Logarithm and Program Testing

KUEN-BANG HOU (FAVONIA), University of Minnesota, USA
ZHUYANG WANG, University of Minnesota, USA

Randomized property-based testing has gained much attention recently, but most frameworks stop short at polymorphic properties. Although Bernardy *et al.* have developed a theory to reduce a wide range of polymorphic properties to monomorphic ones, it relies upon ad-hoc embedding-projection pairs to massage the types into a particular form. This paper skips the embedding-projection pairs and presents a mechanical monomorphization for a general class of polymorphic functions, a step towards automatic testing for polymorphic properties. The calculation of suitable types for monomorphization turns out to be *logarithm*.

CCS Concepts: • **Software and its engineering → Polymorphism**; **Software testing and debugging**; • **Theory of computation → Type theory**.

Additional Key Words and Phrases: parametricity, polymorphism, logarithm

ACM Reference Format:
Kuen-Bang Hou (Favonia) and Zhuyang Wang. 2022. Logarithm and Program Testing. *Proc. ACM Program. Lang.* 6, POPL, Article 64 (January 2022), 26 pages. https://doi.org/10.1145/3498726

1 INTRODUCTION

1.1 Randomized Property-Based Testing

Randomized property-based testing is a popular technique to detect bugs in the early stages of software development. It generalizes traditional unit testing, which suffers from limited numbers of hand-crafted examples. Instead, programmers specify the expected properties of the programs, letting the testing framework generate a vast amount of random test cases for better coverage. Since the popularization of this methodology via the HASKELL library QUICKCHECK [Claessen and Hughes 2000], it has been ported to virtually every popular programming language and inspired other tools such as SMALLCHECK [Runciman et al. 2008] for enumerating finite values.

To see how property-based testing works, consider the reversal function for natural number lists

$$\texttt{reverse} : \texttt{list}(\mathbb{N}) \to \texttt{list}(\mathbb{N}).$$

A programmer can write down properties that `reverse` should satisfy, and the testing engine will verify these properties against a large number of randomly generated test cases. One property is that the `reverse` function should be an involution; that is, for any natural number list l,

$$\texttt{reverse}(\texttt{reverse}(l)) \cong l.$$

The hope is that the amount of randomly generated lists will achieve better coverage than the manually crafted test cases. The function `reverse` passes the test if the result of the comparison is `true` for all generated lists l. The programmer may also provide other properties that the function

Authors' addresses: Kuen-Bang Hou (Favonia), Department of Computer Science and Engineering, University of Minnesota, Minneapolis, Minnesota, 55455, USA, kbh@umn.edu; Zhuyang Wang, Department of Computer Science and Engineering, University of Minnesota, Minneapolis, Minnesota, 55455, USA, wang9163@umn.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2022 Copyright held by the owner/author(s).
2475-1421/2022/1-ART64
https://doi.org/10.1145/3498726

`reverse` should satisfy. As each property is tested against numerous test cases, good coverage can be achieved with only a small number of properties.

In this paper, we focus on the most general property—whether two functions are equivalent. Any property-based testing can be thought as the comparison of that function and the constant function that always returns `true`.

One limit of randomized property-based testing is that the type of a property must be monomorphic, because the test case generator cannot build concrete elements from an abstract type. To test a polymorphic function, one must either instantiate the function by a particular type or rely on the library to automatically choose a type to instantiate; for example, QuickCheck will choose the integer type $\mathbb{Z}$ (**`Integer`** in Haskell). This is not ideal because there are often more efficient ways to distinguish two polymorphic functions. Consider two functions `proj`, `proj′` of the same type

$$\forall a. a \times a \to a.$$

To check the equivalence between `proj` and `proj′`, it suffices to check whether

$$\mathtt{proj}[\mathbb{2}](\langle \mathtt{true}; \mathtt{false}\rangle) \cong \mathtt{proj'}[\mathbb{2}](\langle \mathtt{true}; \mathtt{false}\rangle)$$

because from the type we can tell that there are only two ways to produce an element of a: either taking the first component of the argument or taking the second one. It is sufficient to know which a it is, or in other words, where this a comes from. Thus a type with two elements is enough to index these two different a-elements. Moreover, both functions only need to be run once to compare the results. On the other hand, instantiating the functions with $\mathbb{Z}$ and randomly generating pairs of numbers would be a waste of effort.

Another example is the map function for lists:

$$\mathsf{map} : \forall a.\forall b.(a \to b) \to \mathtt{list}(a) \to \mathtt{list}(b).$$

We are interested in comparing an unverified implementation $\mathsf{map'}$ to the reference function map. It turns out that we only need to test their monomorphic instances $\mathsf{map}[\mathbb{N}][\mathbb{N}]$ and $\mathsf{map'}[\mathbb{N}][\mathbb{N}]$ with the first argument being the identity function and the second being of the form $[0, \dots, n-1]$ for all n. That is, if the following equation holds for all n, then map and $\mathsf{map'}$ are equivalent:

$$\mathsf{map}[\mathbb{N}][\mathbb{N}](\mathsf{id}_{\mathbb{N}})([0, \dots, n-1]) \cong \mathsf{map'}[\mathbb{N}][\mathbb{N}](\mathsf{id}_{\mathbb{N}})([0, \dots, n-1]).$$

The intuition is that parametric polymorphism forces all elements in the output of type $\mathtt{list}(b)$ to be generated from the application of the first argument of type $a \to b$ to elements in the second argument of type $\mathtt{list}(a)$. That is the only way to ever produce an element of type b in $\mathsf{map'}$. The only diversions $\mathsf{map'}$ can make are (1) to omit an element, (2) to duplicate an element, and (3) to permute the output list. The identity function and the lists $[0, \dots, n-1]$ can detect all three forms of discrepancies (omission, duplication, and permutation).

The last example we want to show in the introduction is the length function

$$\mathtt{length} : \forall a.\mathtt{list}(a) \to \mathbb{N}.$$

Let `length′` be an implementation claiming to have the same functionality and $\mathbb{1}$ be the unit type. It is sufficient to check whether their $\mathbb{1}$ instances, $\mathtt{length'}[\mathbb{1}]$ and $\mathtt{length}[\mathbb{1}]$, agree. The reason is that functions of type $\forall a.\mathtt{list}(a) \to \mathbb{N}$ can never inspect an element in the input lists. We can therefore replace all elements with the unique element of type $\mathbb{1}$.

In the above examples, we discovered that equality between polymorphic functions can often be derived from a limited number of inputs at specific instances. This paper aims to develop a general theory to find such suitable types to instantiate polymorphic functions and to fix parts of the inputs if possible, so that unnecessary test cases can be avoided. Specifically, to check the equivalence between two general polymorphic functions f, g of type

$$\forall a.\alpha(a) \to H(a)$$

where $H(-)$ is a *functor* (a type expression with a hole that appears at positive positions; or, $a \in^+ H(a)$ in our notation), this paper provides a mechanical calculation to derive a general enough type a^* to instantiate the function and a function `refill` to fix parts of the input of type $\alpha(a^*)$, such that it suffices to check

$$f[a^*] \circ \texttt{refill} \cong g[a^*] \circ \texttt{refill}.$$

The function `refill` has type $\alpha^-(a^*) \to \alpha(a^*)$, where $\alpha^-(a^*)$ is essentially $\alpha(a^*)$ but with some parts replaced by placeholders to be filled by the function `refill`.

Interestingly, the calculation of the type a^* turns out to be the *logarithm* of the type $\alpha(a)$ with respect to a, and thus the name of this paper. (Strictly speaking, the calculation will in general be an *over-approximation* of the logarithm, for the reasons explained on Page 9.) The type of the `proj` function in the first example provides some hints on why it is logarithm. The argument of `proj` is of type $a \times a = a^2$, and its logarithm $\log_a(a \times a) = \log_a(a^2)$ is 2, a type with two elements, which is exactly the boolean type $\mathbb{2}$. Generally, to test an n-ary projection function of type $a^n \to a$, it is sufficient to use a type with n distinct elements, and $\log(a^n) = n$. This is not a coincidence. The type $\log_a(a^n)$ "counts" the number of a that can be obtained from the type a^n, and a polymorphic projection function can only choose one of the a-elements from a^n and must always choose the same one. A type with n distinct elements is sufficient to tell these choices apart.

To demonstrate practicality, we implemented a Haskell library based on the results in this paper, with two backends QuickCheck and SmallCheck to carry out the testing of the monomorphized instances. Users can test polymorphic functions more efficiently without manually specifying suitable types for instantiation.

Remark (relation to parametricity). The reduction from the equivalence of polymorphic functions to that of their monomorphic instances relies on the parametricity theorem established by Reynolds [1983] and popularized by Wadler [1989] as "free theorems." Parametric polymorphism limits the number of possible programs and thus reduces the needed testing.

1.2 Towards a General Theory of Monomorphization

There were already theoretical developments towards a general theory for testing polymorphic functions. Bernardy et al. [2010] have shown how to monomorphize a wide range of polymorphic function testing. In particular, to check the equivalence between two functions f, g of type

$$\forall a.(F(a) \to a) \times (G(a) \to K) \to H(a)$$

where $F(-)$, $G(-)$, $H(-)$ are functors and K is constant in a, it suffices to check whether

$$f[\mu a.F(a)] \circ \langle \texttt{roll}_{a.F(a)}; - \rangle \cong g[\mu a.F(a)] \circ \langle \texttt{roll}_{a.F(a)}; - \rangle$$

where $\texttt{roll}_{a.F(a)} : F(\mu a.F(a)) \to \mu a.F(a)$ is the initial F-algebra, the "constructor" of the inductive type $\mu a.F(a)$ that rolls an element of type $F(\mu a.F(a))$ into one of type $\mu a.F(a)$. Notice that $G(a)$ and $H(a)$ do not affect our choice for a because they provide no additional ways to construct an element of type a, an important concept we will explore later. For general polymorphic functions, if one can massage the argument type $\alpha(a)$ into $(F(a) \to a) \times (G(a) \to K)$ through an embedding-projection pair, then the theorem still applies. The reduction is done by inserting the mediating projection functions at appropriate places.

$$
\begin{array}{rrl}
\text{Type contexts} & \Delta & \coloneqq \cdot \mid \Delta, a \\
\text{Expression contexts} & \Gamma & \coloneqq \cdot \mid \Gamma, x{:}\tau \\
\text{Monomorphic types} & \tau & \coloneqq a \mid \tau_1 \to \tau_2 \mid \mathbb{0} \mid \tau_1 + \tau_2 \mid \mathbb{1} \mid \tau_1 \times \tau_2 \mid \mu a.\tau \\
\text{Polymorphic types} & T & \coloneqq \tau \mid \forall a.T \\
\text{Expressions} & e, f, g, h & \coloneqq x \mid \Lambda a.e \mid e[\tau] \mid \lambda(x{:}\tau).e \mid e_1(e_2) \mid \\
& & \quad \mathtt{abort}(e) \mid \mathtt{inl}(e) \mid \mathtt{inr}(e) \mid \mathtt{case}(e; x.e_x; y.e_y) \mid \\
& & \quad \star \mid \langle e_1; e_2 \rangle \mid \mathtt{fst}(e) \mid \mathtt{snd}(e) \mid \\
& & \quad \mathtt{roll}_{a.\tau}(e) \mid \mathtt{fold}^{\tau'}_{a.\tau}(e; x.e_1)
\end{array}
$$

Fig. 1. Syntax of the prenex fragment of the extended System F

This paper advances the theory so that the type $\mu a.F(a)$ and the argument $\mathtt{roll}_{a.F(a)}$ can be calculated directly from $\alpha(a)$. It replaces the ad-hoc embedding-projection pairs in the literature with direct calculation on type expressions. It also avoids suspicious reductions that the author(s) found in earlier work. It not only recovers known results (for the calculus in consideration) but also sheds light on the testing semantics of more complicated inductive types, coinductive types, and other type constructors.

2 THE LANGUAGE AND NOTATION

We will use the prenex fragment of System F extended with the following types:

- The empty type $\mathbb{0}$.
- Sum types $\tau_1 + \tau_2$ with injections `inl` and `inr`.
- The unit type $\mathbb{1}$ with the unique element $\star$.
- Product types $\tau_1 \times \tau_2$ with pairs written $\langle -; - \rangle$.
- (Strictly positive) inductive types $\mu a.\tau$ with their recursor $\mathtt{fold}^{\tau'}_{a.\tau}$. The type variable a must be strictly positive in τ, a condition formally written as $a \in^{++} \tau$.

We focus on the prenex fragment (also known as rank-1 polymorphism) with additional built-in types instead of the full System F, because the technology in this paper cannot handle higher-rank polymorphism yet. On the other hand, the fragment is reasonably close to practical functional programming languages. As a piece of evidence, we were able to build a Haskell tool based on the theory to automatically test polymorphic functions.

The syntax and operational semantics are elaborated in Figures 1, 2, 3 and 5. We adopt the standard call-by-value operational semantics. However, choices of evaluation strategies only matter when we start to allow effects (including non-termination) or analyze complexity, which would be out of the scope of this paper.

Finally, we will annotate the types of function arguments unless the presentation is forbiddingly crowded. Also, we write consecutive function applications $f(x)(y)$ as $f(x; y)$.

2.1 Inductive Types and Functoriality

The dynamics of the recursor $\mathtt{fold}^{\tau'}_{a.\tau}(e; x.e_1) : \tau'$ for the inductive type $\mu a.\tau$ is worth more explanation. The expression e is an element of the type $\mu a.\tau$ to be processed, and the recursor $\mathtt{fold}^{\tau'}_{a.\tau}$ will execute these steps:

(1) Evaluate e to $\mathtt{roll}_{a.\tau}(e')$ for some value $e' : \tau[\mu a.\tau / a]$.

$$\boxed{\Delta \vdash \tau} \text{ and } \boxed{\Delta \vdash \sigma : \Delta'} \text{ and } \boxed{\Delta \vdash \Gamma}$$

Types τ, type subsections σ, and expression contexts Γ are well-formed with respect to type contexts Δ.

$$\frac{}{\Delta, a \vdash a} \qquad \frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2} \qquad \frac{}{\Delta \vdash \mathbb{0}} \qquad \frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 + \tau_2} \qquad \frac{}{\Delta \vdash \mathbb{1}} \qquad \frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \times \tau_2}$$

$$\frac{\Delta, a \vdash \tau \qquad a \in^{++} \tau}{\Delta \vdash \mu a.\tau} \qquad \frac{}{\Delta \vdash \cdot} \qquad \frac{\Delta \vdash \Gamma \qquad \Delta \vdash \tau}{\Delta \vdash \Gamma, x{:}\tau}$$

For a substitution σ, we write $\Delta \vdash \sigma : \Delta'$ if $\mathrm{dom}(\sigma) = \Delta'$ and for any $t \in \Delta'$, $\Delta \vdash \sigma(t)$.

$$\boxed{a \in^{-} \tau} \text{ and } \boxed{a \in^{+} \tau}$$

Polarity of type variables a in types τ. Polarities p range over $\{+, -\}$, and $\overline{p}$ is the opposite polarity of p.

$$\frac{}{a \in^{+} a} \qquad \frac{a \neq b}{a \in^{p} b} \qquad \frac{a \in^{\overline{p}} \tau_1 \qquad a \in^{p} \tau_2}{a \in^{p} \tau_1 \to \tau_2} \qquad \frac{}{a \in^{p} \mathbb{0}} \qquad \frac{a \in^{p} \tau_1 \qquad a \in^{p} \tau_2}{a \in^{p} \tau_1 + \tau_2} \qquad \frac{}{a \in^{p} \mathbb{1}}$$

$$\frac{a \in^{p} \tau_1 \qquad a \in^{p} \tau_2}{a \in^{p} \tau_1 \times \tau_2} \qquad \frac{b \in^{p} \tau}{b \in^{p} \mu a.\tau}$$

$$\boxed{a \in^{++} \tau}$$

Strict positivity

$$\frac{}{a \in^{++} b} \qquad \frac{a \notin \tau_1 \qquad a \in^{++} \tau_2}{a \in^{++} \tau_1 \to \tau_2} \qquad \frac{}{a \in^{++} \mathbb{0}} \qquad \frac{a \in^{++} \tau_1 \qquad a \in^{++} \tau_2}{a \in^{++} \tau_1 + \tau_2} \qquad \frac{}{a \in^{++} \mathbb{1}}$$

$$\frac{a \in^{++} \tau_1 \qquad a \in^{++} \tau_2}{a \in^{++} \tau_1 \times \tau_2} \qquad \frac{b \in^{++} \tau}{b \in^{++} \mu a.\tau}$$

Fig. 2. Formation and polarity rules for types in the prenex fragment of the extended System F

(2) Recursively apply itself to $e' : \tau[\mu a.\tau/a]$ to obtain another term $e'' : \tau[\tau'/a]$.
(3) Evaluate the term $e_1[e''/x]$ as the final result.

The recursive calls in the second step are guided by the functorial action induced by the locations of the variable a in the type expression τ. The functorial action is formally written as

$$\langle a^{+}.\tau\rangle(f : \tau_1 \to \tau_2) : \tau[\tau_1/a] \to \tau[\tau_2/a]$$

It covariantly lifts a function f of type $\tau_1 \to \tau_2$ to a function of type $\tau[\tau_1/a] \to \tau[\tau_2/a]$, under the condition that the type variable a is positive in τ. This is noted by the plus sign "+" in the notation $\langle a^{+}.\tau\rangle(f : \tau_1 \to \tau_2)$. Defining this operator for the cases where τ is a function type forces us to

$$\boxed{\Delta;\Gamma \vdash e : \tau} \text{ and } \boxed{\Delta;\Gamma \vdash_{\texttt{poly}} e : T}$$

Expressions e are of monomorphic types τ or polymorphic types T with respect to contexts Δ and Γ.

$$\frac{\Delta \vdash \Gamma \qquad x{:}\tau \in \Gamma}{\Delta;\Gamma \vdash x : \tau} \qquad \frac{\Delta;\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Delta;\Gamma \vdash \lambda(x{:}\tau_1).e : \tau_1 \to \tau_2} \qquad \frac{\Delta;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Delta;\Gamma \vdash e_2 : \tau_1}{\Delta;\Gamma \vdash e_1(e_2) : \tau_2}$$

$$\frac{\Delta;\Gamma \vdash e : \mathbb{0} \qquad \Delta \vdash \tau}{\Delta;\Gamma \vdash \texttt{abort}(e) : \tau} \qquad \frac{\Delta;\Gamma \vdash e : \tau_1}{\Delta;\Gamma \vdash \texttt{inl}(e) : \tau_1 + \tau_2} \qquad \frac{\Delta;\Gamma \vdash e : \tau_2}{\Delta;\Gamma \vdash \texttt{inr}(e) : \tau_1 + \tau_2}$$

$$\frac{\Delta;\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Delta;\Gamma, x{:}\tau_1 \vdash e_x : \tau \qquad \Delta;\Gamma, y{:}\tau_2 \vdash e_y : \tau}{\Delta;\Gamma \vdash \texttt{case}(e; x.e_x; y.e_y) : \tau} \qquad \frac{}{\Delta;\Gamma \vdash \star : \mathbb{1}}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \qquad \Delta;\Gamma \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash \langle e_1; e_2\rangle : \tau_1 \times \tau_2} \qquad \frac{\Delta;\Gamma \vdash e : \tau_1 \times \tau_2}{\Delta;\Gamma \vdash \texttt{fst}(e) : \tau_1} \qquad \frac{\Delta;\Gamma \vdash e : \tau_1 \times \tau_2}{\Delta;\Gamma \vdash \texttt{snd}(e) : \tau_2}$$

$$\frac{\Delta, a \vdash \tau \qquad a \in^{+} \tau \qquad \Delta;\Gamma \vdash e : \tau[\mu a.\tau/a]}{\Delta;\Gamma \vdash \texttt{roll}_{a.\tau}(e) : \mu a.\tau} \qquad \frac{\Delta;\Gamma \vdash e : \mu a.\tau \qquad \Delta;\Gamma, x{:}\tau[\tau'/a] \vdash e_1 : \tau'}{\Delta;\Gamma \vdash \texttt{fold}^{\tau'}_{a.\tau}(e; x.e_1) : \tau'}$$

$$\frac{\Delta;\Gamma \vdash_{\texttt{poly}} e : \tau}{\Delta;\Gamma \vdash e : \tau} \qquad \frac{\Delta;\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash_{\texttt{poly}} e : \tau} \qquad \frac{\Delta, a;\Gamma \vdash_{\texttt{poly}} e : T}{\Delta;\Gamma \vdash_{\texttt{poly}} \Lambda a.e : \forall a.T} \qquad \frac{\Delta;\Gamma \vdash_{\texttt{poly}} e : \forall a.T \qquad \Delta \vdash \tau}{\Delta;\Gamma \vdash_{\texttt{poly}} e[\tau] : T[\tau/a]}$$

Fig. 3. Typing rules of the prenex fragment of the extended System F

$$\boxed{\langle a^p.\tau\rangle(f; e)}$$

Functoriality associated with functions $f : \tau_1 \to \tau_2$ and types τ with respect to a type variable a of polarity p.

$$\begin{aligned}
\langle a^{+}.a\rangle(f;e) &:= f(e)\\
\langle a^{p}.b\rangle(f;e) &:= e \qquad (b \neq a)\\
\langle a^{p}.\mathbb{0}\rangle(f;e) &:= e\\
\langle a^{p}.\tau_{\text{left}} + \tau_{\text{right}}\rangle(f;e) &:= \texttt{case}(e; x.\texttt{inl}(\langle a^{p}.\tau_{\text{left}}\rangle(f;x)); y.\texttt{inr}(\langle a^{p}.\tau_{\text{right}}\rangle(f;y)))\\
\langle a^{p}.\mathbb{1}\rangle(f;e) &:= e\\
\langle a^{p}.\tau_{\text{fst}} \times \tau_{\text{snd}}\rangle(f;e) &:= \langle \langle a^{p}.\tau_{\text{fst}}\rangle(f;\texttt{fst}(e)); \langle a^{p}.\tau_{\text{snd}}\rangle(f;\texttt{snd}(e))\rangle\\
\langle a^{+}.\tau_{\text{dom}} \to \tau_{\text{cod}}\rangle(f;e) &:= \lambda(x{:}\tau_{\text{dom}}[\tau_2/a]).\langle a^{+}.\tau_{\text{cod}}\rangle(f; e(\langle a^{-}.\tau_{\text{dom}}\rangle(f;x)))\\
\langle a^{-}.\tau_{\text{dom}} \to \tau_{\text{cod}}\rangle(f;e) &:= \lambda(x{:}\tau_{\text{dom}}[\tau_1/a]).\langle a^{-}.\tau_{\text{cod}}\rangle(f; e(\langle a^{+}.\tau_{\text{dom}}\rangle(f;x)))\\
\langle a^{+}.\mu b.\tau\rangle(f;e) &:= \texttt{fold}^{\mu b.\tau[\tau_2/a]}_{b.\tau[\tau_1/a]}(e; x.\texttt{roll}_{b.\tau[\tau_2/a]}(\langle a^{+}.\tau\rangle(f;x)))\\
\langle a^{-}.\mu b.\tau\rangle(f;e) &:= \texttt{fold}^{\mu b.\tau[\tau_1/a]}_{b.\tau[\tau_2/a]}(e; x.\texttt{roll}_{b.\tau[\tau_1/a]}(\langle a^{-}.\tau\rangle(f;x)))
\end{aligned}$$

Fig. 4. Functoriality associated with type expressions

$$\boxed{e_1 \mapsto e_2} \text{ and } \boxed{e\ \mathsf{val}}$$

Operational semantics

$$\frac{}{\Lambda a.e\ \mathsf{val}} \qquad \frac{e \mapsto e'}{e[\tau] \mapsto e'[\tau]} \qquad \frac{}{(\Lambda a.e)[\tau] \mapsto e[\tau/a]} \qquad \frac{}{\lambda(x{:}\tau).e_1\ \mathsf{val}} \qquad \frac{e_1 \mapsto e_1'}{e_1(e_2) \mapsto e_1'(e_2)}$$

$$\frac{e_1\ \mathsf{val} \qquad e_2 \mapsto e_2'}{e_1(e_2) \mapsto e_1(e_2')} \qquad \frac{e_2\ \mathsf{val}}{(\lambda(x{:}\tau).e_1)(e_2) \mapsto e_1[e_2/x]} \qquad \frac{e \mapsto e'}{\mathtt{inl}(e) \mapsto \mathtt{inl}(e')} \qquad \frac{e\ \mathsf{val}}{\mathtt{inl}(e)\ \mathsf{val}}$$

$$\frac{e \mapsto e'}{\mathtt{inr}(e) \mapsto \mathtt{inr}(e')} \qquad \frac{e\ \mathsf{val}}{\mathtt{inr}(e)\ \mathsf{val}} \qquad \frac{e \mapsto e'}{\mathtt{case}(e; x.e_x; y.e_y) \mapsto \mathtt{case}(e'; x.e_x; y.e_y)}$$

$$\frac{e\ \mathsf{val}}{\mathtt{case}(\mathtt{inl}(e); x.e_x; y.e_y) \mapsto e_x[e/x]} \qquad \frac{e\ \mathsf{val}}{\mathtt{case}(\mathtt{inr}(e); x.e_x; y.e_y) \mapsto e_y[e/y]} \qquad \frac{}{\star\ \mathsf{val}}$$

$$\frac{e_1 \mapsto e_1'}{\langle e_1; e_2\rangle \mapsto \langle e_1'; e_2\rangle} \qquad \frac{e_1\ \mathsf{val} \qquad e_2 \mapsto e_2'}{\langle e_1; e_2\rangle \mapsto \langle e_1; e_2'\rangle} \qquad \frac{e_1\ \mathsf{val} \qquad e_2\ \mathsf{val}}{\langle e_1; e_2\rangle\ \mathsf{val}} \qquad \frac{e \mapsto e'}{\mathtt{fst}(e) \mapsto \mathtt{fst}(e')}$$

$$\frac{e_1\ \mathsf{val} \qquad e_2\ \mathsf{val}}{\mathtt{fst}(\langle e_1; e_2\rangle) \mapsto e_1} \qquad \frac{e \mapsto e'}{\mathtt{snd}(e) \mapsto \mathtt{snd}(e')} \qquad \frac{e_1\ \mathsf{val} \qquad e_2\ \mathsf{val}}{\mathtt{snd}(\langle e_1; e_2\rangle) \mapsto e_2} \qquad \frac{e\ \mathsf{val}}{\mathtt{roll}_{a.\tau}(e)\ \mathsf{val}}$$

$$\frac{e\ \mathsf{val} \qquad y \text{ fresh}}{\mathtt{fold}^{\tau'}_{a.\tau}(\mathtt{roll}_{a.\tau}(e); x.e_1) \mapsto e_1[\langle a^+.\tau\rangle(\lambda y.\mathtt{fold}^{\tau'}_{a.\tau}(y; x.e_1); e)/x]}$$

Fig. 5. Dynamics of the prenex fragment of the extended System F

consider its dual that *contra*variantly lifts a function because function domains are contravariant:

$$\langle a^-.\tau\rangle(f : \tau_1 \to \tau_2) : \tau[\tau_2/a] \to \tau[\tau_1/a]$$

where a is *negative* in τ (formally written as $a \in^- \tau$). The full definition of the functorial operators is in Figure 4; the polarity changes only when handling function arguments. With the functorial operators $\langle a^p.\tau\rangle(f)$ defined, the dynamics of $\mathtt{fold}^{\tau'}_{a.\tau}$ can be described by

$$\mathtt{fold}^{\tau'}_{a.\tau}(\mathtt{roll}_{a.\tau}(e); x.e_1) \mapsto e_1[\langle a^+.\tau\rangle(\lambda z.\mathtt{fold}^{\tau'}_{a.\tau}(z; x.e'); e)/x]$$

which satisfies the expected homomorphism condition for any $f : \tau[\tau'/a] \to \tau'$,

$$\mathtt{fold}^{\tau'}_{a.\tau}(-; x.f(x)) \circ \mathtt{roll}_{a.\tau} \cong f \circ \langle a^+.\tau\rangle(\mathtt{fold}^{\tau'}_{a.\tau}(-; x.f(x))).$$

Here is the length function for $\mathtt{list}(a) = \mu l.\mathbb{1} + a \times l$ as an example of $\mathtt{fold}$: Suppose we already have $\mathbb{N}$, $\mathtt{zero}$, and $\mathtt{suc}$ defined as the natural number type, the zero, and the successor; then, the polymorphic length function would be $\Lambda a.\lambda x.\mathtt{fold}^{\mathbb{N}}_{l.\mathbb{1}+a\times l}(x; y.\mathtt{case}(y; \_.\mathtt{zero}; z.\mathtt{suc}(\mathtt{snd}(z))))$.

3 LOGARITHM OPERATION

We wish to derive a suitable type a^* from a type expression $\alpha(a)$ so that the behavior of a function

$$f : \forall a.\alpha(a) \to H(a)$$

$$\begin{aligned}
\log_a(a) &:= \mathbb{1} & \log_a(\mathbb{1}) &:= \mathbb{0} \\
\log_a(b) &:= \mathbb{0} \quad (a \neq b) & \log_a(\tau_1 \times \tau_2) &:= \log_a(\tau_1) + \log_a(\tau_2) \\
\log_a(\mathbb{0}) &:= \mathbb{0} & \log_a(\tau_1 \to \tau_2) &:= \tau_1 \times \log_a(\tau_2) \\
\log_a(\tau_1 + \tau_2) &:= \log_a(\tau_1) + \log_a(\tau_2)
\end{aligned}$$

Fig. 6. Logarithm without inductive types

is determined by its instance $f[a^*]$. For example, for the projection function

$$\texttt{proj} : \forall a. a \times a \to a$$

we wish to conclude that $a^* = \mathbb{2}$ from $\alpha(a) = a \times a = a^2$. Logarithm seems to gives the correct answer because $\log_a(a^2) = 2$. However, there are subtleties that can only be revealed with a more delicate example. Consider the case where $\alpha(a) = a \times (a \to a)$; that is,

$$f : \forall a. a \times (a \to a) \to H(a)$$

What would be a suitable type to instantiate the function f for testing? When $H(a)$ is just a, the type is exactly the Church encoding of natural numbers, which hints that the natural number type might be a good choice. Indeed, as we will see, the natural number type is one of the *best* choices for general $H(a)$. Operationally, the function f obtains a pair of two components, one of type a and the other of type $a \to a$. The insight is that the only way to construct an a-element in f is to apply the second component of type $a \to a$ finite numbers of times to the first component of type a; each way to construct an a-element corresponds to a natural number (as the number of times the second component is being applied). Therefore, one suitable choice for a is the natural number type where the argument can be fixed to $\langle \texttt{zero}; \texttt{suc} \rangle$. That is, intuitively, the most general choice for a is the one which records construction histories and nothing else.

What would logarithm say about $\alpha(a) = a \times (a \to a)$? Recall that the arithmetic logarithm obeys the following two laws for suitable real numbers a and b:

$$\log(a \cdot b) = \log a + \log b \qquad\qquad \log b^a = a \cdot \log b.$$

If we follow the steps of the arithmetic logarithm, the logarithm of $\alpha(a) = a \times (a \to a)$ with respect to a would be $\mathbb{1} + a \times \mathbb{1}$ (if "$a \to a$" is treated as "a^a"). The fact that a appears in the logarithm of $\alpha(a)$ signifies the recursive nature of generating a-elements in f—the input to f may be used repeatedly to generate new a-elements from already generated ones. To account for this possibility, we may consider the least fixed point (*i.e.,* the inductive type) of the logarithm, $\mu a. \mathbb{1} + a \times \mathbb{1}$, as the suitable type for testing. This type is isomorphic to the natural number type $\mathbb{N} = \mu a. \mathbb{1} + a$, and thus the logarithm gives the correct answer again, with the help of inductive types. These calculations gave us the hope that perhaps logarithm and the inductive type construction together can generate a suitable type for testing. We proved that it is indeed the case.

In addition to the above intuition where the logarithm represents different ways to construct a-elements, another helpful insight comes from polymorphic data structures (*e.g.,* lists, trees, *etc.*) where a is the element type. Here, the logarithm is the index type of a-elements inside the structure. As a special case, when $\alpha(a) = \texttt{list}(a)$, the natural number type $\mathbb{N}$ is an appropriate choice because a-elements in an a-list can be indexed by natural numbers as their distances from the start of the list. (However, we will delay the discussion of inductive types until Section 3.2 because inductive types significantly complicate the presentation.)

Formally, the logarithm operation is written as $\log_a$ and $a^* = \mu a.\log_a(\alpha(a))$ will give us a general enough type to represent all possible ways to obtain a-elements from an input of type $\alpha(a)$. The logarithm $\log_a(\alpha(a))$ for types is defined by induction on the type expression $\alpha(a)$ as in Figure 6.

- If $\alpha(a) = a$ itself, then there is only one a-element. Thus, $\log_a(\alpha(a)) = \mathbb{1}$.
- If $\alpha(a) = b \neq a$, then an element of type $\alpha(a)$ is useless for the purpose of obtaining a-elements. We assign $\mathbb{0}$ to its logarithm.
- If $\alpha(a)$ is a constant (e.g., $\mathbb{0}$ or $\mathbb{1}$), an element of type $\alpha(a)$ is also useless for obtaining an a-element. Therefore, $\log_a(\alpha(a)) = \mathbb{0}$.
- If $\alpha(a) = \alpha_1(a) \times \alpha_2(a)$, then one may obtain an a-element from an element of type $\alpha(a)$ either via its first component of type $\alpha_1(a)$ or via the second component of type $\alpha_2(a)$. Therefore, its logarithm is the summation of the logarithms of both components; that is, $\log_a(\alpha(a)) = \log_a(\alpha_1(a)) + \log_a(\alpha_2(a))$. This matches the formula $\log(a \cdot b) = \log a + \log b$ in the usual arithmetic.
- If $\alpha(a) = \alpha_1(a) + \alpha_2(a)$, then one may still obtain an a-element from an element of type $\alpha(a)$ by using an element of type $\alpha_1(a)$ or $\alpha_2(a)$, though only one of them is available at any given time. As an over-approximation, we still assign to its logarithm the summation of the logarithms of components; that is, $\log_a(\alpha(a)) = \log_a(\alpha_1(a)) + \log_a(\alpha_2(a))$.
- If $\alpha(a) = \alpha_1(a) \to \alpha_2(a)$, the idea is that any way to obtain an a-element from an element of type $\alpha(a)$ must involve applying this input function to some argument of type $\alpha_1(a)$ and then proceed with its result of type $\alpha_2(a)$ to generate a-elements. In other words, every construction method can be described in two parts:
  - the argument of type $\alpha_1(a)$ fed into this function, and
  - the remaining steps to obtain an a-element from the result of type $\log_a(\alpha_2(a))$.

  This suggests $\log_a(\alpha(a)) = \alpha_1(a) \times \log_a(\alpha_2(a))$, which matches $\log(b^a) = a \cdot \log b$ in the usual arithmetic (where $\alpha_2(a)^{\alpha_1(a)}$ is the aesthetical presentation of $\alpha_1(a) \to \alpha_2(a)$).

The type $\log_a(\alpha(a))$ matches the usual logarithm perfectly when $\alpha(a)$ is made of a, $\mathbb{1}$, $\times$, and $\to$. These are the cases where precise inverses to the exponential can be found. That is, all the types $\alpha(a)$ in this fragment are isomorphic to $\log_a(\alpha(a)) \to a$ (or more stylishly $a^{\log_a(\alpha(a))}$); when $\log_a(\alpha(a))$ is constant in a, we can also say $\alpha(-)$ as a functor is *represented* by its logarithm. The precise invertibility is called *Naperian* by Peter Hancock [Hancock 2019]. Because our goal is to cover as many type expressions as possible, we allow our logarithm operation to overestimate the number of ways to construct a-elements when the inverse to exponential cannot be (easily) found, at the price of losing precise invertibility. Therefore, $\log_a(\alpha(a)) \to a$ is in general not isomorphic to $\alpha(a)$. For example, $\mathbb{0} \neq \left(\log_a(\mathbb{0}) \to a\right) = (\mathbb{0} \to a) \simeq \mathbb{1}$. Perhaps surprisingly, we will see cases where neither $\log_a(\alpha(a)) \to a$ nor $\alpha(a)$ is uniformly larger than the other for all possible types a.

3.1 Correctness Theorem (without Inductive Types)

The following theorem shows that our logarithm operation provides a general enough type.

Theorem 3.1 (correctness without inductive types). *Suppose we have*

- *An ambient kinding context Δ; and*
- *A type substitution δ such that $\cdot \vdash \delta : \Delta$, which applies to all open types in the theorem; and*
- *Two type expressions $\alpha(a)$ and $H(a)$ such that*

$$\Delta, a \vdash \alpha(a) \text{ and } a \in^{++} \log_a(\alpha(a))$$

(The strict positivity makes sure $\mu a.\log_a(\alpha(a))$ is well-formed.)

$$\Delta, a \vdash H(a) \text{ and } a \in^{+} H(a)$$

- *Two functions f and g such that*

$$\cdot;\cdot \vdash_{\mathtt{poly}} f : \delta(\forall a.\alpha(a) \to H(a))$$
$$\cdot;\cdot \vdash_{\mathtt{poly}} g : \delta(\forall a.\alpha(a) \to H(a))$$

Then, $f \cong g$ if and only if the following two conditions hold:
(1) $f[\mathbb{0}] \cong g[\mathbb{0}]$
(2) $f[\delta(\mu a.\log_a(\alpha(a)))] \cong g[\delta(\mu a.\log_a(\alpha(a)))]$

Mystery of the empty type. One might wonder why $f[\mathbb{0}] \cong g[\mathbb{0}]$ is of concern in the above theorem. It seems we failed our mission to find one single type to cover all cases. However, this is almost the best we can do, as shown by this lemma:

Lemma 3.2 (no single type). *For any type $\cdot \vdash \tau$, there exist two functions f and g*

$$\cdot;\cdot \vdash_{\mathtt{poly}} f : \forall a.(a \to \mathbb{0}) + a \to \mathbb{1} + \mathbb{1}$$
$$\cdot;\cdot \vdash_{\mathtt{poly}} g : \forall a.(a \to \mathbb{0}) + a \to \mathbb{1} + \mathbb{1}$$

such that $f[\tau] \cong g[\tau]$ but $f \ncong g$. That is, τ cannot distinguish them.

Proof. Consider these three contextually distinct functions:

$$f := \Lambda a.\lambda(x{:}(a \to \mathbb{0}) + a).\mathtt{case}(x; \_.\mathtt{inl}(\star); \_.\mathtt{inl}(\star))$$
$$g := \Lambda a.\lambda(x{:}(a \to \mathbb{0}) + a).\mathtt{case}(x; \_.\mathtt{inr}(\star); \_.\mathtt{inl}(\star))$$
$$h := \Lambda a.\lambda(x{:}(a \to \mathbb{0}) + a).\mathtt{case}(x; \_.\mathtt{inl}(\star); \_.\mathtt{inr}(\star))$$

On the one hand, if τ is non-empty, $f[\tau]$ and $g[\tau]$ are contextually equivalent, because there is no term of type $(\tau \to \mathbb{0})$ so they always produce $\mathtt{inl}(\star)$. On the other hand, if τ is empty, $f[\tau]$ and $h[\tau]$ are contextually equivalent, because there is no term of type τ so they always produce $\mathtt{inl}(\star)$. In either case, there is a pair of indistinguishable functions for every τ. □

The proof of Lemma 3.2 suggests that, in general, we need at least one empty and one non-empty types for testing. Let's work out the counterexamples used in Lemma 3.2 to see how Theorem 3.1 avoids the trouble. The polymorphic type here is $\forall a.(a \to \mathbb{0}) + a \to \mathbb{1} + \mathbb{1}$. Therefore, the parameters to Theorem 3.1 are $\alpha(a) = (a \to \mathbb{0}) + a$ and $H(a) = \mathbb{1} + \mathbb{1}$, and the general type is

$$\mu a.\log_a((a \to \mathbb{0}) + a) = \mu a.a \times \mathbb{0} + \mathbb{1} \simeq \mathbb{1}$$

which is non-empty. From the counterexamples, we know we should additionally test the empty type, and Theorem 3.1 demands exactly that.

Remark. One might conjecture that we could avoid the empty type case if a is strictly positive or "well-behaved" in the argument type $\alpha(a)$. The conjecture is likely false because a variant of Lemma 3.2 can be proved using functions of the type $\forall a.a + \mathbb{1} \to (a + a) + ((a \to \mathbb{0}) \to \mathbb{1} + \mathbb{1})$ where the parameters to Theorem 3.1 are $\alpha(a) = a + \mathbb{1}$ and $H(a) = (a + a) + ((a \to \mathbb{0}) \to \mathbb{1} + \mathbb{1})$. That is, the lack of strict positivity of a in $H(a)$ alone can necessitate the testing of the empty type. More research is needed to characterize when the extra testing is necessary.

3.2 Logarithm for Inductive Types

With much more effort, the logarithm can also be defined for (strictly positive) inductive types (that is, $\mu b.\beta(b)$ where $b \in^{++} \beta(b)$). As a starting point, consider the binary tree type

$$\mathtt{tree}(a) := \mu t.a \times (\mathbb{1} + t \times t).$$

The logarithm of $\mathtt{tree}(a)$ should intuitively be the type of indexes of a-elements. A suitable indexing scheme for trees would be paths from the root to a-elements in the tree. Such a path is either an immediate stop, pointing to the element at the root, or a choice between immediate subtrees and recursively a path from the root of that subtree to some a-element within the subtree. In other words, $\log_a(\texttt{tree}(a))$ should be

$$\mu t'.\mathbb{1} + (t' + t').$$

From this example, we may conclude that the logarithm of an inductive type should also be inductively defined.

3.2.1 General Formula. To obtain the general formula for inductive types, it is helpful to treat the inductive type $\mu b.\beta(b)$ as the solution to the type isomorphism

$$b \simeq \beta(b).$$

If there is any sensible notion of logarithm, it should also respect type isomorphism, including this particular one. This means the following isomorphism should hold (modulo technical subtleties that we will clean up later):

$$\log_a(b) \simeq \log_a(\beta(b)).$$

The solution to this new isomorphism is again an inductive type. However, the isomorphism does not quite fit the pattern because its left-hand side is not a variable. We cannot directly construct

$$\mu(\log_a(b)).\log_a(\beta(b))$$

as a type—"$\log_a(b)$" is not a type variable. Therefore, we are going to replace $\log_a(b)$ with a fresh variable b' and remember the replacement when calculating the right-hand side $\log_a(\beta(b))$. This leads to the extended logarithm operation $\log_a^{\psi}(\tau)$ parametrized by a mapping ψ from variables bound by the inductive type former (*e.g.*, the b here) to their logarithm representatives (*e.g.*, the b' here). The new isomorphism after the replacement becomes:

$$b' \simeq \log_a^{b\mapsto b'}(\beta(b)) \qquad (b' \text{ fresh}).$$

The solution to the above isomorphism is the inductive type $\mu b'.\log_a^{b\mapsto b'}(\beta(b))$, with the caveat that b can potentially appear in the logarithm. Therefore, we also need to plug in the solution $b = \mu b.\beta(b)$.[1] The eventual result is thus

$$\begin{aligned} \log_a(\mu b.\beta(b)) &= \mu b'.\log_a^{b\mapsto b'}(\beta(b))[\mu b.\beta(b)/b] \qquad (b' \text{ fresh}) \\ \log_a^{b\mapsto b'}(b) &= b' \end{aligned}$$

where the trailing substitution $[\mu b.\beta(b)/b]$ happens *after* the logarithm operation[2] and takes care of any remaining b in $\log_a^{b\mapsto b'}(\beta(b))$.

In general, we should consider $\log_a^{\psi}(\tau)$ with multiple bindings in ψ because inductive types can be nested. The old logarithm $\log_a(\tau)$ is simply $\log_a^{\emptyset}(\tau)$ with an empty mapping $\emptyset$, and we will omit ψ when it is empty. The new rules for the extended logarithm operation are:

$$\begin{aligned} \log_a^{\psi}(\mu b.\tau) &:= \mu b'.\log_a^{\psi,b\mapsto b'}(\tau)[\mu b.\tau/b] \qquad (b' \text{ fresh}) \\ \log_a^{\psi}(b) &:= \psi(b) \qquad (a \neq b \text{ and } b \in \psi) \\ \log_a^{\psi}(b) &:= \mathbb{0} \qquad (a \neq b \text{ and } b \notin \psi) \end{aligned}$$

[1]It can be proved that in the current calculus, b actually will not appear in the logarithm due to strict positivity, but it is a fragile assumption that could be violated by extending inductive types. We thus prefer the more elaborated form.

[2]In general, logarithm and substitution do not commute.

All other rules are the same as the old logarithm except that they now pass the mapping ψ. See Figure 7. The extended logarithm enjoys the same property as in Theorem 3.1. It might be helpful to review the statics of the new logarithm with ψ:

Lemma 3.3 (statics of logarithm). *For any distinct type variables Δ, a, Ξ, and Ξ', any function ψ from Ξ to Ξ', and any type τ such that $\Delta, a, \Xi \vdash \tau$, we have $\Delta, a, \Xi, \Xi' \vdash \log_a^{\psi}(\tau)$.*

Proof. By induction on τ. □

The next lemma says we can construct inductive types from logarithms of inductive types.

Lemma 3.4 (strict positivity of logarithm). *For any distinct type variables Δ, a, Ξ, and Ξ', any function ψ from Ξ to Ξ', and any type τ such that $\Delta, a, \Xi \vdash \tau$ and $\Xi \in^{++} \tau$,*

$$\Delta, a, \Xi' \vdash \log_a^{\psi}(\tau) \quad \textit{and} \quad \Xi' \in^{++} \log_a^{\psi}(\tau).$$

Proof. By induction on τ. □

3.2.2 Commutativity of Logarithm and Unfolding. A nice consequence is that *logarithm commutes with unfolding*. The *unfolding* of an inductive $\mu b.\beta(b)$ refers to the expansion of $\mu b.\beta(b)$ to $\beta(\mu b.\beta(b))$. Let

$$\beta'(\tau) := \left(\log_a^{\psi, b \mapsto b'}(\beta(b))[\mu b.\beta(b)/b]\right)[\tau/b']$$

(where the substitutions happen *after* the logarithm operation) so that $\log_a^{\psi}(\mu b.\tau) = \mu b'.\beta'(b')$. We can show that the logarithm of the unfolding is the unfolding of the logarithm:

Proposition 3.5. $\log_a^{\psi}(\beta(\mu b.\beta(b))) = \beta'(\mu b'.\beta'(b'))$.

Proof. By induction on $\beta(b)$. □

3.2.3 Revisiting Trees and Lists. Going back to the binary tree example, we have

$$\begin{aligned}\log_a(\texttt{tree}(a)) = \log_a(\mu t.a \times (\mathbb{1} + t \times t)) &= \mu t'.\log_a^{t \mapsto t'}(a \times (\mathbb{1} + t \times t)) \\ &= \mu t'.\mathbb{1} + (\mathbb{0} + t' + t') \simeq \mu t'.\mathbb{1} + (t' + t')\end{aligned}$$

as expected. Another example is the logarithm of $\texttt{list}(a) = \mu l.\mathbb{1} + a \times l$ using the new rules:

$$\log_a(\texttt{list}(a)) = \log_a(\mu l.\mathbb{1} + a \times l) = \mu l'.\log_a^{l \mapsto l'}(\mathbb{1} + a \times l) = \mu l'.\mathbb{0} + (\mathbb{1} + l') \simeq \mathbb{N}.$$

The logarithm matches the standard scheme of indexing a-elements in an a-list by natural numbers.

The isomorphism $\log_a(\texttt{list}(a)) \simeq \mathbb{N}$ is more subtle than it seems; in the literature, it was sometimes wrongly assumed that one could embed $\texttt{list}(a)$ into $\mathbb{N} \times (\mathbb{N} \to a)$ where the first $\mathbb{N}$ stores the length. Such an embedding would fail when a is empty because $\texttt{list}(a)$ would be non-empty (due to the empty list) but $\mathbb{N} \times (\mathbb{N} \to a)$ would be empty. In general, τ and $\log_a(\tau) \to a$ are incomparable due to over-approximation and empty types, breaking the invertibility between logarithm and exponential.

4 SKELETONS AND INDEX REFILLING

Theorem 3.1 asserted that logarithm is general enough for testing, but we can reduce the testing further by also fixing parts of the input. For example, in the introduction, we established that we only have to consider the pair $\langle\texttt{true};\texttt{false}\rangle$ for testing `proj` of type $\forall a.a \times a \to a$ and the identity function and lists $[0, 1, \ldots, n{-}1]$ for the `map` function. The specialization not only applies to the types, but also the term-level inputs. Theorem 3.1 helps us identify the logarithms $\log_a(a \times a) = \mathbb{1} + \mathbb{1} \simeq \mathbb{2}$

$$\begin{aligned}
&\log_a^\psi(a) := \mathbb{1} && \log_a^\psi(\mathbb{1}) := \mathbb{0} \\
&\log_a^\psi(b) := \psi(b) \quad (a \neq b \text{ and } b \in \psi) && \log_a^\psi(\tau_1 \times \tau_2) := \log_a^\psi(\tau_1) + \log_a^\psi(\tau_2) \\
&\log_a^\psi(b) := \mathbb{0} \quad (a \neq b \text{ and } b \notin \psi) && \log_a^\psi(\tau_1 \to \tau_2) := \tau_1 \times \log_a^\psi(\tau_2) \\
&\log_a^\psi(\mathbb{0}) := \mathbb{0} && \log_a^\psi(\mu b.\tau) := \mu b'.\, \log_a^{\psi, b \mapsto b'}(\tau)[\mu b.\tau/b] \quad (b' \text{ fresh}) \\
&\log_a^\psi(\tau_1 + \tau_2) := \log_a^\psi(\tau_1) + \log_a^\psi(\tau_2)
\end{aligned}$$

Fig. 7. Logarithm with (strictly positive) inductive types

for `proj` and $\log_a(\texttt{list}(a)) \simeq \mathbb{N}$ for `map`, and we will see that Theorem 4.1, the enhanced theorem introduced in this section, can specialize both the type and the term-level inputs.

The term-level specialization is an inherent part of the generality of logarithm: to prove that logarithm is general enough (*i.e.,* Theorem 3.1), one shows that any general input is related to some specialized input that would witness or disprove the equality. A proof of Theorem 3.1, in a sense, can be seen as a function from generalized inputs to specialized inputs. A *good* proof of Theorem 3.1 will use only a subset of possible inputs that are the most useful for testing; the term-level specialization is to characterize such a useful subset to reduce the testing further.

Let's examine what would happen if we specialize the typing to its logarithm but not the inputs: Take the projection function `proj` : $\forall a.a \times a \to a$ for example. Following Theorem 3.1, we chose to instantiate `proj` with its logarithm (isomorphic to $\mathbb{2}$), but there are still four possible test cases:

$$\begin{array}{ll}
\texttt{proj}[\mathbb{2}](\langle\texttt{true};\texttt{true}\rangle) & \texttt{proj}[\mathbb{2}](\langle\texttt{false};\texttt{true}\rangle) \\
\texttt{proj}[\mathbb{2}](\langle\texttt{true};\texttt{false}\rangle) & \texttt{proj}[\mathbb{2}](\langle\texttt{false};\texttt{false}\rangle)
\end{array}$$

Among these four test cases, only $\langle\texttt{true};\texttt{false}\rangle$ or $\langle\texttt{false};\texttt{true}\rangle$ is needed for Theorem 3.1: the inputs $\langle\texttt{true};\texttt{true}\rangle$ and $\langle\texttt{false};\texttt{false}\rangle$ are particularly useless because they tell nothing about `proj`, and we only need one of the remaining two because the result on $\langle\texttt{true};\texttt{false}\rangle$ must be the negation of that on $\langle\texttt{false};\texttt{true}\rangle$. As for the `map` function, the tests

$$\texttt{map}[\mathbb{N}][\mathbb{N}](\texttt{id}_\mathbb{N})([0,1,2,3]) \quad \texttt{map}[\mathbb{N}][\mathbb{N}](\texttt{id}_\mathbb{N})([1,2,4,8]) \quad \texttt{map}[\mathbb{N}][\mathbb{N}](\texttt{id}_\mathbb{N})([10,20,30,40])$$

are useful for probing its behavior on lists of length 4, though we only need one of them, while

$$\texttt{map}[\mathbb{N}][\mathbb{N}](\texttt{id}_\mathbb{N})([0,0,0,0]) \quad \texttt{map}[\mathbb{N}][\mathbb{N}](\texttt{id}_\mathbb{N})([1,1,1,1]) \quad \texttt{map}[\mathbb{N}][\mathbb{N}](\texttt{id}_\mathbb{N})([9,9,9,9])$$

are less useful; they cannot detect an incorrect `map` function that permutes the output, for example.

Remark. Term-level specialization critically depends on the fact that the underlying function is polymorphic. While we need only one test case for $\mathbb{2}$-instances of polymorphic functions of type $\forall a.a \times a \to a$, we need all four test cases for arbitrary monomorphic functions of type $\mathbb{2} \times \mathbb{2} \to \mathbb{2}$.

How should we identify the most useful test cases and skip others? Recall that logarithm was chosen to be a general enough type to index all a-elements in $\alpha(a)$, or more abstractly all different ways to construct a-elements from an input of type $\alpha(a)$. The maximally useful testing is thus to fill in *distinct* indexes as a-elements. This explains why $\langle\texttt{true};\texttt{false}\rangle$ and $\langle\texttt{false};\texttt{true}\rangle$ are more useful than the other pairs and $[0,1,2,3]$ is more useful than $[0,0,0,0]$. The enhanced Theorem 4.1 states that it is indeed sufficient to consider only the maximally useful test cases.

A systematic way to generate these most useful test cases, the ones with "maximally distinct" a-elements, is to view them as a combination of the following two parts:

- **Skeletons.** A skeleton is an input with a-elements at strictly positive positions replaced by $\star$. For example, the input pairs to `proj` all have the skeleton $\langle\star;\star\rangle$, and polymorphic lists of length four all have the skeleton $[\star,\star,\star,\star]$.
- **Optimal, automatic refilling.** Given skeletons and a general enough type a^* (*i.e.,* logarithm), it is possible to automatically generate test cases where a-elements are *refilled* with distinct indexes. For example, given the skeleton $\langle\star;\star\rangle$ and the boolean type $\mathbb{2}$, the skeleton can be refilled with distinct indexes `true` and `false` and become the test case $\langle$`true`; `false`$\rangle$. Given the skeleton $[\star,\star,\star,\star]$ and the natural number type $\mathbb{N}$, we can refill the skeleton with distinct indexes 0, 1, 2, and 3 to construct one maximally useful input $[0, 1, 2, 3]$.

The idea is to enumerate all possible skeletons and consider only one optimal refilling for each skeleton. An example is that the only possible input skeleton for the function `proj` is $\langle\star;\star\rangle$ and an optimal refiller gives either $\langle$`true`; `false`$\rangle$ or $\langle$`false`; `true`$\rangle$. As for the function map, the possible skeletons for the type $\texttt{list}(a)$ are

$$[], [\star], [\star,\star], [\star,\star,\star], [\star,\star,\star,\star], [\star,\star,\star,\star,\star], \ldots.$$

Each corresponds to lists of a particular length. An optimal refiller may take them and produce

$$[], [0], [0,1], [0,1,2], [0,1,2,3], [0,1,2,3,4], \ldots$$

as concrete inputs for testing. Such a systematic refilling will lead to the enhanced Theorem 4.1 which enumerates only skeletons instead of all possible specialized inputs.

Remark. The over-approximation of logarithm, or in general a larger type for indexes, does not lead to more skeletons as long as the refilling covers all occurances of a in the input type $\alpha(a)$, which is the case when a is strictly positive in $\alpha(a)$. The machine representation of the refilled skeletons might occupy more bits with a larger index type, but the number of them stays the same.

Formally, the type of skeletons, written $\alpha^-(a)$, is defined to be $\alpha(a)$ with all strictly positive a replaced by $\mathbb{1}$, effectively calculating the remaining part that cannot be automatically refilled. The refiller is then written as

$$\{\alpha(a) \uparrow_a\} : \alpha^-(a^*) \to \alpha(a^*)$$

which takes a skeleton of type $\alpha^-(a^*)$ and generates a full test case of type $\alpha(a^*)$. The following subsections will spell out the detail of the refiller. Our goal is to show this enhanced theorem:

Theorem 4.1 (correctness with indexes). *Suppose we have*

- *An ambient kinding context Δ; and*
- *A type substitution δ such that $\cdot \vdash \delta : \Delta$, applied to all open types in the theorem; and*
- *Two type expressions $\alpha(a)$ and $H(a)$ such that*

$$\Delta, a \vdash \alpha(a) \text{ and } a \in^{++} \log_a(\alpha(a))$$
$$\Delta, a \vdash H(a) \text{ and } a \in^{+} H(a)$$

- *Two functions f and g such that*

$$\cdot;\cdot \vdash_{\texttt{poly}} f : \delta(\forall a.\alpha(a) \to H(a))$$
$$\cdot;\cdot \vdash_{\texttt{poly}} g : \delta(\forall a.\alpha(a) \to H(a))$$

Then, $f \cong g$ if and only if the following two conditions hold:

(1) $f[\mathbb{0}] \cong g[\mathbb{0}]$
(2) $f[\delta(a^*)](\delta(\{\alpha(a) \uparrow_a\})(e^-)) \cong g[\delta(a^*)](\delta(\{\alpha(a) \uparrow_a\})(e^-))$
for every (skeleton) e^- such that $\cdot;\cdot \vdash e^- : \delta(\alpha^-(a^*))$

The only difference between Theorems 3.1 and 4.1 is the last condition involving skeletons e^-.

$$\begin{aligned} (a)^-_a &\coloneqq \mathbb{1} & (\mathbb{1})^-_a &\coloneqq \mathbb{1} \\ (b)^-_a &\coloneqq b \quad (a \neq b) & (\tau_1 \times \tau_2)^-_a &\coloneqq (\tau_1)^-_a \times (\tau_2)^-_a \\ (\mathbb{0})^-_a &\coloneqq \mathbb{0} & (\tau_1 \to \tau_2)^-_a &\coloneqq \tau_1 \to (\tau_2)^-_a \\ (\tau_1 + \tau_2)^-_a &\coloneqq (\tau_1)^-_a + (\tau_2)^-_a & (\mu b.\tau)^-_a &\coloneqq \mu b.(\tau)^-_a \end{aligned}$$

Fig. 8. Rules of skeleton

4.1 Skeletons

The calculation of the type of skeletons can be summarized as *killing all strict positive occurrences* of the distinguished type variable a. See Figure 8. Every type constructor in this language preserves strict positivity, except the domain of function types. For function types, we follow the definition of strict positivity and recur only on their codomains, leaving their domains alone. Notationally, for a type expression $\alpha(a)$, we define

$$\alpha^-(\tau) \coloneqq (\alpha(a))^-_a[\tau/a]$$

as the skeleton type of $\alpha(a)$. Note that substitution $[\tau/a]$ happens *after* the calculation of the type.

4.1.1 *Examples.*

Let $\alpha(a)$ be the list type $\texttt{list}(a)$. Its skeleton type $\alpha^-(a)$ is

$$\alpha^-(a) = (\texttt{list}(a))^-_a = (\mu l.\mathbb{1} + a \times l)^-_a = \mu l.(\mathbb{1})^-_a + (a \times l)^-_a = \texttt{list}(\mathbb{1})$$

This makes sense because we can fill in all the a-elements automatically. What's left is a skeleton of type $\texttt{list}(\mathbb{1})$. Let $\alpha(a)$ be the tree type $\texttt{tree}(a)$ instead. Its skeleton type $\alpha^-(a)$ is

$$\alpha^-(a) = (\texttt{tree}(a))^-_a = (\mu t.a \times (\mathbb{1} + t \times t))^-_a = \mu t.\mathbb{1} \times (\mathbb{1} + t \times t) = \texttt{tree}(\mathbb{1})$$

as expected. A skeleton of a tree is of type $\texttt{tree}(\mathbb{1})$.

4.2 Index Refilling without Inductive Types

We will carry out the implementation of $\{\alpha(a) \uparrow_a\}$ by induction on $\alpha(a)$; it is largely forced by its type $\alpha^-(a^*) \to \alpha(a^*)$. To facilitate the induction, we define an auxiliary refiller

$$\{\tau \uparrow_a \sigma\} : \sigma(\log_a(\tau) \to a) \to \sigma((\tau)^-_a) \to \sigma(\tau).$$

The refiller $\{\tau \uparrow_a \sigma\}(\iota; e)$ is taking a skeleton e and an injective function ι that maps indexes local to τ to global indexes in the final output. The type τ such that $\Delta, a \vdash \tau$ is the fragment of the original input type $\alpha(a)$.[3] The substitution σ such that $\Delta \vdash \sigma : a$ delays the unfolding of inductive types when we step into their bodies. This makes sure the inductive definition of $\{\tau \uparrow_a \sigma\}$ is well-founded. Here, the only inductive type is the outermost inductive type $\mu a.\log_a(\alpha(a))$, but we will soon have more once we allow inductive types in $\alpha(a)$. Let's break down how the auxiliary refiller $\{\alpha(a) \uparrow_a \sigma\}(\iota; e)$ populates the skeleton with indexes:

- If $\alpha(a) = a$ itself, the skeleton $e = \star : \mathbb{1}$ and the global index can be obtained by $\iota(e) \cong \iota(\star)$.
- If $\alpha(a) = b \neq a$, the skeleton e is already filled and should be returned directly.
- If $\alpha(a)$ is a constant (*e.g.,* $\mathbb{0}$ or $\mathbb{1}$), then the skeleton e is already filled as well.
- If $\alpha(a) = \alpha_1(a) \times \alpha_2(a)$, one may fill the first component with local indexes prefixed with `inl` and the second with local indexes prefixed with `inr`. The choice of prefixes is induced by how $\log^{\psi}_a(\alpha_1(a) \times \alpha_2(a))$ was defined. The local indexes are then turned into global ones with the help of the index transformer ι.

[3] Δ is the ambient type context as in Theorem 3.1.

$$\boxed{\{\tau \uparrow_a \sigma\}(\iota; e)}$$

Auxiliary refiller without inductive types

$$\begin{aligned}
\{a \uparrow_a \sigma\}(\iota; e) &\coloneqq \iota(e) \\
\{b \uparrow_a \sigma\}(\iota; e) &\coloneqq e \qquad (a \neq b) \\
\{\mathbb{0} \uparrow_a \sigma\}(\iota; e) &\coloneqq e \\
\{\tau_1 + \tau_2 \uparrow_a \sigma\}(\iota; e) &\coloneqq \mathtt{case}(e; x.\mathtt{inl}(\{\tau_1 \uparrow_a \sigma\}(\iota \circ \mathtt{inl}; x)); y.\mathtt{inr}(\{\tau_2 \uparrow_a \sigma\}(\iota \circ \mathtt{inr}; y))) \\
\{\mathbb{1} \uparrow_a \sigma\}(\iota; e) &\coloneqq e \\
\{\tau_1 \times \tau_2 \uparrow_a \sigma\}(\iota; e) &\coloneqq \langle \{\tau_1 \uparrow_a \sigma\}(\iota \circ \mathtt{inl}; \mathtt{fst}(e)); \{\tau_2 \uparrow_a \sigma\}(\iota \circ \mathtt{inr}; \mathtt{snd}(e)) \rangle \\
\{\tau_1 \to \tau_2 \uparrow_a \sigma\}(\iota; e) &\coloneqq \lambda(x{:}\sigma(\tau_1)).\{\tau_2 \uparrow_a \sigma\}(\iota \circ \lambda(y{:}\sigma(\log_a(\tau_2))).\langle x; y\rangle; e(x))
\end{aligned}$$

$$\boxed{\{\alpha(a) \uparrow_a\}(e)}$$

Refiller from the auxiliary refiller

$$\{\alpha(a) \uparrow_a\}(e) \coloneqq \{\alpha(a) \uparrow_a [a \mapsto a^*]\}(\mathtt{roll}_{a.\log_a(\alpha(a))}; e)$$

Fig. 9. Refiller without inductive types

- If $\alpha(a) = \alpha_1(a) + \alpha_2(a)$, then we recur with the local indexes prefixed by `inl` or `inr`, depending on the head symbol of the skeleton e.
- If $\alpha(a) = \alpha_1(a) \to \alpha_2(a)$, we wait for the argument x of type $\sigma(\alpha_1(a^*))$ before proceeding to refill the skeleton $e(x)$ with indexes paired with the argument x.

The eventual refiller can then be defined as follows:

$$\{\alpha(a) \uparrow_a\}(e : \alpha^-(a^*)) \coloneqq \{\alpha(a) \uparrow_a [a \mapsto a^*]\}(\mathtt{roll}_{a.\log_a(\alpha(a))}; e).$$

4.3 Index Refilling with Inductive Types

The subtleties of inductive types force us to significantly extend and modify the (auxiliary) refiller.

(1) We need to pass the mapping ψ from the bounded type variables Ξ to their logarithm representatives as in $\log_a^\psi(\tau)$.
(2) The substitution σ was fixed to $[a^*/a]$ in the earlier section, but now a generic one such that $\Delta \vdash \sigma : a, \Xi$ to delay substitutions for bound variables introduced by inductive type formers.
(3) We also need an additional substitution $\Delta, a, \Xi \vdash \rho : \psi(\Xi)$ to take care of their logarithm representatives, because these representatives are also bound in the logarithm of inductive types.[4] See how $\log_a^\psi(\mu b.\tau)$ was defined in Figure 7.

Putting these together, our extended auxiliary refiller looks like this:

$$\{\tau \uparrow_a^\psi \sigma; \rho\}(\iota; e)$$

The last modification to the refiller is to reconcile the bottom-up nature of $\mathtt{fold}_{a.\tau}^{\tau'}$ and the top-down nature of index generation. This forces $\mathtt{fold}_{a.\tau}^{\tau'}$ to build up a continuation that takes an index transformer ι only at the end of the recursion. The complication can be illustrated by a

[4]Technically, we could fuse ρ and σ into a large substitution, but we found the alternative presentation messier.

$$\boxed{\sigma_a^{\psi;\rho}(b)}$$

Substitutions parametrized by ψ to handle variables bound by inductive type formers

$$
\begin{aligned}
\sigma_a^{\psi;\rho}(b) &\coloneqq \sigma(b) \qquad (b \notin \psi)\\
\sigma_a^{\psi;\rho}(b) &\coloneqq \sigma((\rho(\psi(b)) \to a) \to b) \qquad (b \in \psi)
\end{aligned}
$$

$$\boxed{\{\tau \uparrow_a^{\psi} \sigma; \rho\}(\iota; e)}$$

Auxiliary refiller with inductive types

$$
\begin{aligned}
\{a \uparrow_a^{\psi} \sigma; \rho\}(\iota; e) &\coloneqq \iota(e)\\
\{b \uparrow_a^{\psi} \sigma; \rho\}(\iota; e) &\coloneqq e(\iota) \qquad (a \neq b \text{ and } b \in \psi)\\
\{b \uparrow_a^{\psi} \sigma; \rho\}(\iota; e) &\coloneqq e \qquad (a \neq b \text{ but } b \notin \psi)\\
\{\mathbb{0} \uparrow_a^{\psi} \sigma; \rho\}(\iota; e) &\coloneqq e\\
\{\tau_1 + \tau_2 \uparrow_a^{\psi} \sigma; \rho\}(\iota; e) &\coloneqq \mathtt{case}(e; x.\mathtt{inl}(\{\tau_1 \uparrow_a^{\psi} \sigma; \rho\}(\iota \circ \mathtt{inl}; x)); y.\mathtt{inr}(\{\tau_2 \uparrow_a^{\psi} \sigma; \rho\}(\iota \circ \mathtt{inr}; y)))\\
\{\mathbb{1} \uparrow_a^{\psi} \sigma; \rho\}(\iota; e) &\coloneqq e\\
\{\tau_1 \times \tau_2 \uparrow_a^{\psi} \sigma; \rho\}(\iota; e) &\coloneqq \langle \{\tau_1 \uparrow_a^{\psi} \sigma; \rho\}(\iota \circ \mathtt{inl}; \mathtt{fst}(e)); \{\tau_2 \uparrow_a^{\psi} \sigma; \rho\}(\iota \circ \mathtt{inr}; \mathtt{snd}(e))\rangle\\
\{\tau_1 \to \tau_2 \uparrow_a^{\psi} \sigma; \rho\}(\iota; e) &\coloneqq \lambda(x{:}\sigma_a^{\psi;\rho}(\tau_1)).\{\tau_2 \uparrow_a^{\psi} \sigma; \rho\}(\iota \circ \lambda(y{:}\sigma(\rho(\log_a^{\psi}(\tau_2)))).\langle x; y\rangle; e(x))\\
\{\mu b.\tau \uparrow_a^{\psi} \sigma; \rho\}(\iota; e) &\coloneqq (\mathtt{fold}_{b.\sigma_a^{\psi;\rho}((\tau)_a^-)}^{\hat\sigma((\hat\rho(b') \to a) \to b)}(e; x.\lambda(\iota{:}\hat\sigma(\hat\rho(b') \to a)).\\
&\qquad\qquad \mathtt{roll}_{b.\sigma(\tau)}(\{\tau \uparrow_a^{\psi, b\mapsto b'} \hat\sigma; \hat\rho\}(\iota \circ \mathtt{roll}_{b'.\hat\sigma(\rho(\log_a^{\psi,b\mapsto b'}(\tau)))}; x))))(\iota)\\
&\text{where } \hat\sigma \coloneqq \sigma, b \mapsto \sigma(\mu b.\tau)\\
&\qquad \hat\rho \coloneqq \rho, b' \mapsto \rho(\log_a^{\psi}(\mu b.\tau))
\end{aligned}
$$

Fig. 10. Parametrized substitutions and refiller with inductive types

Haskell programming exercise to turn the skeleton $[\star, \star, \ldots, \star]$ (`[(),(),...,()]` in Haskell) to the specialized input `[0,1,...,n-1]` *using only* `foldr`. Here is the example code:

```
refillFolder :: () -> ((Nat -> Nat) -> [Nat]) -> ((Nat -> Nat) -> [Nat])
refillFolder () continuation = \iota -> iota 0 : continuation (iota . (+1))

refill :: [()] -> (Nat -> Nat) -> [Nat]
refill list iota = foldr refillFolder (\iota -> []) list iota
```

Running "`refill [(),(),(),()] id`" will give us `[0,1,2,3]`, as desired. To account for this continuation-building in the type of the new refiller, we define a special substitution $\sigma_a^{\psi;\rho}$ to indicate that recursive parts should additionally take an index transformer:

$$\sigma_a^{\psi;\rho}(b) \coloneqq \sigma(b) \qquad (b \notin \psi)$$
$$\sigma_a^{\psi;\rho}(b) \coloneqq \sigma((\rho(\psi(b)) \to a) \to b) \qquad (b \in \psi)$$

We can then write down the type of this auxiliary function:

$$\{\tau \uparrow_a^{\psi} \sigma; \rho\} : \sigma(\rho(\log_a^{\psi}(\tau)) \to a) \to \sigma_a^{\psi;\rho}((\tau)_a^-) \to \sigma(\tau).$$

See Figure 10 for the complete definition of this auxiliary function.

The eventual refiller used in Theorem 4.1 is then defined as

$$\{\alpha(a) \uparrow_a\}(e : \alpha^-(a^*)) \coloneqq \{\alpha(a) \uparrow_a^{\emptyset} [a \mapsto a^*]; \emptyset\}(\mathtt{roll}_{a.\log_a(\alpha(a))}; e)$$

which will have the correct type $\alpha^-(a^*) \to \alpha(a^*)$ because

$$\begin{aligned} \mathtt{roll}_{a.\log_a(\alpha(a))} &: \log_a(\alpha(a))[a^*/a] \to a^* \\ (\alpha(a))_a^-[a^*/a] &= \alpha^-(a^*) \\ \alpha(a)[a^*/a] &= \alpha(a^*). \end{aligned}$$

5 PROOF SKETCH OF THEOREM 4.1

(1) **Focus on one function instead of two functions.**
We can compare two functions with a reduced domain because a polymorphic function is completely determined by its selected instances. If both functions are determined by their selected instances, and their selected instances agree, then they themselves agree as well. Thus, we can focus on why a polymorphic function is determined by its own instances.

(2) **Show that composition of skeletonization, refilling, and selection is identity.**
The insight is that every general input of type $\alpha(a)$ is related to a specialized input $\alpha(a^*)$ where a-elements and indexes are related by some left-unique relation $\mathcal{R}$. Given a general input e, the related specialized input may be computed by getting its skeleton e^- and then refilling it with indexes to obtain e^*. The core lemma is to establish an admissible relation between e and e^*. For example, consider the general input $[e_0, e_1, \ldots, e_{n-1}]$ of type $\mathtt{list}(a)$. Its skeleton e^- would be $[\star, \star, \ldots, \star]$ and the corresponding specialized input with the indexes would be $[0, 1, \ldots, n{-}1]$. The relation $\mathcal{R}$ would relate e with i if and only if $e_i \cong e$. The entire process may be summarized by the following diagram:

$$\begin{tikzcd}
\begin{matrix} e = [e_0, e_1, \ldots, e_{n-1}] \\ \text{general inputs} \end{matrix} \arrow[rr, leftrightarrow, "\text{selection}", "\text{specialization}"'] \arrow[dr, bend right, "\text{skeletonization}"'] & & \begin{matrix} e^* = [0, 1, \ldots, n{-}1] \\ \text{specialized inputs} \end{matrix} \\
& \begin{matrix} e^- = [\star, \star, \ldots, \star] \\ \text{skeletons} \end{matrix} \arrow[ur, bend right, "\text{refilling}"'] &
\end{tikzcd}$$

In this section, we will spell out Step (1), showing how Step (2) (which may be summarized as Lemma 5.1 below) implies the main theorem.

Lemma 5.1 (the result of Step (2)). *Suppose we have the following data:*

- *A kinding context Δ.*
- *A closed type substitution $\delta \vdash \Delta$.*
- *The distinguished type variable a.*
- *A type expression $\alpha(a)$ such that $\Delta, a \vdash \alpha(a)$ and $a \in^{++} \log_a(\alpha(a))$.*

- *A type expression $H(a)$ such that $\Delta, a \vdash H(a)$ and $a \in^+ H(a)$.*
- *A non-empty, closed type τ representing the general type to instantiate the type variable a.*[5]
- *A closed element $e : \delta(\alpha(\tau))$ as the general input.*

Let type a^* be $\mu a. \log_a(\alpha(a))$. Given all these data, we can construct:

- *The selection function $s : \delta(a^*) \to \tau$, which relates the indexes of type a^* to the a-elements in the general input e (where a is instantiated by τ); and*
- *The skeleton $e^- : \delta(\alpha^-(a^*))$ from the input e by replacing strict positive occurrences of a with $\star$*

so that the general input e and the specialized input (the result of refilling e^-) are logically related with the selection function s assigned to the type variable a:

$$\delta\{\alpha(a) \uparrow_a\}(e^-) \sim_{\delta(\alpha(a))} e \ [a \mapsto s].$$

Furthermore, for any polymorphic function $f : \delta(\forall a.\alpha(a) \to H(a))$, we can relate the behavior of f on general inputs to that on specialized ones by parametricity:

$$f[\delta(a^*)](\delta\{\alpha(a) \uparrow_a\}(e^-)) \sim_{\delta(H(a))} f[\tau](e) \ [a \mapsto s].$$

Proof Sketch of Theorem 4.1 from Lemma 5.1. The "only if" direction trivially follows the definition of contextual equivalence. For the more interesting "if" direction, by extensionality, it is sufficient to prove that for any closed type τ, $f[\tau] \cong g[\tau]$. Note that it is decidable whether τ is empty. If τ is empty, the equivalence is witnessed by the first condition $f[\mathbb{0}] \cong g[\mathbb{0}]$. If τ is non-empty, we only need to prove that for any general input $e : \delta(\alpha(\tau))$, $f[\tau](e) \cong g[\tau](e)$. With the skeleton e^- and the selection function s given by Lemma 5.1, we have

$$\begin{aligned} f[\delta(a^*)](\delta\{\alpha(a) \uparrow_a\}(e^-)) &\sim_{\delta(H(a))} f[\tau](e) \ [a \mapsto s] \\ g[\delta(a^*)](\delta\{\alpha(a) \uparrow_a\}(e^-)) &\sim_{\delta(H(a))} g[\tau](e) \ [a \mapsto s]. \end{aligned}$$

By assumption, the expressions on the left-hand side are contextually equivalent. We wish to prove that those on the right-hand side are equivalent as well. Because $a \in^+ \delta(H(a))$ and s is a function, we know that the logical relation $- \sim_{\delta(H(a))} - \ [a \mapsto s]$ is also a function. Therefore, the equivalence on the left-hand side implies that on the right-hand side. That is, $f[\tau](e) \cong g[\tau](e)$. □

6 EXAMPLES

6.1 Recovery of Old Theorems

We can recover Theorems 2 in [Bernardy et al. 2010], which is for the functions of the type

$$\forall a.(F(a) \to a) \times (G(a) \to K) \to H(a).$$

This means the $\alpha(a)$ for Theorem 4.1 should be $(F(a) \to a) \times (G(a) \to K)$. Therefore,

$$\begin{aligned} \log_a(\alpha(a)) &= \log_a(F(a) \to a) + \log_a(G(a) \to K) \\ &= F(a) \times \mathbb{1} + \log_a(G(a) \to K) \\ &\simeq F(a) + G(a) \times \log_a(K). \end{aligned}$$

One can show that for any constant K, $\log_a(K) \simeq \mathbb{0}$. Thus,

$$\log_a(\alpha(a)) \simeq F(a) + G(a) \times \mathbb{0} \simeq F(a).$$

[5]The actual lemma developed in the supplemental material additionally asks for a witness to the non-emptiness of τ. We proved the availability of such a witness as part of the decidability of emptiness, so either version of Lemma 5.1 is correct.

Hence, the general type given by Theorem 4.1 is $a^* = \mu a. \log_a(\alpha(a)) \simeq \mu a.F(a)$. The refiller $\{\alpha(a) \uparrow_a\}$ is slightly involved due to the type isomorphisms, but we can check its type

$$\begin{aligned}\alpha^-(a^*) \to \alpha(a^*) &= ((F(a) \to a) \times (G(a) \to K))^-_a[a^*/a] \to (F(a^*) \to a^*) \times (G(a^*) \to K)\\ &= (F(a^*) \to \mathbb{1}) \times (G(a^*) \to (K)^-_a[a^*/a]) \to (F(a^*) \to a^*) \times (G(a^*) \to K).\end{aligned}$$

It can be shown that $(K)^-_a = K$ for any constant K. As a result,

$$\begin{aligned}\alpha^-(a^*) \to \alpha(a^*) &= (F(a^*) \to \mathbb{1}) \times (G(a^*) \to K) \to (F(a^*) \to a^*) \times (G(a^*) \to K)\\ &\simeq \mathbb{1} \times (G(a^*) \to K) \to (F(a^*) \to a^*) \times (G(a^*) \to K)\\ &\simeq (G(a^*) \to K) \to (F(a^*) \to a^*) \times (G(a^*) \to K).\end{aligned}$$

The refiller, forced by its type, is essentially the function $\lambda(g{:}(G(a^*) \to K)).\langle \mathtt{roll}_{a.F(a)}; g\rangle$ up to type isomorphism. There is no need to additionally test the empty type, because when a is instantiated with an empty type, either $F(a)$ is also empty, in which case a^* is also empty and there is no need to test another empty type, or $F(a)$ is not empty, in which case $F(a) \to a$ is empty and all functions of the instantiated type are contextually equivalent.

Thus far, we have fully recovered their Theorem 2, except that our calculus only allows strictly positive inductive types and thus a must be strictly positive in $F(a)$ for $\mu a.F(a)$ to be well-formed. Bernardy et al. [2010] instead imposed a potentially weaker condition that the F-algebra exists. In any case, we can handle all the function types mentioned in their Theorem 5 by calculating their logarithms directly without going through embedding-projection pairs.

6.2 Revisiting Map

The function map has the type $\forall a.\forall b.(a \to b) \to \mathtt{list}(a) \to \mathtt{list}(b)$. We first work on the type variable b and then the type variable a. (The order is insignificant in this example.)

Let $\alpha(b) = (a \to b) \times \mathtt{list}(a)$ (after currying) and $H(b) = \mathtt{list}(b)$. Theorem 4.1 gives us a general type isomorphic to $\mu b.a \simeq a$ and $\{\alpha(b) \uparrow_b\}$ is essentially putting the identity function as the first argument, up to type isomorphism. The testing of map is then reduced to testing of the following instance. (Note that the instance covers the empty type case where $b = \mathbb{0}$: the type $a \to b$ is inhabited only when a is also empty, and its only element is essentially $\mathtt{id}_{\mathbb{0}}$ the identity function on the empty type, which corresponds to the following instance further specialized with $a = \mathbb{0}$.[6])

$$\Lambda a.\mathtt{map}[a][a](\mathtt{id}_a) : \forall a.\mathtt{list}(a) \to \mathtt{list}(a).$$

As for the type variable a, let $\alpha(a) = \mathtt{list}(a)$ and $H(a) = \mathtt{list}(a)$. The theorem gives a type isomorphic to $\mu a.\mathbb{N} \simeq \mathbb{N}$ and $\{\alpha(a) \uparrow_a\}$ is essentially turning a $\mathbb{1}$-list of $\star$ into the list $[0, \ldots, n-1]$ of the same length, again up to type isomorphism.[7] The condition on the empty type is true because there is only one element of type $H(\mathbb{0}) = \mathtt{list}(\mathbb{0})$. This further reduces the testing to

$$\mathtt{map}[\mathbb{N}][\mathbb{N}](\mathtt{id}_{\mathbb{N}})([0, \ldots, n-1]) : \mathtt{list}(\mathbb{N})$$

for all n, as expected.

6.3 Functor Laws

In Haskell, an instance of **Functor** for a functor $F(-)$ comes with a function fmap of type $\forall a.\forall b.(a \to b) \to F(a) \to F(b)$, and it should satisfy these two laws:

(1) $\mathtt{fmap}[a][a](\mathtt{id}_a) \cong \mathtt{id}_{F(a)}$
(2) $\mathtt{fmap}[a][c](f \circ g) \cong \mathtt{fmap}[b][c](f) \circ \mathtt{fmap}[a][b](g)$

[6]A more straightforward argument in the case of the map function is that $H(\mathbb{0}) = \mathtt{list}(\mathbb{0})$ has only one element, but the argument in the main text works for any type constructor, not just $\mathtt{list}(-)$. We need the generality for Section 6.3.
[7]See Section 3.2 for the calculation of $\log_a(\mathtt{list}(a))$.

It can be proved that the second law follows from the first one, using the free theorems [Wadler 1989]. Here we give another proof using our Theorem 4.1. Consider a candidate implementation `fmap′` sharing the same type with `fmap`. Following the same argument for the variable b in Section 6.2 with `list`$(-)$ replaced with $F(-)$, checking whether `fmap′` and `fmap` agree boils down to checking

$$\Lambda a.\texttt{fmap}'[a][a](\texttt{id}_a) \cong \Lambda a.\texttt{fmap}[a][a](\texttt{id}_a) \cong \Lambda a.\texttt{id}_{F(a)}$$

which is exactly the first functor law. If `fmap′` satisfies the first functor law, then it is contextually equivalent to the correct `fmap`, which means it automatically satisfies the second law.

7 HASKELL IMPLEMENTATION AND EXPERIMENTS

We built a Haskell library, PolyCheck, that implements the logarithm, skeletonization and refilling operations; its source code is available in the accompanying artifacts [Hou (Favonia) and Wang 2021] and on GitHub [Wang 2021]. PolyCheck has two backends, QuickCheck and SmallCheck, and users can choose either one. To test a polymorphic function `foo`, one uses the Template Haskell function `monomorphic` to generate a monomorphized function `foo_mono`, with all of its type variables instantiated by the types calculated by the logarithm operation. Based on the skeletonization and refilling operations, new **`Arbitrary`** instances are provided by the QuickCheck backend to generate inputs for the monomorphized function, and new **`Series`** instances by the SmallCheck backend for the same purpose.

7.1 Adaptation to Haskell

PolyCheck assumes that the functions to be tested are type-safe and have no effects. The following Haskell features (and more advanced ones) are not supported due to the limit of the theory: datatypes that are not strictly positive, non-regular datatypes (*e.g.,* finger trees), generalized algebraic datatypes (GADTs), type families, mutually recursive types, higher-rank types and existential types, higher-kinded type variables and datatype parameters, typeclasses, *etc.*

However, because the **`Eq`** typeclass is essential for a polymorphic property to compare polymorphic results, we made an exception to allow the (==) function from the **`Eq`** typeclass, but assumed it is used only for comparing results. If this assumption is violated, for example in the `group` function in Haskell the equality is used to compare inputs, the correctness is not guaranteed. (One has to test `groupBy` instead of `group`, effectively using dictionary passing.)

PolyCheck does not check the usage of unsupported features, but the compilation will likely fail. In some cases, such as non-regular recursive types, the compilation might not terminate. Moreover, PolyCheck does not test the empty type case mainly due to the following two reasons:

(1) The support of the empty type in QuickCheck and SmallCheck is poor: for example, there is no instance of **`Arbitrary`** for the type [**`Void`**] (the type of lists of the empty type) to generate an element of it, even though the empty list is a valid element. In general, we have to insert an **`Arbitrary`** instance for each input type instantiated with the empty type.
(2) It does not seem to be useful in practice. Note that QuickCheck uses **`Integer`** and does not check empty types, either. PolyCheck is on par with QuickCheck in terms of correctness.

The logarithm operation without inductive types largely follows Figure 7, with changes to handle n-ary type constructs more straightforwardly: The binary product types are replaced by n-ary tuple types, and the binary sum types by n-ary sum types in Haskell. For the function types, with the help of n-ary tuples, the logarithm operation sends $\tau_1 \to \ldots \to \tau_n \to \tau$ directly to $\tau_1 \times \ldots \times \tau_n \times \log_a(\tau)$ instead of $\tau_1 \times (\cdots \times (\tau_n \times \log_a(\tau)) \cdots)$, avoiding excessive wrapping.

The inductive types are handled slightly differently from Section 3.2, because in Haskell they are not defined as $\mu a.\tau$, but are explicitly named and defined by equations $\mathfrak{d} = \tau$, where $\mathfrak{d}$ is the name of the inductive type and τ is the body of its definition. The name $\mathfrak{d}$ can occur in τ for recursion. Let's call the τ here $\textsc{body}(\mathfrak{d})$ as the body of $\mathfrak{d}$. Then, for each inductive type $\mathfrak{d}$ of which we need to calculate the logarithm, a new inductive type $\mathfrak{d}'$ will be generated as the logarithm of $\mathfrak{d}$. The mapping ψ in Section 3.2 is repurposed as a global mapping from names $\mathfrak{d}$ to the names of their logarithms $\mathfrak{d}'$. If the logarithm of $\mathfrak{d}$ was already generated (that is, found in ψ), then we will recycle the existing $\mathfrak{d}'$. As a result, the logarithm operation for inductive types now has side effects to inspect or change the global mappings ψ and generate new inductive types when necessary:

$$\log_a(\mathfrak{d}) := \psi(\mathfrak{d}) \quad \text{(when } \mathfrak{d} \in \psi \text{ already)}$$

$$\log_a(\mathfrak{d}) := \mathfrak{d}' \quad \text{(when } \mathfrak{d} \notin \psi \text{ yet, which means we encounter } \mathfrak{d} \text{ for the first time)}$$

after running these steps in order:

1. Generate a fresh name as $\mathfrak{d}'$.
2. Extend ψ with $\mathfrak{d} \mapsto \mathfrak{d}'$.
3. Calculate $\tau \leftarrow \log_a(\textsc{body}(\mathfrak{d}))$ with the new extended ψ.
4. Create a new inductive type $\mathfrak{d}'$ with the body τ.

The skeletonization and refilling operations for inductive types are adjusted accordingly.

Remark. It should be possible to port PolyCheck to other languages with meta-programming facility to manipulate type expressions and to inspect and insert definitions of types and functions. OCaml with its ppx ecosystem [Jane Street Group 2018] is one possibility.

7.2 Experiments

We chose four simple polymorphic functions, and showed PolyCheck can reduce the number of test cases needed to find a counterexample that reveals the disparency between the correct implementation and an incorrect one with a bug. Here are the functions and introduced bugs:

1. `apply3` : $\forall a.a \to (a \to a) \to a$.[8] It is expected to apply the second argument three times on the first argument; the incorrect implementation only applies it two times.
2. `map` : $\forall a.\forall b.(a \to b) \to \texttt{list}(a) \to \texttt{list}(b)$. It should agree with the usual `map` function; the incorrect implementation swaps the first two elements of the input list (if existing).
3. `takeWhile` : $\forall a.(a \to 2) \to \texttt{list}(a) \to \texttt{list}(a)$. It should agree with the `takeWhile` function in the Haskell base library; the incorrect implementation additionally swaps the last two elements of the input list (if existing).
4. `zipWith` : $\forall a.\forall b.\forall c.(a \to b \to c) \to \texttt{list}(a) \to \texttt{list}(b) \to \texttt{list}(c)$. It should agree with the `zipWith` in the Haskell base library; the incorrect implementation swaps the last two elements of the input list (if existing).

We tested these functions using QuickCheck and SmallCheck, both with and without the monomorphization by PolyCheck. For the plain QuickCheck and SmallCheck, we instantiated all type variables with the **`Integer`** type, which is the default approach taken by QuickCheck (see the `polyQuickCheck` function).[9] The source code for the experiments is available in Hou (Favonia) and Wang [2021]. Table 1 and Figure 11 show the numbers of test cases needed to find a counterexample (including the counterexample itself). Because QuickCheck uses random inputs, we ran the tests 10000 times and calculated their means and standard deviations.

Let's discuss the case of SmallCheck first: PolyCheck reduces the needed test cases for all the functions, especially the `map` function, where PolyCheck effectively fixes the functional argument

[8] Interestingly, if we swap the order of the two arguments to the `apply3` function, SmallCheck needs even more test cases to find a counterexample, while PolyCheck can still falsify the equality with one test case.

[9] SmallCheck does not have a built-in way to test polymorphic functions.

Table 1. Numbers of test cases needed to find a counterexample using QuickCheck and SmallCheck

Tested Function	QuickCheck		SmallCheck	
	Original	PolyCheck	Original	PolyCheck
`apply3`	3.06 ± 1.10	1.00 ± 0.00	2	1
`map`	4.73 ± 1.46	4.18 ± 1.17	1339	3
`takeWhile`	8.14 ± 10.76	7.72 ± 10.69	6	5
`zipWith`	5.72 ± 2.32	5.12 ± 2.03	*(timeout)*	10

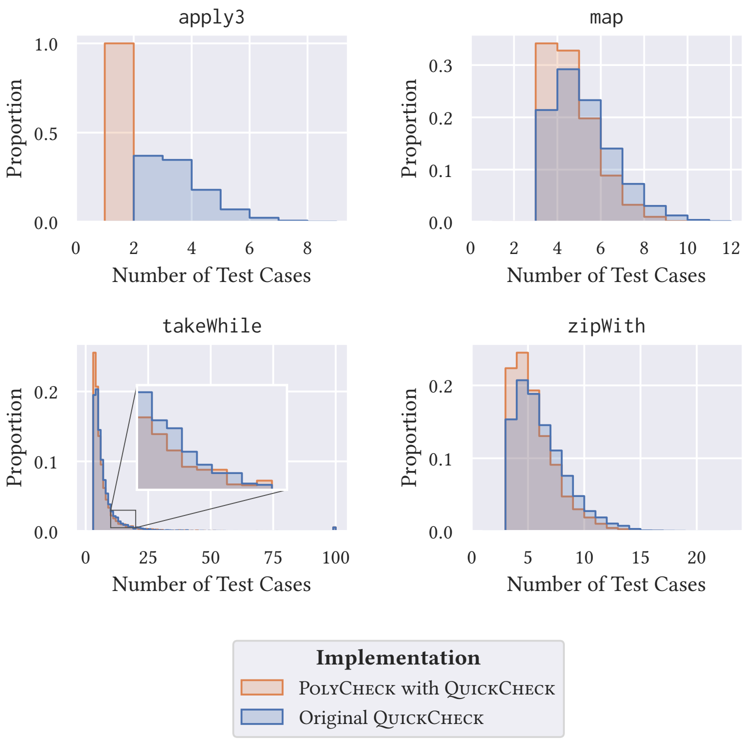

Fig. 11. Distributions of numbers of test cases needed to find a counterexample using QuickCheck

to the identity function and SmallCheck is checking all the function from `Integer` to `Integer` up to depth five. Note that SmallCheck itself could not finish the testing of `zipWith` within one minute even when the maximum depth is set to one, which indicated a bug.

In the case of QuickCheck, PolyCheck also consistently needs fewer test cases. The difference is smaller because the integers randomly generated by QuickCheck are likely distinct, which means they are a good choice for refilling.[10] Still, QuickCheck may generate redundant test cases that PolyCheck can avoid. For example, considering the `map` function, it is possible that the first two elements of the input list happen to be the same, or the input function maps the first two elements into the same integer. Both make the first two elements of the output indistinguishable.

[10]See the discussion in Section 4.

8 DISCUSSION AND FUTURE WORK

We presented a mechanical, direct way to compute a general type to instantiate a polymorphic function for testing. The logarithm operation has been considered by Peter Hancock before [Hancock 2019] though he did not consider its application in program testing. Bernardy [2017] proposed calculating the general type via classical logical negation, which shares similarities with logarithm because negation finds a type τ such that $\tau \to \bot$ matches the argument type, while logarithm is for $\tau \to a$; however, logarithm seems to explain the type variable cases better. The Haskell library METAMORPH [Xia 2017b] comes with operators similar to our logarithm and refilling operations and can monomorphize functions and specialize their inputs, but it does not handle general inductive types or functions with more than one type variable. Other related work includes the research on shapes [Jay 1995] and containers [Abbott et al. 2003; Altenkirch et al. 2015]; our logarithm operation resembles the translation from types to containers in Abbott et al. [2003] because their translator, conceptually, is trying to identify an exponent τ such that a^τ matches the original type.

8.1 Enumeration-Based and Randomized Testing

Our experiments not only showed that both QUICKCHECK and SMALLCHECK can benefit from our technique, but also that the enumeration-based SMALLCHECK can sometimes outperform QUICKCHECK on the monomorphized instances. (See Table 1.) This indicates that randomized testing can lose its edge when the testing case space is significantly reduced. However, a hybrid approach that combines QUICKCHECK and SMALLCHECK should enjoy the best of the two worlds.

8.2 Information Beyond Typing

Our work only relies on typing information and nothing else, which is both its strength and its weakness. In practice, a function often imposes additional requirements on its inputs; for example, a sorting algorithm may demand the element comparator to form a total ordering, or a parallel list reducer might require the input binary reducer to be associative. Such information has not been integrated into our current theory. This issue was noticed by Bernardy et al. [2010], and results such as the 0-1-2 principle established in Voigtländer [2008] still defy systematic analysis.

8.3 Inspectability of Inputs

The types given by Theorem 4.1 are far from being optimal. Consider the length function

$$\texttt{length} : \forall a.\texttt{list}(a) \to \mathbb{N}.$$

The theorem will give the general type $\mathbb{N}$ (up to type isomorphism). However, in this case, $\mathbb{1}$ is enough because there is no way for the `length` function to inspect the elements. In general, Theorem 4.1 is being conservative by reserving distinct indexes for all constructible elements, but this might be overkill. Taking *inspectability* into consideration should improve the testing theory even further. Another example is the function

$$\texttt{exists} : \forall a.(a \to \mathbb{2}) \to \texttt{list}(a) \to \mathbb{2}$$

where the theorem will again give $\mathbb{N}$ as the general type and fix the second argument to $[0, \ldots, n-1]$, leaving the first argument of type $\mathbb{N} \to \mathbb{2}$ unspecified. However, a better solution might be plugging in the type $\mathbb{2}$ for a and fixing the first argument to the identity function, leaving the second argument unspecified. It is arguably easier to enumerate all $\mathbb{2}$-lists than to enumerate all possible functions of type $\mathbb{N} \to \mathbb{2}$. We can potentially derive such a solution from the inspectability of a.

Remark. Inspectability is morally the same as the *terminal* view mentioned in Xia [2017a], but more research is needed to establish a mechanical monomorphization for more general type forms.

8.4 More Expressive Types

There are many possible directions to extend the language: To work with ML-style modules, it is crucial to handle unknown type constructors or at least the prenex fragment of System F_ω. Another direction is to consider coinductive types, which should be achievable with minimum changes. One may also consider nested (non-regular) inductive types [Bird and Meertens 1998]; for example, the nest type `nest`(a) is the least solution to the type isomorphism

$$N(a) \simeq \mathbb{1} + a \times N(a \times a)$$

and its logarithm with respect to a should be derivable by solving these two isomorphisms:

$$\begin{aligned} N(a) &\simeq \mathbb{1} + a \times N(a \times a) \\ \log_a(N(a)) &\simeq \mathbb{0} + (\mathbb{1} + \log_a(N(a \times a))) \simeq \mathbb{1} + \log_a(N(a \times a)) \end{aligned}$$

where $\log_a(N(a \times a))$ should be $\log_a(N(a))[a \times a/a] \times \log_a(a \times a)$, following the "change of base" formula $\log_a(c) = \log_b(c) \times \log_a(b)$ in the usual arithmetic where a, b, and c are positive reals. If we again allocate a fresh variable N' for the logarithm, we are essentially solving these isomorphisms:

$$\begin{aligned} N(a) &\simeq \mathbb{1} + a \times N(a \times a) \\ N'(a) &\simeq \mathbb{1} + N'(a \times a) \times \mathbb{2}. \end{aligned}$$

Because $N'(a)$ does not depend on a, the least solution for $N'(a)$ is $\mu n'.\mathbb{1} + n' \times \mathbb{2}$. Similar analysis can be done for the bush types [Bird and Meertens 1998] and the finger trees [Hinze and Paterson 2006]. The point is that one should be able to calculate their logarithms as well.

Dependent types lead to yet another interesting direction: for example, while $\mathbb{N}$ is the best type for `list`(a), the finite type with n elements is the best for lists *of length n*. The logarithm of the list type of length n should thus be the finite type with n elements. Such precision can only be achieved by dependent types or some forms of them (*e.g.,* indexed containers [Altenkirch et al. 2015]).

Finally, it is possible to lift the prenex restriction but leave $\log_a(\forall b.\tau)$ generally undefined except for $\log_a(K) = \mathbb{0}$ when $a \notin K$. Further research is needed to remove this undefinedness. In addition to the inherent difficulty of type quantifiers, ideally, the logarithm should also commute with the Church encoding. For example, the equation $\log_a^\psi(\tau_1 \times \tau_2) = \log_a^\psi(\tau_1) + \log_a^\psi(\tau_2)$ should imply

$$\log_a^\psi(\forall b.(\tau_1 \to \tau_2 \to b) \to b) \simeq \forall b.(\log_a^\psi(\tau_1) \to b) \to (\log_a^\psi(\tau_2) \to b) \to b$$

by applying the Church encodings on both sides. How the logarithm may be defined so that this equation holds remains open.

ACKNOWLEDGMENTS

We appreciate the insightful feedback from Li-yao Xia on Lemma 3.2, which led to significant simplification. We also appreciate the feedback from the participants of *IFIP Working Group 2.8 — Functional Programming* in May 2019 where the work in progress was presented; in particular, Conor McBride kindly introduced Peter Hancock's work to us. In addition, we thank Carlo Angiuli and anonymous reviewers for their comments on earlier drafts. This research originated from an attempt in Fall 2015 to apply randomized checking to the library code used in the course *15-210 (Parallel and Sequential Data Structures and Algorithms)* at Carnegie Mellon University [CMU 15-210 Staff 2018], and we appreciate the support from Sam Westrick when developing the checker.

DATA AVAILABILITY STATEMENT

The source code for the experiments in Section 7 is available at https://doi.org/10.1145/3462305.

REFERENCES

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2003. Categories of containers. In *International Conference on Foundations of Software Science and Computation Structures*. Springer, 23–38. https://doi.org/10.1007/3-540-36576-1_2

Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. 2015. Indexed containers. *Journal of Functional Programming* 25, e5. https://doi.org/10.1017/S095679681500009X

Jean-Philippe Bernardy. 2017. *A dual view of testing of (polymorphic) programs.* https://jyp.github.io/posts/polytest-two.html

Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. 2010. Testing Polymorphic Properties. In *Programming Languages and Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 125–144. https://doi.org/10.1007/978-3-642-11957-6_8

Richard Bird and Lambert Meertens. 1998. Nested datatypes. In *International Conference on Mathematics of Program Construction*. Springer, 52–67. https://doi.org/10.1007/BFb0054285

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

CMU 15-210 Staff. 2018. *The SEQUENCE signature.* http://www.cs.cmu.edu/afs/cs/academic/class/15210-f15/www/docs/sig/sequence/SEQUENCE.html

Peter Hancock. 2019. Personal communication, along with some notes.

Ralf Hinze and Ross Paterson. 2006. Finger trees: a simple general-purpose data structure. *Journal of functional programming* 16, 2 (2006), 197–217. https://doi.org/10.1017/S0956796805005769

Kuen-Bang Hou (Favonia) and Zhuyang Wang. 2021. *Replication Package for Article: Logarithm and Program Testing.* https://doi.org/10.1145/3462305

Jane Street Group. 2018. *Ppxlib - Meta-programming for OCaml.* https://github.com/ocaml-ppx/ppxlib

C Barry Jay. 1995. A semantics for shape. *Science of Computer Programming* 25, 2-3 (1995), 251–283. https://doi.org/10.1016/0167-6423(95)00015-1 Selected Papers of ESOP'94, the 5th European Symposium on Programming.

John C Reynolds. 1983. Types, abstraction and parametric polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congres*. 513–523.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (Victoria, BC, Canada) *(Haskell '08)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1411286.1411292

Janis Voigtländer. 2008. Much Ado About Two (Pearl): A Pearl on Parallel Prefix Computation. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. ACM, New York, NY, USA, 29–35. https://doi.org/10.1145/1328438.1328445

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (Imperial College, London, United Kingdom) *(FPCA '89)*. ACM, New York, NY, USA, 347–359. https://doi.org/10.1145/99370.99404

Zhuyang Wang. 2021. *PolyCheck: Testing Polymorphic Functions.* https://github.com/hawnzug/polycheck

Li-yao Xia. 2017a. *A terminal view of testing polymorphic functions.* https://blog.poisson.chat/posts/2017-06-29-terminal-monomorphization.html

Li-yao Xia. 2017b. *Test Polymorphic Functions with Metamorph.* https://github.com/Lysxia/metamorph