JFP **33**, e5, 50 pages, 2023.

doi:10.1017/S0956796823000023

Read/write factorizable programs

SIDDHARTH BHASKAR
University of Copenhagen, Copenhagen, Denmark
(*e-mail:* bhask2sk@jmu.edu)

JAKOB GRUE SIMONSEN
University of Copenhagen, Copenhagen, Denmark
(*e-mail:* simonsen@di.ku.dk)

Abstract

In the *cons-free* programming paradigm, we eschew constructors and program using only destructors. Cons-free programs in a simple first-order language with string data capture exactly P, the class of polynomial-time relations. By varying the underlying language and considering other data types, we can capture several other complexity classes. However, no cons-free programming language captures any *functional* complexity class for fundamental reasons. In this paper, we cleanly extend the cons-free paradigm to encompass functional complexity classes. Namely, we introduce programs with data that can either *only* be destructed or *only* be constructed, which we enforce by a type system on the program variables. We call the resulting programs *read/write-* (or *RW-*)factorizable, show that RW-factorizable string programs capture exactly the class FP of polynomial-time functions, and that tail-recursive RW-factorizable programs capture exactly the class FL of logarithmic-space functions. Finally, we state and solve the nontrivial problem of *syntactic composition* of two RW-factorizable programs.

1 Introduction

The primitive operations associated with certain data types can be separated into constructors and destructors. For example, over the natural numbers, the *successor* (or *increment*) function and the constant 0 are constructors, whereas the *predecessor* (or *decrement*) function and equals-zero test are destructors. Over strings on some fixed alphabet Σ, the *cons* function, which prepends a given character onto a given string, is a constructor, as is the constant naming the empty string. On the other hand, the *head* function, which isolates the first character of a given string, and the *tail* function, which deletes it, are destructors, as is the relation which tests whether a string is empty. Similarly, one can define constructors and destructors for trees, heaps, nested lists, and all sorts of other common data types.

A *cons-free* program over any one of these data types is a program in which no constructors occur. For example, consider the following cons-free natural number program, which decides whether its input is even:

$$\texttt{even(n)} = \texttt{if zero(n) then true else if zero(n}-1\texttt{) then false, else even(n}-2\texttt{).}$$

In other words, to check whether a given number is even, we return "true" if it is equal to zero, return "false" if it is equal to one, and otherwise subtract two and recurse. (Here, $\mathtt{n{-}2}$ abbreviates $\mathtt{n{-}1{-}1}$.) This program is clearly cons-free, as $\mathtt{0}$ and $+\mathtt{1}$ do not occur within.

Cons-free programs clearly do not form a Turing complete language. Given their somewhat severe limitations, it is perhaps surprising that cons-free programs retain significant computational power: by a seminal result of Neil Jones (1999), cons-free string programs in a simple first-order functional language capture exactly the class P of polynomial-time relations. In other words, every language in P is decided by some cons-free program, and every language decided by a cons-free program is in P. Moreover, if we restrict our attention to *tail-recursive* programs, we capture the class L of logarithmic-space relations in exactly the same sense.

Cons-free programs, however, fail to capture the *functional* versions of any complexity class, e.g., the classes FP and FL of polynomial-time and logarithmic-space functions respectively. Indeed, they cannot even define the simplest function which increases the size of the input. To remedy this deficiency, we introduce the notion of an *read/write-factorizable program*. Such programs extend the destruct-only (or *read-only*) variables of cons-free programs with construct-only (or *write-only*) variables. For example, the following addition program is RW-factorizable:

$$\mathtt{add(n,m)} = \mathtt{if\ zero(n)\ then\ m\ else\ add(n{-}1,m{+}1)}.$$

Here $\mathtt{n}$ is a read-only variable of type R, $\mathtt{m}$ is a write-only variable of type W, and $\mathtt{add} : R \times W \to W$. Similarly, we can define a *string concatenation* function of type $R \times W \to W$ by

$$\mathtt{cat(x,w)} = \mathtt{if\ null(x)\ then\ w\ else\ cons(hd\ x, cat(tl\ x, w))},$$

where $\mathtt{x}$ is a read-only variable of type R, $\mathtt{w}$ is a write-only variable of type W, $\mathtt{null} : R \to 2$ tests whether read-only string is empty, $\mathtt{cons} : \Sigma \times W \to W$ prepends a character in the alphabet Σ to a write-only string, $\mathtt{hd} : R \to \Sigma$ is the head function, and $\mathtt{tl} : R \to R$ is the tail function.

Consider the problem of programming a RW-factorizable *identity* function of type $R \to W$. Notice that the trivial program $\mathtt{id(x)} = \mathtt{x}$ cannot be consistently typed as $R \to W$; instead, we have to destruct the input and construct it again. Over strings, this would look something like $\mathtt{id(x)} = \mathtt{cat(x, nil)}$. This shows that information can *flow* from R values to W values; however, the reverse is not possible.

Finally, let us try to define the base-2 exponential function $n \mapsto 2^n$. The program

$$\begin{aligned} \mathtt{exp(n)} &= \mathtt{if\ zero(n)\ then\ 1\ else\ double(exp(n{-}1))} \\ \mathtt{double(n)} &= \mathtt{add(n,n)} \end{aligned}$$

is *not* RW-factorizable, because $\mathtt{double} : R \to W$, and $\mathtt{exp}$ cannot be well-typed, as its output must agree with both the input and output of $\mathtt{double}$. However,

$$\mathtt{leap(n)} = \mathtt{f(n,0)} \tag{1}$$

$$\mathtt{f(n,m)} = \mathtt{if\ zero(n)\ then\ m{+}1\ else\ f(n{-}1, f(n{-}1, m))} \tag{2}$$

is a perfectly legitimate RW-factorizable definition of $n \mapsto 2^n$, with $\mathtt{f} : R \times W \to W$.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

1.1 Our contribution

The central results of this paper are that

- RW-factorizable string programs of type $R \to W$ which are *non-nested* capture exactly the class FP of polynomial-time functions, and
- RW-factorizable strings programs of type $R \to W$ which are *tail-recursive* capture exactly the class FL of logarithmic-space functions.

In other words, the passage from cons-free to RW-factorizable programs is exactly what allows us to extend the results of Jones (1999) from relational to functional classes. The purpose of the *non-nested* stipulation is to exclude nested recursive programs like `leap` from line (1), whose outputs can have length exponential in the length of the inputs.[1]

A foundational lesson of implicit computational complexity going back to Bellantoni & Cook (1992) is that any Turing-complete programming language must allow for the *same* piece of data to be *both read and written*. Our results illustrate this phenomenon in a particularly clear way.[2]

A natural question to ask about RW-factorizable programs is how to (syntactically) compose them, i.e., produce a program which computes the sequential composition of two given programs. Naive attempts fail: you cannot stick an output of type W directly into an input of type R. So the problem is nontrivial; however, we solve it at the end of this paper.

Both the capturing and composition results utilize the same technical device, namely *bit-length computability* of a function, which is introduced in this paper. This essentially means computability by two cons-free programs, one computing the bits, the other the length, of a given function on each input. Our results suggest that this idea might have some "legs" and be worthy of further study in its own right.

1.2 Organization of this paper

In Section 2, we review some technical background. We then introduce RW-factorizable programs over strings, our main object of study, in Section 3. In Section 4, we introduce bit-length programs, a dialect of cons-free programs, and define bit-length computability in Section 5. That section also states that FL and FP are captured by pairs of bit-length programs; a sketch of the proofs is postponed to Appendix B.

Sections 6 defines a compiling function from RW-factorizable to (pairs of) bit-length programs; Section 7 defines a compiling function in the other direction. These two sections establish the equivalence of the two models of computation and thus prove our desired capturing results for RW-factorizable programs. Section 8 treats syntactic composition of bit-length programs and hence of RW-factorizable programs by proxy. Finally in Section 9, we discuss further directions for research.

The beating technical heart of this paper consists of four program transformations in Section 6, two in Section 7, and one in Section 8. Some of these are simple and others more intricate, but each transformation implements a conceptually simple idea which we

[1] More precisely, we forbid nested recursive calls of type W, like `f(...f(...))` in `leap`, but allow it for other types.

[2] Thanks to an anonymous referee who pointed this out.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

outline at the top of its respective section. We advise that the reader read these first before diving into the details.

We develop most transformations according to a common template: first describing a transformation of types, then of variables and values, then of terms and programs, and finally proving correctness. In the interests of space, we postpone all the proofs of correctness to Appendix A.

1.3 Related work

The present paper falls squarely into a long tradition of characterizing complexity classes by function algebras, programming languages, and related models of computation, a topic known as *implicit computational complexity* (ICC). ICC has been an area of interest since, at least, the 1960s (Cobham, 1965), with a resurgence in the last 30 years since the groundbreaking work of Bellantoni & Cook (1992). There are multiple approaches to ICC, see, e.g., the survey (Hofmann, 2000) for early developments pre-2000, and Dal Lago (2022) for a recent survey about methods involving higher-order programs. Recently, implicit characterizations have been furnished for complexity classes using different modes of computation, or using different modes of computation as a means of characterizing standard complexity classes; in particular, this includes probabilistic computation (Lago & Toldin, 2015; Lago et al., 2021), reversible computation (Kristiansen, 2022), parallel computation (Baillot & Ghyselen, 2022), and higher-order complexity (Hainry et al., 2022).

Many of these results are obtained by imposing a type system on a base programming language, as we do in this paper. Of the multitude of different flavors of implicit characterizations, we briefly review the most well-known ones with connections to our work:

- *Data ramification.* By this, we mean factoring the base data into two or more copies, and restricting how we can access or modify data in each copy. (Our approach of RW-factorization falls into this broad category.) The most common of these is the *normal/safe* factorization, which underlies many of the foundational papers in the field, e.g., Bellantoni & Cook (1992), Leivant (1995), and has a wide variety of applications. Our work is distinguished from these by our use of a general-purpose programming language with flexible recursive definitions, rather than function algebras based off of primitive recursion.
  There are other instances of data ramification, for example the *secure information flow* of Marion (2011), which partitions data into "higher" and "lower" security tiers. This bears resemblance to W- and R-data respectively, but ultimately yields different results: higher-security data does allow a limited amount of destruction, and FP is characterized by a class of tail-recursive programs. Still, this is perhaps the characterization closest to our own in spirit.
- *Linearity.* By linearity we mean (very roughly) restricting the reuse of variables or primitive operations, e.g., controlling the number of times a constructor can occur in the right-hand side of a recursive definition. Approaches to ICC based on linear logic include proof-theoretic approaches like light linear logic (Girard, 1998) and soft linear logic (Lafont, 2004), as well as the type-based approach of

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

non-size-increasing computation (Hofmann, 2002). From a practical perspective, they seem to yield more intensionally expressive languages than safe primitive recursion and have been applied to, e.g., cryptographic protocols (Baillot et al., 2019).

There does not seem to be any real technical connection between our work and this approach, except perhaps that our "non-nested" restriction of RW-factorizable programs carries a whiff of a linear typing discipline.

- *Restricted termination.* This is (roughly) the study of measures or bounds on programs or computation states, such that only those programs having execution that terminates within the bound, are part of the characterization. This is a flourishing field, both for traditional programs (Bonfante et al., 2011; Aubert et al., 2022) and for non-deterministic models such as term rewriting systems (Avanzini et al., 2011, 2015; Avanzini & Moser, 2016).

  The approach of the present paper does not use restricted termination in any technical sense, and the characterizations we provide are not reliant on termination; there is, however, a tenuous connection in the sense that we use programs with very restricted ability to construct data structures, and this might be amenable to analysis by methods devised for ICC using restricted termination.
- *Cons-free programs.* This is the particular line of inquiry which the present paper belongs to. Jones (1999) was the first to observe that eschewing constructors yields a simple method of capturing polynomial-time and logarithmic-space relations. This paradigm has since been extended to obtain many capturing results in a variety of contexts, e.g., higher types (Jones, 2001), non-determinism (Bonfante, 2006), simultaneous higher types and non-determinism (Kop & Simonsen, 2017), tree-like data (Ben-Amram & Petersen, 1998), term rewriting systems (de Carvalho & Simonsen, 2014; Czajka, 2018), and cons-free time complexity classes (Bhaskar et al., 2022; Jones et al., 2020). The present paper is the first to consider computability of function classes.

2 Preliminaries

We assume a basic familiarity with syntax and semantics of first-order functional programming languages, Turing machines, the Turing machine complexity classes P (polynomial time) and L (logarithmic space), and their functional variants FP and FL. Note that when defining a functional space class, the space bound only applies to the *work* tapes and not the output tape. Hence, the length of the output of function in FL may in general grow polynomially in the length of the input.

Following the set-theoretic convention, we identify any natural number n with its set of predecessors $\{0, 1, \ldots, n-1\}$ and we denote the set of all natural numbers by ω. Therefore, $1 = \{0\}$, $2 = \{0, 1\}$, and 1^n and 2^n are the sets of unary and binary strings of length n, respectively. We also identify 0 and 1 with $\bot$ (*false*) and $\top$ (*true*), respectively, so 2 is also the set of boolean values.

We denote the set of all finite unary and binary strings by $1^\star$ and $2^\star$, respectively, following the convention in computer science.[3] The *empty string* is denoted ε; its underlying

[3] In the set-theoretic convention these are $1^{<\omega}$ and $2^{<\omega}$ respectively.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

alphabet must be inferred from context. The *head* of a string is its first character, and its *tail* is the remainder. The head and tail of ε are undefined. If x is a string, then $|x|$ is its *length*.

We typically use lowercase Latin letters such as m, n, u, v, w, x, y for variables that range over "type-0 objects" such as natural numbers and strings. By, e.g., a "n-ary relation on strings," we mean a subset of $(2^\star)^n$. We use capital Latin letters, e.g., P, Q, X, Y, for variable that range over relations, and f, g, h for variables that range over functions. We use lowercase Greek letters for variables that range over types. We use `typewriter` script for program syntax.

A *partial function* $f : X \rightharpoonup Y$ is a function $f : X' \to Y$ for some $X' \subseteq X$; X' is the *domain of convergence* of f. If x is in the domain of convergence of f, we write $f(x)\downarrow$, otherwise $f(x)\uparrow$, and say that f "converges" or "diverges" on x, respectively. Partial functions converge strictly, meaning that $f(g(x))\downarrow$ implies, in particular, that $g(x)\downarrow$. For two partial functions $f, f' : X \rightharpoonup Y$, we say that $f' \sqsubseteq f$ in case $f'(x) = y \implies f(x) = y$ for all $(x, y) \in X \times Y$. Finally, note that by "partial function" we do mean a possibly total function, otherwise we will say *properly partial*.

The semantics of a program p is the partial function denoted by $[\![p]\!]$. A program p *computes* a partial function f in case $[\![p]\!]$ is identical to f as a partial function. A program p *accepts* a set X in case X is the domain of convergence of $[\![p]\!]$. Finally, a boolean-valued program p *decides* a set X if it computes the characteristic function of X.

A partial function computed by a program may in general have a dependent type. Given a type α and family of types $\beta(x)$ indexed by $x : \alpha$, we will refer to the *dependent sum type* $\sum_{x:\alpha} \beta(x)$ and the *dependent product type* $\prod_{x:\alpha} \beta(x)$. (When β does not depend on x, these reduce to $\alpha \times \beta$ and $\alpha \to \beta$ respectively.) The language of dependent types provides an elegant formulation for *counting modules*, but we do not use the machinery of dependent types in any essential way.

A note on functional complexity. Suppose that $f : 2^\star \to 2^\star$ is contained in the class FP of polynomial-time computable functions. Consider the function $f_\ell : 2^\star \to 1^\star$ defined by $f_\ell(x) = |f(x)|$, i.e., the length of $f(x)$. Then, f_ℓ is also in FP; take, for example, any Turing machine computing f in polynomial time and identify all the characters of its output alphabet. Similarly, consider the function $f_b : 2^\star \times 1^\star \to 2$ defined by $f_b(x, i) = \big(f(x)\big)_i$, meaning the i^{th}-bit of $f(x)$, for any $x \in 2^\star$ and $i < |f(x)|$. This relation is computable in polynomial time as well: on input (x, i), write $f(x)$ on a work tape and extract the ith-bit.[4]

All this is to say, roughly speaking, that if a function of type $2^\star \to 2^\star$ is computable in polynomial time, then so is its *length* function and its *bits* relation. We note that the converse implication is valid too. In other words, if f_ℓ and f_b are both computable in polynomial time, then we can compute f in polynomial time by concatenating the bits $f_b(x, i)$ for each $i < f_\ell(x)$.

In other words, we have reduced the notion of polynomial-time computability of functions of type $2^\star \to 2^\star$ to polynomial-time computability of relations and of functions $2^\star \to 1^\star$. What do we gain from this? We get *some* characterization of polynomial-time

[4] There is some slight imprecision here, since we haven't said what $f_b(x, i)$ is if $i \geq |f(x)|$. It doesn't matter: we view $f_b(x, i) = \big(f(x)\big)_b$ as a *partial* specification. If $f \in$ FP, there is *some* such $f_b \in$ FP; if there is some such $f_b \in$ FP and $f_\ell \in$ FP, then $f \in$ FP.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

computability of functions $2^\star \to 2^\star$ in terms of cons-free programs. (Cons-free programs can handle data of type $1^\star$ with *counting modules*, a device for reckoning with natural number quantities bounded by a polynomial in the length of the input.) So the polynomial-time computability of $f : 2^\star \to 2^\star$ may be witnessed by two cons-free programs: one with string input and counting module output computing the length and another with string and counting module input and boolean outputs, computing the bits. We shall elaborate on this later.

The observations we have made about polynomial time apply respectively to *logarithmic space* and *tail-recursive* cons-free programs. The proof is slightly different: notice that if $f : 2^\star \to 2^\star$ is in FL, then we cannot compute $f_b(x, i)$ by writing $f(x)$ on a work tape, because it will be too long, in general. However, there is a well-known trick in space complexity to circumvent this, viz., replacing the write-only output tape by a work tape that records only the *position* of the head. Similarly, when computing $f(x)$ using f_b and f_ℓ, we compute $f_\ell(x)$ *in binary*, which compresses it enough to stick it on a work tape.

We compile these observations into an official and easily referenced theorem.

Theorem 1. *A function $f : 2^\star \to 2^\star$ is computable in polynomial time, respectively, logarithmic space, iff there are functions $f_\ell : 2^\star \to 1^\star$ and $f_b : 2^\star \times 1^\star \to 2$ computing the length and bits of f which are computable in polynomial time, respectively logarithmic space.*

3 RW-factorizable programs

Let us work over the set $2^\star$ of binary strings. Consider the following set of string primitives[5]

- `hd`, denoting the head function, of type $2^\star \rightharpoonup 2$,
- `tl`, the tail function, of type $2^\star \rightharpoonup 2$
- `null`, the empty test, of type $2^\star \to 2$
- `nil`, a constant naming the empty string, of type $2^\star$, and
- `cons`, a binary function prepending a given character onto a given string, of type $2 \times 2^\star \to 2$.

Now, consider two separate copies of $2^\star$, viz., R, whose strings are read-only, and W, whose strings are write-only. Then, we may retype these primitives, replacing each occurrence of $2^\star$ by either R or W as follows:

$$\texttt{hd} : R \rightharpoonup 2, \quad \texttt{tl} : R \rightharpoonup R, \quad \texttt{null} : R \to 2, \quad \texttt{nil} : W, \quad \texttt{cons} : 2 \times W \to W.$$

Note that this typing is consistent with strings in R being "read-only" and strings in W being "write-only."

From these primitives, we construct a simple first-order programming language. First we define the types, then terms, of our programming language, then we define the programs themselves; finally, we equip these programs with an environment-based big-step semantics.

[5] Here we conflate program syntax, e.g., `hd`, with the function it denotes. We trust that this imprecision causes no confusion. The symbol $\rightharpoonup$ occurs in `hd` and `tl` because they are undefined on the empty string.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

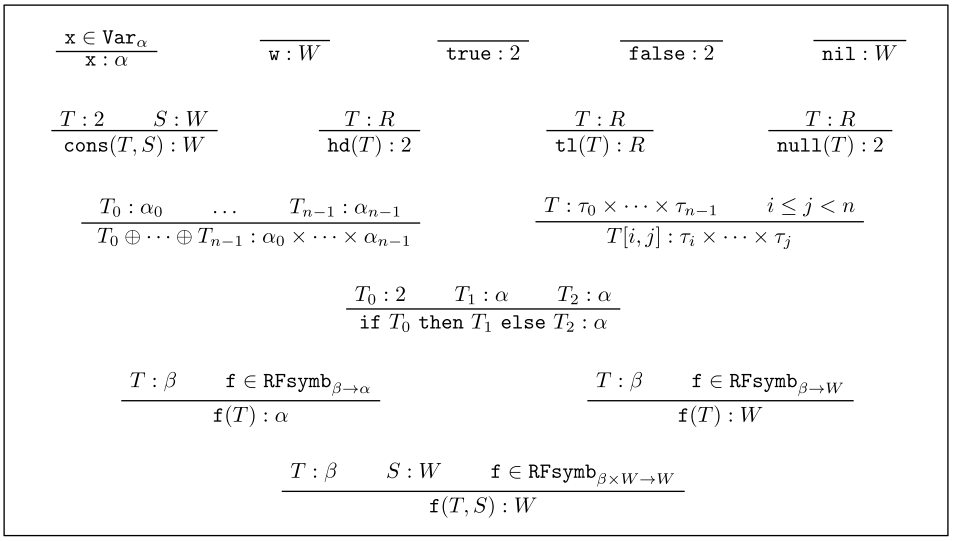

Fig. 1. RW-terms. Letters α, β, and each α_i range over product types and each τ_i ranges over the atomic types 2 and R.

Definition 1. The three *atomic types* are 2, R, and W, denoting the type of booleans, read-only strings, and write-only strings, respectively. A *product type* is any expression of the form $\tau_0 \times \cdots \times \tau_{n-1}$, for $n \geq 0$, where each τ_i is either 2 or R. (W is excluded from product formation.) When $n = 0$, the product is empty. We extend the type constructor $\times$ to apply to product types by

$$(\tau_0 \times \cdots \times \tau_{n-1}) \times (\tau_n \times \cdots \times \tau_{m-1}) = \tau_0 \times \cdots \times \tau_{m-1}.$$

A *function type symbol* is an expression of one of the following forms

$$\beta \to \alpha, \quad \beta \to W, \quad \beta \times W \to W,$$

where β and α are product types, and α is nonempty.

Definition 2. For each product type α, fix an infinite set $\mathtt{Var}_\alpha$ of variables of type α. For each product type β, fix infinite sets $\mathtt{RFsymb}_{\beta \to W}$ and $\mathtt{RFsymb}_{\beta \times W \to W}$ of function symbols of type $\beta \to W$ and $\beta \times W \to W$, respectively; similarly define $\mathtt{RFsymb}_{\beta \to \alpha}$ for each pair or product types (α, β).

By a *variable,* we mean any member of any $\mathtt{Var}_\alpha$, or the symbol $\mathtt{w}$, which is the sole variable of type W. By a *recursive function symbol,* we mean any member of one of the sets $\mathtt{RFsymb}_{\beta \to W}$ or $\mathtt{RFsymb}_{\beta \times W \to W}$, for any β, W.

Definition 3. An *RW term* is any expression that can be derived according to the inference rules in Figure 1.

It is straightforward to show that any RW term has a unique derivation from these inference rules, and that if an RW term contains any subterm of type W, it must itself have type W.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Notice that we restrict the way that W terms can occur: there is only one variable of type W and recursive function symbols of output type W have at most one W input. This restriction, which we may refer to as the *W-thinness* of RW terms, does not ultimately limit the expressive power of the resulting language. Roughly speaking, this is because a W datum is a sort of black box: no information can flow out of it and it cannot affect the *shape* of a computation. Think of it as a write-only output tape of a Turing machine: a single write-only output tape suffices.

Definition 4. We identify a few important properties of RW terms.

- An RW term is *explicit* if it contains no occurrence of a recursive function symbol.
- An RW term is *cons-free* if it is not of type W; equivalently, if it does not contain any subterm of type W.
- An RW term is *non-nested* in case it contains no occurrence of a recursion function symbol of type W inside another occurrence of a recursive function symbol of type W. More precisely, in any application of the bottom-most rule of Figure 1, the second hypothesis S must be explicit.
- An RW term is *tail-recursive* in case there is no occurrence of a recursive function symbol inside any other occurrence of a recursive function symbol, or any primitive call, or in the `if` clause of any `if / then / else` term. More precisely, in any application of a rule in the second row of Figure 1, or any application of the three recursive function calls at the bottom of 1, the terms in the hypotheses must be explicit; when forming `if` T_0 `then` T_1 `else` T_2, T_0 must be explicit.

Remark 1. Notice that every tail-recursive term is also non-nested. Tail recursive RW-factorizable programs will end up capturing FL and non-nested programs will end up capturing FP; the program `leap` of Section 1 is witness to the necessity of this restriction.

Notice also that to be non-nested we only forbid nested recursive calls of output type W. (Surprisingly, we can always eliminate nested recursive calls in the purely cons-free part of a program, though we will not need this result in the present paper.)

Remark 2. We will actually silently use a slightly more general formulation of tail recursion. Notice that tail recursion affords more flexibility to calls to the primitives over recursive calls. For example, it is perfectly legal to write phrases such as `hd(tl(...))` but not `hd(f(...))`, in a tail-recursive program.

However, consider a transformation $T \mapsto T^{\diamond}$ on terms that uses the recursive functions of a previously defined tail-recursive program $p^{\dagger}$. In arguing that $T \mapsto T^{\diamond}$ preserves tail recursion, it suffices to treat calls to the recursive functions of $p^{\dagger}$ like calls to the primitives as opposed to other recursive calls.[6]

[6] Suppose that a function f is computed by a tail-recursive program from the primitives Φ and that a function g is computed by a tail-recursive program from the primitives $\Phi \cup \{f\}$. Then, g can be computed by a tail-recursive program directly from Φ, but the simple transformation replacing each call to f by the Φ program computing it may not preserve tail recursion. So there is a theorem here, but an ancient one, almost certainly folklore. This remark appeals to this theorem, basically saying that to show that g is Φ-tail-recursive, it suffices to show that it's $\Phi \cup \{f\}$-tail-recursive. Cf. x2A.1 of Moschovakis (2018).

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Definition 5. An *RW-factorizable program* consists of a finite list of distinct recursive function symbols $(\mathtt{f}_0, \ldots, \mathtt{f}_k)$ and a definition for each one, which takes one of the following two forms:

$$\mathtt{f}_i(\mathtt{x}_i, \mathtt{w}) = T_i \quad \text{or} \quad \mathtt{f}_i(\mathtt{x}_i) = T_i,$$

where T_i is a RW-term, $\mathtt{x}_i$ is a variable, and every recursive function symbol that occurs in T_i must be one of $(\mathtt{f}_0, \ldots, \mathtt{f}_k)$. Furthermore

- If the definition of $\mathtt{f}_i$ is of the form $\mathtt{f}_i(\mathtt{x}_i, \mathtt{w}) = T_i$, then $\mathtt{f}_i(\mathtt{x}_i, \mathtt{w})$ must be a well-formed RW term of the same type as T_i, and the only variables that may occur in T_i are $\mathtt{x}_i$ and $\mathtt{w}$.
- If the definition of $\mathtt{f}_i$ is of the form $\mathtt{f}_i(\mathtt{x}_i) = T_i$, then $\mathtt{f}_i(\mathtt{x}_i)$ must be a well-formed RW term of the same type as T_i, and the only variable that may occur in T_i is $\mathtt{x}_i$.

The *head (term)* of a program is the left-hand side of its first line, e.g., $\mathtt{f}_0(\mathtt{x}_0)$ or $\mathtt{f}_0(\mathtt{x}_0, \mathtt{w})$. A program is *cons-free* in case all terms occurring within are cons-free.

Remark 3. Any RW-factorizable program can be split into a "purely cons-free part" and a "W-part." The purely cons-free part consists of all recursive function symbols of output type other than W and their definitions. This part is a self-contained "sub-program": it does not contain any recursive function calls of type W, and its semantics is independent of the rest of the program. The W part of the program consists of all recursive function symbols of output type W and their definitions. It lies on top of the purely cons-free part, using it as a black box.

We equip programs with a standard, call-by-value, environment-based big-step semantics. By a *value,* we mean an element of the domain of some type, and by an *environment*, we mean a finite function mapping variables to values.

Definition 6. For a program p, term T, value v of the same type as T, and environment ρ, we define the relation $\rho \vdash_p T \to v$ according to the inference rules in Figure 2. We typically suppress the subscript p from $\vdash$ for legibility.

Definition 7. Given a program p, define $[\![p]\!](x) = v$ iff there is a derivation of $[\mathtt{x} = x] \vdash_p \mathtt{f}_0(\mathtt{x}) \to v$, where $\mathtt{f}_0(\mathtt{x})$ is the head of p. (If the head of p is of the form $\mathtt{f}_0(\mathtt{x}, \mathtt{w})$, then $[\![p]\!](x, w) = v$ iff there is a derivation of $[\mathtt{x} = x, \mathtt{w} = w] \vdash_p \mathtt{f}_0(\mathtt{x}, \mathtt{w}) \to v$.) Since p is deterministic, $[\![p]\!]$ is easily seen to be a partial function.

Note that the relation $\vdash_p$ is independent of the head of p; it is only when defining $[\![p]\!]$ that we care what the head is.[7] Aside from specifying the head, neither $\vdash_p$ nor $[\![p]\!]$ is sensitive to the particular order of the recursive function symbols in a program; it is only because of overwhelming programming intuition that we say *a list* instead of *a set* of function symbols.

[7] A more precise treatment might distinguish *headless programs* from programs but we find this both slightly morbid and excessively pedantic.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

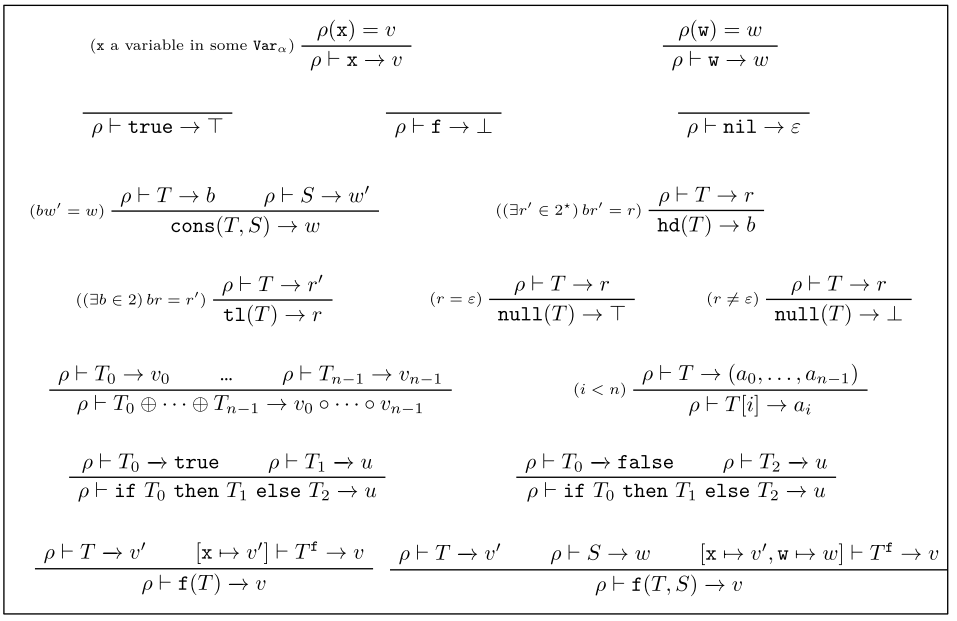

Fig. 2. Program Semantics. In the bottom two rules, $\mathtt{f}(\mathtt{x}) = T^{\mathtt{f}}$ or $\mathtt{f}(\mathtt{x}, \mathtt{w}) = T^{\mathtt{f}}$ is the recursive definition of $\mathtt{f}$ in p. Also, w ranges over values of type W; r and r' range over values of type R; b ranges over values of type 2; the a_i range over values of type 2 and R; v, v', v_i range over values of any product type; and u ranges over values of any type. The value $v_0 \circ \cdots \circ v_{n-1}$ denotes concatenation of the constituent tuples. For example if $v_0 = (a_0, a_1)$ and $v_1 = (a_2, a_3, a_4)$ then $v_0 \circ v_1 = (a_0, a_1, a_2, a_3, a_4)$. In general we will not carefully maintain this correspondence between variable names and their types.

In subsequent sections, we will define programs by *extending* previously defined programs with additional recursive function symbols and their definitions. This simply means: take the old program, forget what its head is, add new lines to the program—we don't care in what order—and pick a new head according to the definition at hand. If q is the old program and p the new one, notice that $\vdash_p$ extends $\vdash_q$.

4 Bit-length programs

In this section, we construct a dialect of cons-free programs and use them to define function computability in the manner sketched in Section 2. In other words, two programs are needed to witness the computability of a function $f : 2^\star \to 2^\star$ in polynomial time: one that computes the *bits* of f and the other that computes the *length*. Hence, we call these programs (rather unimaginatively) *bit-length* (or *BL*) programs.

The bit-length dialect differs from the standard cons-free language in two ways. The first difference is to replace R-data with string indices as follows. Imagine a cons-free program with a single string input. Then any string constructed during computation will be a suffix of that input. Instead, we might as well consider indices of the input string. Instead of querying the head of the various string suffixes, we query the *bit* of the input string at the given index.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Table 1. *A correspondence between languages in the present paper on the left and Jones (1999) on the right. A bit-length program is C-*free *if it contains no counting modules*

bit-length programs	$\texttt{CM}^{rec-poly}$
C-free bit-length programs	$\texttt{CM}^{rec\backslash+}$
tail-recursive bit-length programs	$\texttt{CM}^{poly}$
C-free tail-recursive bit-length programs	$\texttt{CM}^{\backslash+}$

The second difference is that counting modules—a type of natural numbers with magnitude bounded polynomially in the length of the input—are treated as a separate data type, a dependent type which varies with the length of the input string, as opposed to syntactic sugar.

Each of these modifications confers an advantage. Replacing R values by string indices lends a sort of extensional compositionality to bit-length programs that RW-factorizable programs lack. Meaning, the pair of programs computing a function transforms the same type of information about the input (its bits and length, given by the index primitives) into the same type of information about the output (its bits and length, given by the two programs). Moreover, a careful treatment of counting modules allows for cleaner representation strings in $1^\star$. (The relevance of unary strings to computation of functions was discussed in Section 2.)

We note that the original paper (Jones, 1999) contains a language called CM for *counter machine*, which was used as a technical tool in the proofs of the main results of that paper. A variant, $\texttt{CM}^{rec-poly}$, is basically identical to our bit-length language. However, our presentation is different enough to justify a separate treatment. For one, CM and its variants are imperative instead of functional. Secondly, indices and counting modules are conflated in CM, whereas we treat them as different types. A glossary between the present paper and Jones (1999) can be found in Table 1.

As for RW programs, we explain bit-length programs first by discussing types, then terms, then programs, and finally semantics.

Definition 8. For any $n \in \omega$, the *counting module* $C(n)$ is a data type whose domain is the set $n = \{0, 1, \ldots, n-1\}$, and whose primitives consist of constants naming the maximum $(n-1)$, minimum (0), and 1, plus operations of addition, subtraction, and comparison and equality tests. Addition and subtraction "top out" and "bottom out" at the maximum and minimum element, respectively.[8]

Definition 9. For any string $x \in 2^\star$, the set of indices $I(x)$ is a data type whose domain is $|x| + 1 = \{0, 1, \ldots, |x|\}$, and whose primitives consist of constants naming the maximum $(|x|)$, minimum (0), predecessor (which decrements the index), equality to zero, and *bit* (which returns the bit of the input at the given index).

Remark 4. We index strings such that the leftmost character has index the length of the string, and the rightmost character has index 1. That is, for any string x, $x = x_{|x|}x_{|x|-1}\ldots x_2x_1$. The bit x_0 is undefined. This funny indexing makes the compiling

[8] If the sum of two numbers is greater than the maximum, for example, then the result is simply the maximum.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

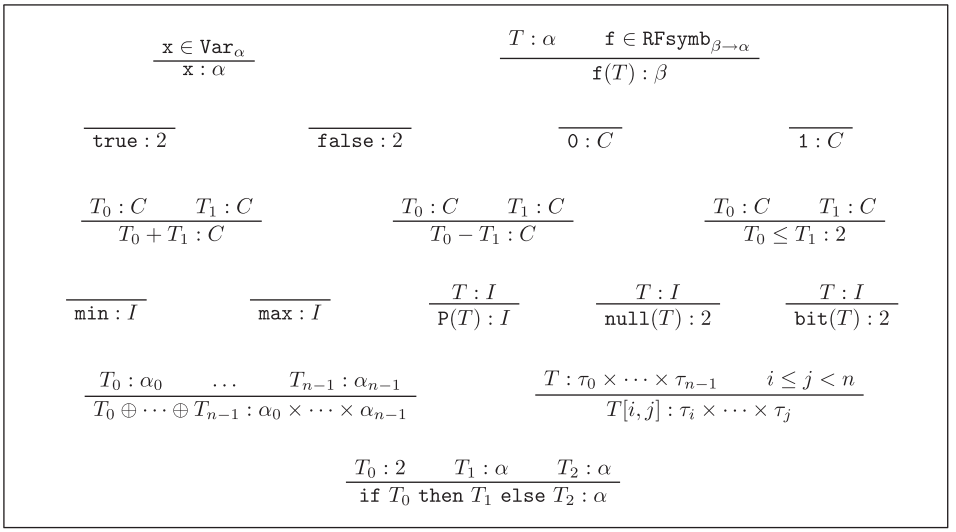

Fig. 3. Bit/length terms. Here α ranges over product types and τ over atomic types.

between bit-length and cons-free programs cleaner, as identifying indices with suffixes identifies `tl` with the predecessor function.

Notice that C and I are *dependent types*; namely, they are families of types indexed by some other value (natural numbers or binary strings). Now we introduce program syntax.

Definition 10. There are three *atomic type symbols*, 2, I, and C, denoting the type of booleans, indices, and counting modules respectively. A *BL-type symbol* is any expression of the form $\tau_0 \times \cdots \times \tau_{n-1}$, for $n \geq 0$, where each τ_i is an atomic type symbol. When $n = 0$, the product is empty, and when $n = 1$, we recover the atomic type symbols. A *BL-function type symbol* is an expression of the form $\beta \to \alpha$, where β and α are product types, and α is nonempty.

Definition 11. For a BL-type symbol α, we write $\alpha(x, n)$ to denote the type obtained by specializing each coordinate of α to x or n as appropriate. (Similarly for BL-function types.)

Remark 5. In the context of a single computation of a bit-length program, x and n will be fixed. Therefore, it is fine to type variables by type *symbols*, like 2, I, C, $2 \times I \times C$, etc., since we do not have multiple instantiations of these types in a single computation. In contrast, *values* have types like $I(x)$, $I(x) \times C(n)$, etc., where x is a string and n a number.

Definition 12. Fix an infinite set $\mathtt{Var}_\alpha$ of variables of type α, for each BL-type α, and an infinite set $\mathtt{RFsymb}_{\beta\to\alpha}$ of BL-function symbols of type $\beta \to \alpha$, for each function type $\beta \to \alpha$. Then, a *bit-length term* is any expression which can be derived according to the inference rules in Figure 3.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

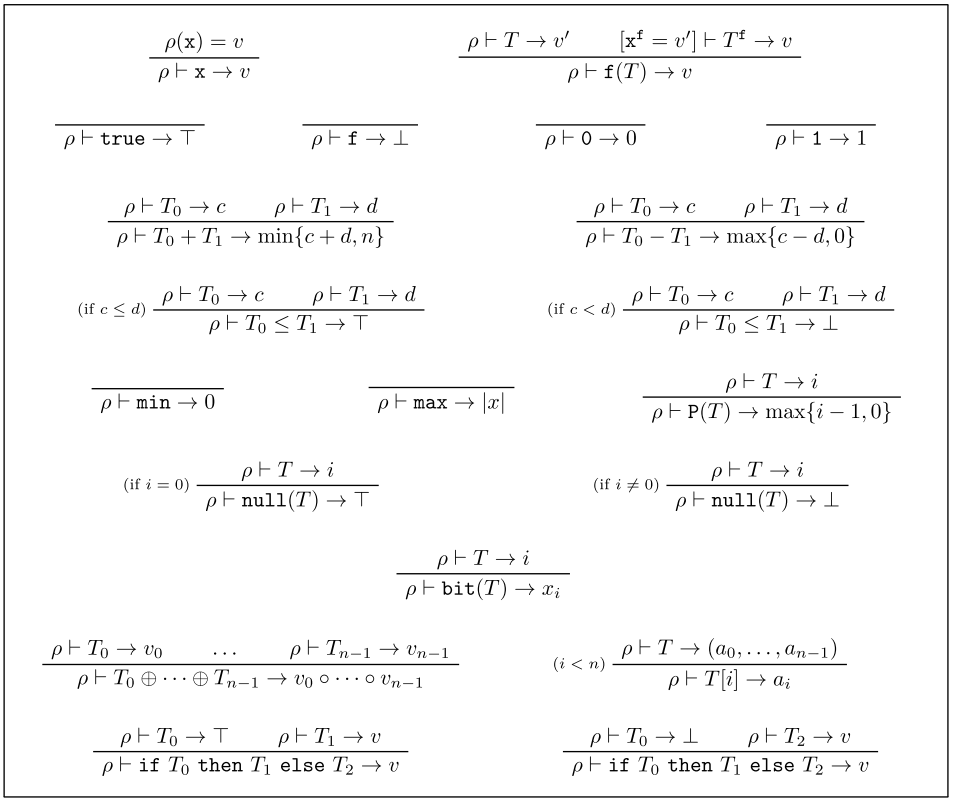

Fig. 4. Semantics for bit-length programs. Dependence of $\vdash$ on x, n, and ρ is suppressed for legibility. $\mathtt{f}(\mathtt{x}^{\mathtt{f}}) = T^{\mathtt{f}}$ is the recursive definition of $\mathtt{f}$ in p and x_i is the i-th bit of $x = x_{|x|} \dots x_1$. Variables c and d range over $C(n)$, i ranges over $I(x)$, the a_i range over values in any atomic type, and v, v' and the v_i range over values of any type. In general we will not carefully maintain this correspondence between variables and their types. As in RW-factorizable programs, note that while we can form larger tuples from smaller proper tuples, we can only decompose tuples into atomic types.

Definition 13. A bit-length term is *explicit* in case it contains no occurrence of a recursive function symbol. It is *tail-recursive* in case no recursive function symbol occurs inside any other recursive call, primitive call, or `if` clause. (We are not concerned with non-nested bit-length terms.)

Definition 14. A *bit-length program* is a finite list of lines of the form $\mathtt{f}_i(\mathtt{x}_i) = T_i$, for $0 \le i \le k$, where $\mathtt{x}_i$ is a variable whose type agrees with the domain of $\mathtt{f}$, and the type of T_i agrees with the codomain of $\mathtt{f}_i$. In addition, the only variable that occurs in T_i must be $\mathtt{x}_i$, and the only recursive function symbols that occur in T_i must be one of $(\mathtt{f}_0, \dots, \mathtt{f}_k)$.

Definition 15. For any bit-length program p, input string $x \in 2^\star$, seed $n \in \omega$, term T of type α, environment ρ binding the free variables of T, and value v of type $\alpha(x, n)$, we define the relation

$$x, n, \rho \vdash_p T \to v$$

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

according to the inference rules in Figure 4. Note that there is at most one v satisfying the above relation for each x, n, ρ, p, and T, which means that p is *deterministic*.

Remark 6. Note the extra two parameters x and n in the definition of semantics of bit-length programs. These may be viewed as analogous to a global variable if you're a programmer or the structure on the left-hand side of the satisfiability relation if you're a logician. Without them, the function symbols `max`, $+$, and `bit` are ill defined.

Definition 16. For any program p and function $\lambda : \omega \to \omega$, we define the relation $[\![p]\!]_\lambda(x, y) = w$ by

$$x, \lambda(|x|), [\mathtt{x}_0 = y] \vdash_p \mathtt{f}_0(\mathtt{x}_0) \to w,$$

where $\mathtt{f}_0(\mathtt{x}_0) = T_0$ is the head of p. By the determinism of p, $[\![p]\!]_\lambda$ is a partial function.

Note that the string variable x becomes the first argument of $[\![p]\!]_\lambda$. Therefore, if we want to compute a function which has a single string input by a bit-length program, the head of that program must be nullary, i.e., have zero inputs.

Definition 17. Fix a string x, natural number n, and program p, and let $\rho \vdash T \to v$ mean $x, n, \rho \vdash_p T \to v$. A *collision* is an occurrence of an inference rule of the form

$$\frac{\rho \vdash T_0 \to v \qquad \rho \vdash T_1 \to w}{\rho \vdash T_0 + T_1 \to \min\{v + w, n\}}$$

such that $\min\{v + w, n\} = n$, in a derivation formed from the inference rules in Figure 4. We say a derivation of $\rho \vdash T \to v$ is *collision-free* in case it contains no collisions.

Informally, a collision occurs when an addition operation attempts to overstep (or even meet) the maximum integer bound n in a derivation. The nice thing about collision-free derivations is that they are oblivious to this bound.

Lemma 1. *If $x, n, \rho \vdash_p T \to v$ by a collision-free derivation and $n' \geq n$, then $x, n', \rho \vdash_p T \to v$.*

Proof Notice that in the inference rules of Figure 4, a collision is the only instance in which the conclusion depends on n. Hence if we take a collision-free derivation and increase n, it is still a valid derivation. ■

Remark 7. As a consequence of Lemma 1, if $[\![p]\!]_\lambda(x, y) = w$ by a collision-free derivation and $\lambda(n) \leq \lambda'(n)$ for all $n \in \omega$, then $[\![p]\!]_{\lambda'}(x, y) = w$.

Finally, let us discuss the *type* of the partial function computed $[\![p]\!]_\lambda$ by a bit-length program p, which will in general be a dependent product type. Suppose the recursive function symbol $\mathtt{f}_0$ in the head of program p has type $\beta \to \alpha$. Then, the type of $[\![p]\!]_\lambda$ is

$$\prod_{x:2^\star} \big(\beta(x, \lambda(|x|)) \to \alpha(x, \lambda(|x|))\big).$$

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

We identify two important special cases. When β is the empty tuple and $\alpha = C$, then

$$[\![p]\!]_\lambda : \prod_{x:2^\star} C(\lambda(|x|)).$$

On the other hand, when $\alpha = 2$ and $\beta = C$, then

$$[\![p]\!]_\lambda : \sum_{x:2^\star} C(\lambda(|x|)) \to 2.$$

(Note that when both β is empty and $\alpha = 2$, then $[\![p]\!]_\lambda : 2^\star \to 2$.)

5 Bit-length computability of functions

The following theorem is a restatement of the theorems $\mathtt{TM}^{ptime} \equiv \mathtt{CM}^{rec-poly}$ and $\mathtt{TM}^{logspace} \equiv \mathtt{CM}^{poly}$ from Jones (1999).

Theorem. *Let f be any function of type $2^\star \to 2$. Then, the following are equivalent:*

- *f is computable in polynomial time.*
- *There is a polynomially bounded function $\lambda : \omega \to \omega$ and bit-length program p such that $[\![p]\!]_\lambda(x) = f(x)$, without collisions, for any $x \in 2^\star$.*

We get an analogous result by replacing FP *with* FL *and "bit-length program" with "tail-recursive bit-length program" throughout.*

We use the following extension of this theorem. The basic observation is that the simulations of Turing machines by programs can be augmented by counting modules that keep track of the *lengths* of tapes or what comes to the same thing, unary strings. So the simulation extends to Turing machines which return unary strings as output. We sketch the proof in Appendix B.

Theorem 2. *For any function $f : 2^\star \to 1^\star$, the following are equivalent:*

- *f is computable in polynomial time.*
- *There is a polynomially bounded function $\lambda : \omega \to \omega$ and a bit-length program p such that $[\![p]\!]_\lambda(x) = f(x)$, without collisions, for any $x \in 2^\star$.*

For any function $f : 2^\star \times 1^\star \to 2$ and polynomially bounded function $\pi : \omega \to \omega$, the following are equivalent:

- *There is a polynomial-time computable function $g : 2^\star \times 1^\star \to 2$ such that $g(x, y) = f(x, y)$ for any string $x \in 2^\star$ and $y < \pi(|x|)$.*
- *There is a polynomially bounded function $\lambda : \omega \to \omega$ and a bit-length program p such that $[\![p]\!]_\lambda(x, y) = f(x, y)$, without collisions, for any $x \in 2^\star$ and $y < \pi(|x|)$.*

Moreover, we get an analogous result by replacing FP *with* FL *and "bit-length program" with "tail-recursive bit-length program" throughout.*

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Note that the recursive function symbol $\mathtt{f}_0$ in the head of the program p should have type C if $f : 2^\star \to 1^\star$ and type $C \to 2$ if $f : 2^\star \times 1^\star \to 2$.

Combining Theorems 1 and 2, we are able to show the fundamental property of bit-length programs; namely that a function is computable in polynomial time iff there is a pair of bit-length programs computing the length and bits of f, respectively. Similarly, a function is computable in logarithmic space iff there is a pair of tail-recursive bit-length programs computing the length and bits of f respectively.

Theorem 3. *For any function $f : 2^\star \to 2^\star$, f is computable in polynomial time if and only if there is a polynomially bounded function $\lambda : \omega \to \omega$ such that $|f(x)| < \lambda(|x|)$ for all $x \in 2^\star$ and a pair of bit-length programs p and q such that*

$$|f(x)| = [\![q]\!]_\lambda(x)$$

without collisions, for all $x \in 2^\star$, and additionally for any $i < |f(x)|$,

$$\big(f(x)\big)_i = [\![p]\!]_\lambda(x, i)$$

without collisions, where $\big(f(x)\big)_i$ is the ith-bit of $f(x)$.

Furthermore, we get the analogous result for logarithmic space by requiring that p and q be tail-recursive.

Proof We go through the proof in the polynomial-time case; the proof for the logarithmic-space case is verbatim, replacing "program" by "tail-recursive program" throughout.

Suppose $f : 2^\star \to 2^\star$ is computable in polynomial time. By Theorem 1, there exist polynomial-time computable functions $f_\ell : 2^\star \to 1^\star$ and $f_b : 2^\star \times 1^\star \to 2$ such that, for all $x \in 2^\star$ $f_\ell(x) = |f(x)|$, and additionally for each $i < |f(x)|$, $f_b(x, i) = \big(f(x)\big)_i$. Let π be a polynomially bounded function such that $|f(x)| < \pi(|x|)$.

By Theorem 2 applied to f_ℓ, there is a polynomially bounded function λ_1 and a bit-length program q such that $[\![q]\!]_{\lambda_1}(x) = f_\ell(x)$, without collisions, for any string x. By Theorem 2 applied to f_b, there is a polynomially bounded function λ_2 and a bit-length program p such that $[\![p]\!]_{\lambda_2}(x, i) = f_b(x, i)$, without collisions, for any string x and $i < \pi(|x|)$. Let λ be a polynomially bounded function dominating both λ_1 and λ_2. By Remark 7 concerning collision-free computation, we can replace λ_1 and λ_2 by λ in the statements above. By definition of f_ℓ, $[\![q]\!]_\lambda(x) = |f(x)|$, for any string x. By definition of f_b and since $\pi(|x|)$ dominates $|f(x)|$, $[\![p]\!]_\lambda(x, i) = \big(f(x)\big)_i$ for any string x and $i < |f(x)|$. This concludes the forward direction.

In the other direction, suppose we have a bound λ and programs p and q satisfying the desired properties. Then by Theorem 2, the function $x \mapsto |f(x)| : 2^\star \to 1^\star$ is computable in polynomial time. Fix a polynomially bounded function π such that $|f(x)| < \pi(|x|)$. Then by Theorem 2 again, there is a function $g : 2^\star \times 1^\star \to 2$, computable in polynomial time, such that $g(x, i) = [\![p]\!]_\lambda(x, i)$ for every string x and $i < \pi(|x|)$. (In particular, $g(x, i) = \big(f(x)\big)_i$ for every string x and $i < |f(x)|$.) Finally, by applying Theorem 1 to $x \mapsto |f(x)|$ and g, we conclude that f is computable in polynomial time. ■

Theorem 3 suggests the following definition.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Definition 18. Let $f : 2^\star \to 2^\star$, $\lambda : \omega \to \omega$, p be a bit-length program whose head recursive function symbol has type $C \to 2$, and q be a bit-length program whose head recursive function symbol has type C.

Then we say (λ, p, q) *properly computes* f in case λ is increasing, $|f(x)| < \lambda(|x|)$ for all $x \in 2^\star$, $|f(x)| = [\![q]\!]_\lambda(x)$, without collisions, for all $x \in 2^\star$, and additionally for any $i < |f(x)|$, $\big(f(x)\big)_i = [\![p]\!]_\lambda(x, i)$ without collisions.

Then, Theorem 3 can be succinctly restated as follows: membership in FP is equivalent to being properly computed by some triple (λ, p, q) with polynomially bounded λ, and membership in FL is equivalent to being properly computed by some triple (λ, p, q) with polynomially bounded λ and p and q tail-recursive.

6 Compiling RW-factorizable to bit-length programs

We now show how to compile an RW-factorizable program into a pair of bit-length programs that properly computes the same function. This consists of four program transformations:

1. The "dagger" transformation †, that translates the RW-terms without W into the bit-length variant without counting modules. (So, substrings of the input string are reinterpreted as indices, `hd` is reinterpreted as `bit`, `tl` is reinterpreted as `P`, etc.)
2. The "diamond" transformation $\diamond$ that takes an RW-term T of type W into a bit-length term $T^\diamond$ of type 2, which detects whether T denotes a string built up from `w` or built up from `nil`.
3. A "length" transformation ℓ that takes an RW-term T of type W into a bit-length term T^ℓ of type C computing the length of T.
4. A "bit" transformation b that takes an RW-term T of type W into a bit-length term T^b of type 2, which has an extra variable `c` and computes the bit of T at `c`.

Each of these transformations requires a translation of types, then terms and values, then programs, and finally a proof of correctness.[9] Moreover, we need to observe that each transformation preserves tail recursion. For that reason, this section is long, even though the main idea of each translation is easy to intuit:

- The dagger transformation is straightforward, essentially amounting to a "renaming" of primitives.
- Operationally, when we evaluate a RW-term T of type W into a value v according to an environment ρ, v is constructed by starting with either the empty string ε or the string $w = \rho(\mathtt{w})$ and `cons`-ing various bits in front. The term $T^\diamond$ detects which of ε or $w\ v$ is "built up" from.
- Suppose p is an RW-factorizable program that contains a recursive function symbol `f` of type $\beta \times W \to W$. Suppose that $\rho \vdash_p \mathtt{f}(T, S) \to v$, $\rho \vdash_p \mathtt{f}(T, \mathtt{nil}) \to u$ and $\rho \vdash_p S \to w$. Then, there are two possibilities. Either $v = uw$, in which case

[9] The map on types is only explicitly defined for the dagger transformation; the remaining three map the type W to 2, C, and 2, respectively.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

– $|v| = |u| + |w|$, and
– $v_i = w_i$ (if $i \leq |w|$), and otherwise is $u_{i-|w|}$.

Otherwise $v = u$, in which case $|v| = |u|$ and $v_i = u_i$. Which case we are in is given to us by the diamond transformation.

The rest of the section essentially consists of making these intuitions formal.

6.1 The dagger transformation

The types in an RW-factorizable program are 2, R, and W. The types in a bit-length program are 2, I, and C. We first define a map from W-free types to bit-length types and then extend it to variables, recursive function symbols, terms, and programs.

Definition 19. Let $2^\dagger = 2$ and $R^\dagger = I$, and extend this to product and function types coordinate-wise.

Definition 20. For RW variables $\mathtt{x}$ not of type W and function symbols $\mathtt{f}$ not containing any type W, let $\mathtt{x} \mapsto \mathtt{x}^\dagger$ and $\mathtt{f} \mapsto \mathtt{f}^\dagger$ be a map from RW-factorizable variables and function symbols into bit-length variables and function symbols, so that for example, if $\mathtt{x}$ is of type τ, then $\mathtt{x}^\dagger$ is of type $\tau^\dagger$. (We may assume these maps are injective.)

Definition 21. Define a map $T \mapsto T^\dagger$ from W-free RW-factorizable terms to bit-length terms by:

- If $T \equiv \mathtt{true}$ or $T \equiv \mathtt{false}$,, then $T^\dagger \equiv T$.
- If $T \equiv \mathtt{x}$, then $T^\dagger \equiv \mathtt{x}^\dagger$.
- If $T \equiv \mathtt{hd}(S)$ then $T^\dagger \equiv \mathtt{bit}(S^\dagger)$.
- If $T \equiv \mathtt{tl}(S)$ then $T^\dagger \equiv \mathtt{P}(S^\dagger)$.
- If $T \equiv \mathtt{null}(S)$ then $T^\dagger \equiv \mathtt{null}(S^\dagger)$.
- If $T \equiv T_0 \oplus \cdots \oplus T_{n-1}$ then $T^\dagger \equiv T_0^\dagger \oplus \cdots \oplus T_{n-1}^\dagger$.
- If $T \equiv S[i]$ then $T^\dagger \equiv S^\dagger[i]$.
- If $T \equiv \mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2$, then $T^\dagger \equiv \mathtt{if}\ T_0^\dagger\ \mathtt{then}\ T_1^\dagger\ \mathtt{else}\ T_2^\dagger$.
- If $T \equiv \mathtt{f}(S)$ then $T^\dagger \equiv \mathtt{f}^\dagger(S^\dagger)$.

Notice that this transformation trivially preserves tail recursion.

Definition 22. For a cons-free program $p = (\mathtt{f}_i(\mathtt{x}_i) = T_i)_{0 \leq i \leq k}$, let the bit-length program $p^\dagger$ be $(\mathtt{f}_i^\dagger(\mathtt{x}_i^\dagger) = T_i^\dagger)_{0 \leq i \leq k}$. Note that $p^\dagger$ is a well-formed program which is tail-recursive if p is.

Finally, we extend † to a map on values. Note that the type of the map $y \mapsto y^\dagger : R \to I(x)$ depends on an "ambient string" x of which y is a suffix.

Definition 23. For any $b \in 2$, let $b^\dagger = b$. For any string $x \in 2^\star$ and any nonempty suffix y of x, let $y^\dagger = |y|$ (as a member of $I(x)$). Furthermore,

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

- Extend the map to tuples of data coordinate-wise, i.e., if $v = (v_0, \ldots, v_{n-1})$ is a decomposition of v into atomic types, then $v^\dagger = (v_0^\dagger, \ldots, v_{n-1}^\dagger)$. Note that $(u \circ v)^\dagger = u^\dagger \circ v^\dagger$, where $\circ$ denotes concatenation.
- For an environment ρ, define the environment $\rho^\dagger$ by $\rho(\mathtt{x}) = v \iff \rho^\dagger(\mathtt{x}^\dagger) = v^\dagger$. (By this, we mean that the domain of $\rho^\dagger$ is the $\dagger$-image of the domain of ρ.)

Next, we prove correctness of the dagger translation.

Definition 24. Let ρ be an RW environment, i.e., a partial, finite map from RW variables to RW data. For a string $x \in 2^\star$, we say that ρ is an *x-environment* in case for every variable $\mathtt{x}$ of type R, $\rho(\mathtt{x})$ is a suffix of x, and similarly, for every variable $\mathtt{x}$ of product type $\tau_0 \times \ldots \tau_{n-1}$, $\rho(\mathtt{x})[i]$ is a suffix of x whenever $\tau_i = R$.

Remark 8. Suppose p is RW-factorizable program, ρ is an x-environment, T is a cons-free term, and $\rho \vdash_p T \to v$. Then, every R value that appears in the derivation of $\rho \vdash T \to v$ must also be a suffix of x.

The proof of the next lemma is postponed to Appendix A.

Lemma 2. *Suppose p is a cons-free program, x is a string, ρ is an x-environment. Then,*

$$\rho \vdash_p T \to v \implies x, \rho^\dagger \vdash_{p^\dagger} T^\dagger \to v^\dagger.$$

Example. Consider the program p defined by

$$\mathtt{evt(y)} = \mathtt{if\ null(y)\ then\ true\ else\ if\ hd(y)\ then\ odt(tl\ y)\ else\ evt(tl\ y)}$$
$$\mathtt{odt(y)} = \mathtt{if\ null(y)\ then\ false,\ else\ if\ hd(y)\ then\ evt(tl\ y)\ else\ odt(tl\ y)}.$$

Then, `evt` and `odt` compute whether the number of occurrences of the character `true` in the string `y` is even or odd, respectively. The program $p^\dagger$ is defined by

$$\mathtt{evt}^\dagger(\mathtt{y}^\dagger) = \mathtt{if\ null}(\mathtt{y}^\dagger)\ \mathtt{then\ true\ else\ if\ bit}(\mathtt{y}^\dagger)\ \mathtt{then\ odt}^\dagger(\mathtt{P}(\mathtt{y}^\dagger))\ \mathtt{else\ evt}^\dagger(\mathtt{P}(\mathtt{y}^\dagger))$$
$$\mathtt{odt}^\dagger(\mathtt{y}^\dagger) = \mathtt{if\ null}(\mathtt{y}^\dagger)\ \mathtt{then\ false,\ else\ if\ bit}(\mathtt{y}^\dagger)\ \mathtt{then\ evt}^\dagger(\mathtt{P}(\mathtt{y}^\dagger))\ \mathtt{else\ odt}^\dagger(\mathtt{P}(\mathtt{y}^\dagger)).$$

Notice that $\mathtt{evt}^\dagger$ and $\mathtt{odt}^\dagger$ compute whether the number of occurrences of `true` with index at most $\mathtt{y}^\dagger$ in the input string is even or odd respectively. Thus, if `y` is bound to a suffix y of some string x, then `evt` and `odt` behave the same on input y as $\mathtt{evt}^\dagger$ and $\mathtt{odt}^\dagger$ behave on input $y^\dagger$ (the index $|y|$) with global input x.

6.2 The diamond transformation

These next three transformations each extend the dagger transformation from pure cons-free terms to terms of type W in different ways. In the diamond transformation, we will transform terms of type W into terms of type 2, the idea being that the transformed term will encode which of `nil` or `w` the original term is built up from.

Definition 25. For each recursive function symbol $\mathtt{f} : \beta \times W \to W$, let $\mathtt{f}^\diamond$ be a recursive function symbol of type $\beta^\dagger \to 2$. (We may assume that the map $\mathtt{f} \mapsto \mathtt{f}^\diamond$ is injective.)

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Definition 26. We define a transformation $T \mapsto T^\diamond$ from RW terms of type W to bit-length terms of type 2 as follows:

- If $T \equiv \mathtt{w}$, then $T^\diamond \equiv \mathtt{false}$,.
- If $T \equiv \mathtt{nil}$, then $T^\diamond \equiv \mathtt{true}$.
- If $T \equiv \mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2$ then $T^\diamond \equiv \mathtt{if}\ T_0^\dagger\ \mathtt{then}\ T_1^\diamond\ \mathtt{else}\ T_2^\diamond$.
- If $T \equiv \mathtt{f}(T')$ for some $\mathtt{f} : \beta \to W$, then $T^\diamond \equiv \mathtt{true}$.
- If $T \equiv \mathtt{f}(T', S)$ for some $\mathtt{f} : \beta \times W \to W$, then $T^\diamond \equiv \mathtt{if}\ S^\diamond\ \mathtt{then}\ \mathtt{true}\ \mathtt{else}\ \mathtt{f}^\diamond((T')^\dagger)$
- If $T \equiv \mathtt{cons}(T', S)$ then $T^\diamond \equiv S^\diamond$.

Remark 9. This transformation transforms tail-recursive terms into tail-recursive terms. It never produces a primitive call, and the only things it puts inside recursive calls are outputs of the † transformation. The only way it possibly produces a non-tail-recursive term is in the case $T \equiv \mathtt{f}(T', S)$, when $S^\diamond$ goes in an `if` clause. But if T is tail-recursive, S is explicit, so $S^\diamond$ is explicit, hence contains no recursive function calls.

Definition 27. Given an RW-factorizable program p of type $R \to W$, we define the bit-length program $p^\diamond$ by extending $p^\dagger$ as follows. For every function symbol $\mathtt{f}_i$ of type $\beta \times W \to W$ with definition $\mathtt{f}_i(\mathtt{x}_i, \mathtt{w}) = T_i$, we add a new line $\mathtt{f}_i^\diamond(\mathtt{x}_i^\dagger) = T_i^\diamond$. Note that $p^\diamond$ is tail-recursive if p is.

We ignore function symbols of type $\beta \to W$. Also, we do not bother to specify a head, because we will only need $\vdash_{p^\diamond}$, not $[\![p^\diamond]\!]$.

Note that $p^\diamond$ is a well-formed program: its recursive function symbols consist exactly of $\mathtt{f}_i^\dagger$ for $\mathtt{f}_i$ of type non-W and $\mathtt{f}_i^\diamond$ for $\mathtt{f}_i$ of type $\beta \times W \to W$. For the latter, the term $\mathtt{f}_i^\diamond(\mathtt{x}_i^\dagger)$ is well-formed and of the same type as $T_i^\diamond$. The only variable that may occur in $T_i^\diamond$ is $\mathtt{x}_i^\dagger$. Notice as well that $p^\diamond$ contains no terms of type C. Hence, when specifying a semantics, we do not need to give a natural number bound n on the upper end of the counting module.

Lemma 3. *Suppose that p is an RW-factorizable program, T is an RW term, $x, v, v', w \in 2^\star$, ρ is an x-environment, and $\mathtt{w}$ is not bound by ρ.*

Suppose that $\rho, [\mathtt{w} = w] \vdash_p T \to v$ and $\rho, [\mathtt{w} = \varepsilon] \vdash_p T \to v'$.[10] *Then either*

- *$x, \rho^\dagger \vdash_{p^\diamond} T^\diamond \to \top$ and $v = v'$, or*
- *$x, \rho^\dagger \vdash_{p^\diamond} T^\diamond \to \bot$ and $v = v'w$.*

(The proof is postponed to Appendix A.)

In particular, Lemma 3 implies that for any x, x-environment ρ, w, and term T, if there exists a v such that $\rho, [\mathtt{w} = w] \vdash T \to v$, then there exists a boolean b such that $x, \rho^\dagger \vdash_{p^\diamond} T^\diamond \to b$. We will silently use this fact in the remainder of this paper.

[10] Note that the convergence of the term T only depends on the environment ρ and not on the binding of the variable $\mathtt{w}$.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Example. Consider the program *id* defined by

$$\begin{aligned}\texttt{id}(\texttt{x}) &= \texttt{cat}(\texttt{x}, \texttt{nil})\\ \texttt{cat}(\texttt{x}, \texttt{w}) &= \texttt{if null}(\texttt{x}) \texttt{ then w else cons}(\texttt{hd x}, \texttt{cat}(\texttt{tl x}, \texttt{w})),\end{aligned}$$

which computes the identity function of type $R \to W$. Both `id` and `cat` have output type W; note that `id` always builds its output from `nil` and `cat` always builds its output from the input `w`. Then,

$$\begin{aligned}\texttt{id}^{\diamond}(\texttt{x}^{\dagger}) &= \texttt{true}\\ \texttt{cat}^{\diamond}(\texttt{x}^{\dagger}) &= \texttt{if null}(\texttt{x}^{\dagger}) \texttt{ then false, else if false, then true else cat}^{\diamond}(\texttt{P}(\texttt{x}^{\dagger})).\end{aligned}$$

We can see that $\texttt{id}^{\diamond}$ and $\texttt{cat}^{\diamond}$ compute the always-true and always-false functions, respectively, as they ought to. Note that the dagger and diamond transformations produce C-free BL-terms. By contrast, the subsequent two transformations make crucial use of counting modules.

6.3 The length transformation

In the length transformation, we will be transforming terms of type W into terms of type C. Since the length of any W output of an RW-factorizable program is bounded by a polynomial in the length of the inputs, it makes sense to try to capture the length within a counting module.

Definition 28. For each recursive function symbol `f` of type $\beta \times W \to W$ or $\beta \to W$, let $\texttt{f}^{\ell}$ be a recursive function symbol of type $\beta^{\dagger} \to C$. (We may assume that the map $\texttt{f} \mapsto \texttt{f}^{\ell}$ is injective.)

Definition 29. We define a transformation $T \mapsto T^{\ell}$ from RW-terms of type W to bit-length terms of type C as follows:

- If $T \equiv \texttt{w}$ or $T \equiv \texttt{nil}$, then $T^{\ell} \equiv 0$.
- If $T \equiv \texttt{if } T_0 \texttt{ then } T_1 \texttt{ else } T_2$, then $T^{\ell} \equiv \texttt{if } T_0^{\dagger} \texttt{ then } T_1^{\ell} \texttt{ else } T_2^{\ell}$.
- If $T \equiv \texttt{f}(T')$, then $T^{\ell} \equiv \texttt{f}^{\ell}((T')^{\dagger})$.
- If $T \equiv \texttt{f}(T', S)$, where S has type W, then

$$T^{\ell} \equiv \texttt{if f}^{\diamond}((T')^{\dagger}) \texttt{ then f}^{\ell}((T')^{\dagger}) \texttt{ else f}^{\ell}((T')^{\dagger}) + S^{\ell}.$$

- If $T \equiv \texttt{cons}(T', S)$, then $T^{\ell} \equiv 1 + S^{\ell}$.

Remark 10. This transformation does *not* preserve tail recursion, the problem being $\texttt{f}^{\ell}((T')^{\dagger}) + S^{\ell}$, where the recursive function symbol $\texttt{f}^{\ell}$ occurs within the primitive $+$. This can be fixed in a couple of ways. For one, we can observe that if T is tail-recursive then T^{ℓ} is *linear recursive*, which can always be transformed into an equivalent tail-recursive term (Greibach, 1975).

But there is an easier fix in this case: just give each $\texttt{f}^{\ell}$ an extra input of type `c`, the idea being that the new $\texttt{f}^{\ell}(T, \texttt{c})$ is the old $\texttt{f}^{\ell}(T) + \texttt{c}$. We have to change each recursive definition $\texttt{f}^{\ell}(\texttt{x}^{\dagger}) = T^{\ell}$ to $\texttt{f}^{\ell}(\texttt{x}^{\dagger}, \texttt{c}) = T^{\ell} + \texttt{c}$. Then $\texttt{f}^{\ell}((T')^{\dagger}) + S^{\ell}$ becomes $\texttt{f}^{\ell}((T')^{\dagger}, S^{\ell})$, and this solves our problem.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Definition 30. Given an RW-factorizable program p of type $R \to W$, we define the bit-length program p^ℓ by extending $p^\diamond$ by adding the following lines for each recursive function symbol of output type W:

For each line in p of the following form,	add the line
$\mathtt{f}_i(\mathtt{x}_i) = T_i \ (\mathtt{f} : \beta \to W)$	$\mathtt{f}_i^\ell(\mathtt{x}_i^\dagger) = T_i^\ell$
$\mathtt{f}_i(\mathtt{x}_i, \mathtt{w}) = T_i \ (\mathtt{f} : \beta \times W \to W)$	$\mathtt{f}_i^\ell(\mathtt{x}_i^\dagger) = T_i^\ell$

Finally, add a new head $\mathtt{h} = \mathtt{f}_0^\ell(\mathtt{max})$. The resulting program is p^ℓ; it is tail-recursive if p is.

Let us check that p^ℓ is a well-defined bit-length program. In addition to $p^\diamond$, it contains recursive function symbols $\mathtt{f}_i^\ell$ for $\mathtt{f}_i$ of output type W. If the only variables that may occur in T_i are $\mathtt{x}_i$ and $\mathtt{w}$, then the only variable that may occur in T_i^ℓ is $\mathtt{x}_i^\dagger$, and the only recursive function symbols that may occur are the recursive function symbols listed above. Finally, the terms in each line are well-formed and the types of each recursive functions symbol and its definition agree. Finally, notice that since p has type $R \to W$, p^ℓ has type C.

Definition 31. For a value v of type W, let $v^\ell = |v|$.

Lemma 4. *For every RW-program p of type $R \to W$, RW term T from p of type W, strings x, v, and w, natural number $n > |v|$, and x-environment ρ binding $\mathtt{w}$ to w:*

$$\rho \vdash_p T \to v \implies x, n, \rho^\dagger \vdash_{p^\ell} T^\ell \to v^\ell - \delta,$$

where $\delta = 0$ if $\rho^\dagger \vdash_\diamond T^\diamond \to \top$ or $\delta = |w|$ if $\rho^\dagger \vdash_\diamond T^\diamond \to \bot$. Moreover, the derivation of the right-hand side is collision-free.

(The proof is postponed to Appendix A.)

Theorem 4. *Suppose that p is an RW-factorizable program of type $R \to W$ and suppose that $\lambda : \omega \to \omega$ satisfies*

$$(\forall x, w \in 2^\star)\ [\![p]\!](x) = w \implies |w| < \lambda(|x|).$$

Then,

$$(\forall x, w \in 2^\star)\ [\![p]\!](x) = w \implies [\![p^\ell]\!]_\lambda(x) = |w|;$$

moreover, without collisions.

Proof Fix x and suppose that $[\![p]\!](x) = w$. Suppose $\mathtt{f}_0(\mathtt{x}_0)$ is the head of p. Then, $[\mathtt{x}_0 = x] \vdash_p \mathtt{f}_0(\mathtt{x}_0) \to w$, so by Lemma 4, since $\lambda(|x|) > w$,

$$x, \lambda(|x|), [\mathtt{x}_0^\dagger = x^\dagger] \vdash_{p^\ell} \mathtt{f}_0^\ell(\mathtt{x}_0^\dagger) \to |w|$$

by a collision-free derivation. Therefore, since $x^\dagger = |x|$ (as a member of $I(x)$), and since $|x|$ is the denotation of $\mathtt{max}$, we have

$$x, \lambda(|x|) \vdash_{p^\ell} \mathtt{f}_0^\ell(\mathtt{max}) \to |w|.$$

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Since $\mathtt{f}_0(\mathtt{max})$ is precisely the head of p^ℓ, we have $[\![p^\ell]\!]_\lambda(x) = |w|$, which is what we wanted to prove. ■

Examples. Consider again the program *id* given by

$$\begin{aligned}\mathtt{id}(\mathtt{x}) &= \mathtt{cat}(\mathtt{x}, \mathtt{nil})\\ \mathtt{cat}(\mathtt{x}, \mathtt{w}) &= \mathtt{if\ null}(\mathtt{x})\ \mathtt{then\ w\ else\ cons}(\mathtt{hd\ x}, \mathtt{cat}(\mathtt{tl\ x}, \mathtt{w})),\end{aligned}$$

Then,

$$\begin{aligned}\mathtt{id}^\ell(\mathtt{x}^\dagger) &= \mathtt{if\ cat}^\diamond(\mathtt{x}^\dagger)\ \mathtt{then\ cat}^\ell(\mathtt{x}^\dagger)\ \mathtt{else\ cat}^\ell(\mathtt{x}^\dagger) + 0\\ \mathtt{cat}^\ell(\mathtt{x}^\dagger) &= \mathtt{if\ null}(\mathtt{x}^\dagger)\ \mathtt{then}\ 0\ \mathtt{else}\ 1 + \big(\mathtt{if\ cat}^\diamond(\mathtt{P}(\mathtt{x}^\dagger))\ \mathtt{then\ cat}^\ell(\mathtt{P}(\mathtt{x}^\dagger))\ \mathtt{else}\\ &\qquad \mathtt{cat}^\ell(\mathtt{P}(\mathtt{x}^\dagger)) + 0\big),\end{aligned}$$

which is semantically equivalent to

$$\begin{aligned}\mathtt{id}^\ell(\mathtt{x}^\dagger) &= \mathtt{cat}^\ell(\mathtt{x}^\dagger)\\ \mathtt{cat}^\ell(\mathtt{x}^\dagger) &= \mathtt{if\ null}(\mathtt{x}^\dagger)\ \mathtt{then}\ 0\ \mathtt{else}\ 1 + \mathtt{cat}^\ell(\mathtt{P}(\mathtt{x}^\dagger)).\end{aligned}$$

It's now easy to see that for sufficiently large λ, $[\![id^\ell]\!]_\lambda(x) = |x|$, as it ought to. (Recall that the head of id^ℓ is $\mathtt{id}^\ell(\mathtt{max})$.)

Consider the following program *leap* based off of modifying the program from line (1) in Section 1 to string data:

$$\begin{aligned}\mathtt{leap}(\mathtt{x}) &= \mathtt{f}(\mathtt{x}, \mathtt{nil})\\ \mathtt{f}(\mathtt{x}, \mathtt{w}) &= \mathtt{if\ null}(\mathtt{x})\ \mathtt{then\ cons}(\mathtt{true}, \mathtt{w})\ \mathtt{else\ f}(\mathtt{tl\ x}, \mathtt{f}(\mathtt{tl\ x}, \mathtt{w})).\end{aligned}$$

Then for any string x, $[\![leap]\!](x)$ is a $2^{|x|}$-length string of only the character `true`. Now,

$$\begin{aligned}\mathtt{leap}^\ell(\mathtt{x}^\dagger) &= \mathtt{if\ f}^\diamond(\mathtt{x}^\dagger)\ \mathtt{then\ f}^\ell(\mathtt{x}^\dagger)\ \mathtt{else\ f}^\ell(\mathtt{x}^\dagger) + 0\\ \mathtt{f}^\ell(\mathtt{x}^\dagger) &= \mathtt{if\ null}(\mathtt{x}^\dagger)\ \mathtt{then}\ 1 + 0\ \mathtt{else}\\ &\qquad \mathtt{if\ f}^\diamond(\mathtt{Px}^\dagger)\ \mathtt{then\ f}^\ell(\mathtt{Px}^\dagger)\ \mathtt{else\ f}^\ell(\mathtt{Px}^\dagger) +\\ &\qquad \big(\mathtt{if\ f}^\diamond(\mathtt{Px}^\dagger)\ \mathtt{then\ f}^\ell(\mathtt{Px}^\dagger)\ \mathtt{else\ f}^\ell(\mathtt{Px}^\dagger) + 0\big).\end{aligned}$$

This is semantically equivalent to

$$\begin{aligned}\mathtt{leap}^\ell(\mathtt{x}^\dagger) &= \mathtt{f}^\ell(\mathtt{x}^\dagger)\\ \mathtt{f}^\ell(\mathtt{x}^\dagger) &= \mathtt{if\ null}(\mathtt{x}^\dagger)\ \mathtt{then}\ 1\ \mathtt{else\ f}^\ell(\mathtt{Px}^\dagger) + \mathtt{f}^\ell(\mathtt{Px}^\dagger),\end{aligned}$$

which we can see correctly computes the base-2 exponential of the length of the input string (given a sufficiently large counting module).

6.4 The bit transformation

In the bit transformation, we will transform terms of type W into terms of type 2, with an additional type-C input. The idea is that the transformed term encodes the *bits* of the original W-term.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Definition 32. For each function symbol $\mathtt{f}$ of type $\beta \times W \to W$ or $\beta \to W$, let $\mathtt{f}^b$ be a recursive function symbol of type $\beta^\dagger \times C \to 2$. (We may assume that the map $\mathtt{f} \mapsto \mathtt{f}^b$ is injective.)

Definition 33. We define a transformation $T \mapsto T^b$ from RW terms of type W to bit-length terms of type 2 as follows. Here $\mathtt{c}$ is a fixed variable of type C.

- If $T \equiv \mathtt{w}$ or $T \equiv \mathtt{nil}$, then $T^b \equiv \mathtt{false}$,.[11]
- If $T \equiv \mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2$, then $T^b \equiv \mathtt{if}\ T_0^\dagger\ \mathtt{then}\ T_1^b\ \mathtt{else}\ T_2^b$.
- If $T \equiv \mathtt{f}(T')$, then $T^b \equiv \mathtt{f}^b((T')^\dagger, \mathtt{c})$
- If $T \equiv \mathtt{f}(T', S)$, where S has type W, then

  $T^b \equiv \mathtt{if}\ \mathtt{f}^\diamond((T')^\dagger)\ \mathtt{then}\ \mathtt{f}^b((T')^\dagger, \mathtt{c})\ \mathtt{else}\ \mathtt{if}\ \mathtt{c} \le S^\ell\ \mathtt{then}\ S^b\ \mathtt{else}\ \mathtt{f}^b((T')^\dagger, \mathtt{c} - S^\ell)$.
- If $T \equiv \mathtt{cons}(T', S)$, then $T^b \equiv \mathtt{if}\ \mathtt{c} \le S^\ell\ \mathtt{then}\ S^b\ \mathtt{else}\ (T')^\dagger$,

Notice that the only variables that may occur in T^b are $\mathtt{c}$ and $\mathtt{x}^\dagger$ for $\mathtt{x}$ which occur in T, and the only recursive function symbols that may occur are $\mathtt{f}^b$ for $\mathtt{f}$ of output type W, and $\mathtt{f}^\dagger$ for $\mathtt{f}$ of output type non-W, which occur in T.

Remark 11. Not only does this transformation preserve tail-recursive terms, but *every* term it produces is tail-recursive (as long as we regard previously defined subterms like S^ℓ and $\mathtt{f}^\diamond((T')^\dagger)$ as explicit, as usual). No matter what T is, no $\mathtt{f}^b$ will occur inside anything other than a `then`- or `else`-clause in T^b.

Definition 34. Given an RW-factorizable program p of type $R \to W$, we define the bit-length program p^b by extending p^ℓ as follows

For each line of the following form,	add the following
$\mathtt{f}_i(\mathtt{x}_i) = T_i\ (\mathtt{f}_i : \beta \to W)$	$\mathtt{f}_i^b(\mathtt{x}_i^\dagger, \mathtt{c}) = T_i^b$
$\mathtt{f}_i(\mathtt{x}_i, \mathtt{w}) = T_i\ (\mathtt{f}_i : \beta \times W \to W)$	$\mathtt{f}_i^b(\mathtt{x}_i^\dagger, \mathtt{c}) = T_i^b$

Finally, add a new head $\mathtt{h}(\mathtt{c}) = \mathtt{f}_0^b(\mathtt{max}, \mathtt{c})$. The resulting program is p^b. It is tail-recursive if p is.

Let us check that for any RW-factorizable program p of type $R \to W$, p^b is a well-defined bit-length program. It contains the recursive function symbols $\mathtt{f}^b$ for $\mathtt{f}$ in p of type W, and $\mathtt{f}^\dagger$ for $\mathtt{f}$ in p of type non-W. Since the only variable that appears in T is $\mathtt{x}_i$, the only variables that may appear in T^b are $\mathtt{x}_i^\dagger$ and $\mathtt{c}$. Finally, the types of $\mathtt{f}_i^b(\mathtt{x}^\dagger, \mathtt{c})$ and T_i agree.

Notice that p^b contains p^ℓ, and hence $p^\diamond$ and $p^\dagger$, as "subprograms." Hence, the semantics of p^b extends the semantics of the previous programs. We now prove its correctness; the proof is postponed to Appendix A.

Lemma 5. *For every RW program p of type $R \to W$, RW term T from p of type W, strings x, w and v, natural numbers $n > |v|$ and $1 \le c \le |v| - \delta$, and x-environment ρ binding* $\mathtt{w}$ *to w,*

[11] This definition will not be important in practice; rather, it is purely so that the transformation is always well-defined. We could have equally well set $T^b \equiv \mathtt{true}$.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

$$\rho \vdash_p T \to v \implies x, n, \rho^\dagger, [\mathtt{c} = c] \vdash_{p^b} T^b \to v_{c+\delta},$$

where $\delta = 0$ *if* $\rho^\dagger \vdash_\diamond T^\diamond \to \top$ *or* $\delta = |w|$ *if* $\rho^\dagger \vdash_\diamond T^\diamond \to \bot$.[12] *Moreover, the computation on the right-hand side is without collisions.*

Theorem 5. *Suppose that p is an RW-factorizable program of type* $R \to W$ *and suppose that* $\lambda : \omega \to \omega$ *satisfies*

$$(\forall x, w \in 2^\star)\ [\![p]\!](x) = w \implies |w| < \lambda(|x|).$$

Then,

$$(\forall x, w \in 2^\star)\ [\![p]\!](x) = w \implies (\forall c \leq |w|)\ [\![p^b]\!]_\lambda(x, c) = w_c,$$

without collisions.

Proof Fix x, v, and c such that $[\![p]\!](x) = v$ and $1 \leq c \leq |v|$. Suppose that $\mathtt{f}_0(\mathtt{x}_0)$ is the head of p, so that $[\mathtt{x}_0 = x] \vdash_p \mathtt{f}_0(\mathtt{x}_0) \to v$. Therefore, $[\mathtt{x}_0 = x, \mathtt{w} = \varepsilon] \vdash_p \mathtt{f}_0(\mathtt{x}_0) \to v$. By Lemma 5, since $\lambda(|x|) > v$,

$$x, \lambda(|x|), [\mathtt{x}_0^\dagger = x^\dagger, \mathtt{c} = c] \vdash_{p^b} \mathtt{f}_0^b(\mathtt{x}_0^\dagger, \mathtt{c}) \to v_c,$$

without collisions. (Notice that $\delta = 0$ as $\mathtt{w}$ is bound to ε.)

Since $x, \lambda(|x|) \vdash_{p^b} \mathtt{max} \to |x|$ and $|x| = x^\dagger$, we have

$$x, \lambda(|x|), [\mathtt{c} = c] \vdash_{p^b} \mathtt{f}_0^b(\mathtt{max}, \mathtt{c}) \to v_c.$$

But since the head of $\mathtt{h}(\mathtt{c})$ is defined to be $\mathtt{f}_0^b(\mathtt{max}, \mathtt{c})$, this implies $[\![p^b]\!]_\lambda(x, c) = v_c$, which is exactly what we wanted to show. ■

Theorems 4 and 5 immediately imply the following result, the statement of correctness for the translation from RW-factorizable to bit-length programs.

Theorem 6. *Let* $f : 2^\star \to 2^\star$ *be a function,* $\lambda : \omega \to \omega$ *satisfy* $\lambda(|x|) > |f(x)|$, *and p be an RW-factorizable program of type* $R \to W$ *computing f. Then* $,(\lambda, p^\ell, p^b)$ *properly computes f.*

Example. Given the program *id* defined by

$$\begin{aligned} \mathtt{id}(\mathtt{x}) &= \mathtt{cat}(\mathtt{x}, \mathtt{nil}) \\ \mathtt{cat}(\mathtt{x}, \mathtt{w}) &= \mathtt{if\ null}(\mathtt{x})\ \mathtt{then\ w\ else\ cons}(\mathtt{hd\ x}, \mathtt{cat}(\mathtt{tl\ x}, \mathtt{w})), \end{aligned}$$

id^b is defined by

$$\begin{aligned} \mathtt{id}^b(\mathtt{x}^\dagger, \mathtt{c}) &= \mathtt{if\ cat}^\diamond(\mathtt{x}^\dagger)\ \mathtt{then\ cat}^b(\mathtt{x}^\dagger, \mathtt{c})\ \mathtt{else\ if\ c} \leq 0\ \mathtt{then\ false}, \\ &\qquad \mathtt{else\ cat}^b(\mathtt{x}^\dagger, \mathtt{c} - 0) \\ \mathtt{cat}^b(\mathtt{x}^\dagger, \mathtt{c}) &= \mathtt{if\ null}(\mathtt{x}^\dagger)\ \mathtt{then\ false, else\ if\ c} \leq S^\ell\ \mathtt{then}\ S^b\ \mathtt{else\ bit}(\mathtt{x}^\dagger), \end{aligned}$$

[12] Recall that $v_{c+\delta}$ refers to a particular bit of v, where $v = v_{|v|} \ldots v_2 v_1$.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

where $S \equiv \texttt{cat(tl x, w)}$. We know that S^{ℓ} correctly computes the length of $\texttt{tl(x)}$, which is (abusing notation) $\texttt{x}^{\dagger} - 1$. Moreover,

$$S^{b} \equiv \texttt{if cat}^{\diamond}(\texttt{Px}^{\dagger})\ \texttt{then cat}^{b}(\texttt{Px}^{\dagger}, \texttt{c})\ \texttt{else if c} \leq 0\ \texttt{then false, else cat}^{b}(\texttt{Px}^{\dagger}, \texttt{c} - 0).$$

Recall that $\texttt{cat}^{\diamond}$ computes the always-false function. Combining this all into semantically equivalent and legible pseudocode, we get

$$\begin{aligned}
\texttt{id}^{b}(\texttt{x}^{\dagger}, \texttt{c}) &= \texttt{if c} \leq 0\ \texttt{then false, else cat}^{b}(\texttt{x}^{\dagger}, \texttt{c}) \\
\texttt{cat}^{b}(\texttt{x}^{\dagger}, \texttt{c}) &= \texttt{if null}(\texttt{x}^{\dagger}) \vee \texttt{c} \leq 0\ \texttt{then false, else} \\
&\qquad \texttt{if c} \leq \texttt{x}^{\dagger} - 1\ \texttt{then cat}(\texttt{Px}^{\dagger}, \texttt{c})\ \texttt{else bit}(\texttt{x}^{\dagger}).
\end{aligned}$$

We can see that $\texttt{cat}^{b}(\texttt{x}^{\dagger}, \texttt{c})$ computes the bit indexed by $\texttt{c}$: operationally, it decrements the index $\texttt{x}^{\dagger}$ until it is equal to the counting module $\texttt{c}$, then spits out the bit indexed by $\texttt{x}^{\dagger}$. Hence, $[\![id^{b}]\!]_{\lambda}(x, c)$ computes the bit x_c for sufficiently large λ and appropriate values of c.

Observe that for sufficiently large λ, $(\lambda, id^{\ell}, id^{b})$ correctly bit-length computes the identity function, as $[\![id^{\ell}]\!]_{\lambda}(x) = |x|$ and $[\![id^{b}]\!]_{\lambda}(x, c) = x_c$.

7 Compiling BL- to RW-factorizable programs

In this section, we show how to compile a pair of bit-length programs into a single RW-factorizable program, thus establishing extensional equivalence for these two notions of computability, at least for total functions. The core of this section consists of two transformations:

- The more significant of these is a transformation ‡, which acts as a sort of inverse to †: it eliminates indices in favor of string suffixes, i.e., R-data, and counting modules in favor of tuples of R-data. (More precisely, this is a *family* of transformations parameterized by how many R values we need to encode a single counting module.) This process transforms every bit-length program into a cons-free program with a "global input variable."
- The simpler transformation takes a cons-free program with a global input variable into one without. We do this in most naive way possible, simply by passing the global input as an additional parameter to every recursive call.

The former transformation uses a well-known trick of cons-free programming, namely that we can simulate a counting module of size polynomial in the input length by a tuple of suffixes of the input. If we just think of a suffix of the input as encoding its length (i.e., forgetting about its bits), then we can identify it with some single digit $\{0, 1, \ldots, n\}$, where n is the length of the input. Therefore, we can identify a k-tuple of suffixes with some k-digit number in base $n + 1$, which we can identify in turn with some number less than $(n + 1)^{k}$. To implement the full data type of a counting module, it simply suffices to implement the fixed-width arithmetic and comparison operations using cons-free programs. In this way, we can replace polynomially bounded counting modules with fixed-width tuples of R values.

A complication is the fact that our language does not accommodate nested tuples. For example, suppose that we had to replace every counting module of type C by three R values

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

of type $R \times R \times R$. Then if we had a tuple of type $I \times C \times 2$, it *should* be replaced by a tuple of type $(R, R^3, 2)$, but since we don't have nested tuples, we are required to "flatten it out" into a tuple of type $(R, R, R, R, 2)$. This makes indexing harder: to get the element of R^3 encoding the counting module, we have to extract the middle three coordinates.

7.1 Programs with global input

Bit-length programs have a global input value, which is the x in the judgment $x, n, \rho \vdash_p T \to v$. RW-factorizable programs, and cons-free programs in particular, do not. It will be convenient to first transform bit-length programs into cons-free programs with some global input variable $\mathtt{in}$, and then eliminate it, instead of trying to cram both in one transformation.

Definition 35. A *RW term term with global input* is obtained by extending the formation rules of Figure 1 by the additional axiom $\mathtt{in} : R$. A *RW-factorizable program with global input* is defined like an ordinary RW-factorizable program, except that the terms T_i are RW-terms with global input.

The semantics relation $\vdash$ has the form $x, \rho \vdash T \to v$ (note the additional x argument on the left-hand side) and is defined by extending the rules of Figure 2 by the axiom $x, \rho \vdash \mathtt{in} \to x$. If $\mathtt{f}_0(\mathtt{x}_0)$ is the head of a program p, then we define

$$[\![p]\!](x, y) = v \iff x, [\mathtt{x}_0 = y] \vdash_p \mathtt{f}_0(\mathtt{x}_0) \to v.$$

An RW-factorizable program with global input is *cons-free* if it contains no term of type W.

We can easily transform a RW program with global input into one without. The basic idea is to pass the global input explicitly into each recursive function symbol as an extra argument which never gets modified. The remainder of this subsection consists of making this intuition formal. Moreover, we will restrict our attention to cons-free programs, because that is the only case we need.

Definition 36. For any RW product term α, let α_R be $\alpha \times R$. For any RW function term $\rho = \beta \to \alpha$, let ρ_R be $\beta_R \to \alpha$. Define an injection $\mathtt{x} \mapsto \mathtt{x}_R$, $\mathtt{f} \mapsto \mathtt{f}_R$ from variables and recursive function symbols of type α and ρ to type α_R and ρ_R, respectively.

The idea of the next transformation is we replace the variable $\mathtt{x}$ with the variable $\mathtt{x}_R$. The last coordinate of $\mathtt{x}_R$ stores the "global input" which gets passed around to all the recursive functions in the program, and the rest of the variable stores the "original variable" $\mathtt{x}$.

Definition 37. Suppose T is a term in which only the variable $\mathtt{x}$ may occur. (Call this an $\mathtt{x}$-*term* for brevity.) Let n be the length of the type of $\mathtt{x}$. Define the map $T \mapsto T_R^{\mathtt{x}}$ from cons-free terms with global input to cons-free terms as follows (for brevity, we omit the superscript $\mathtt{x}$):

- If $T \equiv \mathtt{x}$, then $T_R \equiv \mathtt{x}_R[0, n-1]$.
- If $T \equiv \mathtt{in}$, then $T_R \equiv \mathtt{x}_R[n]$.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

- If $T \equiv \texttt{if}\ T_0\ \texttt{then}\ T_1\ \texttt{else}\ T_2$ then $T_R \equiv \texttt{if}\ (T_0)_R\ \texttt{then}\ (T_1)_R\ \texttt{else}\ (T_2)_R$.
- If $T \equiv \texttt{hd}(S)$, $\texttt{tl}(S)$, or $\texttt{null}(S)$, then $T_R \equiv \texttt{hd}(S_R)$, $\texttt{tl}(S_R)$, or $\texttt{null}(S_R)$ respectively.
- If $T \equiv T_0 \oplus \cdots \oplus T_{n-1}$, then $T_R \equiv (T_0)_R \oplus \cdots \oplus (T_{n-1})_R$.
- If $T \equiv S[i,j]$, then $T_R \equiv S_R[i,j]$.
- If $T \equiv \texttt{f}(S)$, then $T_R \equiv \texttt{f}_R(S_R \oplus \texttt{x}_R[n])$.

Notice that for any RW-term with global input T containing only $\texttt{x}$, then $T_R^{\texttt{x}}$ is a well-formed RW-term of the same type of T. Moreover, $\texttt{x}_R$ is the only variable which may occur in $T_R^{\texttt{x}}$. This transformation is easily seen to preserve tail recursion.

Definition 38. Suppose the cons-free program with global input p consists of the lines $\texttt{f}_i(\texttt{x}_i) = T_i$ for $0 \leq i \leq k$. Then define the program p_R to consist of lines $(\texttt{f}_i)_R((\texttt{x}_i)_R) = (T_i)_R^{\texttt{x}_i}$ for $0 \leq i \leq k$.

Note that p_R is a well-defined program. It contains the recursive function symbols $(\texttt{f}_i)_R$ for each $\texttt{f}_i$ from p. The type of $(\texttt{x}_i)_R$ is the input type of $(\texttt{f}_i)_R$. The type of $(T_i)_R^{\texttt{x}_i}$ is the type of T_i, which is the output type of $\texttt{f}_i$, which is the output type of $(\texttt{f}_i)_R$. The only variable that may occur in $(T_i)_R$ is $(\texttt{x}_i)_R$, and the only recursive function symbols that may occur are among $((\texttt{f}_0)_R, \ldots, (\texttt{f}_k)_R)$.

In the next lemma, recall that if $(u_0, \ldots, u_{n-1})$ is a decomposition of the value u into atomic values, then by $u \circ x$ we mean $(u_0, \ldots, u_{n-1}, x)$. Its proof is postponed to Appendix A.

Lemma 6. *For any cons-free program with global input p, values x u, and v, variable* $\texttt{x}$*, and* $\texttt{x}$*-term T,*

$$x, [\texttt{x} = u] \vdash T \to v \implies [\texttt{x}_R = u \circ x] \vdash T_R^{\texttt{x}} \to v.$$

Hence,

Theorem 7. *For any cons-free program with global input p with nullary input, string x and value v, if* $[\![p]\!](x) = v$*, then* $[\![p_R]\!](x) = v$.

Proof Let $\texttt{f}_0(\texttt{x})$ be the head of p. Since p has nullary input, $\texttt{x}$ is a variable of type the empty product, and $\texttt{x}_R$ is a variable of type R. If $[\![p]\!](x) = v$, then $x \vdash \texttt{f}_0 \to v$. By Lemma 6, $[\texttt{x}_R = x] \vdash (\texttt{f}_0)_R((\texttt{x}_0)_R) \to v$. But since $(\texttt{f}_0)_R((\texttt{x}_0)_R)$ is the head of p_R, $[\![p_R]\!](x) = v$. ■

7.2 Eliminating indices and counting modules

In this subsection, we show how to compile any bit-length program into a cons-free program with global input. We fix a natural number $k \geq 1$ which is the number of copies of R we want to replace every copy of C by. The transformation ‡ defined in this section should be understood as parameterized by this number k. First we will define a map on types, variables, and terms, then values, then programs, and we conclude with a proof of correctness.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

The main idea: Bit-length programs contain two types of data that RW-factorizable programs do not: indices and counting modules. We replace indices by single R values and counting modules by k-tuples of R values.

The former correspondence is straightforward: if the input string x has length n, then its bits are $x_n \ldots x_2 x_1$. R values range over suffixes of x. We encode the index i by the R value $x_i \ldots x_2 x_1$. The "dummy index" 0 is encoded by the empty string. With respect to this encoding, the index primitives `bit`, `P`, `null`, and `max` correspond to `hd`, `tl`, `null`, and `in` (the global input variable) respectively. The only index primitive that needs to be programmed is `min`, which can be replaced by `f(in)`, where

$$\texttt{f(x)} \equiv \texttt{if null(x) then x else f(tl(x))}.$$

Let $\texttt{zero}_I$ be the term `f(in)`.

If we ignore the characters in an R-string, it simply encodes a *length* ranging from 0 to $n+1$; i.e., a single digit base $n+1$. Hence, a k-tuple of R-data encodes a k-digit number base $n+1$, or alternatively, as a natural number less than $(n+1)^k$. Therefore, k-tuples of R data can be used to simulate counting modules which are polynomially bounded in the length of the input.[13] The counting modules primitives $+$, $-$, and $\leq$ can be replaced by cons-free programs `add`, `minus`, and `less`, which simulate the corresponding primitives on k-tuples of R values by mimicking any common algorithm for fixed-width arithmetic.[14] The cons-free constants 0 and 1 can similarly be replaced with programs `zero` and `one`, which compute the constant-$(0, \ldots, 0, 0)$ and constant-$(0, \ldots, 0, 1)$ functions, respectively.[15]

We assume the existence of these five programs without constructing them and note that they can always be made tail-recursive. In fact, all the recursion we need can be collected into a few important subroutines. Thinking of R values as single digits, we can name the largest digit (by the global input variable `in`) and we can decrement digits (by `tl`). Using these primitives, we can define simple tail-recursive subroutines incrementing a digit, comparing two digits, and naming the digit 0. Then, the programs `add`, `minus`, `less`, `zero`, and `one` are *explicit* in these subroutines; i.e., can be defined in terms of them using no additional recursion.

Definition 39. Let $2^\ddagger = 2$, $I^\ddagger = R$, and $C^\ddagger = R^k$. For every bit-length product type α, define the RW product type $\alpha^\ddagger$ by replacing every copy of I by R, replacing every copy of C by R^k, and flattening. For example:

$$C^\ddagger = R^k$$
$$(2 \times I \times C)^\ddagger = 2 \times R^{k+1}$$
$$(2 \times C \times I \times 2 \times C \times C)^\ddagger = 2 \times R^{k+1} \times 2 \times R^{2k}.$$

Extend this to a map on function types by $(\beta \to \alpha)^\ddagger = \beta^\ddagger \to \alpha^\ddagger$.

[13] Indeed, in the original formulation of Jones (1999), counting modules were introduced as syntactic sugar for fixed-width tuples of R data.

[14] When a sum or difference would overflow its bounds of $(n+1)^k - 1$ or 0, `add` "tops out" at $(n+1)^k - 1$ and `minus` "bottoms out" at 0, respectively, thus mimicking the behavior of $+$ and $-$ on counting modules.

[15] When we say, e.g., constant $(0, \ldots, 0, 0)$-function, the values in the tuple are understood to be R-data.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Finally, for each product type α, define the map $s_\alpha : |\alpha^\ddagger| \to |\alpha|$ by mapping each coordinate of $\alpha^\ddagger$ to the coordinate "which it comes from" in α. For example, $s_C : k \to 1$ is defined by $s_C(i) = 0$ for all $i \in k$; $s_{2\times I\times C} : k+2 \to 3$ and $s_{2\times I\times C}(i)$ is 0 if $i = 0$, 1 if $i = 1$, and 2 otherwise; and

$$s_{2\times C\times I\times 2\times C\times C} : 3k+2 \to 6$$

is defined by

$$s_{2\times C\times I\times 2\times C\times C}(i) = \begin{cases} 0 & \text{if } i = 0 \\ 1 & \text{if } 1 \le i \le k \\ 2 & \text{if } i = k+1 \\ 3 & \text{if } i = k+2 \\ 4 & \text{if } k+3 \le i \le 2k+2 \\ 5 & \text{otherwise} \end{cases}$$

Notice that each such map s_α is a monotone (non-decreasing) surjection.

Definition 40. Fix an injection $\mathtt{x} \mapsto \mathtt{x}^\ddagger$ from bit-length variables to RW variables such that if $\mathtt{x}$ has type α, $\mathtt{x}^\ddagger$ has type $\alpha^\ddagger$. Fix an injection $\mathtt{f} \mapsto \mathtt{f}^\ddagger$ from bit-length recursive function symbols of type ρ to RW function symbols of type $\rho^\ddagger$, for every bit-length function type ρ.

In the next definition, `in` is a fixed variable of type R naming the global input, `zero` and `one` are fixed function symbols of type $C^\dagger$, `add` and `minus` are fixed function symbols of type $(C \times C \to C)^\ddagger$, and `less` is a fixed function symbol of type $(C \times C \to 2)^\ddagger$.

Definition 41. Define a map $T \mapsto T^\dagger$ from bit-length terms to cons-free terms with global input by:

- If $T \equiv \mathtt{true}$ or $T \equiv \mathtt{false}$, then $T^\ddagger \equiv T$.
- If $T \equiv \mathtt{x}$, a variable of type α, then $T^\ddagger \equiv \mathtt{x}^\ddagger$.
- If $T \equiv 0$ then $T^\ddagger \equiv \mathtt{zero}$.
- If $T \equiv 1$ then $T^\ddagger \equiv \mathtt{one}$.
- If $T \equiv T_0 + T_1$ then $T^\ddagger \equiv \mathtt{add}(T_0^\ddagger, T_1^\ddagger)$.
- If $T \equiv T_0 - T_1$ then $T^\ddagger \equiv \mathtt{minus}(T_0^\ddagger, T_1^\ddagger)$.
- If $T \equiv T_0 \le T_1$, then $T^\ddagger \equiv \mathtt{less}(T_0^\ddagger, T_1^\ddagger)$.
- If $T \equiv \mathtt{min}$ then $T^\ddagger \equiv \mathtt{zero}_I$.
- If $T \equiv \mathtt{max}$ then $T^\ddagger \equiv \mathtt{in}$.
- If $T \equiv \mathtt{P}(S)$ then $T^\ddagger \equiv \mathtt{tl}(S^\ddagger)$.
- If $T \equiv \mathtt{null}(S)$ then $T^\ddagger \equiv \mathtt{null}(S^\ddagger)$.
- If $T \equiv \mathtt{bit}(S)$ then $T^\ddagger \equiv \mathtt{hd}(S^\ddagger)$.
- If $T \equiv T_0 \oplus \cdots \oplus T_{n-1}$ then $T^\ddagger \equiv T_0^\ddagger \oplus \cdots \oplus T_{n-1}^\ddagger$.
- If $T \equiv S[i,j]$, let m be the length of S and n the length of $S^\ddagger$. Then, $T^\ddagger \equiv S^\ddagger[\imath, \jmath]$, where the interval $[\imath, \jmath] \subseteq m$ is the s_α-pre-image of $[i,j] \subseteq n$.
- If $T \equiv \mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2$ then $T^\ddagger \equiv \mathtt{if}\ T_0^\ddagger\ \mathtt{then}\ T_1^\ddagger\ \mathtt{else}\ T_2^\ddagger$.
- If $T \equiv \mathtt{f}(S)$, then $T \equiv \mathtt{f}^\ddagger(S^\ddagger)$.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Now we define a map on values. For this definition, recall the bijection

$$f : \underbrace{n \times \cdots \times n}_{k \text{ copies}} \to n^k$$

defined by $(d_0, \ldots, d_{k-1}) \mapsto \sum_{i<k} d_i n^i$. Intuitively, this identifies k-digit numerals base n with the numbers they denote.

Remark 12. As long as `zero`, `one`, `add`, `minus`, and `less` are implemented by tail-recursive programs, this transformation is easily seen to preserve tail recursion.

Definition 42. For any string $x \in 2^\star$, define the map $v \mapsto v^\ddagger$ as follows. If $v \in 2$, then $v_x^\ddagger = v$. If $v \in I(x)$, then $v_x^\ddagger$ is the suffix of x of length $|v|$. If $v \in C((|x|+1)^k)$, then $v_x^\ddagger$ is the unique k-tuple of suffixes $(s_0, \ldots, s_{k-1})$ of x such that

$$v = \sum_{i<k} |s_i|(|x|+1)^i.$$

If $(v_0, \ldots, v_{n-1})$ is a decomposition of v into atomic types, then $v^\ddagger = v_0^\ddagger \circ \cdots \circ v_{n-1}^\ddagger$. Notice that for any values u and v, $(u \circ v)^\ddagger = u^\ddagger \circ v^\ddagger$.

Definition 43. Given a bit-length program $p = (\mathtt{f}_i(\mathtt{x}_i) = T_i)$, the cons-free program with global input $p^\ddagger$ is defined to be $(\mathtt{f}_i^\ddagger(\mathtt{x}_i^\ddagger) = T_i^\ddagger)$.

This program is well-defined: if T_i is an $\mathtt{x}_i$-term, then $T_i^\ddagger$ is an $\mathtt{x}_i^\ddagger$-term. The only recursive function symbols that may occur in $T_i^\ddagger$, besides the previously defined $\mathtt{zero}_I$, `zero`, `one`, `add`, `minus` and `less`, are the $\mathtt{f}_j^\ddagger$. Moreover, it is tail-recursive if p is.

The proof of the next result is postponed to Appendix A.

Lemma 7. *For every bit-length program p, bit-length term T from p, string x, x-environment ρ, and value v,*

$$x, (|x|+1)^k, \rho \vdash_p T \to v \implies x, \rho^\ddagger \vdash_{p^\ddagger} T^\ddagger \to v^\ddagger.$$

Lemma 7 and Theorem 7 have the following corollary. Let $p_R^\ddagger$ be $(p^\ddagger)_R$.

Theorem 8. *For any bit-length program p and polynomially bounded $\lambda : \omega \to \omega$, string x and value v, if $[\![p]\!]_\lambda(x) = v$ without collisions, then $[\![p_R^\ddagger]\!](x) = v^\ddagger$.*

Proof Choose k such that $\lambda(n) < (n+1)^k$ for all n and define $\mu : \omega \to \omega$ by $\mu(n) = (n+1)^k$. Suppose that $[\![p]\!]_\lambda(x) = v$ without collisions. Then, $[\![p]\!]_\mu(x) = v$ by monotonicity of collision-free computation. By Lemma 7 and the definition of $p^\ddagger$, $[\![p^\ddagger]\!](x) = v^\ddagger$. By Theorem 7, $[\![p_R^\ddagger]\!](x) = v^\ddagger$. ■

Example. The following program of type $C \times C \to C$ multiplies its two inputs. (Or rather, the two coordinates of its single input—recall these are `c`[0] and `c`[1].)

$$\mathtt{f(c)} = \mathtt{if\ c[0]} \leq 0\ \mathtt{then}\ 0\ \mathtt{else\ c[1]} + \mathtt{f(c[0]} - 1, \mathtt{c[1])}.$$

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Suppose that $k = 2$, so we replace each occurrence of C by two copies of R. Then, the type of $\mathtt{c}^\ddagger$ is R^4, where the former and latter two copies of R correspond to $\mathtt{c}[0]$ and $\mathtt{c}[1]$, respectively. Hence,

$$\mathtt{f}^\ddagger(\mathtt{c}^\ddagger) = \mathtt{if}\ \mathtt{less}(\mathtt{c}^\ddagger[0,1], \mathtt{zero})\ \mathtt{then}\ \mathtt{zero}\ \mathtt{else}\ \mathtt{add}(\mathtt{c}^\ddagger[2,3], \mathtt{f}^\ddagger(\mathtt{minus}(\mathtt{c}^\ddagger[0,1], \mathtt{one}), \mathtt{c}^\ddagger[2,3])).$$

7.3 Building type-W output

So far we have shown that we can convert any bit-length program p into an equivalent cons-free RW-factorizable program $p_R^\ddagger$. Given a polynomially bounded function λ and bit-length programs p and q such that (λ, p, q) properly computes a function f, how do we use the cons-free programs $q_R^\ddagger$ and $p_R^\ddagger$ to form an RW-factorizable program computing f? (As above, $\ddagger$ is dependent on a parameter k, which we choose large enough so that $(n+1)^k$ dominates λ.)

As usual, the basic idea is straightforward. We compute the length of the output string using $q_R^\ddagger$. We iterate through indices of the output string less than $q_R^\ddagger$ and compute the corresponding bit of the output string using $p_R^\ddagger$. For each of these computed bits, we cons-them on to a variable $\mathtt{w}$, which "accumulates" the eventual output.

If we were to write this in a legible but informal imperative pseudocode, it would look like this:

$$\begin{aligned}
&\mathtt{w} = \mathtt{nil}\\
&\mathtt{for}\ 1 \le \mathtt{c} \le [\![q_R^\ddagger]\!](\mathtt{x})\\
&\quad \mathtt{w} = \mathtt{cons}([\![p_R^\ddagger]\!](\mathtt{x}, \mathtt{c}), \mathtt{w})\\
&\mathtt{return}\ \mathtt{w}
\end{aligned}$$

It is easy to see that this program computes $f(x)$ on input x, as it outputs a string of length $[\![q_R^\ddagger]\!](x)$ whose i-th bit is $[\![p_R^\ddagger]\!](x, i)$. Written slightly more carefully, we get this:

$$\begin{aligned}
\mathtt{out}(\mathtt{x}) &= \mathtt{f}(\mathtt{x}, \mathtt{one}, \mathtt{nil})\\
\mathtt{f}(\mathtt{x}, \mathtt{c}, \mathtt{w}) &= \mathtt{if}\ \mathtt{c} = \mathtt{h}_q(\mathtt{x})\ \mathtt{then}\ \mathtt{w}\ \mathtt{else}\ \mathtt{f}(\mathtt{x}, \mathtt{add}(\mathtt{c}, \mathtt{one}), \mathtt{cons}(\mathtt{h}_p(\mathtt{x}, \mathtt{c}), \mathtt{w})),
\end{aligned}$$

where $\mathtt{h}_p$ and $\mathtt{h}_q$ denote the heads of the programs $p_R^\ddagger$ and $q_R^\ddagger$, respectively. But even here, we have sacrificed some precision for legibility: for example, we must replace $=$ by two occurrences of $\mathtt{less}$, and a term like $\mathtt{f}(\mathtt{x}, 0, \mathtt{nil})$ must be understood as $\mathtt{f}(\mathtt{x} \oplus 0, \mathtt{nil})$.

Thus, we have transformed (λ, p, q) into an RW-factorizable program computing the same function. Moreover, note that it is non-nested, and tail-recursive if p and q are.[16] Combined with Theorem 6, we get the following translation result:

Theorem 9. *For every function $f : 2^\star \to 2^\star$ whose length is polynomially bounded, there is a non-nested RW-factorizable program of type $R \to W$ computing f iff there is a polynomially bounded function λ and bit-length programs p and q such that (λ, p, q) properly computes f.*

[16] Recall that "non-nested" only applied to recursive functions of output type W.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Similarly, there is a tail-recursive RW-factorizable program of type $R \to W$ computing f iff there is a polynomially bounded function λ and tail-recursive bit-length programs p and q such that (λ, p, q) properly computes f.

Theorems 9 and 2 have the following immediate consequence, which is the central result of this paper:

Theorem 10. *For any function $f : 2^\star \to 2^\star$, $f \in$ FP iff f is computed by a non-nested RW-factorizable program of type $R \to W$, and $f \in$ FL iff f is computed by a tail-recursive RW-factorizable program of type $R \to W$.*

A few comments are in order. The proof of this theorem gives us a rather strong normal form for non-nested RW-factorizable programs. Namely, each such program can be written with a single `for` loop on top of purely cons-free subroutines. We know that those cons-free subroutines can be made non-nested in general, and can furthermore be made tail-recursive if the original program was tail-recursive.

It remains an open question which complexity class general (i.e., possibly nested) RW-factorizable programs capture, but it seems to be intermediate between the class of polynomial time and polynomial space functions. (Recall that the length of the output of the latter functions can grow exponentially in the length of the input.) A plausible candidate is functions whose length is computable in polynomial space, but whose bits are computable in polynomial time.

8 Composing bit-length programs

Finally, we turn to the problem of *syntactic composition*: producing a program r such that $[\![r]\!] = [\![p]\!] \circ [\![q]\!]$ given p and q. In most programming languages this is trivial: we combine programs p and q and add a new head $\mathtt{h}_p(\mathtt{h}_q(\mathtt{x}))$, where $\mathtt{h}_p$ and $\mathtt{h}_q$ are the recursive function symbols in the heads of p and q respectively. However, this does not work for RW-factorizable programs of type $R \to W$, as the term $\mathtt{h}_p(\mathtt{h}_q(\mathtt{x}))$ would be ill-typed.

However, since we can transform RW-factorizable programs to and from equivalent pairs of bit-length programs, it suffices to syntactically compose these.[17] As we have remarked above, pairs of bit-length programs *are* compositional in a rough extensional sense: we are given bit- and length-access to the input, and we compute bit- and length-access to the output.

More precisely, suppose we have a triple (λ_f, p_f, q_f) that properly computes a function f and a triple (λ_g, p_g, q_g) that properly computes g. We want to transform p_f and q_f so that they output the bits and length of $f(g(x))$, as opposed to $f(x)$. Namely, we must "reinterpret" the primitives in p_f and q_f so that it returns the bits of $g(x)$ instead of bits of x, and the constant `max` so that it returns the maximum index of $g(x)$ instead of the maximum index of x. Luckily this is exactly what we have in p_g and q_g.

[17] To be completely syntactic when we transform an RW-factorizable program into an equivalent triple (λ, p, q), we must provide a finite description for λ as well. This is not hard: if f is computable by an RW-factorizable program, then its length on inputs of length n is bounded by $a(n+1)^b$, where a is something like the number of distinct terms that occur in the program and b is the maximum length of any tuple.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Finally, we have to find a suitable bound for the size of the counting module in the composed program. We assume λ_f and λ_g are increasing and dominate the identity function. Note that if $|f(x)| \leq \lambda_f(|x|)$ and $|g(x)| \leq \lambda_g(|x|)$, then $|f(g(x))| \leq (\lambda_f \circ \lambda_g)(|x|)$. Hence, we may take $\lambda_f \circ \lambda_g$.

We now present the formal development, which follows a familiar rhythm: a map on types, on values, terms, and programs, followed by a proof of that the program transformation behaves how we want it to. In this section, let us fix:

- functions $f, g : 2^\star \rightharpoonup 2^\star$,
- increasing functions $\lambda_f, \lambda_g : \omega \to \omega$, each dominating the identity function, such that $\lambda_f(|x|) > |f(x)|$ and $\lambda_g(|x|) > |g(x)|$ for all x,
- a function $\mu : \omega \to \omega$ satisfying $\mu \geq \lambda_f \circ \lambda_g$,
- a triple (λ_f, p_f, q_f) properly computing f, and
- a triple (λ_g, p_g, q_g) properly computing g.

Note that the map defined in this section will be denoted by a g in the subscript, e.g., $T \mapsto T_g$. For types, terms, and programs, this g is purely formal; not so for values, where it actually depends on the partial function g.

Definition 44. Let $2_g = 2$, $I_g = C$, and $C_g = C$. Extend this map to a map $\alpha \mapsto \alpha_g$ on product types coordinate-wise, and thence to a map $(\beta \to \alpha) \mapsto (\beta_g \to \alpha_g)$ on function types.

Definition 45. For each product type α, fix an injection $\mathtt{x} \mapsto \mathtt{x}_g : \mathtt{Var}_\alpha \to \mathtt{Var}_{\alpha_g}$ and an injection $\mathtt{f} \mapsto \mathtt{f}_g : \mathtt{RFsymb}_\rho \to \mathtt{RFsymb}_{\rho_g}$.

For the next definition, notice that $|g(x)| \leq \lambda_g(|x|) \leq \lambda_f(\lambda_g(|x|)) \leq \mu(|x|)$ and $\lambda_f(|g(x)|) \leq \lambda_f(\lambda_g(|x|)) \leq \mu(|x|)$, by the assumption that λ_f is increasing and dominates the identity.

Definition 46. For each $x \in 2^\star$, we define a map $v \mapsto v_g : I(g(x)) \to C(\mu(|x|))$ and $C(\lambda_f(|g(x)|)) \to C(\mu(|x|))$ by inclusion as initial segments of ω. Extend this to a map $v \mapsto v_g$ on product types coordinate-wise, fixing boolean values.

For the next definition and henceforth, let $\mathtt{h}_p(\mathtt{c})$ and $\mathtt{h}_q$ be the heads of p_g and q_g. Notice that these are terms of type 2 and C respectively, and that $\mathtt{c}$ is a variable of type C.

Definition 47. Given a term T of type α, we define the term T_g of type α_g as follows:

1. If $T \equiv \mathtt{x}$ (a variable) then $T_g \equiv \mathtt{x}_g$.
2. If $T \equiv \mathtt{f}(T')$, then $T_g \equiv \mathtt{f}_g(T'_g)$.
3. If $T \equiv \mathtt{true}$, $\mathtt{false}$,, $\mathtt{0}$, or $\mathtt{1}$, then $T_g \equiv T$.
4. If $T \equiv T_0 + T_1$, $T_0 - T_1$, or $T_0 \leq T_1$, then $T_g \equiv (T_0)_g + (T_1)_g$, $(T_0)_g - (T_1)_g$, or $(T_0)_g \leq (T_1)_g$ respectively.
5. If $T \equiv \mathtt{min}$, then $T_g \equiv 0$.
6. If $T \equiv \mathtt{max}$, then $T_g \equiv \mathtt{h}_q$.
7. If $T \equiv \mathtt{P}(T')$ or $\mathtt{null}(T')$, then $T_g \equiv T'_g - 1$, or $T'_g \leq 0$, respectively.
8. If $T \equiv \mathtt{bit}(T')$, then $T_g \equiv \mathtt{h}_p(T'_g)$.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

9. If $T \equiv (T_0, \ldots, T_{n-1})$, then $T_g \equiv ((T_0)_g, \ldots, (T_{n-1})_g)$.
10. If $T \equiv T'[i]$, then $T_g \equiv (T'_g)[i]$.
11. If $T \equiv \texttt{if}\ T_0\ \texttt{then}\ T_1\ \texttt{else}\ T_2$, then $T_g \equiv \texttt{if}\ (T_0)_g\ \texttt{then}\ (T_1)_g\ \texttt{else}\ (T_2)_g$.

Definition 48. Given a program $p = (\texttt{f}_i(\texttt{x}_i) = T_i)_{0 \le i \le k}$, let program $p_{(g)}$ be defined by combining the programs p_g and q_g, then adding new lines $((\texttt{f}_i)_g((\texttt{x}_i)_g) = (T_i)_g)_{0 \le i \le k}$ on top. (The head of $p_{(g)}$ is the line $(\texttt{f}_0)_g((\texttt{x}_0)_g) = (T_0)_g$, where $\texttt{f}_0(\texttt{x}_0) = T_0$ is the head of p.)

Lemma 8. *For any program p, environment ρ, string x, term T, and value v,*

$$g(x), \lambda_f(|g(x)|), \rho \vdash_p T \to v \implies x, \mu(|x|), \rho_g \vdash_{p_{(g)}} T_g \to v_g,$$

without collisions, if the derivation of the left-hand side has no collisions.

(The proof is in Appendix A.)

Theorem 11. $(\mu, (p_f)_{(g)}, (q_f)_{(g)})$ *properly computes* $f \circ g$.

Proof Let $\texttt{h}'_p(\texttt{c})$ and $\texttt{h}'_q$ be the left-hand-sides of the heads of p_f and q_f respectively. Fix a string x. By definition of q_f,

$$g(x), \lambda_f(|g(x)|) \vdash_{q_f} \texttt{h}'_q \to |f(g(x))|$$

without collisions. Hence by Lemma 8,

$$x, \mu(|x|) \vdash_{(q_f)_{(g)}} (\texttt{h}'_q)_g \to |f(g(x))|,$$

i.e., $[\![(q_f)_{(g)}]\!](x) = |f(g(x))|$, without collisions.

By definition of p_f, for each $i < |f(g(x))|$,

$$g(x), \lambda_f(|g(x)|), [\texttt{c} = i] \vdash_{p_f} \texttt{h}'_p(\texttt{c}) \to (f(g(x)))_i$$

without collisions. Hence, by Lemma 8,

$$x, \mu(|x|), [\texttt{c}_g = i] \vdash_{(p_f)_g} (p_f^0)_g(\texttt{c}_g) \to (f(g(x)))_i,$$

i.e., $[\![(p_f)_{(g)}]\!](x, i) = (f(g(x)))_i$, without collisions. ■

Example. Let q_g be the program

$$\texttt{h}_q = \texttt{q(max)}$$
$$\texttt{q(x)} = \texttt{if null(x) then 0 else 1 + q(P(x))},$$

and let p_g be the program

$$\texttt{h}_p\texttt{(x)} = \texttt{if bit(x) then false, else true}.$$

In other words, for sufficiently large λ, (λ, p_g, q_g) computes the function which flips the bits of the input string. Let us *square* this function, i.e., create an identical pair of programs (p_f, q_f), distinguishing the function symbols of the latter by replacing, e.g., q by q′, and

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

then composing (p_f, q_f) with (p_g, q_g). We get for the length component:

$$\begin{aligned} \mathtt{h}'_q &= \mathtt{q}'(\mathtt{h}_q) \\ \mathtt{q}'(\mathtt{x}_g) &= \mathtt{if}\ \mathtt{x}_g \leq 0\ \mathtt{then}\ 0\ \mathtt{else}\ 1 + \mathtt{q}'(\mathtt{x}_g - 1) \\ \mathtt{h}_q &= \mathtt{q}(\mathtt{max}) \\ \mathtt{q}(\mathtt{x}) &= \mathtt{if}\ \mathtt{null}(\mathtt{x})\ \mathtt{then}\ 0\ \mathtt{else}\ 1 + \mathtt{q}(\mathtt{P}(\mathtt{x})), \end{aligned}$$

and for the bits component,

$$\begin{aligned} \mathtt{h}'_p(\mathtt{x}_g) &= \mathtt{if}\ \mathtt{h}_p(\mathtt{x}_g)\ \mathtt{then}\ \mathtt{false},\ \mathtt{else}\ \mathtt{true} \\ \mathtt{h}_p(\mathtt{x}) &= \mathtt{if}\ \mathtt{bit}(\mathtt{x})\ \mathtt{then}\ \mathtt{false},\ \mathtt{else}\ \mathtt{true}. \end{aligned}$$

By inspection, we can see that this pair of programs computes the identity function, as it should.

9 Discussion and open questions

We have identified RW-factorizable programs as a simple extension of cons-free programs that captures functional polynomial time and logarithmic space, and we have also shown how to compose two such programs. We introduced the notion of bit-length computability as auxiliary machinery to help us show these results. Our work suggests a number of subsequent questions:

1. **Higher orders, non-determinism, and other data types.**
   Cons-free programs have been studied in all of these contexts. Jones (2001) showed that deterministic cons-free programs at higher orders capture the deterministic exponential hierarchy, whose union is the class of elementary relations. Kop & Simonsen (2017) showed that whereas non-deterministic first-order cons-free programs are no more expressive than deterministic ones, even non-deterministic second-order programs capture the entire class of elementary relations. Ben-Amram & Petersen (1998) studied cons-free programs over tree data. Does the RW-factorizable paradigm extend smoothly to all of these situations?
2. **RW-factorizable algorithms**
   Many algorithms seem to be RW-factorizable, in the sense that they have read-only input and write-only output (allowing for some flexibility in what we mean by "read" and "write"). For example, both selection sort and insertion sort seem to follow this paradigm: both algorithms have an input list which gets whittled down over the course of computation while the output list is simultaneously built up. Mergesort, on the other hand, is definitely *not* RW-factorizable: whereas *merge* can be consistently typed by $R \to W$, the recursive definition $sort(u) = merge(sort(u_0), sort(u_1))$ forces $sort(u)$ to be of type R and W simultaneously.
   Are there general techniques or results that apply to RW-factorizable algorithms in general, for example, stronger lower bounds? More generally, can we consider other ways to separate construction from destruction that just variable typing? (e.g., in heapsort we separate them *chronologically*: the same piece of data is first constructed, then destructed.)

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

3. **RW-factorizable programs as an indexing of polynomial-time functions.**
If we fix an encoding of RW-factorizable programs by binary strings, we get an *indexing* of polynomial-time functions, i.e., an identification of strings with functions. Indexings of partial recursive functions are the central topic in recursion theory, but indexings of complexity classes (or *subrecursive indexings*) have received a little bit of attention in the context of proving, e.g., speedup phenomena for non-Turing complete languages (Constable & Borodin, 1972; Alton, 1980).[18] Kozen (1978) has identified several simple axioms of subrecursive indexings with significant consequences, for example, Kleene's second recursion theorem. One of these axioms is the existence of a polynomial-time function which gives a code for $f \circ g$ given codes for f and g. It seems likely that the transformations in this paper can be computed in polynomial time, and that the natural indexing of FP given by RW-factorizable programs satisfies this and the other axioms.
This suggests that we might look for speedup phenomena within the class of RW-factorizable programs itself. Given that cons-free running time corresponds to something like (the exponential of) circuit depth, such speedups might have complexity-theoretic consequences (Bhaskar et al., 2022).

Acknowledgements

We are immensely grateful to Andrzej Filinski, Neil Jones, and Jakob Nordström for their comments on this work. This paper is dedicated to the memory of Neil Jones.

Conflicts of interest

None.

References

Alton, D. A. (1980) "Natural" programming languages and complexity measures for subrecursive programming languages: An abstract approach. In *Recursion Theory, its Generalisations and Applications*. London Mathematical Society Lecture Note Series. Cambridge University Press, pp. 248–285. doi:10.1007/CBO9780511629181.011.

Aubert, C., Seiller, T., Rubiano, T. & Rusch, N. (2022) mwp-Analysis Improvement and Implementation: Realizing Implicit Computational Complexity. In 7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2–5, 2022, Haifa, Israel, Felty, A. P. (ed). LIPIcs, vol. 228. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 26:1–26:23.

Avanzini, M., Eguchi, N. & Moser, G. (2011) A path order for rewrite systems that compute exponential time functions. In Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30–June 1, 2011, Novi Sad, Serbia, Schmidt-Schauß, M. (ed). LIPIcs, vol. 10. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 123–138.

Avanzini, M., Eguchi, N. & Moser, G. (2015) A new order-theoretic characterisation of the polytime computable functions. *Theoret. Comput. Sci.* **585**, 3–24.

[18] These are qualitatively different objects: non-pathological indexings of partial recursive functions always contain a *universal program*, whereas indexings of complexity classes never contain one.

Avanzini, M. & Moser, G. (2016) A combination framework for complexity. *Inf. Comput.* **248**, 22–55.

Baillot, P., Barthe, G. & Lago, U. D. (2019) Implicit computational complexity of subrecursive definitions and applications to cryptographic proofs. *J. Autom. Reason.* **63**(4), 813–855.

Baillot, P. & Ghyselen, A. (2022) Types for complexity of parallel computation in Pi-calculus. *ACM Trans. Program. Lang. Syst.* **44**(3), 15:1–15:50.

Bellantoni, S. & Cook, S. (1992) A new recursion-theoretic characterization of the polytime functions (extended abstract). In Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing (STOC '92). Association for Computing Machinery, pp. 283–293. isbn: 0897915119.

Ben-Amram, A. M. & Petersen, H. (1998) CONS-free programs with tree input (extended abstract). In Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13–17, 1998, Proceedings, Larsen, K. G., Skyum, S. & Winskel, G. (eds). Lecture Notes in Computer Science, vol. 1443. Springer, pp. 271–282.

Bhaskar, S., Kop, C. & Simonsen, J. G. (2022) Subclasses of Ptime Interpreted by Programming Languages. *Theory Comput. Syst*.

Bonfante, G. (2006) Some programming languages for logspace and Ptime. In Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 5–8, 2006, Proceedings, pp. 66–80.

Bonfante, G., Marion, J.-Y. & Moyen, J.-Y. (2011) Quasi-interpretations a way to control resources. *Theoret. Comput. Sci.* **412**(25), 2776–2796.

Cobham, A. (1965) The intrinsic computational difficulty of functions. In Logic, Methodology and Philosophy of Science: Proceedings of the 1964 International Congress (Studies in Logic and the Foundations of Mathematics), Bar-Hillel, Y. (ed). North-Holland Publishing, pp. 24–30.

Constable, R. L. & Borodin, A. B. (1972) Subrecursive programming languages, part I. *J. ACM* **19**(3), 526–568.

Czajka, L. (2018) Term rewriting characterisation of LOGSPACE for finite and infinite data. In 3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9–12, 2018, Oxford, UK, pp. 13:1–13:19.

Dal Lago, U. (2022) Implicit computation complexity in higher-order programming languages: A Survey in Memory of Martin Hofmann. *Math. Struct. Comput. Sci.*, 1–17. doi:10.1017/S0960129521000505.

de Carvalho, D. & Simonsen, J. G. (2014) An implicit characterization of the polynomial-time decidable sets by cons-free rewriting. In Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14–17, 2014. Proceedings, pp. 179–193.

Girard, J.-Y. (1998) Light linear logic. *Inf. Comput.* **143**(2), 175–204.

Greibach, S. A. (1975) *Theory of Program Structures: Schemes, Semantics, Verification*. Lecture Notes in Computer Science, vol. 36. Springer.

Hainry, E., Kapron, B. M., Marion, J.-Y. & Péchoux, R. (2022) A tier-based typed programming language characterizing Feasible Functionals. *Log. Methods Comput. Sci.* **18**(1), 1–31.

Hofmann, M. (2000) Programming languages capturing complexity classes. *SIGACT News* **31**(1), 31–42.

Hofmann, M. (2002) The strength of non-size increasing computation. In Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'02. Portland, Oregon: Association for Computing Machinery, pp. 260–269. isbn: 1581134509.

Jones, N. D. (1999) LOGSPACE and PTIME characterized by programming languages. *Theoret. Comput. Sci.* **228**(1), 151–174.

Jones, N. D. (2001) The expressive power of higher-order types or, life without CONS. *J. Funct. Program.* **11**(1), 5–94.

Jones, N. D., Kop, C., Bhaskar, S. & Simonsen, J. G. (2020) Cons-free programs and complexity classes between LOGSPACE and PTIME. In Proceedings 8th International Workshop on Verification and Program Transformation and 7th Workshop on Horn Clauses for Verification and

Synthesis, VPT/HCVS@ETAPS 2020 2020, and 7th Workshop on Horn Clauses for Verification and Synthesis Dublin, Ireland, 25–26th April 2020, Fribourg, L. & Heizmann, M. (eds), vol. 320. EPTCS, pp. 65–79.

Kop, C. & Simonsen, J. G. (2017) The power of non-determinism in higher-order implicit complexity - characterising complexity classes using non-deterministic cons-free programming. In Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22–29, 2017, Proceedings, Yang, H. (ed). Lecture Notes in Computer Science, vol. 10201. Springer, pp. 668–695.

Kozen, D. (1978) Indexing of subrecursive classes. In Proceedings of the Tenth Annual ACM Symposium on Theory of Computing. STOC'78. San Diego, California, USA: Association for Computing Machinery, pp. 287–295.

Kristiansen, L. (2022) Reversible computing and implicit computational complexity. *Sci. Comput. Program.* **213**, 102723.

Lafont, Y. (2004) Soft linear logic and polynomial time. *Theor. Comput. Sci.* **318**(1–2), 163–180.

Lago, U. D., Kahle, R. & Oitavem, I. (2021) A recursion-theoretic characterization of the probabilistic class PP. In 46th International Symposium on Mathematical Foundations of Computer Science, MFCS 2021, August 23–27, 2021, Tallinn, Estonia, Bonchi, F. & Puglisi, S. J. (eds). LIPIcs, vol. 202. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 35:1–35:12.

Lago, U. D. & Toldin, P. P. (2015) A higher-order characterization of probabilistic polynomial time. *Inf. Comput.* **241**, 114–141.

Leivant, D. (1995) Ramified recurrence and computational complexity I: Word recurrence and poly-time. In *Feasible Mathematics II. Birkhäuser Boston*, pp. 320–343.

Marion, J. (2011) A type system for complexity flow analysis. In 2011 IEEE 26th Annual Symposium on Logic in Computer Science (LICS 2011), pp. 123–132.

Moschovakis, Y. N. (2018) *Abstract Recursion and Intrinsic Complexity.* Lecture Notes in Logic. Cambridge University Press.

Appendix A: Proofs of correctness

In this section, we collect all the proofs of correctness for transformations defined earlier.

Lemma (Lemma 2)**.** *Suppose p is a cons-free program, x is a string, ρ is an x-environment. Then,*

$$\rho \vdash_p T \to v \implies x, \rho^\dagger \vdash_{p^\dagger} T^\dagger \to v^\dagger.$$

Proof The proof is by induction on the length of the derivation of $\rho \vdash T \to v$. Abbreviate $x, \rho^\dagger \vdash_{p^\dagger}$ by $\rho^\dagger \vdash$.

- If $T \equiv \mathtt{true}$ or $T \equiv \mathtt{false}$,, then $v = \top$ or $v = \bot$, and $v^\dagger = v$ and $T^\dagger \equiv T$.
- If $T \equiv \mathtt{x}$, then $T^\dagger \equiv \mathtt{x}^\dagger$, and hence, $\rho^\dagger \vdash T^\dagger \to \rho(\mathtt{x}^\dagger)$, which is $v^\dagger$.
- If $T \equiv \mathtt{hd}(S)$, then let u satisfy $\rho \vdash S \to u$. Then by the suffix property, u is a suffix of x, so the head of u is the bit $x_{|u|}$ and $\rho \vdash T \to x_{|u|}$. By induction, $\rho^\dagger \vdash S \to u^\dagger$ so $\rho^\dagger \vdash \mathtt{bit}(S^\dagger) \to x_{u^\dagger}$, but $u^\dagger = |u|$.
- If $T \equiv \mathtt{tl}(S)$, then there exists a u and $b \in 2$ such that $v = bu$ and $\rho \vdash S \to u$. By induction, $\rho^\dagger \vdash S^\dagger \to |u|$. Hence, $\rho^\dagger \vdash \mathtt{P}(S^\dagger) \to |u| - 1$.
- If $T \equiv \mathtt{null}(S)$, then v is true or false depending on whether S is the empty string. Similarly, $\mathtt{null}(S^\dagger)$ is true or false depending on whether $S^\dagger$ is zero, the length of the empty string.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

- If $T \equiv T_0 \oplus \cdots \oplus T_{n-1}$, then there exist $v_0, \ldots, v_{n-1}$ such that $\rho \vdash T_i \to v_i$ for each $i < n$ and $v = v_0 \circ \cdots \circ v_{n-1}$. By induction, $\rho^\dagger \vdash T_i^\dagger \to v_i^\dagger$ for each $i < n$. Since $v^\dagger = v_0^\dagger \circ \cdots \circ v_{n-1}^\dagger$, $\rho^\dagger \vdash T_0^\dagger \oplus \cdots \oplus T_{n-1}^\dagger \to v^\dagger$; hence, $\rho^\dagger \vdash T^\dagger \to v^\dagger$.
- If $T \equiv S[i]$, then there exists a value u such that $\rho \vdash S \to u$, $(u_0, \ldots, u_{n-1})$ is a decomposition of u into atomic values, and $v = u_i$. By induction, $\rho^\dagger \vdash S^\dagger \to u^\dagger$, where $u^\dagger = (u_0^\dagger, \ldots, u_{n-1}^\dagger)$. Thus, $\rho^\dagger \vdash S^\dagger[i] \to u_i^\dagger$, which says $\rho^\dagger \vdash T^\dagger \to v^\dagger$.
- If $T \equiv \texttt{if}\ T_0\ \texttt{then}\ T_1\ \texttt{else}\ T_2$, assume $\rho \vdash T_0 \to \top$. (The case $\rho \vdash T_0 \to \bot$ is similar.) Then, $\rho \vdash T_1 \to v$. By induction, $\rho^\dagger \vdash T_0^\dagger \to \top$ and $\rho^\dagger \vdash T_1^\dagger \to v^\dagger$. Hence, $\rho^\dagger \vdash T \to v^\dagger$.
- If $T \equiv \texttt{f}(S)$, then let $\texttt{f}(\texttt{x}^\texttt{f}) = T^\texttt{f}$ be the definition of $\texttt{f}$ in p. Then there exists some u such that $\rho \vdash S \to u$ and $[\texttt{x}^\texttt{f} = u] \vdash T^\texttt{f} \to v$. By induction, $\rho^\dagger \vdash S^\dagger \to u^\dagger$ and $[(\texttt{x}^\texttt{f})^\dagger = u^\dagger] \vdash (T^\texttt{f})^\dagger \to v^\dagger$. Hence ,$\rho^\dagger \vdash \texttt{f}^\dagger(S^\dagger) \to v^\dagger$, and $\rho^\dagger \vdash T^\dagger \to v^\dagger$. ■

Lemma (Lemma 3)**.** *Suppose that p is an RW-factorizable program, T is an RW-term, $x, v, v', w \in 2^\star$, ρ is an x-environment, and $\texttt{w}$ is not bound by ρ.*

Suppose that $\rho, [\texttt{w} = w] \vdash_p T \to v$ and $\rho, [\texttt{w} = \varepsilon] \vdash_p T \to v'$.[19] *Then either*

- *$x, \rho^\dagger \vdash_{p^\diamond} T^\diamond \to \top$ and $v = v'$, or*
- *$x, \rho^\dagger \vdash_{p^\diamond} T^\diamond \to \bot$ and $v = v'w$.*

Proof Abbreviate $\rho, [\texttt{w} = w] \vdash_p T \to v$ by $[\rho, w] \vdash T \to v$, and similarly abbreviate $\rho, [\texttt{w} = \varepsilon] \vdash_p T \to v$ by $[\rho, \varepsilon] \vdash T \to v$. Abbreviate $x, \rho^\dagger \vdash_{p^\diamond} T^\diamond \to b$ by $\rho^\dagger \vdash_\diamond T^\diamond \to b$. The proof proceeds by induction on the size of the derivation of $[\rho, w] \vdash T \to v$ and breaks up into cases depending on the form of T.

- If $T \equiv \texttt{w}$, then $v = w$, $\rho^\dagger \vdash_\diamond T^\diamond \to \bot$, and $v' = \varepsilon$. In this case, $v = w = \varepsilon w = v'w$, so we're done.
- If $T \equiv \texttt{nil}$, then $v = \varepsilon$, $\rho^\dagger \vdash_\diamond T^\diamond \to \top$, and $v' = \varepsilon$. In this case $v = \varepsilon = v'$, so we're done.
- If $T \equiv \texttt{if}\ T_0\ \texttt{then}\ T_1\ \texttt{else}\ T_2$, first suppose that $\rho \vdash T_0 \to \top$. By Lemma 2, $\rho^\dagger \vdash_\diamond T_0^\dagger \to \top$. By definition of T, $[\rho, w] \vdash T_1 \to v$ and $[\rho, \varepsilon] \vdash T_1 \to v'$. By induction, either $\rho^\dagger \vdash_\diamond T_1^\diamond \to \top$ or $\rho^\dagger \vdash_\diamond T_1^\diamond \to \bot$.
  Suppose that $\rho^\dagger \vdash_\diamond T_1^\diamond \to \top$. By induction $v = v'$. By definition of $T^\diamond$, $\rho^\dagger \vdash_\diamond T^\diamond \to \top$. Similarly, suppose that $\rho^\dagger \vdash_\diamond T^\diamond \to \bot$. By induction, $v = v'w$. By definition of $T^\diamond$, $\rho^\dagger \vdash_\diamond T^\diamond \to \bot$, and we're done.
  (The case $\rho \vdash T_0 \to \bot$ is similar.)
- Suppose that $T \equiv \texttt{f}(T')$ for some $\texttt{f} : \beta \to W$. Then, the variable $\texttt{w}$ does not occur in T, so $v' = v$. Since $T^\diamond \equiv \texttt{true}$, $\rho^\dagger \vdash_\diamond T^\diamond \to \top$, and we are done.
- Suppose that $T \equiv \texttt{f}(T', S)$ for some $\texttt{f} : \beta \times W \to W$. Suppose that $\texttt{f}(\texttt{x}^\texttt{f}, \texttt{w}) = T^\texttt{f}$ is the recursive definition of $\texttt{f}$ in p. Then there exist t and s such that $\rho \vdash T' \to t$, $[\rho, w] \vdash S \to s$, and $[\texttt{x}^\texttt{f} = t, \texttt{w} = s] \vdash T^\texttt{f} \to v$, and there exists s' such that $[\rho, \varepsilon] \vdash S \to s'$ and $[\texttt{x}^\texttt{f} = t, \texttt{w} = s'] \vdash T^\texttt{f} \to v'$. By Lemma 2, $\rho^\dagger \vdash (T')^\dagger \to t^\dagger$. Since convergence is independent of the binding of $\texttt{w}$, there exists a v'' such that $[\texttt{x}^\texttt{f} = t, \texttt{w} = \varepsilon] \vdash T^\texttt{f} \to v''$.

[19] Note that the convergence of the term T only depends on the environment ρ and not on the binding of the variable $\texttt{w}$.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Now

$$[\mathtt{x}^{\mathtt{f}} = t, \mathtt{w} = s] \vdash T^{\mathtt{f}} \to v \text{ and } [\mathtt{x}^{\mathtt{f}} = t, \mathtt{w} = \varepsilon] \vdash T^{\mathtt{f}} \to v''.$$

By induction,

$$\left([(\mathtt{x}^{\mathtt{f}})^\dagger = t^\dagger] \vdash_\diamond (T^{\mathtt{f}})^\diamond \to \top \wedge v = v''\right) \vee \left([(\mathtt{x}^{\mathtt{f}})^\dagger = t^\dagger] \vdash_\diamond (T^{\mathtt{f}})^\diamond \to \bot \wedge v = v''s\right).$$

Similarly,

$$[\mathtt{x}^{\mathtt{f}} = t, \mathtt{w} = s'] \vdash T^{\mathtt{f}} \to v' \text{ and } [\mathtt{x}^{\mathtt{f}} = t, \mathtt{w} = \varepsilon] \vdash T^{\mathtt{f}} \to v''.$$

By induction,

$$\left([(\mathtt{x}^{\mathtt{f}})^\dagger = t^\dagger] \vdash_\diamond (T^{\mathtt{f}})^\diamond \to \top \wedge v' = v''\right) \vee \left([(\mathtt{x}^{\mathtt{f}})^\dagger = t^\dagger] \vdash_\diamond (T^{\mathtt{f}})^\diamond \to \bot \wedge v' = v''s'\right).$$

First assume $[(\mathtt{x}^{\mathtt{f}})^\dagger = t^\dagger] \vdash_\diamond (T^{\mathtt{f}})^\diamond \to \top$. Then, $v = v'' = v'$ and (by definition of $T^\diamond$) $\rho^\dagger \vdash_\diamond T^\diamond \to \top$, which completes this case.
Next assume then that $[(\mathtt{x}^{\mathtt{f}})^\dagger = t^\dagger] \vdash_\diamond (T^{\mathtt{f}})^\diamond \to \bot$. In this case, $v = v''s$ and $v' = v''s'$. Now

$$[\rho, w] \vdash S \to s \text{ and } [\rho, \varepsilon] \vdash S \to s'.$$

Then by induction,

$$\left(\rho^\dagger \vdash_\diamond S^\diamond \to \top \wedge s = s'\right) \vee \left(\rho^\dagger \vdash_\diamond S^\diamond \to \bot \wedge s = s'w\right).$$

If $\rho^\dagger \vdash_\diamond S^\diamond \to \top$, then $v = v''s = v''s' = v'$ and $\rho^\dagger \vdash_\diamond T^\diamond \to \top$ (by definition of $T^\diamond$). If, on the other hand, $\rho^\dagger \vdash_\diamond S^\diamond \to \bot$, then $v = v''s = v''s'w = v'w$ and $\rho^\dagger \vdash_\diamond T^\diamond \to \bot$ (by definition of $T^\diamond$). This finishes the case that $T \equiv \mathtt{f}(T', S)$.

- If $T \equiv \mathtt{cons}(T', S)$, then there are strings s and s' and a character $c \in 2$ such that $[\rho, w] \vdash S \to s$, $[\rho, \varepsilon] \vdash S \to s'$, $v = cs$, and $v' = cs'$. By induction, either $\rho^\dagger \vdash_\diamond S^\diamond \to \top$ and $s = s'$ or $\rho^\dagger \vdash_\diamond S^\diamond \to \bot$ and $s = s'w$. In the first case, $\rho^\dagger \vdash_\diamond T^\diamond \to \top$ and $v = cs = cs' = v'$. In the second case, $\rho^\dagger \vdash_\diamond T^\diamond \to \bot$ and $v = cs = cs'w = v'w$.

■

Lemma (Lemma 4)**.** *For every RW program p of type $R \to W$, RW term T from p of type W, strings x, v, and w, natural number $n > |v|$, and x-environment ρ binding $\mathtt{w}$ to w:*

$$\rho \vdash_p T \to v \implies x, n, \rho^\dagger \vdash_{p^\ell} T^\ell \to v^\ell - \delta,$$

where $\delta = 0$ if $\rho^\dagger \vdash_\diamond T^\diamond \to \top$ or $\delta = |w|$ if $\rho^\dagger \vdash_\diamond T^\diamond \to \bot$. Moreover, the derivation of the right-hand side is collision-free.

Proof Let $\rho \vdash T \to v$ abbreviate $\rho \vdash_p T \to v$ and $\rho \vdash_\ell T \to v$ abbreviate $x, n, \rho \vdash_{p^\ell} T \to v$. The proof proceeds by induction on the derivation of $\rho \vdash T \to v$ and breaks up into cases depending on the form of T. To verify that the derivation is collision-free, we note that any time we add counting modules we do not introduce a new collision. Note that since p^ℓ contains $p^\diamond$, $\vdash_\ell$ extends $\vdash_\diamond$.

- If $T \equiv \mathtt{w}$, then $v = w$ and $v^\ell = |w|$. Moreover, $\rho^\dagger \vdash_\ell T^\diamond \to \bot$, so $\delta = |w|$ and $v^\ell - \delta = 0$. But in this case, $T^\ell \equiv 0$.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

- If $T \equiv \mathtt{nil}$, then $v = \varepsilon$. Moreover, $\rho^\dagger \vdash_\ell T^\diamond \to \top$, so $\delta = \varepsilon$ and $v^\ell - \delta = 0$. But in this case, $T^\ell \equiv 0$ as well.
- If $T \equiv \mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2$, assume $\rho \vdash T_0 \to \top$ (the case $\bot$ is similar). By Lemma 2, $\rho^\dagger \vdash_\ell T_0^\dagger \to 1$, so by definition of $T^\diamond$, $\rho^\dagger \vdash_\ell T^\diamond \to \top \iff \rho^\dagger \vdash_\ell T_1^\diamond \to \top$. Therefore, the δ associated with T_1 is identical to the δ associated with T, and we can simply say δ unambiguously.
  Since $\rho \vdash T_0 \to 1$, $\rho \vdash T_1 \to v$, so (by induction) $\rho^\dagger \vdash_\ell T_1^\ell \to v^\ell - \delta$. Since $\rho^\dagger \vdash_\ell T_0^\dagger \to \top$, $\rho^\dagger \vdash_\ell T^\ell \to v^\ell - \delta$ by definition of T^ℓ.
- If $T \equiv \mathtt{f}(T')$, let $\mathtt{f}(\mathtt{x^f}, \mathtt{w}) = T^\mathtt{f}$ be the definition of $\mathtt{f}$ in p. There exists a value t such that $\rho \vdash T' \to t$ and $[\mathtt{x^f} = t] \vdash T^\mathtt{f} \to v$. By induction, $[(\mathtt{x^f})^\dagger = t^\dagger] \vdash_\ell (T^\mathtt{f})^\ell \to v^\ell$. Hence, $\rho^\dagger \vdash_\ell \mathtt{f}^\ell((T')^\dagger) \to v^\ell$, and $\rho^\dagger \vdash_\ell T^\ell \to v^\ell$. (In this case, $\rho^\dagger \vdash_\ell T^\diamond \to \top$.)
- If $T \equiv \mathtt{f}(T', S)$, let $\mathtt{f}(\mathtt{x^f}, \mathtt{w}) = T^\mathtt{f}$ be the definition of $\mathtt{f}$ in p. There exist values t and s such that $\rho \vdash T' \to t$, $\rho \vdash S \to s$, and $[\mathtt{x^f} = t, \mathtt{w} = s] \vdash T^\mathtt{f} \to v$.
  Let us first suppose that $\rho^\dagger \vdash_\diamond \mathtt{f}^\diamond((T')^\dagger) \to \top$. By Lemma 2, $\rho^\dagger \vdash_\ell (T')^\dagger \to t^\dagger$. Since the recursive definition of $\mathtt{f}^\diamond$ is $\mathtt{f}^\diamond((\mathtt{x^f})^\dagger) = (T^\mathtt{f})^\diamond$, we have that $[(\mathtt{x^f})^\dagger = t^\dagger] \vdash_\ell (T^\mathtt{f})^\diamond \to \top$. By induction, $[(\mathtt{x^f})^\dagger = t^\dagger] \vdash_\ell (T^\mathtt{f})^\ell \to v^\ell$; hence, $\rho^\dagger \vdash_\ell \mathtt{f}^\ell((T')^\dagger) \to v^\ell$. By definition of T^ℓ, $\rho^\dagger \vdash_\ell T^\ell \to v^\ell$, which is what we wanted to show.
  Now suppose that $\rho^\dagger \vdash_\diamond \mathtt{f}^\diamond((T')^\dagger) \to \bot$. By definition of $T^\diamond$, $\rho^\dagger \vdash_\ell T^\diamond \to b$ iff $\rho^\dagger \vdash_\ell S^\diamond \to b$ for both boolean values b. Therefore, since $\mathtt{w}$ is bound to the same value (w) in S as in T, the value of δ can be used unambiguously in both the inductive hypothesis at S and the conclusion at T.
  Moreover, $[(\mathtt{x^f})^\dagger = t^\dagger] \vdash_\ell (T^\mathtt{f})^\diamond \to \bot$. Thus, by the inductive hypothesis applied to $[\mathtt{x^f} = t, \mathtt{w} = s] \vdash T^\mathtt{f} \to v$, $[(\mathtt{x^f})^\dagger = t^\dagger] \vdash_\ell (T^\mathtt{f})^\ell \to v^\ell - |s|$, and therefore, $\rho^\dagger \vdash_\ell \mathtt{f}^\ell((T')^\dagger) \to v^\ell - |s|$. By the inductive hypothesis at $\rho \vdash S \to s$, $\rho^\dagger \vdash_\ell S^\ell \to s^\ell - \delta$. As $s^\ell = |s|$, $\rho^\dagger \vdash_\ell \mathtt{f}^\ell(T^\dagger) + S^\ell \to v^\ell - \delta$. (Note that this uses the assumption that $n > |v|$, which also verifies that this addition does not introduce a new collision.)
  Finally, since $\rho^\dagger \vdash_\diamond \mathtt{f}^\diamond((T')^\dagger) \to \bot$ and by definition of T^ℓ, $\rho^\dagger \vdash_\ell T^\ell \to v^\ell - \delta$, which is what we wanted to show.
- If $T \equiv \mathtt{cons}(T', S)$, then there exists a character $c \in 2$ and a string s such that $\rho \vdash T' \to b$, $\rho \vdash S \to s$, and $v = cs$. As in the previous case, the values of δ at S and T are identical. By induction, $\rho^\dagger \vdash_\ell S^\ell \to |s| - \delta$, so $\rho^\dagger \vdash_\ell T^\ell \to |s| + 1 - \delta$, which is $|v| - \delta$. (We again use the assumption that $n > |v|$ here, which again verifies that this addition does not introduce a new collision.) ■

Lemma (Lemma 5)**.** *For every RW program p of type $R \to W$, RW term T from p of type W, strings x, w and v, natural numbers $n > |v|$ and $1 \le c \le |v| - \delta$, and x-environment ρ binding $\mathtt{w}$ to w,*

$$\rho \vdash_p T \to v \implies x, n, \rho^\dagger, [\mathtt{c} = c] \vdash_{p^b} T^b \to v_{c+\delta},$$

where $\delta = 0$ if $\rho^\dagger \vdash_\diamond T^\diamond \to \top$ or $\delta = |w|$ if $\rho^\dagger \vdash_\diamond T^\diamond \to \bot$.[20] *Moreover, the computation on the right-hand side is without collisions.*

[20] Recall that $v_{c+\delta}$ refers to a particular bit of v, where $v = v_{|v|} \dots v_2 v_1$.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Proof Let $\rho \vdash T \to v$ abbreviate $\rho \vdash_p T \to v$ and $[\rho^\dagger, c] \vdash_b T^b \to a$ abbreviate $x, n, \rho^\dagger, [\mathtt{c} = c] \vdash_{p^b} T^b \to a$. The proof is by induction on the size of the derivation of $\rho \vdash T \to v$ and breaks up into cases depending on the form of T.

- If $T \equiv \mathtt{w}$ or $T \equiv \mathtt{nil}$, then $|v| - \delta = 0$, so the universal quantification over c is empty and the conclusion is vacuously true.
- If $T \equiv \mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2$, then suppose $\rho \vdash T_0 \to \top$ (the case $\rho \vdash T_0 \to \bot$ is similar). Then $\rho \vdash T_1 \to v$. By Lemma 2, $\rho^\dagger \vdash_\ell T_0^\dagger \to \top$, and hence $\rho^\dagger \vdash_\ell T^\diamond \to b$ if and only if $\rho^\dagger \vdash_\ell T_1^\diamond \to b$.
  Let us apply the induction hypothesis to $\rho \vdash T_1 \to v$. The value of δ at the induction hypothesis is identical to the value of δ for $\rho \vdash T \to v$, so by induction $[\rho^\dagger, c] \vdash_b T_1^b \to v_{c+\delta}$, and by definition of T^b, $[\rho^\dagger, c] \vdash_b T^b \to v_{c+\delta}$.
- If $T \equiv \mathtt{f}(T')$, let $\mathtt{f}(\mathtt{x}^\mathtt{f}) = T^\mathtt{f}$ be the definition of $\mathtt{f}$ in p. Then, there exists a value t such that $\rho \vdash T' \to t$ and $[\mathtt{x}^\mathtt{f} = t] \vdash T^\mathtt{f} \to v$. In this case, $\rho^\dagger \vdash T^\diamond \to \top$, so $\delta = 0$. By Lemma 2, $\rho^\dagger \vdash_b (T')^\dagger \to t^\dagger$.
  Since $[\mathtt{x}^\mathtt{f} = t] \vdash T^\mathtt{f} \to v$, $[\mathtt{x}^\mathtt{f} = t, \mathtt{w} = \varepsilon] \vdash T^\mathtt{f} \to v$. By induction on the latter, $[(\mathtt{x}^\mathtt{f})^\dagger = t^\dagger, \mathtt{c} = c] \vdash_b (T^\mathtt{f})^b \to v_c$. (Notice that the denotation of $(T^\mathtt{f})^\diamond$ is irrelevant if $|\mathtt{w}| = 0$.) Hence, $[\rho^\dagger, c] \vdash_b \mathtt{f}((T')^\dagger, \mathtt{c}) \to v_c$, i.e., $[\rho^\dagger, c] \vdash_b T^b \to v_c$.
- If $T \equiv \mathtt{f}(T', S)$, let $\mathtt{f}(\mathtt{x}, \mathtt{w}) = T^\mathtt{f}$ be the definition of $\mathtt{f}$ in p. Then, there are values (t, s) such that $\rho \vdash T' \to s$, $\rho \vdash S \to s$, and $[\mathtt{x}^\mathtt{f} = t, \mathtt{w} = s] \vdash T^\mathtt{f} \to v$.
  First let us suppose that $\rho^\dagger \vdash \mathtt{f}^\diamond((T')^\dagger) \to \top$. In this case, $\rho^\dagger \vdash_b T^\diamond \to \top$, so $\delta = 0$. Moreover, $[(\mathtt{x}^\mathtt{f})^\dagger = t^\dagger] \vdash_b (T^\mathtt{f})^\diamond \to \top$, so by induction applied to $T^\mathtt{f}$, $[(\mathtt{x}^\mathtt{f})^\dagger = t^\dagger, \mathtt{c} = c] \vdash_b (T^\mathtt{f})^b \to v_c$, and hence, $[\rho^\dagger, c] \vdash_b \mathtt{f}^b((T')^\dagger, \mathtt{c}) \to v_c$. But by definition of T^b, $[\rho^\dagger, c] \vdash_b T^b \to v_c$.
  Next let us suppose that $\rho^\dagger \vdash \mathtt{f}^\diamond((T')^\dagger) \to \bot$. In this case, $[(\mathtt{x}^\mathtt{f})^\dagger] \vdash_b (T^\mathtt{f})^\diamond \to \bot$, and the denotation of $T^\diamond$ is equivalent to that of $S^\diamond$. Since it is the same environment ρ at T and S as well, the value of $\delta = |\rho(\mathtt{w})|$ is identical at both S and T. Therefore, by induction applied to $\rho \vdash S \to s$, $[\rho^\dagger, c] \vdash_b S^b \to s_{c+\delta}$ for any $1 \le c \le |s| - \delta$. By Lemma 3, since $[\mathtt{x}^\mathtt{f} = t, \mathtt{w} = s] \vdash T^\mathtt{f} \to v$, s appears as a suffix of v, and hence for $1 \le c \le s - |\delta|$, $s_{c+\delta} = v_{c+\delta}$. Hence for any $1 \le c \le |s| - \delta$, $[\rho^\dagger, c] \vdash_b S^b \to v_{c+\delta}$.
  By induction applied to $[\mathtt{x}^\mathtt{f} = t, \mathtt{w} = s] \vdash T^\mathtt{f} \to v$, $[(\mathtt{x}^\mathtt{f})^\dagger, \mathtt{c} = c] \vdash_b (T^\mathtt{f})^b \to v_{c+|s|}$, and hence $[\rho^\dagger, c] \vdash_b \mathtt{f}^b((T')^\dagger, \mathtt{c}) \to v_{c+|s|}$, for any $1 \le c \le |v| - |s|$. Said another way, for any $|s| - \delta + 1 \le c \le |v| - \delta$, $[\rho^\dagger, c - |s| + \delta] \vdash_b \mathtt{f}^b((T')^\dagger, \mathtt{c}) \to v_{c+\delta}$. By Lemma 4, $\rho^\dagger \vdash_b S^\ell \to |s| - \delta$. Hence for any $|s| - \delta + 1 \le c \le |v| - \delta$, $[\rho^\dagger, c] \vdash_b \mathtt{f}^b((T')^\dagger, \mathtt{c} - S^\ell) \to v_{c+\delta}$.
  Since $\rho^\dagger \vdash_b S^\ell \to |s| - \delta$, $[\rho^\dagger, c] \vdash_b \mathtt{c} \le S^\ell \to \top$ just in case $c \le |s| - \delta$, otherwise $[\rho^\dagger, c] \vdash_b \mathtt{c} \le S^\ell \to \bot$. We have now shown that for all $1 \le c \le |v| - \delta$,

$$[\rho^\dagger, c] \vdash_b \mathtt{if}\ \mathtt{c} \le S^\ell\ \mathtt{then}\ S^b\ \mathtt{else}\ \mathtt{f}^b((T')^\dagger, \mathtt{c} - S^\ell) \to v_{c+\delta}.$$

  Since we are in the case $\rho^\dagger \vdash \mathtt{f}^\diamond((T')^\dagger) \to \bot$, we have now shown that $[\rho^\dagger, c] \vdash_b T^b \to v_{c+\delta}$, which is what we wanted to show.
- If $T \equiv \mathtt{cons}(T', S)$, then $T^\diamond$ and $S^\diamond$ are identical, so we can use δ unambiguously at both T and S as in the previous case. Let $\rho \vdash S \to s$. By induction, for any $1 \le c \le |s| - \delta$, $[\rho^\dagger, c] \vdash S^b \to v_{c+\delta}$. If $c = |v| - \delta$, then $v_{c+\delta}$ is the first character of v, so $\rho \vdash T' \to v_{c+\delta}$. By Lemma 2, $\rho^\dagger \vdash T' \to v_{c+\delta}$ ($v_{c+\delta} = v_{c+\delta}^\dagger$ as it is a boolean value).

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Since $|v| - \delta = (|s| - \delta) + 1$ and $[\rho^\dagger, c] \vdash_b \mathtt{c} \leq S^\ell \to \top$ if and only if $c \leq |s| - \delta$, otherwise $[\rho^\dagger, c] \vdash_b \mathtt{c} \leq S^\ell \to \bot$, then for any $1 \leq c \leq |v| - \delta$,

$$[\rho^\dagger, c] \vdash_b \mathtt{if\ c} \leq S^\ell \mathtt{\ then\ } S^b \mathtt{\ else\ } (T')^\dagger \to v_{c+\delta};$$

hence $[\rho^\dagger, c] \vdash_b T^b \to v_{c+\delta}$, which is what we wanted to show.

That there are no collisions stems from the fact that the transformation $T \mapsto T^b$ introduces no new occurrences of counting module addition that were not already contained in the transformation $T \mapsto T^\ell$. ■

Lemma (Lemma 6)**.** *For any cons-free program with global input p, values x u, and v, variable* x*, and* x*-term T,*

$$x, [\mathtt{x} = u] \vdash T \to v \implies [\mathtt{x}_R = u \circ x] \vdash T^{\mathtt{x}}_R \to v.$$

Proof As usual, the proof proceeds by induction on the size of the derivation of the hypothesis and breaks up into cases according to the form of T. Let us abbreviate $T^{\mathtt{x}}_R$ by T_R.

- If $T \equiv \mathtt{x}$, then $v = u$, and $[\mathtt{x}_R = u \circ x] \vdash \mathtt{x}_R[0, n-1] \to u$.
- If $T \equiv \mathtt{in}$, then $v = x$, and $[\mathtt{x}_R = u \circ x] \vdash \mathtt{x}_R[n] \to x$.
- If $T \equiv \mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2$, suppose that $x, [\mathtt{x} = u] \vdash T_0 \to \top$. (The case $x, [\mathtt{x} = u] \vdash T_0 \to \bot$ is similar.) Then, $x, [\mathtt{x} = u] \vdash T_1 \to v$, so by induction, $[\mathtt{x}_R = u \circ x] \vdash (T_0)_R \to \top$ and $[\mathtt{x}_R = u \circ x] \vdash (T_1)_R \to v$. Therefore, $[\mathtt{x}_R = u \circ x] \vdash T_R \to v$.
- Suppose that $T \equiv \varphi(S)$, where $\varphi \in \{\mathtt{hd}, \mathtt{tl}, \mathtt{null}\}$. Then, there exists some v' such that $x, [\mathtt{x} = u] \vdash S \to v'$ and $\varphi(v') = v$. By induction, $[\mathtt{x}_R = u \circ x] \vdash S_R \to v'$. Therefore, $[\mathtt{x}_R = u \circ x] \vdash T_R \to v$.
- Suppose that $T \equiv T_0 \oplus \cdots \oplus T_{n-1}$. Then, there exist $v_0, \ldots, v_{n-1}$ such that $x, [\mathtt{x} = u] \vdash T_i \to v_i$ for each $i < n$ and $v = v_0 \circ \cdots \circ v_{n-1}$. By induction, for each $i < n$, $[\mathtt{x}_R = u \circ x] \vdash T_i \to v_i$, so $[\mathtt{x}_R = u \circ x] \vdash T \to v$.
- If $T \equiv S[i, j]$, there is some v' such that $x, [\mathtt{x} = u] \vdash S \to v'$. Let $(v_0, \ldots, v_{n-1})$ be a decomposition of v' into a tuple of atomic values; then, $v = (v_i, \ldots, v_j)$. By induction, $[\mathtt{x}_R = u \circ x] \vdash S_R \to (v_0, \ldots, v_{n-1})$, so $[\mathtt{x}_R = u \circ x] \vdash S_R[i, j] \to (v_i, \ldots, v_j)$, i.e., $[\mathtt{x}_R = u \circ x] \vdash T_R \to v$.
- Finally, suppose $T \equiv \mathtt{f}(S)$. Let $\mathtt{f}(\mathtt{x}^\mathtt{f}) = T^\mathtt{f}$ be the definition of $\mathtt{f}$ in p. Then, there is some v' such that $x, [\mathtt{x} = u] \vdash S \to v'$ and $x, [\mathtt{x}^\mathtt{f} = v'] \vdash T^\mathtt{f} \to v$. By induction, $[\mathtt{x}_R = u \circ x] \vdash S_R \to v'$ and $[\mathtt{x}^\mathtt{f}_R = v' \circ x] \vdash (T^\mathtt{f})^{\mathtt{x}^\mathtt{f}}_R \to v$. Since $[\mathtt{x}_R = u \circ x] \vdash S_R \to v'$, $[\mathtt{x}_R = u \circ x] \vdash S_R \oplus \mathtt{x}_R[n] \to v' \circ x$.
  But $\mathtt{f}_R(\mathtt{x}^\mathtt{f}_R) = (T^\mathtt{f})^{\mathtt{x}^\mathtt{f}}_R$ is the recursive definition of $\mathtt{f}_R$ in p_R. Hence, $[\mathtt{x}_R = u \circ x] \vdash \mathtt{f}_R(S_R \oplus \mathtt{x}_R[n]) \to v$, which is what we wanted to prove. ■

Lemma (Lemma 7)**.** *For every bit-length program p, bit-length term T from p, string x, x-environment ρ, and value v,*

$$x, (|x| + 1)^k, \rho \vdash_p T \to v \implies x, \rho^\ddagger \vdash_{p^\ddagger} T^\ddagger \to v^\ddagger.$$

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Proof The proof is by induction on the length of x, $(|x|+1)^k, \rho \vdash_p T \to v$ and then breaks up into cases depending on the form of T. Abbreviate $x, (|x|+1)^k, \rho \vdash_p T \to v$ by $\rho \vdash T \to v$ and $x, \rho^\dagger \vdash_{p^\ddagger} T^\ddagger \to v^\ddagger$ by $\rho^\ddagger \vdash T^\ddagger \to v^\ddagger$.

- If $T \equiv \texttt{true}$ or $\texttt{false}$,, then $v = v^\ddagger = \top$ or $\bot$, and $T^\ddagger \equiv \texttt{true}$ or $\texttt{false}$, respectively.
- If $T \equiv \texttt{x}$, then $v = \rho(x)$, so $v^\ddagger = \rho^\ddagger(\texttt{x}^\ddagger)$, $T^\ddagger \equiv \texttt{x}^\ddagger$, and $\rho^\ddagger \vdash T^\ddagger \to v^\ddagger$.
- If $T \equiv 0$ then $v = 0$ (as a counting module), $v^\ddagger = (0, \ldots, 0, 0)$ (as an element of R^k), and $T^\ddagger \equiv \texttt{zero}$. By correctness of $\texttt{zero}$, $\rho^\ddagger \vdash T^\ddagger \to v^\ddagger$.
- The case $T \equiv 1$ is similar to the previous one, replacing $(0, \ldots, 0, 0)$ by $(0, \ldots, 0, 1)$ and $\texttt{zero}$ by $\texttt{one}$.
- If $T \equiv T_0 + T_1$, then $T^\ddagger \equiv \texttt{add}(T_0^\ddagger, T_1^\ddagger)$. By induction, there exist u_0 and u_1 such that $\rho \vdash T_i \to u_i$ for $i < 2$ and $v = \max\{u_0 + u_1, (|x|+1)^k - 1\}$. By induction, $\rho^\ddagger \vdash T_i^\ddagger \to u_i^\ddagger$ for $i < 2$. By correctness of $\texttt{add}$, $[\texttt{u}_0 = u_0^\ddagger, \texttt{u} = u_1^\ddagger] \vdash \texttt{add}(\texttt{u}_0, \texttt{u}_1) \to v^\ddagger$. Hence, $\rho^\ddagger \vdash T^\ddagger \to v^\ddagger$.
- The cases $T \equiv T_0 - T_1$ and $T \equiv T_0 \leq T_1$ are similar to the previous one, by the correctness of $\texttt{minus}$ and $\texttt{less}$ respectively.
- If $T \equiv \texttt{min}$, then $v = 0$ (as an index), $v^\ddagger = \varepsilon$, and $T^\ddagger = \texttt{zero}_I$. By correctness of $\texttt{zero}_I$, $\rho^\ddagger \vdash T^\ddagger \to v^\ddagger$.
- If $T \equiv \texttt{max}$, then $v = |x|$, $v^\ddagger = x$, $T^\ddagger \equiv \texttt{in}$, so $x, \rho^\ddagger \vdash \texttt{in} \to x$ implies $x, \rho^\ddagger \vdash T^\ddagger \to v^\ddagger$.
- If $T \equiv \texttt{P}(S)$, then there exists some u such that $\rho \vdash S \to u$ and $u - 1 = v$. In this case, $v^\ddagger$ is the tail of $u^\ddagger$; by induction, $\rho^\ddagger \vdash S^\ddagger \to u^\ddagger$, and thus, $\rho^\ddagger \vdash T^\ddagger \to v^\ddagger$.
- If $T \equiv \texttt{null}(S)$, then there exists some u such that $\rho \vdash S \to u$ and $v = \texttt{true} \iff u = 0$. In this case $v^\ddagger = v$ and $u^\ddagger = \varepsilon \iff u = 0$; by induction, $\rho^\ddagger \vdash S^\ddagger \to u^\ddagger$, and thus $\rho^\ddagger \vdash T^\ddagger \to v^\ddagger$.
- If $T \equiv \texttt{bit}(S)$, then there exists some u such that $\rho \vdash S \to u$ and $x_u = v$. In this case, $v^\ddagger = v$ and $u^\ddagger$ is a suffix of x with head $x_u = v = v^\ddagger$; by induction, $\rho^\ddagger \vdash S^\ddagger \to u^\ddagger$, and thus, $\rho^\ddagger \vdash T^\ddagger \to v^\ddagger$.
- If $T \equiv T_0 \oplus \cdots \oplus T_{n-1}$, then there exist $v_0, \ldots, v_{n-1}$ such that $\rho \vdash T_i \to v_i$ for each $i < n$; by induction, $\rho^\ddagger \vdash T_i^\ddagger \to v_i^\ddagger$ and $\rho^\ddagger \vdash T^\ddagger \to v_0^\ddagger \circ \cdots \circ v_{n-1}^\ddagger$. But this is exactly $v^\ddagger$.
- If $T \equiv S[i,j]$, then there exists a u such that $\rho \vdash S \to u$. Let $(u_0, \ldots, u_{m-1})$ be a decomposition of u into atomic values; then $v = u_i \circ \ldots u_j$. Now $\rho^\ddagger \vdash S^\ddagger \to u^\ddagger$ by induction, and $u^\ddagger = u_0^\ddagger \circ \cdots \circ u_{m-1}^\ddagger$. Suppose that $(t_0, \ldots, t_{n-1})$ is a decomposition of $u^\ddagger$ into atomic values. By definition of s_α, where α is the type of S, $t_\imath \circ \cdots \circ t_\jmath = u_i^\ddagger \circ \ldots u_j^\ddagger = v^\ddagger$. Hence, $\rho^\ddagger S^\ddagger[\imath, \jmath] \to v^\ddagger$, which is what we wanted to show.
- If $T \equiv \texttt{if } T_0 \texttt{ then } T_1 \texttt{ else } T_2$, then suppose that $\rho \vdash T_0 \to \top$. (The case $\rho \vdash T_0 \to \bot$ is similar.) Then, $\rho \vdash T_1 \to v$. By induction, $\rho^\ddagger T_0^\ddagger \to \top$ and $\rho^\ddagger \vdash T_1^\ddagger \to v^\ddagger$; hence, $\rho^\ddagger \vdash T^\ddagger \to v^\ddagger$.
- If $T \equiv \texttt{f}(S)$, let $\texttt{f}(\texttt{x}^\texttt{f}) = T^\texttt{f}$ be the recursive definition of $\texttt{f}$ in p. Then, there exists a u such that $\rho \vdash S \to u$ and $[\texttt{x}^\texttt{f} = u] \vdash T^\texttt{f} \to v$. By induction $\rho^\ddagger \vdash S^\ddagger \to u^\ddagger$ and $[(\texttt{x}^\texttt{f})^\ddagger = u^\ddagger] \vdash (T^\texttt{f})^\ddagger \to v^\ddagger$. Since $\texttt{f}^\ddagger((\texttt{x}^\texttt{f})^\ddagger) = (T^\texttt{f})^\ddagger$ is the definition of $\texttt{f}^\ddagger$ in $p^\ddagger$, $\rho^\ddagger \vdash \texttt{f}^\ddagger(S^\ddagger) \to v^\ddagger$. ■

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

Lemma (Lemma 8)**.** *For any program p, environment ρ, string x, term T, and value v,*

$$g(x), \lambda_f(|g(x)|), \rho \vdash_p T \to v \implies x, \mu(|x|), \rho_g \vdash_{p_{(g)}} T_g \to v_g,$$

without collisions, if the derivation of the left-hand side has no collisions.

Proof For legibility, we will abbreviate $g(x), \lambda_f(|g(x)|), \rho \vdash_p$ by $\rho \vdash$ and $x, \mu(|x|), \rho_g \vdash_{p_{(g)}}$ by $\rho_g \vdash_g$. The proof proceeds by induction on the size of the derivation of $\rho \vdash T \to v$ and is broken into cases depending on the form of T.

1. If $T \equiv \mathtt{x}$, a single variable, then $T_g \equiv \mathtt{x}_g$, and $v = \rho(\mathtt{x})$, so $v_g = \rho(\mathtt{x}_g)$. Hence, $\rho_g \vdash_g T_g \to v_g$.
2. If $T \equiv \mathtt{f}(T')$, let $\mathtt{f}(\mathtt{x}) = T^{\mathtt{f}}$ be the recursive definition of $\mathtt{f}$ in p. Then, there exists a w such that $\rho \vdash T' \to w$ and $[\mathtt{x} = w] \vdash T^{\mathtt{f}} \to v$. By induction, $\rho_g \vdash_g T'_g \to w_g$ and $[\mathtt{x}_g = w_g] \vdash T^{\mathtt{f}}_g \to v_g$.
   But $\mathtt{f}_g(\mathtt{x}_g) = T^{\mathtt{f}}_g$ is the recursive definition of $\mathtt{f}_g$ in p_g. Hence, $\rho_g \vdash_g \mathtt{f}_g(T'_g) \to v_g$.
   But $T_g \equiv \mathtt{f}_g(T'_g)$, so $\rho_g \vdash_g T_g \to v_g$.
3. If $T \equiv \mathtt{true}$, $\mathtt{false}$,, 0, or 1, then the conclusion is immediate.
4. If $T \equiv \varphi(T_0, T_1)$, for $\varphi \in \{+, -, \leq\}$, then there exists v_0, v_1, such that $\rho \vdash T_i \to v_i$ for $i < 2$ and $\varphi(v_0, v_1) = v$ in $C(\lambda_f(|g(x)|))$. Since, by assumption, there are no collisions, $\varphi(v_0, v_1) = v$ in $C(\mu(|x|))$. Note that $v_i = (v_i)_g$ and $v = v_g$, when we identify $C(\lambda_f(|g(x)|))$ as a subset of $C(\mu(|x|))$.
   By induction, $\rho_g \vdash_g (T_i)_g \to (v_i)_g$. Since $T_g \equiv \varphi((T_0)_g, (T_1)_g)$, $\rho_g \vdash T_g \to v_g$. Since $C(\mu(|x|))$ contains $C(\lambda_f(|g(x)|)$, no new collisions are introduced.
5. If $T \equiv \mathtt{min}$, then v is the index 0, so $T_g \equiv 0$, and v_g is $0 \in C(\mu(|x|))$. Hence, $\rho_g \vdash_g T_g \to v_g$.
6. If $T \equiv \mathtt{max}$, then v is the index $|g(x)|$, so $v_g = |g(x)| \in C(\mu(|x|))$.
   Since $x, \lambda_g(|x|) \vdash_{q_g} \mathtt{h}_q \to |g(x)|$ with no collisions, $x, \mu(|x|) \vdash_{q_g} \mathtt{h}_q \to |g(x)|$. Since q_g is a fragment of p_g, $x, \mu(|x|) \vdash_{p_g} \mathtt{h}_q \to |g(x)|$, i.e., $\vdash_g \mathtt{h}_q \to |g(x)|$. Hence, $\vdash_g \mathtt{h}_q \to v_g$, but $T_g \equiv \mathtt{h}_q$, so $\vdash_g T_g \to v_g$.
7. If $T \equiv \varphi(T')$ for $\varphi \in \{\mathtt{P}, \mathtt{null}\}$, then there exists an index w of $g(x)$ such that $\rho \vdash T' \to w$ and $v = \varphi(w)$. By induction, there exists some $w_g \in C(\mu(|x|))$ such that $v_g = w_g - 1$ (if $\varphi = \mathtt{P}$) or $v_g \iff w_g = 0$ (if $\varphi = \mathtt{null}$), and $\rho_g \vdash_g T'_g \to w_g$. In the former case, $\rho_g \vdash_g T'_g - 1 \to v_g$ and in the latter, $\rho_g \vdash_g (T'_g) \to v_g$; in either case, $\rho_g \vdash T_g \to v_g$.
8. If $T \equiv \mathtt{bit}(T')$, then there is some w such that $\rho \vdash T' \to w$ and $(g(x))_{w-1} = v$. Notice that $v_g = v$ and w_g is natural number equivalent to w in the counting module $C(\mu(|x|))$. If we identify $C(\lambda_g(|x|))$ as a subtype of $C(\mu(|x|))$, w_g becomes a member of $C(\lambda_g(|x|))$ too. Then,

   $$x, \lambda_g(|x|), [\mathtt{c} = w_g] \vdash_{p_g} \mathtt{h}_p(\mathtt{c}) \to v,$$

   moreover with no collisions. Hence,

   $$x, \mu(|x|), [\mathtt{c} = w_g] \vdash_{p_g} \mathtt{h}_p(\mathtt{c}) \to v,$$

   and therefore $[\mathtt{c} = w_g] \vdash_g \mathtt{h}_p(\mathtt{c}) \to v$.
   By induction, $\rho_g \vdash T'_g \to w_g$. Hence, $\rho_g \vdash \mathtt{h}_p(T'_g) \to v$; as $T_g \equiv \mathtt{h}_p(T'_g)$ and $v = v_g$, this is what we wanted to show.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

9. If $T \equiv (T_0, \ldots, T_{n-1})$, then $v = (v_0, \ldots, v_{n-1})$, where $\rho \vdash T_i \to v_i$. By induction, $\rho_g \vdash (T_i)_g \to (v_i)_g$, and $\rho_g \vdash T_g \to v_g$, as $T_g \equiv ((T_0)_g, \ldots, (T_{n-1})_g)$ and $v_g = ((v_0)_g, \ldots, (v_{n-1})_g)$.
10. If $T \equiv T'[i]$, then there exists some $(v_0, \ldots, v_{n-1})$ such that $v = v_i$ and $\rho \vdash T' \to (v_0, \ldots, v_{n-1})$. By induction, $\rho_g \vdash_g T'_g \to ((v_0)_g, \ldots, (v_{n-1})_g)$, and therefore, $\rho_g \vdash_g T_g \to (v_i)_g$.
11. If $T \equiv \texttt{if}\ T_0\ \texttt{then}\ T_1\ \texttt{else}\ T_2$, then there exists some $b \in 2$ such that $\rho \vdash T_0 \to b$ and $\rho \vdash T_b \to v$. By induction, $\rho_g \vdash_g (T_0)_g \to b$ (as $b = b_g$) and $\rho_g \vdash (T_b)_g \to v_g$. Therefore, since $T_g \equiv \texttt{if}\ (T_0)_g\ \texttt{then}\ (T_1)_g\ \texttt{else}\ (T_2)_g$, $\rho_g \vdash T_g \to v_g$. ■

Appendix B: Bit-length programs and Turing machines

In this section, we sketch a proof of Theorem 2. We split the recursive/polynomial time correspondence and the tail-recursive/logarithmic space correspondence into two different parts, as they require different constructions. We try to give just enough detail so that these are perspicuous and unburdened by excess notation.

Theorem. *For any function $f : 2^\star \to 1^\star$, the following are equivalent:*

- *f is computable in polynomial time.*
- *There is a polynomially bounded $\lambda : \omega \to \omega$ and a bit-length program p such that $[\![p]\!]_\lambda(x) = f(x)$, without collisions, for any $x \in 2^\star$.*

For any function $f : 2^\star \times 1^\star \to 2$ and polynomially bounded function $\pi : \omega \to \omega$, the following are equivalent:

- *There is a polynomial-time computable function $g : 2^\star \times 1^\star \to 2$ such that $g(x, y) = f(x, y)$ for any string $x \in 2^\star$ and $y < \pi(|x|)$.*
- *There is a polynomially bounded function $\lambda : \omega \to \omega$ and a bit-length program p such that $[\![p]\!]_\lambda(x, y) = f(x, y)$, without collisions, for any $x \in 2^\star$ and $y < \pi(|x|)$.*

Proof Let f be a function of type $2^\star \times (1^\star)^k \to \alpha$ for some $k \in \omega$ and $\alpha \in \{1^\star, 2\}$. (This generalizes both cases above.) Suppose that f is computable in polynomial time, and let $\mathcal{M}$ be a Turing machine witnessing as much. We may assume that $\mathcal{M}$ has read-only input tapes, a one-way write-only output tape if the output alphabet is $1^\star$, and any finite number of work tapes, each with tape alphabet Σ containing a blank character. All tapes of $\mathcal{M}$ extend infinitely to the right; the heads of $\mathcal{M}$ are initialized to the left; the work and output tapes are initialized to all blank.

We now describe a bit-length program simulating $\mathcal{M}$. The global input variable is identified with the single input of type $2^\star$. The tuple $\vec{\mathtt{x}}$ of counting module variables is identified with the remaining unary inputs; these keep getting passed around, unchanged, in each recursive call. For each tape of $\mathcal{M}$ we define a recursive functions $\mathtt{position} : C \to C$ and $\mathtt{character} : C \times C \to \Sigma$. The function $\mathtt{position}(\mathtt{t}, \vec{\mathtt{x}})$ describes the position of the head after $\mathtt{t}$ steps, and $\mathtt{character}(\mathtt{t}, \mathtt{p}, \vec{\mathtt{x}})$ describes the character on that tape after $\mathtt{t}$ steps at position $\mathtt{p}$. In addition, there is a single recursive function $\mathtt{state}(\mathtt{t}, \vec{\mathtt{x}})$ describing the state

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

of $\mathcal{M}$ after $\mathtt{t}$ steps. The codomain of $\mathtt{state}$ is a finite type, which we can encode by a sufficiently wide tuple of booleans.

The *surface configuration* of $\mathcal{M}$ at a given time $\mathtt{t}$, consists of the current state ($\mathtt{state}(\mathtt{t}, \vec{\mathtt{x}})$) and the characters read by the heads ($\mathtt{character}(\mathtt{t}, \mathtt{position}(\mathtt{t}), \vec{\mathtt{x}})$ for each tape). The surface configuration tells us what state $\mathcal{M}$ transitions to, how the heads move, and what they write in each step. (Exactly how this happens is specified by the transition function of $\mathcal{M}$.) More precisely, for $\mathtt{t} > 0$, $\mathtt{state}(\mathtt{t}, \vec{\mathtt{x}})$ is given by the previous surface configuration, $\mathtt{position}(\mathtt{t}, \vec{\mathtt{x}}) = \mathtt{position}(\mathtt{t} - 1, \vec{\mathtt{x}}) + \delta$, where $\delta = 0, 1,$ or -1 depending on the previous surface configuration, and $\mathtt{character}(\mathtt{t}, \mathtt{p}, \vec{\mathtt{x}})$ is equal to $\mathtt{character}(\mathtt{t} - 1, \mathtt{p}, \vec{\mathtt{x}})$ unless $\mathtt{position}(\mathtt{t} - 1) = \mathtt{p}$, in which case it is given by the previous surface configuration.

The initial position of each head is $\mathtt{0}$. The initial characters all work and output tapes are blank, and the initial characters of the input tapes are given by the $\vec{\mathtt{x}}$. Once $\mathcal{M}$ reaches a halting state, the state, positions, and characters of the program remain fixed. Therefore, to name the "final" state, tape positions, etc., we need a function $\mathtt{seed} : C$ which is big enough to bound the largest possible running time on some given input.

Notice that we can always "copy" an index value to a counting module value in a bit-length program. In particular, we can copy the maximum index $\mathtt{max}$ into the counting module value n, the length of the input. Using the counting module primitives, we can define multiplication on counting modules. Then, all we need to do is construct $(n+1)^k$ for a large enough constant k. All of this is done without collisions in the context of a sufficiently large (but still polynomially bounded) counting module.[21]

The final state is named by $\mathtt{state}(\mathtt{seed}, \vec{\mathtt{x}})$, which is the output of $\mathcal{M}$ if f is boolean-valued. If $\mathcal{M}$ has output of type $1^\star$, then the output is $\mathtt{position}(\mathtt{seed}, \vec{\mathtt{x}})$, the final position of the output tape. Since the output tape is one-way and write-only, this position is precisely the output string. Finally, this program contains no collisions, since it contains no counting module additions except those in the construction of $\mathtt{seed}$.

Conversely, suppose that there is a polynomially bounded $\lambda : \omega \to \omega$ and a bit-length program p such that $[\![p]\!]_\lambda(x) = f(x)$, without collisions, for any $x \in 2^\star$. For a fixed input x, there are at most polynomially many-in-$|x|$ possible environments ρ obtained by binding variables of p to indices of x and counting modules bounded by $\lambda(|x|)$. We make a table of all pairs (ρ, T) of environments and possible terms and then, using dynamic programming, evaluate each term on each environment. One of these is the output of the program, and the whole process takes polynomial time. ■

Theorem. *For any function $f : 2^\star \to 1^\star$, the following are equivalent:*

- *f is computable in logarithmic space.*
- *There is a polynomially bounded $\lambda : \omega \to \omega$ and a tail-recursive bit-length program p such that $[\![p]\!]_\lambda(x) = f(x)$, without collisions, for any $x \in 2^\star$.*

For any function $f : 2^\star \times 1^\star \to 2$ and polynomially bounded function $\pi : \omega \to \omega$, the following are equivalent:

[21] In earlier papers (Jones, 2001; Kop & Simonsen, 2017), $\mathtt{seed}$ was originally a counting module *primitive* naming the largest element of the counting module. We did not follow that approach here, because its inclusion would break Lemma 1, the independence of collision-free computation from the bound of the counting module.

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press

- *There is a logarithmic-space computable function* $g : 2^\star \times 1^\star \to 2$ *such that* $g(x, y) = f(x, y)$ *for any string* $x \in 2^\star$ *and* $y < \pi(|x|)$.
- *There is a polynomially bounded function* $\lambda : \omega \to \omega$ *and a tail-recursive bit-length program* p *such that* $[\![p]\!]_\lambda(x, y) = f(x, y)$*, without collisions, for any* $x \in 2^\star$ *and* $y < \pi(|x|)$.

Proof As above, let f be a function of type $2^\star \times (1^\star)^k \to \alpha$ for some $k \in \omega$ and $\alpha \in \{1^\star, 2\}$, and let $\mathcal{M}$ be a Turing machine computing f in logarithmic space. In case $\alpha = 1^\star$, $\mathcal{M}$ has a write-only one-way output tape; note that the logarithmic space bound does not apply to this tape.

For a fixed finite alphabet Δ, call a *short string* a Δ-string of length $O(\log n)$, for any given n. We can encode short strings by polynomially bounded-in-n counting modules, roughly, by identifying Δ with some finite initial segment of ω, and identifying bounded-width Δ-strings with base-Δ numerals.

Let the *configuration* of $\mathcal{M}$ at a given time consist of the current state, the positions of each head, and the entire contents of each tape. (Contrast this with the surface configuration above.) The point is, in a logarithmic-space Turing machine on input of length n, the entire configuration of n can be encoded by a short string in n and thus in a polynomially bounded counting module. Moreover, we can compute the next configuration from the previous one by a tail-recursive program using the counting module primitives.

Ultimately, the program p looks something like this. As above, the single binary string input is identified with the (invisible) global input variable, and the string $\vec{\mathtt{x}}$ is identified with any remaining variables of type $1^\star$.

$$\mathtt{f}_0(\vec{\mathtt{x}}) = \mathtt{f}_1(\mathtt{init}(\vec{\mathtt{x}}))$$
$$\mathtt{f}_1(\mathtt{c}) = \mathtt{if}\ \mathtt{halt}(\mathtt{c})\ \mathtt{then}\ \mathtt{out}(\mathtt{c})\ \mathtt{else}\ \mathtt{f}_1(\mathtt{next}(\mathtt{c})),$$

where $\mathtt{halt}(\mathtt{c})$ detects whether the configuration $\mathtt{c}$ is halting, $\mathtt{out}(\mathtt{c})$ correctly returns the output of a halting configuration, $\mathtt{init}(\vec{\mathtt{x}})$ is the initial configuration, and $\mathtt{next}(\mathtt{c})$ is the configuration following $\mathtt{c}$. This program is tail-recursive and correctly computes f.

Conversely, suppose that there is a polynomially bounded $\lambda : \omega \to \omega$ and a tail-recursive bit-length program p such that $[\![p]\!]_\lambda(x) = f(x)$, without collisions, for any $x \in 2^\star$. It is a well-known fact that every tail-recursive program can be rewritten as a while program with a single while loop and no additional data structures (e.g., a stack). We can directly simulate a program with a single while loop on a Turing machine by, e.g., storing each program variable on a tape of the machine. But it takes only logarithmic space to encode indices and counting modules which are bounded polynomially in the length of the input. ■

https://doi.org/10.1017/S0956796823000023 Published online by Cambridge University Press