ELSEVIER Science of Computer Programming 32 (1998) 109–143

Science of Computer Programming

Linear-time hierarchies for a functional language machine model

Eva Rose*

DIKU, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen Ø, Denmark

Abstract

In STOC 1993, Jones showed the existence of a hierarchy within problems decidable in linear time by a canonical first-order functional language based on tree-structured data (F), as well as for an extension of that language based on graph-structured data maintained through selective updating (F^{su}). In this paper, we prove the existence of a linear-time hierarchy for an authentic and realistic intermediate "machine" language featuring higher order constructs: the *Categorical Abstract Machine*. We show the existence of such a hierarchy for the Categorical Abstract Machine based on tree-structured data (CAM) as well as on graph-structured data (CAM^{su}). The existence is shown by constructing mutually efficient interpreters between CAM and F, and CAM^{su} and F^{su}, two robustness results establishing that first-order functional programs written in F, and in F^{su}, define the same class of linear-time decidable problems as the higher-order functional programs run in CAM, and in CAM^{su}, respectively.

Keywords: Applicative Languages (ML); Interpreters; Models of Computation (Categorical Abstract Machine); Complexity Classes; Operational semantics

1. Introduction

First we indicate why we regard this study as being significant, before describing the approach adopted. Then we detail how this paper is structured.

1.1. Motivation

Usually complexity theory is presented on its own. However, there is a growing trend to formulate and present it from a *programming language perspective*. This entails viewing the programming language as its own computation model, as can be seen in the many recent books introducing it this way [20, 34, 18].

* Correspondence address: Ecole Normale Sup. de Lyon, Labo. d'Informatique et Parallélisme, 46, Allee d'Italie, 69364 Lyon Cedex 07, France. E-mail: eva.rose@ens-lyon.fr.

0167-6423/98/$19.00
PII: S0167-6423(97)00032-4

Recently, this trend has resulted in the definition of a computational complexity property called a "constant-factor time hierarchy" by Jones [16], based on the observation that many automatic program transformers yield only constant speed-up. By a "time hierarchy" we understand a partitioning of decision problems, where each partition contains problems which are solvable within a distinct time interval but not faster. When these time intervals can be factored from a specific time class, such as linear time or exponential time, we talk about a "factored" time hierarchy. A time hierarchy is "constant-factor" when moreover the factorization is given by multiplication with a global constant. We say that a computation model *has* a constant-factor time hierarchy, for some time class, when at least one decision problem belongs to each of these intervals.

This property may or may not exist for a computation model. It requires that *time* can be measured, i.e., that the computation model can be assigned a well-defined time-complexity measure. For programming languages without higher order features, an intuitive time-complexity measure, based on the notion of a "computation step" in the language, can easily be assigned. But for programming languages which include higher order features (such as closures implementing higher type procedures), only a rudimentary theory of their computational complexity currently exists. For languages based on ML [26, 36], which are excellent languages for stating algorithms, we typically have to resort to loose informal narratives when specifying performance (such as time-complexity requirements) of programs and, in particular, modules.

For general purpose, higher-order computation models like the untyped lambda calculus, it is not possible to assign any realistic measure of cost. This has recently been argued by Lawall and Mairson [22]. Hence, from a complexity viewpoint it makes no sense to discuss directly whether the lambda calculus has a constant-factor time hierarchy or not.

Taking a programming approach to complexity has introduced new concerns with respect to the computational strength of a programming language.[1] This is one point where the existence and elaboration of constant-factor time hierarchies is useful, e.g., it is a long-standing open problem whether languages, which include selective update facilities[2] (here the two su-languages), differ from languages without. However, in simple cases this has been proven to be true, as is explained in Section 8.2. Another issue which concerns computational strength at a more fine-grained level, is how the number of variables in the programming language may influence the existence of a constant-factor time hierarchy. As pointed out by Jones [16], it is essential for the proof of the linear time hierarchy for first-order models that the number of variables with constant access-time is bounded. When it comes to higher-order functional languages, what can we expect from adding an unlimited number of functions, or from permitting a branching construct with an unlimited number of branches? As discussed elsewhere by the author [31], the cost of adding different language constructs, however, essentially

[1] By strength we mean whether programs in one model run asymptotically faster than another.

[2] "Selective updating" refers to the fact that one can change, or "mutate", parts of a data structure after it has been initially stored in memory.

depends on which cost model, and which implementation model, we assume for the language.

The most important consequence of the existence of a constant-factor time-hierarchy for some computation model is that it invalidates the so-called "Constant" (or "Linear") Speed-up Theorem for that model. This Theorem, which was originally stated for Turing Machines [12, 13], says that it is possible to speed-up the computation time by an arbitrary constant (except that execution can never become better than $(1 + \varepsilon)n$, for any $\varepsilon > 0$, where n is the size of the input).

The theorem depends crucially on the fact that the cost-measure of Turing Machine theory does not account for the (counterintuitive!) ability to extend the machine alphabet during a computation. This is exploited to reduce the size of the tape arbitrarily by "compressing" several symbols into one, from a correspondingly larger alphabet. (Hühne [15] showed that this dependency is indeed critical, by proving that Constant Speed-up does not hold for a variant Turing Machine with "tree-like storage". This is because the symbols in a tree cannot be compressed in a way that reduces the depth of the tree). Unfortunately, this has lead to the widely accepted, theoretical viewpoint that linear time speed-up is a *trivial* matter.

In contrast to this, language implementors have always taken care that compilation or interpretation of programs do not add more than a constant overhead. For these practical disciplines, the *failure* of the Constant Speed-up Theorem for any realistic computation model is regarded as a kind of folklore. Clearly, it is of interest to formalize this folklore to gain insight into how theory and practice relate for the complexities of real programming languages.

In a paper from 1994, Jones [17] writes that "*imperative programs, first-order functional programs*, and *higher order functional programs* all define the same class of linear-time decidable problems". He proves the equivalence of the first two, using two small eager[3] representative languages, but not the last. Our goal is to advance this formalization of the folklore from the descriptions for first-order computational models of Hühne and Jones to include *higher order* constructions; indeed for programming languages having a practical use.

1.2. Approach

In order for a constant-factor time hierarchy to exist for some computation model, a proper cost measure has to be assigned. As argued above, there is no developed theory today which can provide a universal cost model for real, practical programming languages with their variety of language features. A first approach is to study their implementation model. For (eager) higher-order functional languages, the traditional solution is to translate into an "environment machine" [7, 21]. One very successful machine of this kind, and the first based on a formal idea, is the "Categorical Abstract

[3] Technically we understand eager, or strict, evaluation as applicative-order evaluation to weak-head normal form.

Machine" [6]; this being a canonical, abstract machine. Firstly, it has a well-defined semantics which is easily assigned realistic running times [11]. Secondly, most of the common language constructs that we find in high-level, higher-order languages can be found here; in particular closures, which are explicitly represented. Thirdly, not only is it close to many practical, higher-order functional languages, but it has also served as an intermediate language in early versions of CAML [36]. Finally, it is a combinator-based language, i.e., operates without variable and function names, thus side-stepping questions of varying access-times.

In recent years, linear-time complexity has gained much attention. First of all, many optimisation problems in real life are to be found here, e.g., compiler optimisation, or studying the effect of program transformations [30, 27]. This is understandable, as studying problems with a linear time-complexity provides the most fine-grained perspective on complexity, since linear complexity is sensitive to changes in the computation model. (In Section 8.2, we comment on the extent of robustness of linear-time complexity.) Hence, in this paper we focus on the complexity class LIN, i.e., problems decidable in linear time. In particular, we will use the term "linear-time hierarchy" instead of "constant-factor time hierarchy" when this property exists with respect to LIN.

Another issue is the widely believed conjecture that the presence of selective updating (hence cyclic graphs) makes the computational model stronger in an asymptotic sense. Recent results of Pippenger [28] support this conjecture. Hence we will treat the linear time hierarchy property separately, for languages with and without selective update facilities.

In order to obtain the same description form of the different semantical language descriptions, we present the languages in the style of natural (operational) semantics [19], instrumented with the assumed running times. Thus the semantics of programming languages is defined through judgements of the shape

$$\vdash program, input \xrightarrow{time} value$$

We label the turn-stile with the relevant language, e.g., $\vdash_L$, when needed to resolve any ambiguities.

1.3. Overview

We start, in Section 2, with an introduction to the concept of a hierarchy within linear-time decidable sets. In particular, the notation used throughout this paper is presented together with a restatement of the basic concepts upon which this work is based.

In Section 3 we proceed by presenting a simple, first-order functional language in two versions: one which allows selective updating (hence cyclic graphs), and one which does not (hence only tree-structures). Then, in Section 4, we present the Categorical Abstract Machine, again in two versions, and illustrate how it realises higher-order functional languages (in Section 5).

The main results are presented in Sections 6 (and 7): there exists a hierarchy within linear-time decidable sets defined by CAM (CAM^{su}) programs which defines the same

linear-time decidable sets, as defined by F (F^{su}) programs. Finally, we summarise the work in Section 8, giving references to related work as well as outlining possible future directions.

2. The linear time hierarchy concept

In this section we will take a programming language approach to complexity when defining the concepts on which we will base our proofs. In particular, we are concerned with the definition of a *linear time hierarchy* assuming a programming language as a basis of computation. Definitions 2–6 and 8 are adapted from Jones [16]. Further, we specify what to understand by a *representation function* of Rose [32] in order for an efficient interpretation to be well-defined.

In the context of this paper we will specify a programming language by its operational semantics in the style of natural semantics [19]. Assigning a complexity-measure may hence be obtained in a straightforward manner by instrumenting each inference rule by the assumed running time of the specified computation.

Definition 1 (*Instrumented operational semantics*). By an *operational semantics instrumented with running times* we understand an inference system defining judgements of the form

$$\vdash_{\mathrm{L}} p, d \xrightarrow{t} v$$

which reads "In the programming language L, the program $p \in$ L-programs with input $d \in$ L-data computes the value $v \in$ L-data in finite time t". When we are only interested in the running time and not the value, we will write $time_p^{\mathrm{L}}(d)$ for the t satisfying this judgement.

Decision problems [13, 34, 25] deal with whether an element belongs to a given set or not. Hence, a *problem* may be identified by a *set*. When we can identify the elements belonging to a given set, we say that the problem is solved and that the elements constitute the *solution*. Another way to express this is by a so-called characteristic function: given a set, the associated decision problem can be mathematically represented by a boolean function where the input values for which the function returns "true" are the solutions. This function can then be encoded as an *algorithm* which computes the function. (We will only be concerned about *decidable* problems which correspond to *total* functions and thus to *terminating* algorithms.) Taking a programming language approach to complexity implies further identifying such an algorithm by a *program* of some programming language, where the output values have an appropriate boolean interpretation.

Hence, a decision problem becomes a subset of the encoding programming language L's input domain, that is L-data, a view point which is formally expressed by the following definition:

Definition 2 (*Decision problem*). Given a programming language L where the value domain L-data contains a distinct value TRUE.[4] For an L-program p, $\text{ACC}^{\text{L}}(p)$ denotes the decision problem defined by

$$\text{ACC}^{\text{L}}(p) = \{d \in \text{L-data} \mid \exists t\colon \vdash_{\text{L}} p, d \xrightarrow{t} \text{TRUE}\}$$

Provided that a measure of time-complexity can be assigned to programs of some programming language, we can define what it means for a problem to be solvable within a time-bound given by a function:

Definition 3 (*Time-decidable problems*).

(i) A function f is *time-constructible* in L if there exists an L-program p satisfying
   (a) $\forall d \in \text{L-data}\colon time_p^{\text{L}}(d) \leqslant c \cdot f(|d|)$, and
   (b) $\forall n \in \mathbb{N}\ \exists d \in \text{L-data}\colon |d| = n \wedge time_p^{\text{L}}(d) = f(|d|)$.

(ii) The functional o is defined, for any $f\colon \mathbb{N} \to \mathbb{N}$ and $n \in \mathbb{N}$, by $o(f)(n) = \varepsilon(n) \cdot f(n)$ where $\varepsilon\colon \mathbb{N} \to \mathbb{N}$ is any function where $\lim_{n \to \infty} \varepsilon(n) = 0$.

(iii) Given a time-constructible function $f\colon \mathbb{N} \to \mathbb{N}$. $\text{TIME}^{\text{L}}(f)$ denotes the class of problems decidable within the time $f(|d|)$, formally

$$\text{TIME}^{\text{L}}(f) = \{\text{ACC}^{\text{L}}(p) \mid \forall d\colon \vdash_{\text{L}} p, d \xrightarrow{t} v \text{ such that } t \leqslant (f + o(f))(|d|)\}$$

where $|d|$ is the size of the input d in each case.

Remark 4. The function $f + o(f)$ signifies that the running time t must not exceed the time-bounding function f for that class by more than $o(f)$ on any input size $n \in \mathbb{N}$. To illustrate the meaning of this, assume that a class of problems are time-bounded by the time-constructible function f. For any input size $n \in \mathbb{N}$, we will naturally like to consider times which only exceed $f(n)$ by some constant c. By Definition 3 we have

$$\begin{aligned} & time_p^{\text{L}}(d) \leqslant (f + o(f))(n) \\ & \quad \Leftrightarrow\ c + f(n) \leqslant (1 + \varepsilon(n)) \cdot f(n) \\ & \quad \Leftrightarrow\ c / f(n) \leqslant \varepsilon(n) \end{aligned}$$

Hence adding constants to running times is unproblematic.

By applying Definition 3 to a linear function in particular, we obtain the class of problems decidable within linear time.

Definition 5 (*Linear-time decidable problems*). For $a \in \mathbb{N}$, $\ell_a\colon \mathbb{N} \to \mathbb{N}$ is the *linear function* given by $\ell_a(n) = a \cdot n$ for any $n \in \mathbb{N}$. The class of *linear-time decidable problems*, denoted by $\text{LIN}^{\text{L}}(a)$, is defined by

$$\text{LIN}^{\text{L}}(a) = \text{TIME}^{\text{L}}(\ell_a)$$

[4] It suffices that there is an interpretation of L-data values as booleans.

Assuming some programming language L as a basis of computation to which a time-complexity measure is associated, this concept can formally be defined from a programming perspective as:

Definition 6 (*Linear-time hierarchy*). The class of problems decidable in linear time forms a hierarchy iff

$$\exists b \geqslant 1 \ \forall a \geqslant 1: \quad \text{LIN}^{\text{L}}(a) \subset \text{LIN}^{\text{L}}(a \cdot b)$$

Notice how the constant factor b is significant for any concrete hierarchy, that is for the underlying programming language L. Actually, a b has been experimentally determined (as the number 249) for a first-order imperative language by Hessellund and Dahl [8].

We need a notion of *representation* to be able to relate programs and data terms of different languages. In particular we want to be able to avoid that (p,d) is represented as p paired with the result of running p on d, since the "complexity", i.e., the time it takes to run the translated term, in the latter representation "collapses" trivially. Hence we will require representations to be *compositionally* defined over the syntax. This is ensured by requiring that representations are specified as systems of recursive equations [23] that are easily verified to be compositional. This is not quite enough, however, because a compositional definition can still copy subcomponents, upsetting the size of the represented program or value. In order to avoid this, we impose a limit to the number of equation unfoldings to the size of the term.

Definition 7 (*Representation*). A map from one set of terms T_1 to another T_2, $\overline{\cdot}: T_1 \to T_2$, is a *representation of* T_1 *as* T_2 if it is defined as a system of recursive equations which is compositional over the syntactic structure of T_1 such that the number of equation unfoldings is bounded by the size of the term.

We can now define the notion of an efficient interpretation cf. [16], adapted to our more general notion of representation.

Definition 8 (*Efficient interpretation*). Given two programming languages L and M, each with an instrumented operational semantics, and a representation $\overline{\cdot}$ of L-programs and L-data as M-data. Furthermore, assume that M has a pairing operation $(\cdot,\cdot)$ which uses constant time to pair two M-data values. An M-program m is an *efficient interpreter of* L *written in* M if there exists a global constant $e \geqslant 1$ such that for all L-programs p and L-data d,

(i) $\vdash_{\text{L}} p,d \xrightarrow{t_p} v$ iff $\vdash_M m,(\overline{p},\overline{d}) \xrightarrow{t_m} \overline{v}$, and

(ii) $time_m^M(d) \leqslant e \cdot time_p^{\text{L}}(d)$.

We then say that there is an *efficient interpretation* $M \preccurlyeq \text{L}$.

In other words, an interpreter is efficient if the interpretation overhead is bounded by a positive number which is *globally independent* of the interpreted program and its input.

Provided that L-data and M-data are defined over the same domain,[5] and time_p^{L} is bounded by some linear ℓ_a, with $a \geqslant 1$, then Definition 8 can be formulated as

$$\exists e, a \geqslant 1\colon \text{LIN}^{\text{L}}(a) \subseteq \text{LIN}^{M}(e \cdot a)$$

Notice the difference between Definition 6 and this: The first defines the concept of a linear time hierarchy based on the *same* language. The second defines a relation based on two (possibly) *different* languages. However, if a linear time hierarchy is known for one of these languages, say L, then two mutually efficient interpretations, say $M \preccurlyeq \text{L}$ and $M \succcurlyeq \text{L}$, can be used to construct a linear time hierarchy for M with the factor $b = e \cdot b' \cdot e'$ obtained by the chain of inclusions

$$\text{LIN}^{M}(a) \subseteq \text{LIN}^{\text{L}}(a \cdot e) \subset \text{LIN}^{\text{L}}(a \cdot e \cdot b') \subseteq \text{LIN}^{M}(a \cdot e \cdot b' \cdot e')$$

This is the technique we will use to establish a linear time hierarchy for the Categorical Abstract Machine in Section 6 (and 7).

It is note-worthy that this not only clarifies the existence of a constant-factor time-hierarchy for some time-class: If it exists, the proof method establishes that the two programming languages, related through mutual efficient interpretation, provide a basis for computation which defines the *same* problem-solving hierarchy-structure for that time-class. Another way to express this is to say that switching between such closely related languages is *robust* with respect to that time-class.

3. The languages F and F^{su}

In this section we formally define the two small functional subsets of Lisp [24], shown to have linear time hierarchies by Jones [16], on which we base our development.

The languages are very restricted in that they allow only one first-order recursive function (named `f`) to be defined, and only one variable name (`x`), which is used to denote both the input to the program and the formal parameter of the function, thus the program input data is hidden to the body of the function. However, mutually recursive functions as well as multiple variables can be simulated easily – both languages are Turing complete. The languages are eager and have running times based on standard Scheme[6] implementation technology [5] (in fact they can be implemented on a unit-cost RAM in times proportional to those given here). Basically, they differ in the data values on which they operate: F manipulates *tree-structured data*, i.e., finite, directed trees, with "*NIL*" for leaves, and whose internal nodes, the "*CONS*-cells", each have out-degree two. F^{su}, however, manipulates *graph-structured data* by allowing *selective updating* as in Scheme. More precisely, graph-structured data is defined as finite and directed graphs in the sense of Barendregt et al. [3] with leaves labelled "*NIL*", and

[5] Actually, a structure-preserving isomorphism between them is sufficient.

[6] Like traditional Lisp implementations [24] but with `hd 'nil = tl 'nil = NIL`.

where the internal nodes, labelled "*CONS*", have out-degree two; further, each node is identified by a unique number (its location). In the following, graph-structured data are called "boxes", and each node-identifier, a "location". We notice that the definition allows cyclic paths in the graph.[7]

After defining each language in a separate section, we state the constant-factor time hierarchy property for the languages.

3.1. Semantics of F

Definition 9 (F).
Syntax.

$$P \in \text{Program} ::= E \texttt{ whererec f(x) = } E'$$
$$E \in \text{Expression} ::= \texttt{x} \mid \texttt{'nil} \mid \texttt{hd } E \mid \texttt{tl } E \mid \texttt{cons}(E', E'')$$
$$\mid \texttt{if } E \texttt{ then } E' \texttt{ else } E'' \mid \texttt{f}(E)$$

Semantic sorts.

$$d, v \in \text{Value} ::= NIL \mid CONS\ (v_1, v_2)$$

Semantic rules.

$\vdash P, d \xrightarrow{t} v$: The program P, given input d, evaluates to the output v with a time cost of t.

$d, E' \vdash E \xrightarrow{t} v$: The expression E evaluates to the value v with a time cost t where the variable x is bound to the data structure d, and the function f has body E'.

The rules are shown in Fig. 1; we use _ for unique variables that are otherwise ignored.

In this description, we exploit the fact that in F there are always exactly the two bindings of the symbols x and f in the "environment", which we have therefore marked implicitly: Instead of $[\texttt{x} \mapsto d; \texttt{f} \mapsto E']$ we simply write d, E'.

3.2. Semantics of F^{su}

F^{su} is a store-based version of F extended with `setcar!` and `setcdr!`, with the same meaning and running times as in Scheme,[8] defined following Plotkin [29]. This means that the variable binding description becomes a two-level description, introducing boxes as the intermediate step. Hence, the variable binding $\texttt{x} \mapsto v$ becomes $\texttt{x} \mapsto l \overset{\sigma}{\mapsto} v$ in

[7] This is, in fact, the only difference in implementations: while it is not semantically observable, tree-structured data is invariably implemented as *directed acyclic graphs*. This becomes clear when one observes that the time it takes to construct $CONS(t, t)$ is constant rather than proportional to the size of t as one should expect if a true tree structure was built.

[8] The same meaning as rplaca and rplacd in traditional Lisp [24].

$$\frac{d,E' \vdash E \xrightarrow{t} v}{\vdash E \texttt{ whererec f(x) = } E', d \xrightarrow{t+1} v} \quad \text{(F1)}$$

$$\frac{}{d,E' \vdash \texttt{x} \xrightarrow{1} d} \quad \text{(F2)}$$

$$\frac{d,E' \vdash E_1 \xrightarrow{t_1} v_1 \quad d,E' \vdash E_2 \xrightarrow{t_2} v_2}{d,E' \vdash \texttt{cons}(E_1,E_2) \xrightarrow{t_1+t_2+1} CONS\ (v_1,v_2)} \quad \text{(F3)}$$

$$\frac{}{d,E' \vdash \texttt{'nil} \xrightarrow{1} NIL} \quad \text{(F4)}$$

$$\frac{d,E' \vdash E \xrightarrow{t} CONS\ (v_1,\_)}{d,E' \vdash \texttt{hd}\,E \xrightarrow{t+1} v_1} \quad \text{(F5)}$$

$$\frac{d,E' \vdash E \xrightarrow{t} NIL}{d,E' \vdash \texttt{hd}\,E \xrightarrow{t+1} NIL} \quad \text{(F6)}$$

$$\frac{d,E' \vdash E \xrightarrow{t} CONS\ (\_,v_2)}{d,E' \vdash \texttt{tl}\,E \xrightarrow{t+1} v_2} \quad \text{(F7)}$$

$$\frac{d,E' \vdash E \xrightarrow{t} NIL}{d,E' \vdash \texttt{tl}\,E \xrightarrow{t+1} NIL} \quad \text{(F8)}$$

$$\frac{d,E' \vdash E \xrightarrow{t_1} CONS\ (\_,\_) \quad d,E' \vdash E_1 \xrightarrow{t_2} v_1}{d,E' \vdash \texttt{if}\,E\,\texttt{then}\,E_1\,\texttt{else}\,E_2 \xrightarrow{t_1+t_2+1} v_1} \quad \text{(F9)}$$

$$\frac{d,E' \vdash E \xrightarrow{t_1} NIL \quad d,E' \vdash E_2 \xrightarrow{t_2} v_2}{d,E' \vdash \texttt{if}\,E\,\texttt{then}\,E_1\,\texttt{else}\,E_2 \xrightarrow{t_1+t_2+1} v_2} \quad \text{(F10)}$$

$$\frac{d,E' \vdash E \xrightarrow{t_1} d' \quad d',E' \vdash E' \xrightarrow{t_2} v}{d,E' \vdash \texttt{f}(E) \xrightarrow{t_1+t_2+1} v} \quad \text{(F11)}$$

Fig. 1. F semantics and running times.

F^{su}, where l is a *location*[9] and σ is a *store*, mapping locations to boxes (where a location identifies the root of its box in that store). We introduce a special notation, a partial function, $\sigma@l$, to denote the tree-structured value obtained by unravelling the box $\sigma(l)$ from its root l in the store σ; it is only defined when no cyclic paths are reachable from l.

Definition 10 (F^{su}).
Syntax same as F but extended with

$$E \in \text{Expression} ::= \ldots \mid \texttt{setcar!}\ E\ E' \mid \texttt{setcdr!}\ E\ E'$$

Semantic sorts same as F but extended with

$$\sigma \in \text{Store} = \text{Location} \to \text{Box}$$
$$l \in \text{Location} = \text{Nat}$$
$$\text{Box} ::= NIL \mid CONS\ (l_1, l_2)$$

Semantic functions.

$$\cdot @ \cdot : \text{Store} \times \text{Location} \to \text{Value}_\perp \qquad \textit{Extract value (partial function)}$$

[9] Following Plotkin, a location is an "address" which is independent of any concrete machine technology.

$$\frac{\sigma_0, l, E' \vdash E \xrightarrow{t} \sigma', l'}{E \texttt{ whererec f(x) = } E', d \xrightarrow{t+1} \sigma'@\, l' \quad \textit{where } \sigma_0@\, l = d\ ,\ \sigma_0@\, l_{\text{nil}} = NIL} \qquad (\text{F}^{\text{su}}1)$$

$$\frac{\sigma, l, E' \vdash E_1 \xrightarrow{t_1} \sigma_1, l_1 \quad \sigma_1, l, E' \vdash E_2 \xrightarrow{t_2} \sigma_2, l_2}{\sigma, l, E' \vdash \texttt{cons}(E_1, E_2) \xrightarrow{t_1+t_2+1} \sigma_2[\, l_{\text{fresh}} \mapsto CONS\ (l_1, l_2)\,],\ l_{\text{fresh}} \quad \textit{where } l_{\text{fresh}} \notin \text{Dom}(\sigma_2)} \qquad (\text{F}^{\text{su}}3)$$

$$\frac{\sigma, l, E' \vdash E_1 \xrightarrow{t_1} \sigma_1, l_1 \quad \sigma_1, l, E' \vdash E_2 \xrightarrow{t_2} \sigma_2, l_2}{\sigma, l, E' \vdash \texttt{setcar!}E_1\ E_2 \xrightarrow{t_1+t_2+1} \sigma_2[\, l_1 \mapsto CONS\ (l_2, l_1'')\,],\ l_1 \quad \textit{where } \sigma_2(l_1) = CONS\ (l_1', l_1'')} \qquad (\text{F}^{\text{su}}12)$$

$$\frac{\sigma, l, E' \vdash E_1 \xrightarrow{t_1} \sigma_1, l_1 \quad \sigma_1, l, E' \vdash E_2 \xrightarrow{t_2} \sigma_2, l_2}{\sigma, l, E' \vdash \texttt{setcdr!}E_1\ E_2 \xrightarrow{t_1+t_2+1} \sigma_2[\, l_1 \mapsto CONS\ (l_1', l_2)\,],\ l_1 \quad \textit{where } \sigma_2(l_1) = CONS\ (l_1', l_1'')} \qquad (\text{F}^{\text{su}}13)$$

Fig. 2. F^{su} semantics and running times.

The value "extracted" from a store is obtained by traversing the graph; if the graph is infinite then $\perp$ is extracted.

Semantic rules as F modified to use a store:

$\sigma, l, E' \vdash E \xrightarrow{t} \sigma', l'$: In the store σ, with x bound to location l and f is bound to E', the expression E evaluates to location l' in the store σ'.

Rules in Fig. 2.

The constant location, $[l_{\text{nil}} \mapsto NIL]$, is invariantly part of any store since it is part of the initial store (and because it cannot be updated as they are not *CONS* nodes).

The only place where the store is updated is in the cons-rule ($\text{F}^{\text{su}}3$), where a new memory location, l_{fresh}, is allocated in constant time, and in the `setcar!`, `setcdr!` rules ($\text{F}^{\text{su}}12$ and 13), where cyclic structures might be introduced. Hence only these rules have been explicitly formalised in Fig. 2. Notice the two different uses of updating. In rule ($\text{F}^{\text{su}}3$) l_{fresh} is not a location in σ and thus the store is extended. In rules ($\text{F}^{\text{su}}12$ and 13) l is already a location in σ and thus we are effectively overwriting the box $\sigma(l)$.

As should be expected, F^{su} behaves like F when no selective updating is invoked; the input and output data domains are the same, so cyclic data cannot be output.

Proposition 11. *Given an F-program p.* $\vdash_{\text{F}} P, d \xrightarrow{t_{\text{F}}} v$ *iff* $\vdash_{\text{F}^{\text{su}}} P, d \xrightarrow{t_{\text{F}^{\text{su}}}} v$.

Proof. The internal parts of F- and F^{su}-evaluation without selective updating are completely equivalent. This is trivially true except for rules ($\text{F}/\text{F}^{\text{su}}1$), which requires understanding that $\sigma@l = d$ implies $\sigma'@l' = v$, and rules ($\text{F}/\text{F}^{\text{su}}3$), where $\sigma@l_1 = v_1$ and $\sigma@l_2 = v_2$ implies that

$$\sigma_2[l_{\text{fresh}} \mapsto CONS\ (l_1, l_2)]@l_{\text{fresh}} = CONS\ (l_1, l_2) = v$$

and that if either proof is infinite so is the other. □

3.3. *F and* F^{su} *have linear time hierarchies*

Finally we quote from Jones [16]:

Theorem 12. *The constant-hierarchy theorem,*[10] [...] *holds for F as well as for* F^{su}.

4. The categorical abstract machine (CAM)

Our target machine is the environment-based, categorical abstract machine CAM, developed on a categorical foundation by Cousineau et al. [6]. Its instructions form a fixed set of (categorical) combinators, constructed to be faithful to β-reduction in the λ-calculus, and acting on a graph-environment (stack). It is the binding-height which defines a variable binding – since no variables are explicit in the model. As described in [16], it is essential for program independent interpretation that the number of variable names is bounded. This is why we consider a model like CAM (and the reason for which we cannot consider higher-order functional languages in general).

The CAM implements a call-by-value evaluation strategy, and is suitable for implementing ML, an eager, higher-order functional language [6, 36]. Originally there are two versions of CAM: one where recursion and branching are implicitly represented [6, Table 1], hence operating on tree-structured values, and one where general recursion and branching facilities have been made explicit [6, Table 6], i.e., working on graph-structured values. We use this classification for our CAM versions: plain *CAM* for the first case, CAM^{su} for the latter. However, we present the languages using natural semantics following Kahn [19]. Actually, CAM^{su} has been slightly extended: the original *wind*-instruction is replaced by the identically defined *rplacd*,[11] and we add its symmetrical instruction, *rplaca*, which has no counterpart in CAM originally; this is of no complexity-consequence since one can simulate the other in constant time.

To ease the proof developments, we omit integers and integer operations since they can be encoded in F (F^{su}) and in CAM (CAM^{su}) in the same way (with respect to complexity), e.g., as Church numerals or using Peano arithmetic (the example in the next section illustrates this).

We present the rules of CAM in Fig. 3, and CAM^{su} in Fig. 4, instrumented with realistic execution times. These are based on an analysis of CAM by Hannan [11].

4.1. *Semantics of CAM*

Definition 13 (*CAM*).
Syntax.

$$\text{P} \in \text{Program} ::= \textit{program}(\text{Cs})$$
$$\text{Cs} \in \text{Commands} ::= \varnothing \mid \text{C}; \text{Cs}$$

[10] The "constant-hierarchy theorem" simply states the existence of a constant-factor time-hierarchy for LIN, i.e., a linear time hierarchy within the terminology of this paper.

[11] Kahn's recursion operator *rec* [19], is essentially defined in terms of *rplacd*.

$$\frac{() \cdot \alpha \vdash \mathrm{Cs} \xrightarrow{t} s \cdot \beta}{\vdash \mathit{program}(\mathrm{Cs}), \alpha \xrightarrow{t+1} \beta} \quad \text{(CAM1)}$$

$$\frac{}{s \cdot \alpha \vdash \emptyset \xrightarrow{1} s \cdot \alpha} \quad \text{(CAM2)}$$

$$\frac{s \cdot \alpha \vdash \mathrm{C} \xrightarrow{t} s_1 \cdot \beta \qquad s_1 \cdot \beta \vdash \mathrm{Cs} \xrightarrow{t'} s_2 \cdot \gamma}{s \cdot \alpha \vdash \mathrm{C}; \mathrm{Cs} \xrightarrow{t+t'+1} s_2 \cdot \gamma} \quad \text{(CAM3)}$$

$$\frac{}{s \cdot \beta \vdash \mathit{quote}(\alpha) \xrightarrow{1} s \cdot \alpha} \quad \text{(CAM4)}$$

$$\frac{}{s \cdot (\alpha, \beta) \vdash \mathit{car} \xrightarrow{1} s \cdot \alpha} \quad \text{(CAM5)}$$

$$\frac{}{s \cdot (\alpha, \beta) \vdash \mathit{cdr} \xrightarrow{1} s \cdot \beta} \quad \text{(CAM6)}$$

$$\frac{}{s \cdot \alpha \cdot \beta \vdash \mathit{cons} \xrightarrow{1} s \cdot (\alpha, \beta)} \quad \text{(CAM7)}$$

$$\frac{}{s \cdot \alpha \vdash \mathit{push} \xrightarrow{1} s \cdot \alpha \cdot \alpha} \quad \text{(CAM8)}$$

$$\frac{}{s \cdot \alpha \cdot \beta \vdash \mathit{swap} \xrightarrow{1} s \cdot \beta \cdot \alpha} \quad \text{(CAM9)}$$

$$\frac{}{s \cdot \rho \vdash \mathit{cur}(\mathrm{Cs}) \xrightarrow{1} s \cdot [\mathrm{Cs}, \rho]} \quad \text{(CAM10)}$$

$$\frac{s \cdot (\rho, \alpha) \vdash \mathrm{Cs} \xrightarrow{t} s_1 \cdot \beta}{s \cdot ([\mathrm{Cs}, \rho], \alpha) \vdash \mathit{app} \xrightarrow{t+1} s_1 \cdot \beta} \quad \text{(CAM11)}$$

Fig. 3. CAM semantics and running times.

$$\frac{\sigma_0, () \cdot l \vdash \mathrm{Cs} \xrightarrow{t} \sigma', s \cdot l'}{\vdash \mathit{program}(\mathrm{Cs}), \alpha \xrightarrow{t} \beta} \quad \sigma_0 @ l_{()} = (),\ \sigma_0 @ l = \alpha,\ \sigma' @ l' = \beta \qquad (\mathrm{CAM}^{\mathrm{su}}1)$$

$$\frac{}{\sigma, s \cdot l \vdash \mathit{quote}(\alpha) \xrightarrow{|\alpha|} \sigma_1, s \cdot l_1} \quad \sigma \subseteq \sigma_1,\ (\sigma_1 \backslash \sigma) @ l_1 = \alpha \qquad (\mathrm{CAM}^{\mathrm{su}}4)$$

$$\frac{}{\sigma, s \cdot l_1 \cdot l_2 \vdash \mathit{cons} \xrightarrow{1} \sigma[\, l_{\mathrm{fresh}} \mapsto (l_1, l_2)\,],\ s \cdot l_{\mathrm{fresh}}} \quad l_{\mathrm{fresh}} \notin \mathrm{Dom}(\sigma) \qquad (\mathrm{CAM}^{\mathrm{su}}7)$$

$$\frac{}{\sigma, s \cdot l \vdash \mathit{cur}(\mathrm{Cs}) \xrightarrow{1} \sigma[\, l_{\mathrm{fresh}} \mapsto [\mathrm{Cs}, l]\,], s \cdot l_{\mathrm{fresh}}} \quad l_{\mathrm{fresh}} \notin \mathrm{Dom}(\sigma) \qquad (\mathrm{CAM}^{\mathrm{su}}10)$$

$$\frac{}{\sigma, s \cdot l \cdot l_2 \vdash \mathit{rplaca} \xrightarrow{1} \sigma[\, l \mapsto \mathit{CONS}(l_2, l'')\,], s \cdot l} \quad \sigma(l) = \mathit{CONS}(l', l'') \qquad (\mathrm{CAM}^{\mathrm{su}}12)$$

$$\frac{}{\sigma, s \cdot l \cdot l_2 \vdash \mathit{rplacd} \xrightarrow{1} \sigma[\, l \mapsto \mathit{CONS}(l', l_2)\,], s \cdot l} \quad \sigma(l) = \mathit{CONS}(l', l'') \qquad (\mathrm{CAM}^{\mathrm{su}}14)$$

Fig. 4. $\mathrm{CAM}^{\mathrm{su}}$ semantics and running times.

$$\mathrm{C} \in \mathrm{Command} ::= \mathit{quote}(\alpha) \mid \mathit{car} \mid \mathit{cdr} \mid \mathit{cons} \mid \mathit{push} \mid \mathit{swap} \mid \mathit{cur}(\mathrm{Cs}) \mid \mathit{app}$$

Semantic sorts

$$s \in \mathrm{Stack} ::= s \cdot \alpha \mid ()$$

$$\alpha, \beta, \rho \in \mathrm{Value} ::= (\alpha, \beta) \mid [\mathrm{Cs}, \alpha] \mid ()$$

( ) is both the empty stack and the only atomic value.

Semantic rules

$\vdash$ *program*(Cs), $\alpha \xrightarrow{t} \beta$: The program *program*(Cs) with input α evaluates to the output β with a time cost of t

$s \vdash \text{Cs} \xrightarrow{t} s'$: Commands Cs transforms the stack s into the output stack s' with a time cost of t.

The rules are shown in Fig. 3.

4.2. Semantics of CAM^{su}

In Definition 14, we present the CAM^{su} as a store-semantic version of CAM following Plotkin [29] as for F^{su} in Section 3.

Definition 14 (CAM^{su}).

Syntax same as CAM but extended with

$$\text{C} \in \text{Command} \ ::= \ \cdots \mid \mathit{rplaca} \mid \mathit{rplacd}$$

Semantic sorts same as CAM but extended with

$$\sigma \in \text{Store} = \text{Location} \to \text{Box}$$
$$l \in \text{Location} = \text{Nat}$$
$$s \in \text{Stack} ::= s \cdot l \mid ( )$$
$$\text{Box} ::= (l_1, l_2) \mid [\text{Cs}, l] \mid ( )$$

Semantic functions

$\cdot @ \cdot$: Store $\times$ Location $\to$ Value$_\perp$	Extract value
$\cdot \setminus \cdot$: Store $\times$ Store $\to$ Store	Store difference

Extraction is as for F^{su}. The store "difference" is the first store restricted to the locations not in the second store.

Semantic rules as CAM modified to use a store:

$\sigma, s \vdash \text{Cs} \xrightarrow{t} \sigma', s'$: Commands Cs transforms the stack s with store σ to the stack s' with store σ' using a time cost of t. The rules are given in Fig. 4; in analogy with F^{su}, we only list those rules which have an effect on the store.

$l_{( )}$ is a constant location: $[l_{( )} \mapsto ( )]$ is invariantly part of any store since it is part of the initial store σ_0 (and cannot be updated since ( ) is not a pair node).

As for F^{su}, updating is used in two ways. In rules (CAM^{su}7 and 10) l_{fresh} is not a location in σ and thus the store is extended with new box. In rules (CAM^{su}12 and 13) l is already a location in σ and thus we are effectively overwriting the box $\sigma(l)$.

The *quote* rule (CAM^{su}4) deserves special mention. Its purpose is to add a value to the store. In CAM without selective updating, this can be done in time 1 because constant values remain constant. However, in CAM^{su} a *quote*(α) command takes time

$|\alpha|$ since the model must allocate a fresh copy each time (this is represented by the requirement that $(\sigma_1 \backslash \sigma)@ l_1 = \alpha$) to allow selective updating of this copy without destroying any data (this is represented by $\sigma \subseteq \sigma_1$). Notice, however, that we do not charge time for loading the initial data d into memory as this can only be done once and hence there is no need to be careful that repeated executions yield the same result. Again the input and output data domains are the same in CAM and CAM^{su}, so cyclic data cannot be output.

Proposition 15. *For any CAM-program* P, $\vdash_{\text{CAM}} \text{P}, \alpha \overset{t_{\text{CAM}}}{\rightarrow} \beta$ *iff* $\vdash_{\text{CAM}^{su}} \text{P}, \alpha \overset{t_{\text{FSU}}}{\rightarrow} \beta$.

Proof. As for Proposition 11, only tree-like data is created and thus it is easy to see that what is extracted with @ from the graph case is correct. □

5. CAM is almost a higher-order functional language!

In this section we demonstrate the proximity of CAM to a higher-order functional programming language by showing how an example program with extensive use of higher-order functions can be translated almost directly into CAM. The example is the encoding of the *Church numerals* in the λ-calculus [2, Definition 6.4.4]. The idea is that the two basic constructors for "zero" and "successor" are encoded as functionals (actually *combinators*, since they do not depend on the environment):

$$Z\,f\,x = x \qquad S\,n\,f\,x = f(n\,f\,x)$$

or, in λ-notation, $Z = \lambda f x.x$ and $S = \lambda n f x.f(n f x)$. The number 1, for example, is represented as $S(Z)$ which reduces by substitution (β-reduction) as follows in λ-notation:

$$S(Z) \rightarrow \lambda f x.f(Z f x) \rightarrow \lambda f x.f x$$

In fact the natural number n is represented by the λ-term $\lambda f x.f^n(x)$ (of type $(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$).

In order to read a number out of CAM we should convert it to a data value; this is easily done with the two additional functions

$$D\,n = n\,C(\,) \quad \text{where } C\,n = (n(\,))$$

Now, assume we wish to compute the value $D(SZ)$ using CAM. The first thing that a compiler will do when translating is to build an *environment*, and we will do the same and decide that the global environment should contain Z, S, D, C, in that order. If we use numeric indices instead of names for these, and index from 0, just like on a stack, then we can write the main program as 2(1 0). If furthermore we add the convention that function parameters are "pushed" onto the environment, then we can write the four functions as follows (using leading λs to indicate how many arguments they take):

$$Z \equiv \lambda\lambda 0 \qquad S \equiv \lambda\lambda\lambda 1(2\,1\,0) \qquad D \equiv \lambda 0\,4(\,) \qquad C \equiv \lambda(0,(\,))$$

Notice how the index for C is 4 inside D: 3 (the depth of C in the global environment) plus 1 (the number of parameters to D).

What we have invented here is, in fact, well known as de Bruijn's indexing technique for the λ-calculus [9]. The interesting observation is that now we can translate the higher-order functional program into CAM in a one-to-one style, i.e., we can also translate back, if we wish. This is why we claim CAM *is* a higher-order functional language. Here is the CAM code for the CAM-program P computing $D(SZ)$ (including the used abbreviations; furthermore C^n means that C is used n times):

$$\begin{aligned}
\mathrm{P} \equiv\ & \mathit{program}(N; !C; !D; !S; !Z; \\
& \mathit{push}^2; ?2; \mathit{swap}; \mathit{push}; ?1; \mathit{swap}; ?0; \mathit{cons}; \mathit{app}; \mathit{cons}; \mathit{app})
\end{aligned}$$

where

$$\begin{aligned}
N &\equiv \mathit{quote}(( )) \\
!\mathrm{Cs} &\equiv \mathit{push}; \mathrm{Cs}; \mathit{swap}; \mathit{cons} \\
?n &\equiv \mathit{car}^n; \mathit{cdr} \\
C &\equiv \mathit{cur}(?0; \mathit{push}; N; \mathit{cons}) \\
D &\equiv \mathit{cur}(\mathit{push}; ?0; \mathit{swap}; ?4; \mathit{cons}; \mathit{app}; \mathit{push}; N; \mathit{cons}; \mathit{app}) \\
Z &\equiv \mathit{cur}^2(?0) \\
S &\equiv \mathit{cur}^3(\mathit{push}^2; ?1; \mathit{swap}; \mathit{push}^2; ?2; \mathit{swap}; ?1; \mathit{cons}; \mathit{app}; \mathit{swap}; \\
&\qquad ?0; \mathit{cons}; \mathit{app}; \mathit{swap}; \mathit{cons}; \mathit{app})
\end{aligned}$$

Running this program will build the CAM value $((),())$ as follows: The first line of instructions in P will build an "environment" value $(((((),C,),D),S),Z)$ in which we can later lookup these using their de Bruijn indices (in fact, the environment overwrites the input data which is not used). The second line pushes D, builds and computes SZ, and finally applies D to the result.

Clearly, this code looks very much like the de Bruijn style functional program above. In fact, the only slightly non-trivial matter is to insert the right $\mathit{push}; \ldots; \mathit{swap}$ pairs to make sure that a copy of the environment is always available at the top of the stack when it is needed. Hence we can conclude that CAM contains (as a subset) a higher-order functional programming language except for the use of *car*/*cdr*-sequences instead of variable names.

6. A linear time hierarchy for CAM

This section proves our main result for CAM.

Theorem 16. *There exists a linear-time hierarchy for CAM.*

```
i ≡ tl(f(('nil.tl x).hd x)) whererec f(x) = Loop
where Loop ≡
 LET  stack.(instruction.arg) = x  IN
  CASE  instruction  OF
   'empseq -> stack
   'seq ->    LET c1.c2 = arg          IN f(f(stack.c1).c2)
   'quote ->  LET rest._ = stack       IN rest.arg
   'car ->    LET rest.(a._) = stack   IN rest.a
   'cdr ->    LET rest.(_.b) = stack   IN rest.b
   'cons ->   LET (rest.a).b = stack   IN rest.(a.b)
   'push ->   LET rest.a  = stack      IN (rest.a).a
   'swap ->   LET (rest.a).b = stack   IN (rest.b).a
   'cur ->    LET rest.rho = stack     IN rest.(arg.rho)
   'app ->    LET rest.((code.rho).a) = stack
                                       IN f((rest.(rho.a)).code)
```

Fig. 5. Interpreter in F for CAM-programs.

Proof. We establish that there exists a b such that for all $a \geqslant 1$ we have $\text{LIN}^{\text{CAM}}(a) \subset \text{LIN}^{\text{CAM}}(a \cdot b)$, namely $b = e \cdot b' \cdot e'$ obtained by the following inclusions:

$$\begin{aligned} \text{LIN}^{\text{CAM}}(a) &\subseteq \text{LIN}^{\text{F}}(a \cdot e) && \text{by Lemma 20} \\ &\subset \text{LIN}^{\text{F}}(a \cdot e \cdot b') && \text{by Theorem 12} \\ &\subseteq \text{LIN}^{\text{CAM}}(a \cdot e \cdot b' \cdot e') && \text{by Lemma 17} \quad \square \end{aligned}$$

The rest of this section is devoted to state and prove Lemmas 17 and 20 in some detail (the full proof is in the technical report [33]). The major part of the effort is in the construction of efficient interpretors witnessing $\text{F} \geqslant \text{CAM}$ and $\text{CAM} \geqslant \text{F}$.

6.1. *An efficient interpreter in F for CAM*

Lemma 17. *There is an efficient interpretation* $\text{F} \geqslant \text{CAM}$.

The proof, presented below, is essentially a correctness proof for the interpreter for CAM written in F, $i_{\text{F}}^{\text{CAM}}$, shown in Fig. 5, including a careful analysis of the running time of interpretation. Before indulging in the proof we will explain the interpreter notation and how interpretation proceeds.

Definition 18. The interpreter in Fig. 5 makes use of some abbreviation *macros*:
Infix cons. We write $E_1.E_2$ instead of $\texttt{cons}(E_1, E_2)$.
Simple definitions. We permit the form

LET pattern = E_1 IN E_2

where pattern may only contain cons and new variable names. It is a shorthand defining each of the names in the pattern as a macro. The value is obtained by reducing E_2 after expanding these names inside E_2 to an expression which applies a sequence of

hd and tl to E_1. Thus effectively it defines a pseudo-F-rule

$$\frac{d,E' \vdash_F E \xrightarrow{t} \text{PATTERN}}{d,E' \vdash_F n \xrightarrow{t+D} v} \quad (FLET)$$

where n can be any name in the pattern and PATTERN denotes pattern with . interpreted as infix *CONS*; D is the number of hd and tl needed to get to the occurrence of the variable in the pattern. We use _ for names that are not used. For example, LOOP above has the form

```
LET stack.(instruction.arg)=x IN ...
```

which means that within ... three new macros are defined: stack≡hd x, instruction≡hd tl x, and arg≡tl tl x, and the pseudo-rules

$$\frac{d,E' \vdash_F x \xrightarrow{1} CONS(v,\_)}{d,E' \vdash_F \texttt{stack} \xrightarrow{1+1} v} \quad (FLET)$$

$$\frac{d,E' \vdash_F x \xrightarrow{1} CONS(\_,CONS(v,\_))}{d,E' \vdash_F \texttt{instruction} \xrightarrow{1+2} v} \quad (FLET)$$

$$\frac{d,E' \vdash_F x \xrightarrow{1} CONS(\_,CONS(\_,v))}{d,E' \vdash_F \texttt{arg} \xrightarrow{1+2} v} \quad (FLET)$$

Atomic constants. A finite number of *atoms*[12] 'a abbreviate distinct cons-patterns. One way to do this for n atoms, which we will assume below, is to encode each as a "bit-pattern" of the form

$$\texttt{cons}(b_1,\texttt{cons}(b_2,\texttt{cons}(\cdots(b_{k-1},b_k)\cdots)))$$

with $k=\lceil \log_2 n \rceil$ and each b_i either equal to 'nil or to the special macro 'true≡cons('nil,'nil) (we exploit the convention of F that 'nil denotes false and everything else true, including all atoms). In particular, LOOP uses the $n=10$ atoms 'empseq, 'seq, etc., hence here $k=4$.

Multi-way branch statement. The form

$$\texttt{CASE } E \texttt{ OF } \ldots \texttt{'}a_i \texttt{ -> } E_i \ldots$$

is short for a nested if–then–else statement which evaluates $E \to v$, determines (using a "decision tree") which "atomic" value $\texttt{'}a_i = v$, and chooses the corresponding E_i. Using the representation above, it is clear that this requires testing (at most) k times, namely one for each of the expressions hd v, hd tl $v,\ldots,$ hd^{k-1} tl v, thus the maximal number of decomposition commands is $T=\sum_{i=1}^{k} i$, i.e., in $O(\log_2^2 n)$ steps. This leads to a pseudo-inference rule

$$\frac{d,E' \vdash_F E \xrightarrow{t} \texttt{'}a_i \quad d,E' \vdash_F E_i \xrightarrow{t'} v_i}{d,E' \vdash_F \texttt{CASE } E \texttt{ OF } \ldots \texttt{'}a_i \texttt{ -> } E_i \ldots \xrightarrow{t+T+t'} v_i} \quad (FCASE)$$

[12] We hereby mean entities that can be compared in constant time.

where a_i is any of the atoms. Since LOOP uses ten atoms, selection of the appropriate branch is done in $T = \sum_{i=1}^{4} i = 10$ steps.

The interpretation proceeds by running i_F^{CAM} on input data cons($\overline{P}$, $\overline{\alpha}$), where p is some CAM program and α some CAM input value with the following compositionally defined representations:

Definition 19. *F-representation* $\overline{\cdot}$ *of CAM-programs*:

$$\overline{program(Cs)} = \overline{Cs}$$

$$\overline{\varnothing} = CONS('empseq, NIL)$$

$$\overline{C;Cs} = CONS('seq, CONS(\overline{C}, \overline{Cs}))$$

$$\overline{quote(\alpha)} = CONS('quote, \overline{\alpha})$$

$$\overline{car} = CONS('car, NIL)$$

$$\overline{cdr} = CONS('cdr, NIL)$$

$$\overline{cons} = CONS('cons, NIL)$$

$$\overline{push} = CONS('push, NIL)$$

$$\overline{swap} = CONS('swap, NIL)$$

$$\overline{cur(Cs)} = CONS('cur, \overline{Cs})$$

$$\overline{app} = CONS('app, NIL)$$

F-representation $\overline{\cdot}$ *of the CAM-stack*:

$$\overline{S \cdot \alpha} = CONS(\overline{S}, \overline{\alpha})$$

F-*representation* $\overline{\cdot}$ *of CAM-values*:

$$\overline{S \cdot \alpha} = CONS(\overline{S}, \overline{\alpha})$$

$$\overline{(\alpha, \beta)} = CONS(\overline{\alpha}, \overline{\beta})$$

$$\overline{[Cs, \alpha]} = CONS(\overline{Cs}, \overline{\alpha})$$

$$\overline{()} = NIL$$

To give the reader an idea about how the interpreter works, we will show fragments of the interpretation that would result from running the CAM program P of Section 5. We will run it on the simplest input data, ( ), since the input data does not matter anyway. The interpreter, i_F^{CAM}, will thus start executing with the x variable bound to the value $CONS(\overline{P}, NIL)$ where $\overline{P}$ has a value that looks like the following (using [...] for lists build with *CONS* and terminated with *NIL*, e.g., [a,b] means

$CONS(\mathtt{a}, CONS(\mathtt{b}, NIL))$):

$$\overline{P} = \overline{program(N; !C; \ldots; cons; app)}$$
$$= \mathtt{['seq,['quote],'seq,['push],'seq,['cur,'seq,'cdr,'empseq],}$$
$$\mathtt{\ldots,'seq,['cons],'seq,['app],'empseq]}$$

From looking at (F1) we see that the first piece of code that is executed is `tl(f(('nil.tl x).hd x))` of i_F^{CAM}; this means that `f` is applied and hence with (F11) we start by evaluating `('nil.tl x).hd x` which in our case builds the value $CONS(CONS(NIL, NIL), \overline{P})$, corresponding to the fact that the initial state of a CAM execution has the input (NIL) on the top of an otherwise empty stack. This is bound to `x` and then the LOOP of the interpreter is executed and simulates all the instructions in $\overline{P}$, returning just the stack; the result is then extracted from the top of the stack by the final `tl` call. The correctness of this is the first part of the proof of Lemma 17 below.

Running LOOP works similarly: the part of `x` denoted `instruction` – actually `hd tl x` – is investigated: here it is `'seq`. This means that the result of that instruction execution is `f(f(stack.c1).c2)`, or rather, `f(f(hd x,hd tl tl x).tl tl tl x)`, which means that the F evaluation mechanism will now first evaluate the first instruction, that happen to be a `'quote`, using the current stack, and then execute the following instructions using the resulting stack. In short: we have started simulating execution of the CAM instructions. The correctness of the actual simulation is the second half of the proof of Lemma 17 below.

Now we prove the correctness and efficiency of the interpretation $\mathrm{F} \geqslant \mathrm{CAM}$. Since this involves rather large inference trees, we write

$$\begin{array}{l} \overline{\mathrm{prem}_1}\ (\mathrm{Axiom}) \\ \vdots \\ \overline{\mathrm{prem}_k}\ (\mathrm{Axiom}) \\ \hline \mathrm{conclusion}\ (\mathrm{Rule}) \end{array} \quad \text{for} \quad \frac{\overline{\mathrm{prem}_1}(\mathrm{Axiom}) \quad \cdots \quad \overline{\mathrm{prem}_k}(\mathrm{Axiom})}{\mathrm{conclusion}}(\mathrm{Rule})$$

Proof of Lemma 17. Since the first proof steps of the left and right sides do not depend on c and α, we first prove that the *initialisation is correct*:

$$\exists e \geqslant 1\ \forall program(\mathrm{c}), \alpha : \tag{1}$$
$$(\vdash_F\ i_F^{CAM}, CONS(\overline{program(\mathrm{c})}, \overline{\alpha}) \xrightarrow{t_1} \overline{\beta})$$
$$\Leftrightarrow (\vdash_{CAM}\ program(\mathrm{c}), \alpha \xrightarrow{t_2} \beta)$$
$$\text{for } t_1 \leqslant e \cdot t_2$$

if and only if there exists some CAM stack s_1 such that

$$\exists e \geqslant 1\ \forall \mathrm{c}, \alpha : (CONS(\overline{()\cdot\alpha}, \overline{\mathrm{c}}), \mathrm{LOOP} \vdash_F\ \mathrm{LOOP} \xrightarrow{t_1} \overline{s_1 \cdot \beta}) \tag{2}$$
$$\Leftrightarrow ((\,)\cdot\alpha \vdash_{CAM}\ \mathrm{c} \xrightarrow{t_2} s_1 \cdot \beta) \quad \text{for } t_1 \leqslant e \cdot t_2$$

This is done by simply constructing the initial segment of both derivations synchronously: For all CAM programs *program*(Cs) and CAM values α we have:

$$\dfrac{\dfrac{\boxed{\textit{Hypothesis}}}{() \cdot \alpha \vdash \mathrm{Cs} \xrightarrow{t_1} s \cdot \beta}}{\vdash \textit{program}(\mathrm{Cs}), \alpha \xrightarrow{1+t_1} \beta}\ (\mathrm{CAM1})$$

Running the CAM interpreter $i_{\mathrm{F}}^{\mathrm{CAM}}$ on this program will always result in the following execution, with $d = \overline{CONS(\textit{program}(\mathrm{Cs}), \bar{\alpha})}$ and $E = \texttt{cons(cons('nil,tl x),hd x)}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{d, \textsc{Loop} \vdash \texttt{'nil} \xrightarrow{1} \mathit{NIL}}\,(\mathrm{F4}) \quad \dfrac{\dfrac{}{d, \textsc{Loop} \vdash \texttt{x} \xrightarrow{1} d}\,(\mathrm{F2})}{d, \textsc{Loop} \vdash \texttt{tl x} \xrightarrow{2} \bar{\alpha}}\,(\mathrm{F7})}{d, \textsc{Loop} \vdash \texttt{cons('nil,tl x)} \xrightarrow{4} \overline{() \cdot \alpha}}\,(\mathrm{F3}) \quad \dfrac{\dfrac{}{d, \textsc{Loop} \vdash \texttt{x} \xrightarrow{1} d}\,(\mathrm{F2})}{d, \textsc{Loop} \vdash \texttt{hd x} \xrightarrow{2} \overline{\textit{program}(\mathrm{Cs})}}\,(\mathrm{F7})}{d, \textsc{Loop} \vdash E \xrightarrow{7} \overline{CONS(() \cdot \alpha, \textit{program}(\mathrm{Cs}))}}\,(\mathrm{F3}) \quad \dfrac{\boxed{\textit{Hypothesis}}}{\overline{CONS(\bar{\alpha}, \textit{program}(\mathrm{Cs}))}, \textsc{Loop} \vdash \textsc{Loop} \xrightarrow{t_1} \overline{s_1 \cdot \beta}}\,(\mathrm{F11})}{d, \textsc{Loop} \vdash \texttt{f}(E) \xrightarrow{8+t_1} CONS(s_1, \bar{\beta})}\,(\mathrm{F7})}{d, \textsc{Loop} \vdash \texttt{tl(f}(E)\texttt{)} \xrightarrow{9+t_1} CONS(s_1, \bar{\beta})}\,(\mathrm{F1})$$
$$\vdash \texttt{tl(f}(E)\texttt{)}\ \texttt{whererec f(x) = Loop}\ ,\ d \xrightarrow{10+t_1} \bar{\beta}$$

From this we conclude that the initialisation is correct and adds only a constant interpretation overhead.

Then we prove by induction over the height of the (remaining part of the) proof that the Loop macro executes *one iteration of the interpreter in time independent of the CAM program*, i.e., that there exists some CAM stack s_R such that the invariant

$$\exists e \geqslant 1\ \forall \mathrm{c}, \alpha : (\overline{CONS(\overline{s_L \cdot \alpha}, \bar{\mathrm{c}})}, \textsc{Loop} \vdash_{\mathrm{F}} \textsc{Loop} \xrightarrow{t_1} \overline{s_R \cdot \beta}) \tag{3}$$
$$\Leftrightarrow (s_L \cdot \alpha \vdash_{\mathrm{CAM}} \mathrm{c} \xrightarrow{t_2} s_R \cdot \beta) \quad \text{for } t_1 \leqslant e \cdot t_2$$

Here we exploit that the operational semantic is compositional such that all derivations can be enumerated by structural induction over syntax.

We must consider all possible proof trees. The only possible variation, however, is in the Cs-instruction sequences, since the inference system is *syntax directed.* Thus varying Cs makes it possible to do induction. Furthermore it is *compositional* thus we can use this to do induction over the height of the proof trees. We will do both $\Leftarrow$ and $\Rightarrow$ at the same time, however, the inductions should be seen as separate: given a finite right hand side proof tree we can construct the left hand side one by well founded induction, and vice versa.

This yields an induction with eight base cases (one for each of the CAM constructions ø, *quote*(α), *car*, *cdr*, *cons*, *push*, *swap*, and *cur*($\cdot$), where the corresponding inference rules have no premises), and two inductive cases ($\cdot\,;\,\cdot$ and *app*). These are very similar; we show only the case which is particular to the higher-order aspects of CAM, namely the base case for *cur* and the inductive case for *app* (all the other cases are simpler):

– Case *cur*(Cs).

For any $s_L = s, \rho$, with $s_R = s$ and $d = CONS(\overline{s \cdot \rho}, CONS(\text{'cur}, \overline{\text{Cs}}))$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{d, \text{LOOP} \vdash \texttt{x} \xrightarrow{1} d}\,(\text{F1})}{d, \text{LOOP} \vdash \texttt{instruction} \xrightarrow{3} \text{'cur}}\,(\text{FLET})}{d, \text{LOOP} \vdash \texttt{rest} \xrightarrow{3} \overline{s}}\,(\text{FLET})}{d, \text{LOOP} \vdash \texttt{arg} \xrightarrow{3} \overline{\text{Cs}}}\,(\text{FLET})}{d, \text{LOOP} \vdash \texttt{rho} \xrightarrow{3} \overline{\rho}}\,(\text{FLET})}{d, \text{LOOP} \vdash \texttt{arg.rho} \xrightarrow{7} CONS(\text{Cs}, \overline{\rho})}\,(\text{F3})}{d, \text{LOOP} \vdash \texttt{rest.(arg.rho)} \xrightarrow{11} CONS(\overline{s}, CONS(\overline{\rho}, \overline{\text{Cs}}))}\,(\text{F3})}{d, \text{LOOP} \vdash \text{LOOP} \xrightarrow{T+14} \overline{s \cdot [\text{Cs}, \rho]}}\,(\text{FCASE})$$

and

$$\dfrac{}{s \cdot \rho \vdash cur(\text{Cs}) \xrightarrow{1} s \cdot [\text{Cs}, \rho]}\,(\text{CAM 10})$$

This means that this case adds the requirement that $T + 14 \leqslant e \cdot 1$.

– Case *app*.

For any $s_L = s, \rho, \alpha, \mathrm{Cs}$, with $s_R = s_1$ and $d = CONS(\overline{s \cdot ([\mathrm{Cs}, \rho], \alpha)}, CONS(\text{'app}, NIL))$:

$$\dfrac{\dfrac{}{d, \text{LOOP} \vdash \texttt{x} \xrightarrow{1} d}\ \text{(F1)}}{d, \text{LOOP} \vdash \texttt{instruction} \xrightarrow{3} \text{'app}}\ \text{(FLET)}$$

$$\dfrac{}{d, \text{LOOP} \vdash \texttt{x} \xrightarrow{1} d}\ \text{(F1)}$$
$$d, \text{LOOP} \vdash \texttt{stack} \xrightarrow{2} \overline{s \cdot ([\mathrm{Cs}, \rho], \alpha)}\ \text{(FLET)}$$
$$d, \text{LOOP} \vdash \texttt{rest} \xrightarrow{3} \overline{s}\ \text{(FLET)}$$

$$\dfrac{}{d, \text{LOOP} \vdash \texttt{x} \xrightarrow{1} d}\ \text{(F1)}$$
$$d, \text{LOOP} \vdash \texttt{stack} \xrightarrow{2} \overline{s \cdot ([\mathrm{Cs}, \rho], \alpha)}\ \text{(FLET)}$$
$$d, \text{LOOP} \vdash \texttt{rho} \xrightarrow{5} \overline{\rho}\ \text{(FLET)}$$

$$\dfrac{}{d, \text{LOOP} \vdash \texttt{x} \xrightarrow{1} d}\ \text{(F1)}$$
$$d, \text{LOOP} \vdash \texttt{stack} \xrightarrow{2} \overline{s \cdot ([\mathrm{Cs}, \rho], \alpha)}\ \text{(FLET)}$$
$$d, \text{LOOP} \vdash \texttt{a} \xrightarrow{4} \overline{\alpha}\ \text{(FLET)}$$

$$d, \text{LOOP} \vdash \texttt{rho.a} \xrightarrow{10} CONS(\overline{\rho}, \overline{\alpha})\ \text{(F3)}$$
$$d, \text{LOOP} \vdash \texttt{rest.(rho.a)} \xrightarrow{14} \overline{s \cdot (\rho, \alpha)}\ \text{(F3)}$$

$$\dfrac{}{d, \text{LOOP} \vdash \texttt{x} \xrightarrow{1} d}\ \text{(F1)}$$
$$d, \text{LOOP} \vdash \texttt{stack} \xrightarrow{2} \overline{s \cdot ([\mathrm{Cs}, \rho], \alpha)}\ \text{(FLET)}$$
$$d, \text{LOOP} \vdash \texttt{code} \xrightarrow{5} \overline{\mathrm{Cs}}\ \text{(FLET)}$$

$$d, \text{LOOP} \vdash \texttt{(rest.(rho.a)).code} \xrightarrow{20} CONS(\overline{s \cdot (\rho, \alpha)}, \overline{\mathrm{Cs}})\ \text{(F3)}$$

Induction hypothesis

$$CONS(\overline{s \cdot (\rho, \alpha)}, \overline{\mathrm{Cs}}), \text{LOOP} \vdash \text{LOOP} \xrightarrow{t_1'} \overline{s_1 \cdot \beta}$$
$$d, \text{LOOP} \vdash \texttt{f((rest.(rho.a)).code)} \xrightarrow{t_1'+21} \overline{s_1 \cdot \beta}\ \text{(F11)}$$
$$d, \text{LOOP} \vdash \text{LOOP} \xrightarrow{t_1'+24+T} \overline{s_1 \cdot \beta}\ \text{(FCASE)}$$

and

Induction hypothesis

$$\dfrac{s \cdot (\rho, \alpha) \vdash \mathrm{Cs} \xrightarrow{t_2'} s_1 \cdot \beta}{s \cdot ([\mathrm{Cs}, \rho], \alpha) \vdash app \xrightarrow{t_2'+1} s_1 \cdot \beta}\ \text{(CAM 11)}$$

So: $(T + 24) + t_1' \leqslant e \cdot (1 + t_2')$. By hypothesis $t_1' \leqslant e \cdot t_2'$. Hence this case adds the requirement that $T + 24 \leqslant e$.

Collecting the requirements to e for all the cases reveals that the *app* case is, indeed, the most severe, so we have shown that (3) implies (2) with $e \geqslant T + 24$, and consequently that i_F^{CAM} is correct and efficient. □

6.2. An efficient interpreter in CAM for F

Lemma 20. *There is an efficient interpretation CAM* $\geqslant$ F.

$$
\begin{aligned}
i &= program(\mathrm{Cs}_{init}; \mathrm{Cs}_{loop}) \quad \text{where}\\
\mathrm{Cs}_{init} &= push; push; cdr; swap; car; cdr; cons; swap; car; car; cons\\
\mathrm{Cs}_{loop} &= push; cdr; car; swap; push; car; swap; cdr; cdr; cons; cons; app\\
\mathrm{Cs}_{NIL} &= cdr; car; swap; cons; app\\
\mathrm{Cs}_{CONS} &= cdr; cdr; swap; cons; app\\
\mathrm{Cs}_{x} &= cdr; car; car\\
\mathrm{Cs}_{nil} &= quote(([\mathrm{Cs}_{NIL}, ()], ()))\\
\mathrm{Cs}_{cons} &= cdr; push; push; car; swap; cdr; cdr; cons; swap; push; car; swap;\\
&\qquad cdr; car; cons; \mathrm{Cs}_{loop}; swap; \mathrm{Cs}_{loop};\\
&\qquad cons; push; quote([\mathrm{Cs}_{CONS}, ()]); swap; cons\\
\mathrm{Cs}_{hd} &= cdr; \mathrm{Cs}_{loop}; push; car; push; quote([\mathrm{Cs}_{htNIL}, ()], [\mathrm{Cs}_{htCONS}, ()]);\\
&\qquad cons; app; car\\
\mathrm{Cs}_{tl} &= cdr; \mathrm{Cs}_{loop}; push; car; push; quote([\mathrm{Cs}_{htNIL}, ()], [\mathrm{Cs}_{htCONS}, ()]);\\
&\qquad cons; app; car\\
&\quad \textit{where } \mathrm{Cs}_{htNIL} = cdr; push; cons\\
&\quad \textit{and } \mathrm{Cs}_{htCONS} = cdr; cdr\\
\mathrm{Cs}_{if} &= cdr; push; push; car; swap; cdr; cdr; cons; swap; push; car; swap;\\
&\qquad cdr; car; cons; \mathrm{Cs}_{loop}; car; push;\\
&\qquad quote([\mathrm{Cs}_{ifNIL}, ()], [\mathrm{Cs}_{ifCONS}, ()]); cons; app\\
&\quad \textit{where } \mathrm{Cs}_{ifNIL} = cdr; push; car; swap; cdr; cdr; cons; \mathrm{Cs}_{loop}\\
&\quad \textit{and } \mathrm{Cs}_{ifCONS} = cdr; push; car; swap; cdr; cdr; cons; \mathrm{Cs}_{loop}\\
\mathrm{Cs}_{call} &= cdr; push; \mathrm{Cs}_{loop}; swap; car; cdr; cons; push; cdr;\\
&\qquad cons; \mathrm{Cs}_{loop}
\end{aligned}
$$

Fig. 6. Interpreter in CAM for F-programs.

In Fig. 6 we present $i^{\mathrm{F}}_{\mathrm{CAM}}$, a CAM interpreter of F, which we will prove efficient. (Note that this proof is not difficult as F has no higher-order features which we must simulate.) The interpretation of the F-program P on F-input d proceeds by running $i^{\mathrm{F}}_{\mathrm{CAM}}$ on the input data $\alpha = (\underline{P}, \underline{d})$, using the CAM-representation $\underline{\cdot}$ defined compositionally on F-programs/expressions as follows.

Definition 21. *CAM-representation* $\underline{\cdot}$ *of F-programs/expressions*:

$$\begin{aligned}
\underline{E \texttt{ whererec f(x) = } E'} &= (\underline{E'}, \underline{E}) \\
\underline{\mathtt{x}} &= ([\mathrm{Cs}_{\mathtt{x}}, (\,)], (\,)) \\
\underline{\texttt{'nil}} &= ([\mathrm{Cs}_{\mathtt{nil}}, (\,)], (\,)) \\
\underline{\mathtt{cons}(E_1, E_2)} &= ([\mathrm{Cs}_{\mathtt{cons}}, (\,)], (\underline{E_1}, \underline{E_2})) \\
\underline{\mathtt{hd}\, E} &= ([\mathrm{Cs}_{\mathtt{hd}}, (\,)], \underline{E}) \\
\underline{\mathtt{tl}\, E} &= ([\mathrm{Cs}_{\mathtt{tl}}, (\,)], \underline{E}) \\
\underline{\mathtt{if}\, E \,\mathtt{then}\, E_1 \,\mathtt{else}\, E_2} &= ([\mathrm{Cs}_{\mathtt{if}}, (\,)], (\underline{E}, (\underline{E_1}, \underline{E_2}))) \\
\underline{\mathtt{f}(E)} &= ([\mathrm{Cs}_{\mathtt{call}}, (\,)], \underline{E})
\end{aligned}$$

CAM-representation $\underline{\cdot}$ *of F-values*:

$$\begin{aligned}
\underline{NIL} &= ([\mathrm{Cs}_{NIL}, (\,)], (\,)) \\
\underline{CONS(d_1, d_2)} &= ([\mathrm{Cs}_{CONS}, (\,)], (\underline{d_1}, \underline{d_2}))
\end{aligned}$$

Proof of Lemma 20. Again the first proof steps of the left- and right-hand sides do not depend on P and d, so we first prove that the *initialisation is correct*:

$$\begin{aligned}
&\exists e \geqslant 1 \ \forall E \texttt{ whererec f(x) = } E' \,,\, d: \qquad (4) \\
&\quad (\vdash_{\mathrm{CAM}} \ i^{\mathrm{F}}_{\mathrm{CAM}}, (\underline{E \texttt{ whererec f(x) = } E', \underline{d}}) \xrightarrow{t_1} \underline{v}) \\
&\quad \Leftrightarrow \ (\vdash_{\mathrm{F}} \ E \texttt{ whererec f(x) = } E', d \xrightarrow{t_2} v) \quad \text{for } t_1 \leqslant e \cdot t_2
\end{aligned}$$

if and only if

$$\begin{aligned}
&\exists e \geqslant 1 \ \forall E, E', d : ((\,) \cdot ((\underline{d}, \underline{E'}), \underline{E}) \ \vdash_{\mathrm{CAM}} \ \mathrm{Cs}_{\mathrm{LOOP}} \xrightarrow{t_1} (\,) \cdot \underline{v}) \qquad (5) \\
&\quad \Leftrightarrow \ (d, E' \ \vdash_{\mathrm{F}} \ E \xrightarrow{t_2} v) \quad \text{for } t_1 \leqslant e \cdot t_2
\end{aligned}$$

For all F-programs E `whererec f(x) = E'` and F-values d we have that running $i^{\mathrm{F}}_{\mathrm{CAM}}$ on the program gives the following initial execution, with $\alpha = ((\underline{E},\underline{E'}),\underline{d})$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{()\cdot\alpha \vdash push \xrightarrow{1} ()\cdot\alpha}(\mathrm{CAM8})}{()\cdot\alpha\cdot\alpha \vdash push \xrightarrow{1} ()\cdot\alpha\cdot\alpha\cdot\alpha}(\mathrm{CAM8})}{()\cdot\alpha\cdot\alpha\cdot\alpha \vdash cdr \xrightarrow{1} ()\cdot\alpha\cdot\alpha\cdot\underline{d}}(\mathrm{CAM6})}{()\cdot\alpha\cdot\alpha\cdot\underline{d} \vdash swap \xrightarrow{1} ()\cdot\alpha\cdot\underline{d}\cdot\alpha}(\mathrm{CAM9})}{()\cdot\alpha\cdot\underline{d}\cdot\alpha \vdash car \xrightarrow{1} ()\cdot\alpha\cdot\underline{d}\cdot(\underline{E},\underline{E'})}(\mathrm{CAM5})}{()\cdot\alpha\cdot\underline{d}\cdot(\underline{E},\underline{E'}) \vdash cdr \xrightarrow{1} ()\cdot\alpha\cdot\underline{d}\cdot\underline{E'}}(\mathrm{CAM4})}{()\cdot\alpha\cdot\underline{d}\cdot\underline{E'} \vdash cons \xrightarrow{1} ()\cdot\alpha\cdot(\underline{d},\underline{E'})}(\mathrm{CAM7})}{()\cdot\alpha\cdot(\underline{d},\underline{E'}) \vdash swap \xrightarrow{1} ()\cdot(\underline{d},\underline{E'})\cdot\alpha}(\mathrm{CAM9})}{()\cdot(\underline{d},\underline{E'})\cdot\alpha \vdash car \xrightarrow{1} ()\cdot(\underline{d},\underline{E'})\cdot(\underline{E},\underline{E'})}(\mathrm{CAM5})}{()\vdash car \xrightarrow{1} ()\cdot(\underline{d},\underline{E'})\cdot\underline{E}}(\mathrm{CAM5})}{()\vdash cons \xrightarrow{1} ()\cdot((\underline{d},\underline{E'}),\underline{E})}(\mathrm{CAM7})}{()\cdot\alpha \vdash \mathrm{Cs}_{\mathrm{INIT}} \xrightarrow{23} ()\cdot((\underline{d},\underline{E'}),\underline{E})}(\mathrm{Cs})$$

$$\dfrac{\dfrac{\boxed{\textit{Hypothesis}}}{()\cdot((\underline{d},\underline{E'}),\underline{E}) \vdash \mathrm{Cs}_{\mathrm{LOOP}} \xrightarrow{t_1} \underline{v}}}{\alpha \vdash \mathrm{Cs}_{\mathrm{INIT}};\mathrm{Cs}_{\mathrm{LOOP}} \xrightarrow{22+t_1} v}(\mathrm{Cs}_{;})$$

$$\dfrac{\alpha \vdash \mathrm{Cs}_{\mathrm{INIT}};\mathrm{Cs}_{\mathrm{LOOP}} \xrightarrow{22+t_1} v}{\vdash program(\mathrm{Cs}_{\mathrm{INIT}};\mathrm{Cs}_{\mathrm{LOOP}}),\alpha \xrightarrow{22+t_1} v}(\mathrm{CAM1})$$

This corresponds to

$$\dfrac{\dfrac{\boxed{\textit{Hypothesis}}}{d,E' \vdash E \xrightarrow{t_2} v}}{\vdash E\ \texttt{whererec f(x) = } E'\ ,\ d \xrightarrow{t_2+1} v}(\mathrm{F1})$$

so the interpreter initialisation adds only a constant overhead.

Then we prove by induction over the height of the (remaining part of the) proof that the $\mathrm{Cs}_{\mathrm{LOOP}}$ macro executes *one iteration of the interpreter in time independent of the*

F-program, i.e., that for all CAM-stacks s, the invariant

$$\begin{aligned}&\exists e \geqslant 1\ \forall E, E', d:\\ &\quad (s \cdot ((\underline{d}, \underline{E'}), \underline{E}) \vdash_{\mathrm{CAM}} \mathrm{Cs}_{\mathrm{LOOP}} \xrightarrow{t_1} s \cdot \underline{v}) \\ &\quad \Leftrightarrow\ (d, E' \vdash_{\mathrm{F}} E \xrightarrow{t_2} v) \quad \text{for } t_1 \leqslant e \cdot t_2\end{aligned} \tag{6}$$

holds. The CAM representation of an F-expression E uses the general pattern $(E_{\mathrm{label}}, E_{\mathrm{arg}})$, where E_{label} is a distinct CAM-value identifying that E and E_{arg} is a representation of any F-entities nested inside E. The fact that the $\mathrm{Cs}_{\mathrm{LOOP}}$ macro ends with *app* is reflected by the form of E_{label} which is always a closure $[\mathrm{Cs}_{E_{\mathrm{label}}}, ( )]$ where $\mathrm{Cs}_{E_{\mathrm{label}}}$ is code to interpret E. Hence it turns out that the crucial step in proving the invariant is to show that

$$\begin{aligned}&s \cdot (E_{\mathrm{label}}, ((d, E'), E_{\mathrm{arg}})) \vdash \mathit{app} \xrightarrow{t+1} s \cdot \underline{v}\\ &\quad \Leftrightarrow\ s \cdot (( ), ((d, E'), E_{\mathrm{arg}})) \vdash \mathrm{Cs}_{E_{\mathrm{label}}} \xrightarrow{t} s \cdot \underline{v}\end{aligned} \tag{7}$$

which as before is a straightforward (if tedious) enumeration of all possible derivations. Again the only possible variation, however, is in the F-expressions. Thus varying E makes it possible to do induction over the height of the proof trees, completely analogously to the situation for the other interpreter, so we will do this, again collecting "requirements", this time to e'.

There are two base cases ($E = \mathtt{x}$ and `'nil`) that are immediate, and five inductive cases ($E =$ `hd`, `tl`, `cons`, `if`, and `f( )`); we will only show the last and most interesting one, namely function call; the remaining details are in the technical report [33]. To make it manageable, we will make use of two pseudo-rules for CAM to avoid excessive nesting: a pseudorule grouping a sequence of (CAM2) and (CAM3) commands at the same level,

$$\frac{s \vdash \mathrm{c}_1 \xrightarrow{t_1} s_1 \cdot \alpha_1 \quad \cdots \quad s_{k-1} \vdash \mathrm{C}_k \xrightarrow{t_k} s_k \cdot \alpha_k}{s \vdash \mathrm{C}_1; \ldots; \mathrm{C}_k \xrightarrow{t_1 + \cdots + t_k + k + 1} s_k \cdot \alpha_k} \quad k \geqslant 1 \qquad (\mathrm{Cs})$$

correcting for only including $\varnothing$ once for each sequence, and we will permit abbreviating a sequence of premises that are obviously axioms (one of CAM4–10) into "superaxioms" with the generic pseudorule-name (CAM$*$); finally we omit the leading "$( ) \cdot$" of stacks.

– $E = \mathtt{f}(E_1)$, assuming $d, E' \vdash E_1 \xrightarrow{t_2'} d'$ and $d', E' \vdash E' \xrightarrow{t_2''} v$.

$$\dfrac{\dfrac{\dfrac{}{((),((\underline{d},\underline{E'}),\underline{E_1})) \vdash \mathit{cdr};\mathit{push} \xrightarrow{5} ((\underline{d},\underline{E'}),\underline{E}) \cdot ((\underline{d},\underline{E'}),\underline{E})}\ (\text{CAM}*) \quad \boxed{\textit{Induction hypothesis}}}{((\underline{d},\underline{E'}),\underline{E}) \cdot ((\underline{d},\underline{E'}),\underline{E}) \vdash \mathrm{Cs}_{\mathrm{LOOP}}((\underline{d},\underline{E'}),\underline{E}) \cdot \underline{d'}}}{((\underline{d},\underline{E'}),\underline{E}) \cdot \underline{d'} \vdash \mathit{swap};\mathit{car};\mathit{cdr};\mathit{cons};\mathit{push};\mathit{cdr};\mathit{cons} \xrightarrow{15} ((\underline{d'},\underline{E'}),\underline{E'})}\ (\text{CAM}*) \quad \boxed{\textit{Induction hypothesis}}$$

$$\dfrac{((\underline{d'},\underline{E'}),\underline{E'}) \vdash \mathrm{Cs}_{\mathrm{LOOP}} \xrightarrow{t_1''} \underline{v}}{((),((\underline{d},\underline{E'}),\underline{E_1})) \vdash \mathrm{Cs}_{\mathtt{call}} \xrightarrow{25+t_1'+t_1''} \underline{v}}\ (\mathrm{Cs})$$

and

$$\dfrac{\dfrac{}{d,E' \vdash E \xrightarrow{t_2'} d'} \quad \boxed{\textit{Induction hypothesis}} \qquad \dfrac{}{d',E' \vdash E' \xrightarrow{t_2''} v} \quad \boxed{\textit{Induction hypothesis}}}{d',E \vdash \mathtt{f}(E') \xrightarrow{1+t_2'+t_2''} v}\ (\mathrm{F11})$$

The requirement added is $25 \leqslant e'$.

Since the strictest requirement is $98 \leqslant e'$ (contributed by the rule for cons), we conclude that (6) implies (5) whence $i^{\mathrm{F}}_{\mathrm{CAM}}$ is correct and efficient with factor $e' \geqslant 98$. □

7. A linear time hierarchy for CAM^{su}

In this section we prove our main result for CAM^{su}:

Theorem 22. *There exists a linear-time hierarchy for CAM^{su}.*

Proof. We establish that there exists b such that for all $a \geqslant 1$ we have $\mathrm{LIN}^{\mathrm{CAM}^{su}}(a) \subset \mathrm{LIN}^{\mathrm{CAM}^{su}}(a \cdot b)$, namely $b = e \cdot b' \cdot e'$ obtained by the following inclusions:

$$\begin{aligned}
\mathrm{LIN}^{\mathrm{CAM}^{su}}(a) &\subseteq \mathrm{LIN}^{\mathrm{F}^{su}}(a \cdot e) && \text{by Lemma 23}\\
&\subset \mathrm{LIN}^{\mathrm{F}^{su}}(a \cdot e \cdot b') && \text{by Theorem 12}\\
&\subseteq \mathrm{LIN}^{\mathrm{CAM}^{su}}(a \cdot e \cdot b' \cdot e') && \text{by Lemma 24}
\end{aligned}$$

□

The rest of this section is devoted to state and outline the proofs of the Lemmas 23 and 24, which follow the pattern from the F and CAM cases of the previous section, so we will merely outline where selective updating changes things.

Again the major part of the effort is in the construction of efficient interpretors witnessing $\mathrm{F}^{\mathrm{su}} \geqslant \mathrm{CAM}^{\mathrm{su}}$ and $\mathrm{CAM}^{\mathrm{su}} \geqslant \mathrm{F}^{\mathrm{su}}$, however, most of the previous section can be reused.

7.1. An efficient interpreter in F^{su} for CAM^{su}

Lemma 23. *There is an efficient interpretation $F^{su} \geqslant CAM^{su}$.*

Proof. In Fig. 5 we presented an efficient interpreter, $i_{\mathrm{F}}^{\mathrm{CAM}}$, of CAM programs, written in F. First we notice that $i_{\mathrm{F}}^{\mathrm{CAM}}$ is equally an F^{su}-program, interpreting the subset of $\mathrm{CAM}^{\mathrm{su}}$ which corresponds to CAM. $i_{\mathrm{F}^{\mathrm{su}}}^{\mathrm{CAM}^{\mathrm{su}}}$ is obtained by adding the following two rules for selective update to the loop-macro:

```
'rplaca -> LET (rest.a).b = stack IN setcar! a b
'rplacd -> LET (rest.a).b = stack IN setcdr! a b
```

With the representations of Definition 19 extended with
F^{su}-representation $\overline{\cdot}$ of CAM^{su}-programs:

$$\overline{rplaca} = CONS(\texttt{'rplaca}, NIL)$$
$$\overline{rplacd} = CONS(\texttt{'rplacd}, NIL)$$

The two new atoms do not change the complexity of CASE since $k = 4$ remains valid.
F^{su}-representation $\overline{\cdot}$ of the CAM^{su}*-stack/values*: Use the extension of the value translation for CAM generalised to term graphs [3].

The interpreter should, at input–output level, meet the requirements of efficiency, similar to the requirements for $i_{\mathrm{F}}^{\mathrm{CAM}}$:

$$\exists e \geqslant 1\ \forall \mathit{program}(\mathrm{C}), \alpha : \tag{8}$$
$$(\vdash_{\mathrm{F}^{\mathrm{su}}} i_{\mathrm{F}^{\mathrm{su}}}^{\mathrm{CAM}^{\mathrm{su}}}, CONS(\overline{\mathit{program}(\mathrm{C})}, \overline{\alpha}) \xrightarrow{t_1} \overline{\beta})$$
$$\Leftrightarrow\ (\vdash_{\mathrm{CAM}^{\mathrm{su}}} \mathit{program}(\mathrm{C}), \alpha \xrightarrow{t_2} \beta) \quad \text{for } t_1 \leqslant e \cdot t_2$$

Essentially the same proof is used to show that this is equivalent to finding some $\mathrm{CAM}^{\mathrm{su}}$ stack such that we can construct a "CAM graph stack". The initial situation established by the startup code is the following:

$$\exists e \geqslant 1\ \forall \mathrm{C}, \sigma, l : \tag{9}$$
$$(\sigma', l', \textsc{Loop} \vdash_{\mathrm{F}^{\mathrm{su}}} \textsc{Loop} \xrightarrow{t_2} \overline{\sigma_1, s_1 \cdot l_1})$$
$$\text{where } \sigma' = (\pi_1(\overline{\sigma, () \cdot l}))[l' \mapsto CONS(\pi_2(\overline{\sigma, () \cdot l}), \overline{\mathrm{C}})]$$
$$\Leftrightarrow\ (\sigma, () \cdot l \vdash_{\mathrm{CAM}^{\mathrm{su}}} \mathrm{C} \xrightarrow{t_2} \sigma_1, s_1 \cdot l_1)$$
$$\text{for } \alpha = \sigma @ l,\ \beta = \sigma_1 @ l_1,\ l' \notin \mathrm{Dom}(\sigma),\ t_1 \leqslant e \cdot t_2$$

(π_1 and π_2 are projections extracting the store and location components of a pair created by the representation function.) While the technical details of this property are more complex, the principle is exactly the same as for i_F^{CAM}: the initialisation portion of the interpretation establish the right store with a representation of the input value α, and correctly outputs the location containing the output value β at the end. In fact, since the notion of location is the same in the two models this turns out to be rather simple. It generalises to the LOOP invariant analogously to (3) for i_F^{CAM}:

$$\begin{aligned}
&\exists e \geqslant 1\ \forall \mathrm{C}, \sigma, s, l: \qquad (10)\\
&\quad (\sigma', l', \textsc{Loop} \vdash_{\mathrm{F^{su}}} \textsc{Loop} \xrightarrow{t_2} \overline{\sigma_1, s_1 \cdot l_1})\\
&\qquad \text{where } \sigma' = (\pi_1(\overline{\sigma, s \cdot l}))[l' \mapsto \mathit{CONS}(\pi_2(\overline{\sigma, s \cdot l}), \overline{\mathrm{C}})]\\
&\qquad \Leftrightarrow\ (\sigma, s \cdot l \vdash_{\mathrm{CAM^{su}}} \mathrm{C} \xrightarrow{t_2} \sigma_1, s_1 \cdot l_1)\\
&\qquad \text{for } l' \notin \mathrm{Dom}(\sigma),\ t_1 \leqslant e \cdot t_2
\end{aligned}$$

We conclude that $i_{F^{su}}^{CAM^{su}}$ is correct and efficient (the proof details for locations are in the technical report [33]). □

7.2. An efficient interpreter in CAM^{su} for F^{su}

Lemma 24. *There is an efficient interpretation* $\mathrm{CAM^{su}} \geqslant \mathrm{F^{su}}$.

Proof. We just need to extend the proof of Lemma 20 with cases for `setcar!` and `setcdr!`. This amounts to the following additions to the representation of Definition 21 and interpreter of Fig. 6:

$i_{CAM^{su}}^{F^{su}}$ is as i_{CAM}^{F} in Fig. 6 extended with these macros:

$$\begin{aligned}
\mathrm{Cs}_{\texttt{setcar!}} = {} & \mathit{cdr}; \mathit{push}; \mathit{push}; \mathit{car}; \mathit{swap}; \mathit{cdr}; \mathit{cdr}; \mathit{cons}; \mathit{swap}; \mathit{push};\\
& \mathit{car}; \mathit{swap}; \mathit{cdr}; \mathit{car}; \mathit{cons}; \mathrm{Cs}_{\textsc{Loop}}; \mathit{swap}; \mathrm{Cs}_{\textsc{Loop}};\\
& \mathit{cons}; \mathit{push}; \mathit{push}; \mathit{car}; \mathit{cdr}; \mathit{swap}; \mathit{cdr}; \mathit{rplaca};\\
& \mathit{cons}; \mathit{car}; \mathit{cons}; \mathit{car}; \mathit{car}\\
\mathrm{Cs}_{\texttt{setcdr!}} = {} & \mathit{cdr}; \mathit{push}; \mathit{push}; \mathit{car}; \mathit{swap}; \mathit{cdr}; \mathit{cdr}; \mathit{cons}; \mathit{swap}; \mathit{push};\\
& \mathit{car}; \mathit{swap}; \mathit{cdr}; \mathit{car}; \mathit{cons}; \mathrm{Cs}_{\textsc{Loop}}; \mathit{swap}; \mathrm{Cs}_{\textsc{Loop}};\\
& \mathit{cons}; \mathit{push}; \mathit{push}; \mathit{car}; \mathit{cdr}; \mathit{swap}; \mathit{cdr}; \mathit{rplacd};\\
& \mathit{cons}; \mathit{car}; \mathit{cons}; \mathit{car}; \mathit{car}
\end{aligned}$$

CAM^{su}-representation $\underline{\ }$ of F^{su}-expressions: as Definition 21 plus

$$\begin{aligned}
\underline{\texttt{setcar!}\ E_1\ E_2} &= ([\mathrm{Cs}_{\texttt{setcar!}}, (\,)], (\underline{E_1}, \underline{E_2}))\\
\underline{\texttt{setcdr!}\ E_1\ E_2} &= ([\mathrm{Cs}_{\texttt{setcdr!}}, (\,)], (\underline{E_1}, \underline{E_2}))
\end{aligned}$$

The proof carries through using the same generalisation as for the proof of Lemma 23 (proof details are in the technical report [33]).

We conclude that $i^{F^{su}}_{CAM^{su}}$ is correct and efficient. □

8. Conclusions

We will first summarize the contribution of this paper, then we will describe related results. Finally, we give our view of possible future directions and perspectives.

8.1. Contribution

The main contribution of this paper is the proof of the existence of a linear time hierarchy for an authentic and realistic intermediate language featuring higher order constructs: the Categorical Abstract Machine.

The existence was established by constructing mutually efficient interpreters between CAM and F, and CAM^{su} and F^{su}. Hence we conclude that the "pure" Categorical Abstract Machine, CAM, is robust with respect to linear time computations in F, and that the "impure" Categorical Abstract Machine with selective updating, CAM^{su}, is robust for linear time computations in F^{su}.

Since we have also argued why the Categorical Abstract Machine is itself almost a higher-order functional language, we believe to have taken a significant step towards the general statement that "first-order and higher-order functional languages define the same linear time hierarchies".

8.2. Related work

The basis for our work is the techniques and results presented by Jones in his STOC 1993 paper [16]. We will start by elaborating on how these differ from what is presented above. For two simple first-order sublanguages of Lisp, an imperative one (I) and the functional one used in this paper (F), Jones proved the existence of a linear time hierarchy based on the running times given in their semantics. The proof of this was structured as follows: first, by proving the property for I directly, then by proving it for F on the basis of the proof for I. This exposed two ways to show the property. (1) The technique used for I is to provide a self-interpreter,[13] and then essentially applying a diagonalisation argument to conclude about its efficiency. This technique is significant since it is independent of the existence of a related language which has the constant-factor time hierarchy. Moreover, the proof can easily be reused to determine legal constant-factors. This is the method used by Dahl and Hessellund [8], in which they decide on a constant-factor for I to be of value 249, which yields a rather fine-grained hierarchy-structure. (2) The technique used for F is to provide a pair of efficient interpreters between the language in question and the language which is already known to have this property. (Actually, a self-interpreter for F, efficient with respect to the cost

[13] A "Self-interpreter" is written in the interpreted language.

model in [16], has been given directly and proven correct by Andersen [1], however, without explicitly stating this efficiency property.)

The practical significance of LIN was discussed in Section 1.2. Unfortunately, both linear and sub-linear complexity classes have been shown sensitive (i.e. not robust) with respect to the underlying computation model. This means that the problems which belong to, e.g., LIN on one computation model may belong to LOGTIME on another computation model. It has been shown that it depends on the complexity class whether these problems appear. In 1985, Gurevich and Shelah [10] determined a time-limit for which any complexity class with a greater time-bound would be robust with respect to the computation model, whereas classes with a lower time-bound would be sensitive to their computation model. The limit is called "Nearly Linear Time", and is determined to be n times a poly-logarithmic function, where n is given by the size of the input. Consequently, proving that F has a linear-time hierarchy by application of technique (2) has the important consequence that it establishes F and I as mutually robust with respect to LIN.

The effect of permitting selective update facilities (hence cyclic graphs) in a language, was extensively discussed in Section 1.2. A widely believed conjecture is that the presence of these makes the computational model stronger in an asymptotic sense; consequently, two separate versions of the Categorical Abstract Machine were studied. This distinction was also originally made for the treatment of F and F^{su}, and I and I^{su}. Recently, a proof by Pippenger [28] has actually proved this conjecture for linear-time, on-line computations[14] for two versions of Lisp, one featuring selective update (impure Lisp), and one without (pure Lisp). Pippenger shows that for such a pure Lisp program to compute (on-line!) what an impure Lisp program does in n steps, $O(n \log n)$ steps are sufficient, but in some cases $\Omega(n \log n)$ steps are necessary.

The discussion of Constant Speed-up in Section 1.1, revealed a tension between traditional Turing Machine based complexity theory and practical experience; essentially, Constant Speed-up is an artifact of the Turing Machine model. This was not only pointed out by Jones [16], but also shown by Hühne [15]. Hühne showed that Constant Speed-up does not hold for a variant Turing Machine with "tree-like storage" because the symbols in a tree cannot be compressed in a way that reduces the depth of the tree. If we interpret these two observations, we may take the viewpoint that Turing Machines with tree-structured storage is in a way a first step or "lifting" towards a programming language model operating on tree-structured data.

8.3. Future directions

One of our primary concerns, which in fact initiated this work, was the problem of how to state time-performance specifications in a more formal and precise way for real-life functional languages. As recently discussed elsewhere by the author [31],

[14] A computation is *on-line* if its input and output each comprise an unbounded sequence of symbols and if, for every n, the nth output is produced by the computation before the $(n+1)$th input is received.

many conventional language features seem to be incompatible with the existence of a hierarchy. Being able to define any number of functions, and still assume that they can be accessed in constant time, seems a natural thing to have for any practical, high-level language. Permitting case-constructs, under the assumption that any of the branches can be accessed in constant time, is another example of the same kind. It could be interesting to be able to give back some kind of user feed-back as to which facilities should not be used in order to permit a constant-factor time hierarchy. It would also be nice to know more about hierarchies within other time bounds, in particular identifying whether these hierarchies correspond to real programming considerations.

In this paper we have tacitly limited ourselves to study linear time complexity in relation to eager functional languages. However, many *lazy* languages, such as Haskell [14] and Miranda [35], are widely used today. The main problem is to assign a reasonable cost to "suspensions", i.e., unevaluated expressions. Useful cost models for lazy languages have only just started to emerge; one particularly promising family of models is $\lambda\sigma_w^a$ of Benaissa et al. [4], since it exists in both tree- and graph-structured versions.

All models considered in this paper were deterministic. For the first-order case Jones shows that results for LIN generalise to NLIN, the class of problems decidable in *non-deterministic* linear time. Will the generalisation to higher-order languages remain valid for non-deterministic computation?

The last perspective of this work comes from Pippenger's proof that an "impure" Lisp and a "pure" Lisp variant (synonymous with graph- and tree-structured data in this paper) have different computational strength with respect to linear time, when restricted to on-line computations. However, most practical languages must be viewed as off-line computation devices by their semantics. It is an interesting perspective whether it is possible to extend these results by relating or comparing the constant-factor time hierarchies of languages in the two groups, e.g., CAM and CAM^{su}.

Acknowledgements

This work started as a student project supervised by Neil Jones. My thanks go to Kristoffer Rose for his sharp and valuable criticism and comments throughout, to Neil Jones, Olivier Danvy, and Peter Sestoft, for their comments to drafts of this paper, and to Ross Moore for his comments on the last version. Finally, I would like to thank the anonymous referees for insightful critique to the submitted version. The research has been supported by DIKU, the DART project, NSF, and the Torkil Holm Foundation.

References

[1] L.O. Andersen, Correctness proof for a self-interpreter, Student report 91-12-16, DIKU (University of Copenhagen), Department of Computer Science, Universitetsparken 1, 2100 Copenhagen Ø, Denmark, 1991.

[2] H.P. Barendregt, The Lambda Calculus: Its Syntax and Semantics, revised ed., North-Holland, Amsterdam, 1984.
[3] H.P. Barendregt, M.C.D.J. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, M.R. Sleep, Term graph rewriting, in: J.W. de Bakker, A.J. Nijman, P.C. Treleaven (Eds.,) PARLE '87 – Parallel Architectures and Languages Europe, Vol. II, Eindhoven, The Netherlands, June 1987, Lecture Notes in Computer Science, Vol. 256, Springer, Berlin, 1987, pp. 141–158.
[4] Z.-E.-A. Benaissa, P. Lescanne, K.H. Rose, Modeling sharing and recursion for weak reduction strategies using explicit substitution, in: H. Kuchen, D. Swierstra (Eds.), PLILP '96 – 8th Internat. Symp. on Programming Languages: Implementation, Logics and Programs, Aachen, Germany, September 1996, Lecture Notes in Computer Science, Vol. 1140, Springer, Berlin, 1996, pp. 393–407.
[5] W. Clinger, J. Rees et al., Revised^4 Report on the Algorithmic Language Scheme, November 1991.
[6] G. Cousineau, P.-L. Curien, M. Mauny, The categorical abstract machine, Science of Computer Programming 8 (1987) 173–202.
[7] P.-L. Curien, An abstract framework for environment machines, Theoret. Comput. Sci. 82 (2) (1990) 389–402.
[8] C. Dahl, M. Hessellund, Determining the constant coefficients in a time hierarchy, Student report 94-2-2, DIKU (University of Copenhagen), Department of Computer Science, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark, February 1994.
[9] N.G. de Bruijn, Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation with application to the Church–Rosser theorem, Koninklijke Nederlandse Akademie van Wetenschappen, Ser. A, Math. Sci. 75 (1972) 381–392.
[10] Y. Gurevich, S. Shelah, Nearly linear time, in: Logic at Botik, Lecture Notes in Computer Science, Vol. 363, Springer, Berlin, 1985, pp. 108–118.
[11] J. Hannan, Making abstract machines less abstract, in: Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science, Vol. 523, Springer, Berlin, 1991, pp. 618–635.
[12] J. Hartmanis, R.E. Stearns, On computational complexity of algorithms, Trans. Amer. Math. Soc. 117 (1965) 285–306.
[13] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley Series in Computer Science, Addison-Wesley, Reading, MA, 1979.
[14] P. Hudak, S.L. Peyton Jones, P. Wadler et al., Report on the programming language Haskell, SIGPLAN Notices 27 (5) (1992) Version 1.2.
[15] M. Hühne, Linear speed-up does not hold on Turing machines with tree storages, Inform. Process. Lett. 47 (1993) 313–318.
[16] N.D. Jones, Constant time factors *do* matter, in: S. Homer (Ed.), STOC '93. Symposium on Theory of Computing, ACM Press, New York, 1993, pp. 602–611.
[17] N.D. Jones, Program speedups in theory and practice, in: B. Pehrson, I. Simon (Eds.), 13th World Computer Congress 94, Vol. 1, IFIP, Elsevier, Amsterdam, 1994.
[18] N.D. Jones, Computability and Complexity, Addison-Wesley, Reading, MA, 1997.
[19] G. Kahn, Natural Semantics, Rapport de Recherche, 601, INRIA, Sophia-Antipolis, France, 1987.
[20] A.J. Kfoury, R.N. Moll, M.A. Arbib, A Programming Approach to Computability, Texts and monographs in Computer Science, Springer, Berlin, 1982.
[21] P. Landin, The mechanical evaluation of expressions, Comput. J. 6 (1964) 308–320.
[22] J.L. Lawall, H.G. Mairson, Optimality and inefficiency: What isn't a cost model of the lambda calculus?, in: Internat. Conf. of Functional Programming, ACM Press, New York, 1996, pp. 92–101.
[23] J. McCarthy, Recursive functions of symbolic expressions, Comm. ACM 3 (4) (1960) 184–195.
[24] J. McCarthy, P.W. Abrahams, D.J. Edwards, T.P. Hart, M. Levin, Lisp 1.5 Programmer's Manual, MIT Press, Cambridge, MA, 1965.
[25] A.R. Meyer et al., Algorithm and complexity, in: J. Van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. A, Elsevier, Amsterdam, 1990.
[26] R. Milner, M. Tofte, R. Harper, The Definition of Standard ML, MIT Press, Cambridge, MA, 1990.
[27] R. Paige, F. Henglein, Mechanical translation of set theoretic problem specifications into efficient RAM code – a case study, Lisp and Symbolic Computation 4 (2) (1987) 207-232.
[28] N. Pippenger, Pure versus Impure LISP, in: POPL '96 – 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, St. Petersburg Beach, Florida, January 1996, pp. 104–109.
[29] G.D. Plotkin, A structural approach to operational semantics, Technical Report FN-19, DAIMI, Aarhus University, Aarhus, Denmark 1981.

[30] K. Regan, Linear speed-up, information vicinity, and finite-state machines, in: IFIP Proceedings, North-Holland, Amsterdam, 1994.
[31] E. Rose, Characterizing computation models with a constant factor time hierarchy, in: B. Kapron (Ed.), DIMACS Workshop On Computational Complexity and Programming Languages, New Jersey, USA, July 1996. DIMACS, RUTCOR, Rutgers University. Workshop proceedings available from http://dimacs.rutgers.edu/Workshops/Programming.
[32] E. Rose, Linear time hierarchies for a functional language machine model, in: H.R. Nielson (Ed.), Programming Languages and Systems – ESOP '96, Linköping University, Linköping, Sweden, April 1996, Lectures Notes in Computer Science, Vol. 1058, Springer, Berlin, 1996, pp. 311–325.
[33] E. Rose, Linear-time hierarchies for a functional language machine model, DIKU student report, Available through URL: http://www.diku.dk/research-groups/topps/personal/evarose.html, 1997.
[34] R. Sommerhalder, S.C. van Westrhenen, The Theory of Computability – Programs, Machines, Effectiveness and Feasibility, Addison-Wesley, Reading, MA, 1988.
[35] D.A. Turner, Miranda: A non-strict funtional language with polymorphic types, in: J.-P. Jouannaud (Ed.), FPCA '85 – IFIP Internat. Conf. on Functional Programming Languages and Computer Architecture, Nancy, France, September 1985, Lectures Notes in Computer Science, Vol. 201, Springer, Berlin, pp. 1–16.
[36] P. Weis et al., The CAML Reference Manual, Version 2.5, INRIA-ENS, December 1987.