Logarithm and Program Testing

KUEN-BANG HOU (FAVONIA), 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 layers of ad-hoc embedding-projection pairs which require creativity and obscure the testing semantics. This paper presents a mechanical monomorphization, a step towards automatic testing for polymorphic properties. It turns out the calculation of a general enough type for monomorphization is exactly *logarithm*.

Additional Key Words and Phrases: parametricity, polymorphism, logarithm

1 INTRODUCTION

1.1 Background

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 the number of limited 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 as the Haskell library QuickCheck [Claessen and Hughes 2000], it has inspired tools such as SmallCheck [Runciman et al. 2008] for finite values and has been ported to virtually every popular programming language.

In this paper, we focus on the ultimate correctness criterion—whether a particular implementation is equivalent to the reference one. This criterion boils down to testing whether two functions of some given type are equivalent. In other words, we want to understand the semantics of types in terms of testing. For example, consider the map function:

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

We are interested in comparing a claimed implementation $\texttt{map}'$ of the same type to this reference function. It turns out that we only need to test their monomorphic instances $\texttt{map}[\mathbb{N}][\mathbb{N}]$ and $\texttt{map}'[\mathbb{N}][\mathbb{N}]$ with the first argument being the identity function and the second being of the form $[0, \ldots, n-1]$ for all n. The intuition is that polymorphism forces the application of the function (the first argument) exactly once to each element in the list (the second argument), and the only mistakes one can make are to omit an item, to duplicate an item, and to permutate the output. The identity function and the lists $[0, \ldots, n-1]$ can detect any discrepancy.

In general, polymorphism limits the number of possible programs and should theoretically reduce the needed testing. Nonetheless, most testing frameworks can only handle monomorphic functions; some arbitrarily choose `int` as the type for instantiation but still fail to exploit the full potential of polymorphism.

Bernardy et al. [2010] have shown that we can monomorphize a wide range of polymorphic function testing. In particular, for functions

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

Author's address: Kuen-Bang Hou (Favonia), Computer Science Department, University of Minnesota, Minneapolis, Minnesota, 55455, USA, favonia@umn.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

$$\begin{aligned}
\tau &\coloneqq a \mid \tau_1 \to \tau_2 \mid \mathbb{0} \mid \tau_1 + \tau_2 \mid \mathbb{1} \mid \tau_1 \times \tau_2 \mid \mu a.\tau \\
\rho &\coloneqq \tau \mid \forall a.\rho \\
e, f, g, h &\coloneqq x \mid \Lambda a.e \mid e[\tau] \mid \lambda x{:}\tau.e \mid e_1(e_2) \mid \\
&\qquad \mathtt{abort}(e) \mid \mathtt{inl}(e) \mid \mathtt{inl}(e) \mid \mathtt{case}(e; x.e_x; y.e_y) \mid \\
&\qquad \star \mid \langle e_1; e_2 \rangle \mid \mathtt{fst}(e) \mid \mathtt{snd}(e) \mid \\
&\qquad \mathtt{roll}_{a.\tau}(e) \mid \mathtt{unroll}_{a.\tau}(e) \mid \mathtt{fold}_{a.\tau}(x.e_x; e)
\end{aligned}$$

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

where F, G, H are functors (whose arguments only appear at positive positions) and K is constant in a, it suffices to check $f[\mu a.F(a)](\mathtt{roll}_{a.F(a)})$ where $\mathtt{roll}_{a.F(a)}$ is the initial F-algebra. For more general forms

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

if one can embed τ into $(F(a) \to a) \times (G(a) \to K)$ then the theorem still applies, after inserting the mediating projection functions.

This paper aims at advancing the theory by providing a direct, mechanical calculation to derive F from τ. It replaces layers of embedding-projection pairs in the literature which require creativitiy and arguably obscure the testing semantics. As it turns out, it matches the notion of the logarithm function, and thus its name. It not only recovers known results but also sheds light on the testing semantics of nested (nonregular) data types and other type constructors.

2 THE LANGUAGE AND NOTATION

We will use the prenex fragment of System F extended with the empty type $\mathbb{0}$, sum types $\tau_1 + \tau_2$, the unit type $\mathbb{1}$, product types $\tau_1 \times \tau_2$, and data types $\mu a.\tau$ (where a is positive in τ) as our base language. We focus on the prenex fragment with additional built-in types instead of the full System F, because the technology in this paper cannot handle all higher-rank polymorphism yet and the fragment already covers most cases in practice. In other words, it characterizes a fully solved part used in practical functional programming languages.[1] In later sections, we will discuss possible further extensions such as union types $\tau_1 \cup \tau_2$.

We follow the standard syntax as in Figure 1, where the term $\star$ is the unique element of the unit type $\mathbb{1}$. For brevity, we will omit the subscripts $a.\tau$ in $\mathtt{roll}_{a.\tau}$, $\mathtt{unroll}_{a.\tau}$ and $\mathtt{fold}_{a.\tau}$ when they are clear from the context.

3 LOGARITHM OPERATION

Our first goal is to derive a general enough type τ' from a type expression τ so that for a function

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

it suffices to check $f[\tau']$. One benchmark from the literature [Bernardy et al. 2010] is to derive $\mu a.F(a)$ when $\tau = (F(a) \to a) \times (G(a) \to K)$. The intuition is that $\mu a.F(a)$ captures all possible ways to use a function of type $F(a) \to a$ to construct an element of type a. An element of type $\mu a.F(a)$ is considered the *recipe* of, the *label* of, or the *position* of an a-element available to the tested function. The operation we will define will capture the same concept but work on more general forms.

For the reason that will be clear later, we write the type operation as $\log_a$ where $\log_a(\tau)$ will give us a general enough type for representing all possible ways to construct an element of type a.

[1] Alternatively, we can lift the prenex restriction; see Section 8.

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

Fig. 2. Basic rules of logarithm

If τ has the form $\tau_1 \times \tau_2$, then one may generate such an element using the τ_1 part or the τ_2 part. If τ is of the form $\tau_1 + \tau_2$, one may still construct an element using τ_1 or τ_2, though only one of them is available at any given time. If τ is constant in a, then it is useless. If τ is a itself, then $\log_a(a)$ should be the type $\mathbb{1}$ because there is exactly one way to obtain an element of type a. We summarize these observations as the first six rules in Figure 2 (all the rules except the last one).

When it comes to the function types $\tau_1 \to \tau_2$, how can we exploit this functional argument? The idea is that any a-element construction must involve applying this function to some input of type τ_1 and then constructing the a-element from its result of type τ_2. In other words, every construction can be described by a pair of the input of type τ_1 together with a "recipe" of type $\log_a(\tau_2)$ to continue the construction based on the result of the application, justifying the last equation in Figure 2.

The rules we have so far made it clear that the operation we want is *logarithm*; $\log_a(\tau)$ matches the usual definition of logarithm perfectly when τ is a, $\mathbb{1}$, $\tau_1 \times \tau_2$, and $\tau_1 \to \tau_2$. Because our goal is to cover as many type expressions as possible, our logarithm is no longer inverse to exponential. That is, $\log_a(\tau) \to a$ is in general not isomorphic to τ. For example, $\mathbb{0} \not\cong \log_a(\mathbb{0}) \to a \simeq \mathbb{1}$. We will even see cases where neither $\log_a(\tau) \to a$ nor τ is larger than the other. People interested in precise invertability may refer to Naperian functors coined by Peter Hancock.

We can show that our logarithm fulfills our purpose by proving the following theorem:

Theorem 3.1 (correctness). *For any two functions f, g of type*

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

where H is a functor, if $\log_a(\tau)$ exists and a is positive in $\log_a(\tau)$, then $f \cong g$ if and only if

$$f[\mu a.\log_a(\tau)] \cong g[\mu a.\log_a(\tau)].$$

Note the ambient typing context may have free type variables around, but not free expression variables.

3.1 Optimization with Union Types

The logarithm of sum types leaves some room to be desired. The type $\log_a(\tau_1) + \log_a(\tau_2)$ assigns district labels to all possible ways to construct an element using the τ_1 part or the τ_2 part, but only one of them exists at any given time. Therefore, it is more optimized for the two parts to share their labels. If the system is further extended with union types $\tau_1 \cup \tau_2$ with (possibly overlapping) constructors $\texttt{inl}^{\cup}$ and $\texttt{inr}^{\cup}$, then we may refine the rule for sum types as follows:

$$\log_a(\tau_1 + \tau_2) = \log_a(\tau_1) \cup \log_a(\tau_2)$$

Intuitively, the union type corresponds to the max operator and thus should commute with the logarithm. For real numbers, the logarithm of the maximums is the maximum of the logarithms. Therefore, one might consider the following optional rule:

$$\log_a(\tau_1 \cup \tau_2) = \log_a(\tau_1) \cup \log_a(\tau_2)$$

This rule implies that labels overlap in a way compatible with how elements in the original union type overlap. The implementation details, however, are beyond the scope of this paper.

3.2 Logarithm for Data Types

The logarithm can also be defined for (recursive) data types. The idea is that the logarithm of a recursive definition is still recursive. As an example, consider the binary tree type

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

The logarithm of $\mathtt{tree}(a)$ should intuitively be the type of paths from the root to nodes 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. In other words, $\log_a(\mathtt{tree}(a))$ should be $\mu t'.\mathbb{1} + (t' + t')$. The logarithm itself is also recursive.

In general, the recursive type $\mu b.\tau$ may be seen as a solution to the type isomorphism $b \simeq \tau$, and the calculation of the logarithm tries to solve the following two isomorphisms at the same time:

$$b \simeq \tau$$
$$\log_a(b) \simeq \log_a(\tau)$$

The solution to the second isomorphism is again a recursive type; however, its left-hand side $\log_a(b)$ is not a variable, and thus, we need to transform it into a fresh variable b' to represent its solution as $\mu b'.\tau'$. We also have to remember the mapping from b to b' while recursing on the type expression. This leads to the extended logarithm operation $\log_a^{\psi}(\tau)$ indexed by a mapping ψ from variables bound by the data type constructor (b) to their counterparts (b'). 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 data types are:

$$\log_a^{\psi}(\mu b.\tau) = \mu b'.\log_a^{\psi, b \mapsto b'}(\tau)[\mu b.\tau/b] \quad (b' \text{ fresh})$$
$$\log_a^{\psi}(b) = \psi(b) \quad (a \neq b \text{ and } b \in \psi)$$
$$\log_a^{\psi}(b) = \mathbb{0} \quad (a \neq b \text{ but } b \notin \psi)$$

All other rules are the same as the old logarithm except that it now carries the mapping ψ. The substitution $[\mu b.\tau/b]$, which happens *after* the logarithm calculation, takes care of any remaining b in $\log_a^{\psi, b \mapsto b'}(\tau)$. These seemingly complicated rules for $\log_a^{\psi}(\mu b.\tau)$ are largely forced by the invariant that logarithm should commute with the unrolling of recursive types. Let

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

The invariant refers to the equations

$$\log_a^{\psi}(\mu b.\tau) = \mu b'.\beta(b')$$
$$\log_a^{\psi}(\tau[\mu b.\tau/b]) = \beta(b')[\mu b'.\beta(b')/b']$$

which will play an important role in proving Theorem 4.1 (more precisely, the correctness of the operation $\mathtt{fill}_a$ defined in Section 4.2).

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

Fig. 3. Advanced rules of logarithm with union types and data types

Going back to the binary tree example, we have

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

or, with the union type optimization in Section 3.1,

$$
\log_a(\texttt{tree}(a)) = \mu t'.\mathbb{1} \cup (\mathbb{1} + t' + t')
$$

which would be isomorphic to the expected $\mu t'.\mathbb{1} + (t' + t')$ with very optimized union types. Another example is the logarithm of $\texttt{list}(a) = \mu l.\mathbb{1} + a \times l$. Following the new rules, we have

$$
\begin{aligned}
\log_a(\texttt{list}(a)) &= \log_a(\mu l.\mathbb{1} + a \times l) \\
&= \mu l'.\log_a^{l \mapsto l'}(\mathbb{1}) \cup (\log_a^{l \mapsto l'}(a) + \log_a^{l \mapsto l'}(l)) \\
&= \mu l'.\mathbb{0} \cup (\mathbb{1} + l').
\end{aligned}
$$

With reasonably optimized union types, $\log_a(\texttt{list}(a)) \simeq \mu l'.\mathbb{1} + l' = \mathbb{N}$. This matches the usual scheme to index elements in a list by natural numbers.

3.3 Further Remarks

The rules of logarithm with all the extensions in this section are summarized in Figure 3. The correctness criterion is the same as Theorem 3.1, but for the new logarithm operation.

In the previous section, we have learned that $\log_a(\texttt{list}(a)) \simeq \mathbb{N}$ under reasonable assumptions. This turns out to be more subtle than it seems to be; 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. When a is empty, $\texttt{list}(a)$ is non-empty, but $\mathbb{N} \times (\mathbb{N} \to a)$ is empty, which means $\texttt{list}(a)$ is not always embeddable into $\mathbb{N} \times (\mathbb{N} \to a)$. For the same reason, $\texttt{list}(a)$ and $\mathbb{N} \to a$ are in general incomparable, breaking the invertibility between logarithm and exponential.

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

Fig. 4. Rules of residual

4 LABEL GENERATION

Theorem 3.1 only asserted that logarithm gives us a general enough type, but we can reduce the number of testing cases further. The power of our logarithm lies in its ability to record the construction steps of each a-element; the most general testing case is when each a-element is the construction history itself. We can achieve this by placing suitable label generators at suitable positions to reconstruct the history.

When the function is of type $\forall a.(F(a) \to a) \times (G(a) \to K) \to H(a)$, we know that it is sufficient to consider the cases where the initial F-algebra $\mathtt{roll}_{a.F(a)}$ is the first argument [Bernardy et al. 2010]. In this paper, we want to prove a similar theorem for functions of more general types

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

(We wrote $\alpha(a)$ instead of τ as in previous sections to avoid clumsy notation in further developments.) Let

$$a^* = \mu a.\log_a(\alpha(a))$$

be the general type given by Theorem 3.1. We wish to create an element of type $\alpha(a^*)$ for testing. The insight is that *there are suitable label generators for all strictly positive occurrences of a in $\alpha(a)$*. Let $\alpha^-(a)$ be the $\alpha(a)$ with all strictly positive a replaced by $\mathbb{1}$, effectively calculating the residual. We can then define a *filler*:

$$\mathtt{fill}_a^{\alpha(a)} : \alpha^-(a^*) \to \alpha(a^*)$$

which incorporates label generators to assemble a full element of type $\alpha(a^*)$ for testing. See the following subsections for the precise definitions of these new operations. The new, enhanced theorem goes as follows:

Theorem 4.1 (correctness, upgraded). *For any two functions f, g of type*

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

where H is a functor, if a is positive in $\log_a(\alpha(a))$, then $f \cong g$ if and only if

$$f[\mu a.\log_a(\alpha(a))](\mathtt{fill}_a^{\alpha(a)}(h)) \cong g[\mu a.\log_a(\alpha(a))](\mathtt{fill}_a^{\alpha(a)}(h))$$

for every $h : \alpha^-(a)$. Again, the ambient typing context may have free type variables around, but not free expression variables.

$$\mathtt{fill}^{a}_{a;a\mapsto a^*}(e)(f) = f(\star)$$
$$\mathtt{fill}^{b}_{a;a\mapsto a^*}(e)(f) = e \quad (a \neq b)$$
$$\mathtt{fill}^{\mathbb{0}}_{a;a\mapsto a^*}(e)(f) = e$$
$$\mathtt{fill}^{\tau_1+\tau_2}_{a;a\mapsto a^*}(e)(f) = \mathtt{case}(e; x.\mathtt{inl}(\mathtt{fill}^{\tau_1}_{a;a\mapsto a^*}(x)(f\circ \mathtt{inl}^{\cup})); y.\mathtt{inr}(\mathtt{fill}^{\tau_2}_{a;a\mapsto a^*}(y)(f\circ \mathtt{inr}^{\cup})))$$
$$\mathtt{fill}^{\tau_1\cup\tau_2}_{a;a\mapsto a^*}(e)(f) = \mathtt{case}^{\cup}(e; x.\mathtt{inl}^{\cup}(\mathtt{fill}^{\tau_1}_{a;a\mapsto a^*}(x)(f\circ \mathtt{inl}^{\cup})); y.\mathtt{inr}^{\cup}(\mathtt{fill}^{\tau_2}_{a;a\mapsto a^*}(y)(f\circ \mathtt{inr}^{\cup})))$$
$$\mathtt{fill}^{\mathbb{1}}_{a;a\mapsto a^*}(e)(f) = \star$$
$$\mathtt{fill}^{\tau_1\times\tau_2}_{a;a\mapsto a^*}(e)(f) = \langle \mathtt{fill}^{\tau_1}_{a;a\mapsto a^*}(\mathtt{fst}(e))(f\circ \mathtt{inl}); \mathtt{fill}^{\tau_2}_{a;a\mapsto a^*}(\mathtt{snd}(e))(f\circ \mathtt{inr})\rangle$$
$$\mathtt{fill}^{\tau_1\to\tau_2}_{a;a\mapsto a^*}(e)(f) = \lambda x{:}\tau_1[a^*/a].\mathtt{fill}^{\tau_2}_{a;a\mapsto a^*}(e(x))(f\circ \lambda y{:}\log_a(\tau_2)[a^*/a].\langle x; y\rangle)$$

Fig. 5. Basic rules to generate the labels

$$\mathtt{fill}^{\psi;a}_{a;\sigma}(e)(f) = f(\star)$$
$$\mathtt{fill}^{\psi;b}_{a;\sigma}(e)(f) = e(f) \quad (a \neq b \text{ and } b \in \psi)$$
$$\mathtt{fill}^{\psi;b}_{a;\sigma}(e)(f) = e \quad (a \neq b \text{ but } b \notin \psi)$$
$$\mathtt{fill}^{\psi;\mathbb{0}}_{a;\sigma}(e)(f) = e$$
$$\mathtt{fill}^{\psi;\tau_1+\tau_2}_{a;\sigma}(e)(f) = \mathtt{case}(e; x.\mathtt{inl}(\mathtt{fill}^{\psi;\tau_1}_{a;\sigma}(x)(f\circ \mathtt{inl}^{\cup})); y.\mathtt{inr}(\mathtt{fill}^{\psi;\tau_2}_{a;\sigma}(y)(f\circ \mathtt{inr}^{\cup})))$$
$$\mathtt{fill}^{\psi;\tau_1\cup\tau_2}_{a;\sigma}(e)(f) = \mathtt{case}^{\cup}(e; x.\mathtt{inl}^{\cup}(\mathtt{fill}^{\psi;\tau_1}_{a;\sigma}(x)(f\circ \mathtt{inl}^{\cup})); y.\mathtt{inr}^{\cup}(\mathtt{fill}^{\psi;\tau_2}_{a;\sigma}(y)(f\circ \mathtt{inr}^{\cup})))$$
$$\mathtt{fill}^{\psi;\mathbb{1}}_{a;\sigma}(e)(f) = \star$$
$$\mathtt{fill}^{\psi;\tau_1\times\tau_2}_{a;\sigma}(e)(f) = \langle \mathtt{fill}^{\psi;\tau_1}_{a;\sigma}(\mathtt{fst}(e))(f\circ \mathtt{inl}); \mathtt{fill}^{\psi;\tau_2}_{a;\sigma}(\mathtt{snd}(e))(f\circ \mathtt{inr})\rangle$$
$$\mathtt{fill}^{\psi;\tau_1\to\tau_2}_{a;\sigma}(e)(f) = \lambda x{:}\sigma(\tau_1).\mathtt{fill}^{\psi;\tau_2}_{a;\sigma}(e(x))(f\circ \lambda y{:}\sigma(\log^{\psi}_a(\tau_2)).\langle x; y\rangle)$$
$$\mathtt{fill}^{\psi;\mu b.\tau}_{a;\sigma}(e)(f) =$$
$$\mathtt{fold}(x.\lambda y{:}\sigma(\log^{\psi}_a(\mu b.\tau)\to a).\mathtt{roll}(\mathtt{fill}^{\psi, b\mapsto b';\tau}_{a;\sigma, b\mapsto\sigma(\mu b.\tau), b'\mapsto\sigma(\log^{\psi}_a(\mu b.\tau))}(x)(y\circ \mathtt{roll})); e)(f)$$

Fig. 6. Advanced rules to generate the labels

4.1 Calculation of Residual

The calculation of the residual type follows the idea of *killing all strict positive occurrences* of the distinguished type variable a. See Figure 4. Every type constructor in this language preserves strict positivity, except function types. For function types, we follow the definition of strict positivity and recurse only on the codomains, leaving the domains alone. Notationally, for a type expression $\alpha(a)$, we define

$$\alpha^{-}(\tau) = (\alpha(a))^{-}_{a}[\tau/a].$$

Note that substitution $[\tau/a]$ happens *after* the calculation of residual.

4.2 Fill in the Generators

The definition of the filler $\mathtt{fill}_a^{\alpha(a)}$ is forced by its type $\alpha^-(a^*) \to \alpha(a^*)$. However, due to the subtlety of logarithm and (recursive) data types, we define an auxiliary filler $\mathtt{fill}_{a;\sigma}^{\psi;\tau}(e)(f)$ by induction on τ. Roughly speaking, the mapping ψ plays the same role as ψ in $\log_a^\psi(\tau)$, remembering the variables bound by the data type constructor; the substation σ accumulates the substitutions on types as we step into the bodies of data types; the expression e is the input before filling the label generators; and the transformer f maps "local" labels of type $\log_a^\psi(\tau)[a^*/a]$ to "global" labels of type a^*, making it easy to generate a valid a^*-element deep inside the expression.

Perhaps it is easier to first look at the basic auxiliary filler $\mathtt{fill}_{a;a\to a^*}^{\tau}(e)(f)$ which can handle everything except data types. See Figure 5 for its definition. It has the type

$$\mathtt{fill}_{a;a\mapsto a^*}^{\tau}(e : (\tau)_a^-[a^*/a])(f : \log_a(\tau)[a^*/a] \to a^*) : \tau[a^*/a].$$

As we dive into subexpressions of τ, we extend the transformer f further to maintain the invariant that it always maps a local label to a global one.

Recursive data types demand extra care as the label transformer f is constructed top-down but `fold` works bottom-up. This forces the recursive parts to be parametrized by a label transformer that will be fed only at the end of the recursion. Another complexity is that the ambient substitution grows whenever we step into the body of a data type. To express the input type, we replace the fixed substitution $[a^*/a]$ by a generic σ and define a special substitution $\overline{\sigma}$ to indicate that recursive parts of e should additionally take a transformer:

$$\overline{\sigma}(b) = \sigma(b) \quad (b \notin \psi)$$
$$\overline{\sigma}(b) = (\log_a^\psi(\sigma(b)) \to a^*) \to \sigma(b) \quad (b \in \psi)$$

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

$$\mathtt{fill}_{a;\sigma}^{\psi;\tau}(e : \overline{\sigma}((\tau)_a^-))(f : \sigma(\log_a^\psi(\tau) \to a)) : \sigma(\tau).$$

See Figure 6 for the complete definition of this auxiliary function.

The eventual filler used in Theorem 4.1 is then defined as

$$\mathtt{fill}_a^{\alpha(a)}(e : \alpha^-(a^*)) = \mathtt{fill}_{a;a\mapsto a^*}^{\emptyset;\alpha(a)}(e)(\mathtt{roll}_{a.\,\log_a(\alpha(a))})$$

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

$$\mathtt{roll}_{a.\,\log_a(\alpha(a))} : \log_a(\alpha(a))[a^*/a] \to a^*$$
$$(\alpha(a))_a^-[a^*/a] = \alpha'(a^*)$$
$$\alpha(a)[a^*/a] = \alpha(a^*).$$

5 PROOF SKETCH OF THEOREM 4.1

Let the general type given by the theorem as $a^* = \mu a.\log_a(\alpha(a))$. The central idea is to prove that the behavior of $f : \forall a.\alpha(a) \to H(a)$ is completely determined by $f[a^*](\mathtt{fill}_a^{\alpha(a)}(-))$. This is done by the following two steps:

(1) Prove that, for any type τ and any $e : \alpha(\tau)$, there exist $h^* : \alpha^-(a^*)$ and a partially functional relation $\mathcal{R}$ from a^* to τ such that $\mathtt{fill}_a^{\alpha(a)}(h^*)$ and e are related at type $\alpha(a)$ where the type variable a is associated with the relation $\mathcal{R}$.
The intuition is that a^* is general enough for $\mathcal{R}$ to assign distinct labels to all constructible elements of type τ (that is, uniformly constructible without knowing what τ is).

(2) By parametricity [Reynolds 1983], $f[a^*](\mathtt{fill}_a^{\alpha(a)}(h^*))$ and $f[\tau](e)$ are related at type $H(a)$. Because H is a functor and $\mathcal{R}$ is (partially) functional, the latter is determined by the former.

The same argument also applies to the other function g, which means $g[\tau](e)$ is also determined by $g[a^*](\mathtt{fill}_a^{\alpha(a)}(h^*))$. However, by assumption the two functions match in this special case, and thus they match in all cases.

The construction of the input h^* is essentially the reverse operation of $\mathtt{fill}_a$ which removes data from e at strictly positive occurrences of a. The relation $\mathcal{R}$ associates a label of type a^* to the represented element of τ (if existing).

6 EXAMPLES

6.1 Old Theorems

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

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

The $\alpha(a)$ in Theorem 4.1 is $(F(a) \to a) \times (G(a) \to K)$; 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,

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

Hence, the general type given by Theorem 4.1 is

$$a^* = \mu a.\log_a(\alpha(a)) \simeq \mu a.F(a).$$

The filler $\mathtt{fill}_a^{\alpha(a)}$ 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 filler, 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 isomorphisms. Thus far, we have fully recovered the theorem. We can also handle all function types mentioned in Theorem 5 in that paper by calculating their logarithms directly without going through embedding-projection pairs.

6.2 Revisiting Map

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

Let $\alpha(b) = (a \to b) \times \mathtt{list}(a)$ (after currying) and $H(b) = \mathtt{list}(a)$. Theorem 4.1 gives us a general type $\mu b.a \simeq a$ and $\mathtt{fill}_b^{\alpha(b)}$ is essentially putting the identity function as the first argument, up to type isomorphisms. The testing of map is then reduced to testing

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

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 $\mathtt{fill}_a^{\alpha(a)}$ 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.[2] This further reduces the testing to

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

for all n, as expected.

7 FURTHER IMPROVEMENTS

7.1 Inspectability

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

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

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 labels for all constructible elements, but this might be overkilling. Taking *inspectability* into consideration should improve the testing theory even further.

Another example is the function

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

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.

8 DISCUSSIONS

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. Related work includes the research on shapes [Jay 1995] and containers [Abbott et al. 2003]. Our logarithm operation resembles the translation from strictly positive types to containers in Abbott et al. [2003] because their translator, conceptually, is trying to find an exponent τ such that a^τ matches (part of) the original type.

There is still a long way to go for the maximally automatic testing. Various functions in practice have additional requirements on their inputs; for example, a sorting algorithm might demand the element comparator to form a total ordering, or a parallel list reducer might require the input binary reducer to be associative. However, such information has not been integrated into the current

[2] See Section 3.2 for the calculation of $\log_a(\mathtt{list}(a))$.

theory. This issue was noticed by Bernardy et al. [2010], and results like Voigtländer [2008] still defy systematic analysis. For languages with ML-style modules or type constructor abstraction, it is also crucial to advance the theory further to handle unknown type constructors or at least the prenex fragment of System F_ω.

The technology presented in this paper should apply to codata types with minimum changes. Another possible low-hanging fruit is nested (nonregular) data types [Bird and Meertens 1998]. For example, the nest type $\mathtt{nest}(a)$ is the least solution to the type isomorphism

$$N(a) \simeq 1 + a \times N(a \times a)$$

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

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

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

$$\log_a(c) = \log_b(c) \times \log_a(b)$$

when a, b, and c are positive real numbers. If we again allocate a fresh variable N' for the logarithm, we are essentially solving these isomorphisms:

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

Because $N'(a)$ does not depend on a, the least solution for $N'(a)$ is $\mu n'.1 + n' \times \mathbb{2}$. 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 their logarithms can be calculated mechanically as well.

Finally, it is possible to lift the prenex restriction but leave $\log_a(\forall b.\tau)$ generally undefined except that $\log_a(K) = \mathbb{0}$ 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

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

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

Peter Hancock. 2019. Napier's combinators and Böhm's λogarhythms. http://docs.hancock.fastmail.fm/arithmetic.pdf

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

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

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 (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 (POPL '08)*. ACM, New York, NY, USA, 29–35. https://doi.org/10.1145/1328438.1328445