translations from SF-calculus, and finally show that wave calculus is intensionally complete.

The irreducible applications, or *compounds* of wave calculus take one of the two forms $\sim P$ and $\sim PQ$. Hence, to capture the conditional rewriting rules for F in Section 3.5 requires a means of distinguishing $\sim$ from $\sim P$ from $\sim PQ$.

Given terms $\mathsf{is0}$, $\mathsf{is1}$ and $\mathsf{is2}$ define q by

$$q\ x = \ \sim x(K^2\mathsf{is0})(K^4\mathsf{is2})(KI)\ \mathsf{is1}\ .$$

Then we have

$$\begin{aligned} q\ \sim &\longrightarrow \mathsf{is0} \\ q\ (\sim P) &\longrightarrow \mathsf{is0}\ (KI)\ \mathsf{is1} \\ q\ (\sim PQ) &\longrightarrow \mathsf{is2}\ . \end{aligned}$$

Hence, if $\mathsf{is0}$ is K and $\mathsf{is1}$ and $\mathsf{is2}$ are both KI then $q = q_0$ is a test for being an operator. Similarly, if $\mathsf{is1}$ is K and the other two are KI then $q = q_1$ is a test

for being a stem. If $\mathsf{is2}$ is K and the other two are KI then $q = q_2$ is a test for being a fork.

Define

$$\begin{aligned} \mathsf{swap}\ N &= \lambda^* x.\lambda^* y.Nyx \\ \mathsf{compose}\ M\ N &= \lambda^* x.M(Nx) \end{aligned}$$

and

$$\begin{aligned} \mathsf{factor}(x, M, N) = \mathsf{is0}\ x\ M\ (\mathsf{is1}\ x\ (\sim (x \sim) M(\mathsf{swap}\ N)) \\ (\sim xM(\mathsf{compose}\ N\ \ \sim))) \ . \end{aligned}$$

Now

$$\begin{aligned} \mathsf{factor}(\sim, M, N) &\longrightarrow M \\ \mathsf{factor}(\sim y, M, N) &\longrightarrow \sim (\sim y \sim) M(\mathsf{swap}\ N) \\ &\longrightarrow \mathsf{swap}\ Ny \sim \\ &\longrightarrow N \sim y \\ \mathsf{factor}(\sim yz, M, N) &\longrightarrow \sim (\sim yz) M(\mathsf{compose}\ N \sim) \\ &\longrightarrow \mathsf{compose}\ N \sim yz \\ &\longrightarrow N(\sim y)z \end{aligned}$$

shows that factor factorizes, so define

$$F = \lambda^* x.\lambda^* y.\lambda^* z.\mathsf{factor}(x, y, z) \ .$$

Although this F does indeed factorise in wave calculus, the function which maps F of SF-calculus to F in wave calculus does *not* preserve reduction, as it maps the operator F to a compound. To preserve reduction, F must be mapped to a combination C that treats C itself as if it were an atom. However, developing this translation would take us too far from our exploration of texts.

Theorem 4. *Wave calculus is intensionally complete.*

Proof. The Gödel function can be defined by recursively factoring the argument to reveal its tree structure.

7 Types

Wave calculus support a simple type system. Leaves, stems and forks all have distinct types. Then subtyping is used to coerce forks to take function types. The *types* are given by

$$T ::= \mathsf{Leaf} \mid \mathsf{Stem}\ T \mid \mathsf{Fork}\ T\ T \mid T \to T \ .$$

The*subtyping relation* is given by

$$\begin{aligned} \mathsf{Fork}\ \mathsf{Leaf}\ U &< U \to V \to U \\ \mathsf{Fork}\ (U \to V)\ (U \to V \to T) &< U \to T \\ \mathsf{Fork}\ (\mathsf{Fork}\ T U)\ V &< (T \to U \to W) \to W\ . \end{aligned}$$

Note that since sub-typing relates fork types to function types, it is neither reflexive nor transitive.

The *type derivation rules* are given by

$$\frac{}{\sim : \mathsf{Leaf}} \qquad \frac{t : \mathsf{Leaf} \quad u : U}{tu : \mathsf{Stem}\ U} \qquad \frac{t : \mathsf{Stem}\ T \quad u : U}{tu : \mathsf{Fork}\ T\ U}$$

$$\frac{t : U}{t : T}\ U < T \qquad \frac{t : U \to T \quad u : U}{tu : T}$$

Theorem 5. *Reduction preserves typing.*

Proof. The proof has been verified in Coq by straightforward, but tedious, induction.

8 Conclusions

By basing computation on syntax instead of numbers we can avoid a whole layer of meta-computation required to encode syntax as numbers and decode numbers as syntax. However, this exposes the limitations of traditional combinatory logic: despite being complete for all the Turing-computable numerical functions, it is not complete for all the computable functions of syntax. In the past,we have expressed this limitation in terms of the inability to compute with its own normal forms, but the social impact of the result has been small, since few care about combinators at all, much less their normal forms. Now we can see the impact in terms of things that we all care about, namely words and texts.

Combinatory logic is unable to represent computations on trees directly and so must rely on heavy encodings that lie outside the calculus. Generalizing the concerns to SK-combinators in normal form does not help, as SK-calculus is not intensionally complete: there are computable functions of its normal forms that are not definable within the calculus.

SF-calculus is a little better than combinatory logic. It is still unable to represent syntax directly, but it is intensionally complete, and so able to work with trees whose leaves are labelled by S and F.

By contrast, wave calculus handles trees directly: its combinations are exactly the rose trees; its normal forms are exactly the binary trees; and it is intensionally complete, i.e. is Turing complete (with respect to numeric functions) and can define the Gödel function of its normal forms. It has a simple types system that reflects the nature of the trees.

All of this suggests that trees form a better foundation for computation than natural numbers. They have the same mathematical or ontological status as the natural numbers. They have an intrinsic interest to computer scientists. As syntax, they are of interest to the literati, as well the numerati.

Acknowledgments Thanks to Xuanyi Chew, Thomas Given-Wilson, Achim Jung, Reuben Rowe and Jose Vergara for stimulating discussions.

References

1. H.B. Curry and R. Feys. *Combinatory Logic*, volume 1 of *Combinatory Logic*. North-Holland Publishing Company, 1958.
2. R. Hindley and J.P. Seldin. *Introduction to Combinators and Lambda-calculus*. Cambridge University Press, 1986.
3. 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.
4. Barry Jay and Thomas Given-Wilson. A combinatory account of internal structure. *Journal of Symbolic Logic*, 76(3):807–826, 2011.