The tree machine*

Arnaud Spiwack

MINES ParisTech
`arnaud@spiwack.net`

Abstract. A variant of Turing machines is introduced where the tape is replaced by a single tree which can be manipulated in a style akin to purely functional programming. This yields two benefits: first, the extra structure on the tape can be leveraged to write explicit constructions of machines much more easily than with Turing machines. Second, this new kind of machines models finely the asymptotic complexity of functional programming languages, and may allow to answer questions such as "is this problem inherently slower in functional languages".

1 Intro

This article came to be as I seem to find myself all to often in two kinds of discussions: one of them is the functional programmer's complaint that Turing machine make an unpleasant computation model as it is so unstructured that writing any explicit Turing machine is a chore usually left to the gods of hand-waving. The second one is a common interrogation about some computational problem: "is it actually slower by a logarithmic factor to solve with a purely functional program, rather than an imperative one".

My inner functional programmer's reflex would be to turn to λ-calculus to answer such questions. However, there is no denying that it is easier to quantify over automata-like machines, such as Turing-machines, for the purpose of proving complexity results. With that in mind, I will introduce, in this article, a variant of Turing machines, the tree machine, with better structured data which correspond faithfully to the cost-model of purely functional programming languages.

I will not attempt to answer, either positively or negatively, whether concrete problems are slower or not in purely functional style; I will however, give an explicit description of a machine implementing λ-calculus in section 5, to demonstrate that it is effectively possible to write non-trivial machines explicitly in this model.

Acknowledgment I want to thank Alexandre Miquel and Guyslain Naves who, most independently, planted the seed of this article in my mind through very entertaining and enlightening discussions.

2 Eilenberg's machines

We will work with a generic notion of machines introduced by Eilenberg [4, Chapter 10], which can be instantiated to yield finite automata as well as Turing machines. The tree machine introduced in Section 3 is yet another instantiation of Eilenberg's machine (in fact we will give several equivalent definitions).

A type of machine is given by a set X of *data* and a set $\Phi \subseteq \mathcal{P}(X \times X)$ of *instructions*. Most of the times the instructions will be partial relations. A machine of type (X, Φ) is given by a finite set Q of *states*, and subsets I and F of initial and finite states, as usual for automata. Transitions are labelled with relations of Φ. A path $q_0, \dots, q_n$ computes the composition of the relations on the successive edges. The machine itself compute the union of the relation computed by path from an initial state to a final state.

It does not change the expressiveness to close Φ by composition $(i_1 \cdot i_2)$, union $(i_1 + i_2)$, identity $(\mathbf{1})$[1] and empty relation $(\mathbf{0})$. We shall use this fact implicitly.

In fact, relations computed by a machine of type (X, Φ) are exactly the relations in the sub-Kleene algebra of $\mathcal{P}(X \times X)$ generated by Φ: the result for finite automata lifts naturally to Eilenberg's machines. So Eilenberg machines are equivalent to regular expressions with alphabet Φ. However, if closing Φ by all the regular expression operations does not change what relation the machines compute, it does change the complexity. Since we are concerned with complexity properties, we may therefore refer to regular expressions as machines, while we will reserve the term *instructions* to star-free expressions.

This definition of Eilenberg's machines is naturally non-deterministic. It would be more accurate to work with deterministic machines in the setting of this article, but it does not really change anything of substance, and would unnecessarily clutter the presentation. So the machines throughout this article will be non-deterministic, but all of them could be made deterministic, and actually should, for practical applications.

3 Tree machines

Let us define the set T of (rooted, unlabeled, binary) trees as the set generated by the following grammar:

$$u, v \ ::= \ () \ | \ (u, v)$$

Such trees will be the data of our tree machines. Take notice of the fact that trees do not replace the alphabet of Turing machines but the whole tape: there will not be a tape of trees, just one tree.

---

[1] In Eilenberg's formulation, a more general kind of relation is considered, in order to be able, typically, to count the multiplicity of successful path. In that case, closure by $\mathbf{1}$ (*i.e.* adding ϵ-transitions) is not permitted.

Let us define the following partial functions on T:

$$
\begin{array}{llll}
\delta(x) & = & (x, x) & \\
\pi_1(x) & = & y & (\text{if } \exists z^{\in T}.\ x = (y, z)) \\
\pi_2(x) & = & z & (\text{if } \exists y^{\in T}.\ x = (y, z)) \\
(i, j)\,(x) & = & (i(y), j(z)) & (\text{if } x = (y, z), \text{ for } i \text{ and } j \text{ partial functions}) \\
\varepsilon(x) & = & () & \\
()\,(x) & = & () & (\text{if } x = ())
\end{array}
$$

The set of instruction Φ is chosen to be the smallest set containing the partial functions $\{\delta; \pi_1; \pi_2; \varepsilon; ()\,; \mathbf{1}\}$ and closed by $(\cdot, \cdot)$. This set of instructions has been chosen to correspond to the presentation of cartesian products and terminal elements in categories as adjunctions.

We call *tree machine* a machine of type (T, Φ). Notice that, contrary to Turing machines, tree machines are not parametrised by an alphabet: the tree structure offers enough power on its own.

Tree machines, by virtue of the $(\cdot, \cdot)$ instruction scheme, has an infinite number of instructions which make it possible to observe the tree and modify it at arbitrary depths. However each individual instruction affects trees at a bounded depth, which is considered a constant time operation in functional language, which is the important property we want to ensure. As we shall see in Section 4.2, this choice of an infinite set of instruction is pure convenience: a finite set suffices.

3.1 A language of guards and actions

One way to read the instruction in Φ, is to think of them as combinators giving means to match tree prefixes and rearrange the corresponding subtrees. That is, instructions of tree machines perform pattern-matching. We shall give an alternative set of instructions for tree machines which is suggestive of the pattern-matching notation functional programmers all know and love.

We write $\gamma \Rightarrow \alpha$ for a partial function whose domain is denoted by the *guard*, or pattern, γ and whose functional action is denoted by the *action* α. Both γ and α are trees with variables, with the following restrictions: variables occur at most once γ, and all variables of α appear in γ (variables in γ bind variables in α). We may use $\gamma_1 \Rightarrow \alpha_1 \mid \gamma_2 \Rightarrow \alpha_2$ instead of $(\gamma_1 \Rightarrow \alpha_1) + (\gamma_2 \Rightarrow \alpha_2)$ when γ_1 and γ_2 denote disjoint domains (*i.e.* γ_1 and γ_2 are not unifiable). We can also use the wildcard pattern "_" to represent a variable in γ which does not bind a variable in α.

For instance, the instructions of Φ can be represented using this notations as follows:

$$
\begin{array}{rcll}
\mathbf{1} & = & x \Rightarrow x \\
\delta & = & x \Rightarrow (x, x) \\
\pi_1 & = & (x, \_) \Rightarrow x \\
\pi_2 & = & (\_, y) \Rightarrow y \\
(i, j) & = & (\gamma_i, \gamma_j) \Rightarrow (\alpha_i, \alpha_j) & \quad \text{(for } i = \gamma_i \Rightarrow \gamma_j \text{ and } j = \gamma_j \Rightarrow \alpha_j) \\
\varepsilon & = & \_ \Rightarrow () \\
() & = & () \Rightarrow ()
\end{array}
$$

Conversely, the language of guard and action is subsumed by Φ, which we shall use as a definition rather than giving and independent definition and prove it as a theorem (which would be, of course, theoretically possible but not practically useful). The definition is lexicographically recursive on the subterm ordering of γ then that of α.

$$
\begin{array}{rcll}
(x \Rightarrow x) & = & \mathbf{1} \\
(x \Rightarrow ()) & = & \varepsilon \\
(() \Rightarrow ()) & = & () \\
((\gamma_1, \gamma_2) \Rightarrow ()) & = & ((\gamma_1 \Rightarrow ()), (\gamma_2 \Rightarrow ())) \cdot \varepsilon \\
((\gamma_1, \gamma_2) \Rightarrow x) & = & ((\gamma_1 \Rightarrow x), (\gamma_2 \Rightarrow ())) \cdot \pi_1 & \quad \text{(when } x \text{ occurs in } \gamma_1) \\
((\gamma_1, \gamma_2) \Rightarrow x) & = & ((\gamma_1 \Rightarrow ()), (\gamma_2 \Rightarrow x)) \cdot \pi_2 & \quad \text{(when } x \text{ occurs in } \gamma_2) \\
(\gamma \Rightarrow (\alpha_1, \alpha_2)) & = & \delta \cdot ((\gamma \Rightarrow \alpha_1), (\gamma \Rightarrow \alpha_2))
\end{array}
$$

The language of pattern and action can be used to conveniently defined the following examples:

$$
\begin{array}{rcl}
\sigma & = & ((x, y) \Rightarrow (y, x)) \\
& = & \delta \cdot ((\varepsilon, \mathbf{1}) \cdot \pi_2, (\mathbf{1}, \varepsilon) \cdot \pi_1) \\
\mathsf{push} & = & ((x, y), z) \Rightarrow (x, (y, z)) \\
& = & \delta \cdot (((\mathbf{1}, \varepsilon) \cdot \pi_1, \varepsilon) \cdot \pi_1, \delta \cdot (((\varepsilon, \mathbf{1}) \cdot \pi_2, \varepsilon) \cdot \pi_1, ((\varepsilon, \varepsilon) \cdot \varepsilon, \mathbf{1}) \cdot \pi_2))
\end{array}
$$

This procedure does not give the smallest possible definition in terms of Φ of σ and push (or of pretty much anything for that matter). Here are better candidates for this particular award:

$$
\begin{array}{rcl}
\sigma & = & \delta \cdot (\pi_2, \pi_1) \\
\mathsf{push} & = & \delta \cdot (\pi_1 \cdot \pi_1, \delta \cdot (\pi_2 \cdot \pi_1, \pi_2))
\end{array}
$$

In either case, however, it is fair to claim that the language of guard and patterns gives a clearer account of the intent and semantics of instructions than the more elementary Φ. In consequence we will peruse the guard and actions in the remainder of the article.

3.2 Constants

It will be useful to embed natural numbers in trees. Any embedding will do. We choose a binary encoding for the sake of fun and compactness:

$$
\begin{aligned}
0 &= () \\
2 \times n + 1 &= (() , n) \\
2 \times n + 2 &= ((() , ()) , n)
\end{aligned}
$$

A direct consequence of this encoding is that any finite set of symbols can be easily represented as particular trees, by mapping them to arbitrary distinct natural numbers. We will do so quite liberally.

3.3 Zipper

In [5] Huet presents a purely functional data structure, the *zipper* to implement "pointers" in trees, *i.e.* a way to walk through a tree from parent to child or back in constant time and to replace the pointed subtree also in constant time.

The zipper can be adapted to the tree machines as a set of instructions. The zipper instruction provide us with an ability we have not had so far: modifying a subtree at an *a priori* unbounded depth in a tree.

The idea is that a zipper is represented as a pair (c, t) of the pointed subtree t together with a "reversed tree" c which represents the context and gives enough information to rebuild the tree when walking up towards the parent (with the instruction up below). Here are the relevant instructions:

$$
\begin{aligned}
\mathsf{open} &= x \Rightarrow (() , x) \\
\mathsf{left} &= (x, (y, z)) \Rightarrow ((0, (x, z)) , y) \\
\mathsf{right} &= (x, (y, z)) \Rightarrow ((1, (x, y)) , z) \\
\mathsf{up} &= ((0, (x, z)) , y) \Rightarrow (x, (y, z)) \mid ((1, (x, y)) , z) \Rightarrow (x, (y, z)) \\
\mathsf{exit} &= (() , x) \Rightarrow x
\end{aligned}
$$

Where open and exit transform a tree into a pointer to its root and back, left walks down to the left child (and marks, in the reversed tree, with 0 that it did walk down left), right to the right child, and up walks up to the parent, and reconstruct the tree according to the mark left by either left or right. For further detail, the reader unfamiliar with these concepts is deeply encouraged to read Huet's paper [5]

4 Comparisons

With the basic material now set in place, we can now turn to the use of the tree machine as a computational complexity model. It is important to be precise on what is meant here: clearly, complexity classes are very robust, and it does not matter what computation model is taken to define them; the tree machine is no exception. However, for more fine grained accounts of asymptotic complexity,

the model will matter a lot: in Turing machines already, multiple-tape Turing machines can provide a quadratic speedup over single-tape ones.

It is this sort of complexity that is our concern, and the claim of this article is that the tree machine is a good complexity model for purely functional computations.

4.1   Turing machines as tree machines

It is straightforward to implement Turing machines as tree-machines: fixing a coding for the alphabet, we arrange the tree to be a pair (L, R) of lists of symbols. The list $R = (a_0, (a_1, (\ldots, () \ldots)))$ represents the part of the tape just under and to the right of the head (a_0 is the symbol under the head). The reversed list $L = (((\ldots () , \ldots), b_2), b_1)$ represents the part of the tape which sits to the left of the head. With this representation there is no need for a special symbol to stand at empty slots on the tape: instead the symbol under the head is empty when R is the empty list $()$.

The instruction of Turing machines are implemented as instruction of the tree machines (in guard-and-action style):

- Write symbol a under the head: $(L, (\_, R)) \Rightarrow (L, (a, R)) \mid (x, ()) \Rightarrow (x, (a, ()))$
  (the second case extends the tape if we reached the end)
- Move right: $(L, (x, R)) \Rightarrow ((L, x), R)$
- Move left: $((L, x), R) \Rightarrow (L, (x, R))$
- Check that symbol a is under the head: $(L, (a, R)) \Rightarrow (L, (a, R))$

Therefore, a Turing machines is translated to a tree machine with the same state and transitions, except the instructions labelling transitions are replaced with their respective implementation as tree machine instructions.

This translation highlights a point which is occasionally overlooked: Turing machines are not a very good model of imperative programs. Turing machines can be simulated in constant time in a purely functional language, and so can multiple-tape Turing machines. To get a better model of imperative programs, we shall turn to random-access machines in Section 4.4.

4.2   A finite type for tree machines

In order to build the converse translation of tree machines into Turing machines, it will be convenient for the set instruction to be presented by a finite set Ψ. To obtain this finite presentation we will use the zipper instructions from Section 3.3. To be more specific, we will implement the instruction $(i, j) \in \Phi$ as a (finite) sequence of zipper operations which will walk through the tree to apply the appropriate actions to the appropriate subtrees.

Remember that a zipper is a pair (c, t) of a focused subtree t together with a reversed tree c representing the necessary context to rebuild the tree. Our goal is to lift the instructions of Φ so that they apply to the focused subtree instead of the root of the complete tree. In other words, we are looking for a $[\![i]\!] = (1, i)$ for each $i \in \Phi$.

This property that $[\![i]\!] = (1, i)$ acts as a perfectly fine definition for each of the generators of Φ:

$$
\begin{aligned}
[\![\mathbf{1}]\!] &= (\mathbf{1}, \mathbf{1}) \\
[\![()]\!] &= (\mathbf{1}, ()) \\
[\![\varepsilon]\!] &= (\mathbf{1}, \varepsilon) \\
[\![\pi_1]\!] &= (\mathbf{1}, \pi_1) \\
[\![\pi_2]\!] &= (\mathbf{1}, \pi_2) \\
[\![\delta]\!] &= (\mathbf{1}, \delta)
\end{aligned}
$$

For the case (i, j) however, in order to avoid introducing infinitely many instructions in Ψ, we need to find an alternative definition in terms of the zipper operations. We define $[\![(i, j)]\!]$ recursively as follows:

$$
[\![(i, j)]\!] = \mathsf{left} \cdot [\![i]\!] \cdot \mathsf{up} \cdot \mathsf{right} \cdot [\![j]\!] \cdot \mathsf{up}
$$

It is a straightforward exercise of symbol pushing to verify that indeed, by induction, $[\![(i, j)]\!] = (\mathbf{1}, (i, j))$.

Every instruction $i \in \Phi$ can be implemented as $\mathsf{open} \cdot [\![i]\!] \cdot \mathsf{exit}$. Therefore we can take the set Φ as being

$$
\Psi = \{\mathsf{open}; \mathsf{left}; \mathsf{right}; \mathsf{up}; \mathsf{exit}; (\mathbf{1}, \mathbf{1}); (\mathbf{1}, ()); (\mathbf{1}, \varepsilon); (\mathbf{1}, \pi_1); (\mathbf{1}, \pi_2); (\mathbf{1}, \delta)\}
$$

4.3   Tree machines as Turing machines

Translating tree machines into Turing machines is not as direct as the converse. One way to translate trees into word so that it fits a Turing machine tape is to use the Polish notations: we take the alphabet to include $\{\mathsf{p}; \mathsf{u}\}$ (for *pair* and *unit* respectively). The tree $((), ((), ()))$ is then translated to pupuu.

As always, giving a concrete definition of a Turing machine – or even of a translation to Turing machine – is not very easy nor particularly enlightening, and, in fact, would lead us way over the page limit. On the other hand, equipped with the finite presentation of Section 4.2, it is quite clear that it can in principle be done.

Let us sketch how such a construction could be achieved. In a first step we can ignore the open and exit instruction, and assume we are always working on a zipper. The rational is that $\mathsf{exit} \cdot \mathsf{open}$ acts as the identity on a zipper which is focused at the root, so we may simply represent non-zipper trees as zipper focused at the root and both instructions become the identity. One may be tempted to replace calls to exit to test that the context is a u, but this is not even necessary as it is an invariant of the translation.

To represent the zipper (c, t), the most convenient way is to have two tapes, one holding c and the other holding t. A third tape will be used to store a counter to navigate through Polish-notation trees, and a fourth to store intermediate trees which are to be moved or copied.

The instructions of tree machines are not translated as constant time instructions. However, they are all in a polynomial P (which depends on the details of

the translation, but is at least of degree 1) of the current size of the tape. Hence, if the complexity of a tree machine is $O(f(n))$, then the corresponding Turing machine has complexity $O(P(f(n)) \times f(n))$, which is in the same complexity class.

Therefore, tree machines and Turing machines have the same complexity classes. However, the translation from tree machines to Turing machines is non-trivial both in term of slowdowns of the translated machine and complexity of the translation itself. It would be quite hard to get an explicit description of the translation. On the other hand the translation of Turing machines into tree machines is quite direct. Tree machines fare pretty well on that front.

4.4   Tree machines and random-access machines

Random-access machines – which happen to be yet another instance of Eilenberg machine – support more natural translations of tree machines: the encoding of algebraic data types of functional programming languages. In such a translation, a tree is encoded as an address, at this address there is 0 if the tree is empty, and 1 if the tree is a pair. In the latter case, the two following addresses contain the addresses of the two subtrees.

Unfortunately, this translation would not fit these pages either, as it has to solve the problem of memory allocation and garbage collection in order to preserve the space complexity of tree machines. Garbage collection can be done achieved via reference counting [8] since the tree cannot have cycles, but it still would not make a program under a page long (or anything near it).

Nevertheless, this translation is quite concrete and serves as a good test for the tree machine. Tree machines, indeed, can implemented in the traditional complexity model of practical computer and, assuming garbage collection away, this implementation preserves the complexity of machines. Assuming that garbage collection is constant time may seem unreasonable, but it is in fact the way functional programmers think about their programs: as having negligible overhead due to garbage collection.

This translation also serves to remark a limitation of the tree machine for space efficiency. Indeed trees can have various representations in a random-access machine with more or less sharing between subtrees. In the worst case, a maximally shared subtree is exponentially smaller than its sharing-free equivalent. Therefore, in an accurate cost model for tree-machine space consumption, the space occupied by a tree cannot be read on the tree itself: it depends on the history of how the tree was built. There is no particular problem in defining such a dynamic space-cost semantics though, see [3] for a much more ambitious case. It also means that the translation to Turing machines in Section 4.3 is not accurate as far as space complexity is concerned.

Conversely, it is not difficult to implement a random-access machine as a tree machine. Random-access machines are composed of arithmetic operations and an addressed memory. Arithmetic is straightforward, and addressed memory can be stored in a tree (most likely a trie such as in [7, Chapter 10]).

Random-access machines typically assume constant time arithmetic operations and memory access. The translation of random-access machines into tree machines preserves neither. For the case of constant-time arithmetic, the assumption is not realistic for big numbers, it is actually meant to model the fact that no big numbers will appear and that bounded arithmetic is sufficient for the modelled algorithms. To make such an hypothesis in tree machines, they would need to be outfitted with a primitive notion of integers like random-access machines.

Constant-time memory access is more subtle: it is a reasonable assumption for a vast majority of programs, yet, a more accurate model may use logarithmic-time memory access [2]. Logarithmic-time memory access is precisely what is provided by the tree machine, so it might look as though the superiority of the random-access machine were illusory. It is not, however, as when a model with logarithmic-time memory access apply, it will not be *a priori* reasonable to see the instructions of the tree machines as running in constant time.

5 Encoding λ-calculus

To conclude, I want to present, with few words, an encoding of λ-calculus in tree machines to demonstrate that it is really practical to make explicit definitions of non-trivial machines.

We shall use the simple explicit substitution calculus called λv-calculus [6]. It uses de Bruijn indices and has three kinds of substitutions: $[v]$ for a term v, $[\uparrow]$ (read *shift*) and $[\Uparrow s]$ (read *lift s*) for a substitution s.

$$
\begin{array}{lcl}
(\lambda u)v & \rightsquigarrow & u[v] \\
(\lambda u)[s] & \rightsquigarrow & \lambda(u[\Uparrow s]) \\
(uv)[s] & \rightsquigarrow & (u[s])(v[s]) \\
0[v] & \rightsquigarrow & v \\
(n+1)[v] & \rightsquigarrow & n+1 \\
0[\Uparrow s] & \rightsquigarrow & 0 \\
(n+1)[\Uparrow s] & \rightsquigarrow & n[s][\uparrow] \\
n[\uparrow] & \rightsquigarrow & n+1
\end{array}
$$

Let us suppose fixed an encoding for the symbols that we will use in the encoding of terms: $\{\lambda; \mathsf{app}; \mathsf{succ}; \mathsf{nought}; \mathsf{subst}; \mathsf{term}; \mathsf{lift}; \mathsf{shift}; \mathsf{ok}\}$. The terms λu and uv are represented, respectively (λ, u) and $(\mathsf{app}, (u, v))$. The de Bruijn index 2 is represented as $(\mathsf{succ}, (\mathsf{succ}, \mathsf{nought}))$.

We will use the non-determinism of tree machines to represent the non-determinism of β-reduction, which is achieved navigating non-deterministically in λ-terms with zipper-like instructions.

$$
\begin{array}{lcl}
\mathsf{open} & = & u \Rightarrow (()\,, u) \\
\mathsf{down}_\lambda & = & (c, (\lambda, u)) \Rightarrow ((c, \lambda)\,, u) \\
\mathsf{down}_\sigma & = & (c, (\mathsf{subst}, (u, s))) \Rightarrow ((c, (\mathsf{subst}, s))\,, u) \\
\mathsf{left} & = & (c, (\mathsf{app}, (u, v))) \Rightarrow ((c, (\mathsf{app}, (0, v)))\,, u) \\
\mathsf{right} & = & (c, (\mathsf{app}, (u, v))) \Rightarrow ((c, (\mathsf{app}, (1, u)))\,, v)
\end{array}
$$

$$
\begin{aligned}
\mathsf{up}_\lambda &= ((c, \lambda), u) \Rightarrow (c, (\lambda, u)) \\
\mathsf{up}_\sigma &= ((c, (\mathsf{subst}, s)), u) \Rightarrow (c, (\mathsf{subst}, (u, s))) \\
\mathsf{up}_{\mathsf{app}_0} &= ((c, (\mathsf{app}, (0, v))), u) \Rightarrow (c, (\mathsf{app}, (u, v))) \\
\mathsf{up}_{\mathsf{app}_1} &= ((c, (\mathsf{app}, (1, u))), v) \Rightarrow (c, (\mathsf{app}, (u, v))) \\
\mathsf{up} &= \mathsf{up}_\lambda \mid \mathsf{up}_\sigma \mid \mathsf{up}_{\mathsf{app}_0} \mid \mathsf{up}_{\mathsf{app}_1} \\
\mathsf{exit} &= ((), u) \Rightarrow u \\
\mathsf{move} &= \mathsf{down}_\lambda + \mathsf{down}_\sigma + \mathsf{left} + \mathsf{right} + \mathsf{up} \\
\mathsf{zip} &= \mathsf{up}^* \cdot \mathsf{exit}
\end{aligned}
$$

The reduction rules are given in a context independent manner:

$$
\begin{aligned}
\beta &= (\mathsf{app}, ((\lambda, u), v)) \Rightarrow (\mathsf{subst}, (u, (\mathsf{term}, v))) \\
\lambda_\sigma &= (\mathsf{subst}, ((\lambda, u), s)) \Rightarrow (\lambda, (\mathsf{subst}, (u, (\mathsf{lift}, s)))) \\
\mathsf{app}_\sigma &= (\mathsf{subst}, ((\mathsf{app}, (u, v)), s)) \Rightarrow (\mathsf{app}, ((\mathsf{subst}, (u, s)), (\mathsf{subst}, (v, s)))) \\
\mathsf{nought}_{\mathsf{term}} &= (\mathsf{subst}, (\mathsf{nought}, (\mathsf{term}, v))) \Rightarrow v \\
\mathsf{succ}_{\mathsf{term}} &= (\mathsf{subst}, ((\mathsf{succ}, n), (\mathsf{term}, \_))) \Rightarrow \mathsf{succ}(n) \\
\mathsf{nought}_{\mathsf{lift}} &= (\mathsf{subst}, (\mathsf{nought}, (\mathsf{lift}, \_))) \Rightarrow \mathsf{nought} \\
\mathsf{succ}_{\mathsf{lift}} &= (\mathsf{subst}, ((\mathsf{succ}, n), (\mathsf{lift}, s))) \Rightarrow (\mathsf{subst}, ((\mathsf{subst}, (x, s)), \mathsf{shift})) \\
\mathsf{nought}_{\mathsf{shift}} &= (\mathsf{subst}, (\mathsf{nought}, \mathsf{shift})) \Rightarrow (\mathsf{succ}, \mathsf{nought}) \\
\mathsf{succ}_{\mathsf{shift}} &= (\mathsf{subst}, ((\mathsf{succ}, n), \mathsf{shift})) \Rightarrow (\mathsf{succ}, (\mathsf{succ}, n)) \\
\mathsf{var}_{\mathsf{shift}} &= \mathsf{nouhgt}_{\mathsf{shift}} \mid \mathsf{succ}_{\mathsf{shift}} \\
\sigma &= \lambda_\sigma \mid \mathsf{app}_\sigma \mid \mathsf{nought}_{\mathsf{term}} \mid \mathsf{succ}_{\mathsf{term}} \mid \mathsf{nought}_{\mathsf{lift}} \mid \mathsf{succ}_{\mathsf{lift}} \mid \mathsf{var}_{\mathsf{shift}}
\end{aligned}
$$

A step of β-reduction is represented as one β rule followed by eagerly applying σ rules. To ensure that explicit substitutions have been eliminated, substitution-free terms are marked with ok.

$$
\begin{aligned}
\mathsf{nought}_{\mathsf{ok}} &= \mathsf{nought} \Rightarrow (\mathsf{ok}, \mathsf{nought}) \\
\mathsf{succ}_{\mathsf{ok}} &= (\mathsf{succ}, n) \Rightarrow (\mathsf{ok}, (\mathsf{succ}, n)) \\
\lambda_{\mathsf{ok}} &= (\lambda, (\mathsf{ok}, u)) \Rightarrow (\mathsf{ok}, (\lambda, u)) \\
\mathsf{app}_{\mathsf{ok}} &= (\mathsf{app}, ((\mathsf{ok}, u), (\mathsf{ok}, v))) \Rightarrow (\mathsf{ok}, (\mathsf{app}, (u, v))) \\
\mathsf{rule}_{\mathsf{ok}} &= \mathsf{nought}_{\mathsf{ok}} + \mathsf{succ}_{\mathsf{ok}} + \lambda_{\mathsf{ok}} + \mathsf{app}_{\mathsf{ok}} \\
\mathsf{check}_{\mathsf{ok}} &= (\mathsf{ok}, u) \Rightarrow u
\end{aligned}
$$

A step of β-reduction is encoded as the following machine[2]:

$$
\begin{aligned}
\mathsf{all}_\sigma &= (\mathsf{move}^* \cdot (\mathbf{1}, \sigma))^* \\
\mathsf{zip}_{\mathsf{ok}} &= (\mathsf{move}^* \cdot (\mathbf{1}, \mathsf{rule}_{\mathsf{ok}}))^* \cdot \mathsf{exit} \cdot \mathsf{check}_{\mathsf{ok}} \\
\mathsf{step} &= \mathsf{open} \cdot \mathsf{move}^* \cdot (\mathbf{1}, \beta \cdot \mathsf{open} \cdot \mathsf{all}_\sigma \cdot \mathsf{zip}_{\mathsf{ok}}) \cdot \mathsf{zip}
\end{aligned}
$$

6 Conclusion

The tree machine is designed to stand as a standard model of complexity for purely functional algorithms, much like the random-access machine is for imperative algorithms. At the cost of not being quite as minimalist as Turing machines,

---

[2] Notations are abused a little in this example, as $(\cdot, \cdot)$ is used with relations which are not instructions from the type Φ. It can be made formal using Section 4.2.

tree machines are quite expressive and make it reasonably easy to write explicit machines to implement desired behaviours.

In fact, if in this article I have given direct descriptions of machines, a technique, favoured by Danvy, allows to compile programs written in a very small purely functional idealised Scheme language, with full recursion and higher order, into a tree machine. This is achieved by apply sequentially to the program the following transformations: defunctionalisation, continuation-passing style transformation, and defunctionalisation again [1]. This sequence of transformation yields a mutually recursive first-order program where all calls (in particular recursive calls) are tail, while preserving the structure of the program. At this point we are one tiny step shy of having a tree machine: function environments must be reified as data and variables as accessors in this data. Then, using the guard-and-action presentation of tree machines, we can build a machine with just one state per function in the transformed program (the functions of the original program plus one or two administrative functions). Yielding another, more indirect, way to produce explicit tree machines.

References

1. Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming - PPDP '03*, number March, pages 8–19, New York, New York, USA, 2003. ACM Press.
2. Alok Aggarwal, Bowen Alpern, Ashok K Chandra, and Marc Snir. A model for hierarchical memory. *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 305—-314, 1987.
3. Guy Blelloch and Robert Harber. Cache and I/O efficent functional algorithms. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '13*, page 39, New York, New York, USA, 2013. ACM Press.
4. Samuel Eilenberg. *Automata, languages, and machines.* 1974.
5. Gérard Huet. Functional Pearl: The Zipper. *Journal of Functional Programming*, 7:7–5, 1997.
6. Pierre Lescanne. From $\lambda\sigma$ to λv: a journey through calculi of explicit substitutions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '94*, pages 60–69, New York, New York, USA, 1994. ACM Press.
7. Chris Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.
8. Paul R Wilson. Uniprocessor garbage collection techniques. *Memory Management*, 1992.