パラメトリック多相関数の真偽値への単相化によるテスト

森畑 明昌

パラメトリック多相型をもつ関数をテストする際には、何らかの具体的な型で多相型を単相化する必要があるが、テストの有効性はどの型で単相化するかに依存しかねない。この問題に対し、いくつかの先行研究が多相型関数のテストのための適切な単相化手法を与えている。これらの手法は、多相型変数に対応する値がどのように得られたかを精緻に追跡するという着想に基づき、最終的には元の多相関数と等価な単相関数を構築する。そのため、テストにかかわらない部分についてまで実行トレースを全て覚えるような挙動になりやすい。本研究では、値が最終的にどのように観察され検証されるか、という観点から単相化の手法を与える。そして、多相型変数を真偽値に単相化したとしても意図せぬ挙動を必ず発見できるための十分条件を与える。さらに、この十分条件を満たさないプログラムに対し、条件をより満たしやすいプログラムへと変換するプログラム変換を与える。提案手法の有用性を確かめるため、Haskell 98 の Prelude ライブラリに含まれる関数に対して網羅的な適用を試みた。真偽値型への単相化では十分なテストが本質的に不可能な場合を除いて、提案手法はほとんど全ての多相関数を「真偽値型への単相化で十分テストできる」と正しく判定できた。

1 はじめに

テストはプログラムの誤りを発見する最も基本的で広く使われている手法である。にもかかわらず、著者の知る限り、多相関数のテストについての議論は多くない。例として、リストの要素を逆順に並び替える関数 *reverse* を考える。

$$reverse :: \forall \alpha.\ [\alpha] \to [\alpha]$$

これはリストの要素がどんな型であっても動作する多相関数である。ごく単純に考えると、この関数をテストするためには、リストの要素としてあらゆる型のものを試す、言い換えると型変数 α をあらゆる型で**単相化**する必要がありそうに見える。例えば、リストの要素が *Bool* 型の場合にうまく動作したとしても、*Int* 型の場合はうまく動作しないかもしれない。しかし、ありうる型は有限種類ではなく、またプログラム中で新たに定義されることもあるので、あらゆる型を試すのは原理的に不可能である。

もちろん、テストは明らかな間違いを見つけることはできても、正しさを保証するようなものではない。そのため、すべての型を試すというのはテストの趣旨からしてそもそも必要ないという主張もありうる。そうだとしても、一定の標準的な指針があることが望ましい。

パラメトリック多相型を用いる関数型言語では、「適当な型（例えば *Int*）だけを試せば十分である」というような主張がしばしばなされる。例えば、Sannellar らによる教科書 [19] には以下の記述がある。

> *You might think that polymorphic properties should be tested for values of many different types, in order to catch errors that might arise for one type but not another. But since polymorphic functions behave uniformly for all type instances, there's no need to test them on different types. It suffices to test polymorphic prop-*

[†]Testing Parametrically Polymorphic Functions via Finite Monomorphization.

[†]This is an unrefereed paper. Copyrights belong to the author.

Akimasa Morihata, 東京大学大学院総合文化研究科, Graduate School of Arts and Sciences, the University of Tokyo.

erties for Int or some other infinite type.

しかし、このような主張にははっきりしないところがある。まず、そもそもこの主張が正しいのかどうか、一見しては判じがたい。さらに、より少ない要素しか含まない型（例えば 2 つの要素をもつ *Bool* 型）では何故だめなのか、またそうだとすれば本当に *Int* 型で十分なのか、といったこともわからない。また、*Int* 型の要素なら何を使ってもよいのか、それともあらゆる *Int* 型の値をテストしなければならないのかもはっきりしない。

この問題に対し、いくつかの研究[3] [4] [8] [26] が、パラメトリック多相性を持つ関数を完全にテストできる単相化の方法を論じている。これらの手法では、多相型に対応する値の具体的な内容の代わりに、その値が**どのように得られたか**を用いる。例えば、前述の *reverse* 関数にこの手法を適用すると次のようになる。この関数の場合、α 型の値を得る方法は、入力リストの中身を取り出すしかない。つまり、「入力リストの何番目の要素から得られたか」に着目し、「リストの何番目の要素か」で単相化すればよい。よって、

$$reverse\ [0, 1, \ldots, n-1] = [n-1, n-2, \ldots, 0]$$

を確認すれば、長さ n のあらゆるリストに対して正しく動作することが確かめられる。

これらの手法は多相関数をテストする上での非常に良い指針を与える。しかし、これらの手法は、バグを発見するのに十分な情報をもてる値を使うことを目指しており、時として複雑で巨大な値を用いて単相化することになる。このことは、いくつかの問題をもたらしている。

- どのようなテストが行われるかを考慮していないために冗長な情報を保持している可能性がある。例えば、リストの長さを計算する関数 $length :: \forall\alpha.[\alpha] \to Int$ であれば、入力リストの要素はテストに関係しないはずだが、この手法では *reverse* と同様に自然数が順に並んだリストでのテストを求める。
- 実行トレースをほとんど完全に記憶するような結果になることも珍しくない。そのため、実行トレースが多少異なっても問題ないような例では、うまくテストできないことがある。抽象データ型を用いるプログラムや、型クラスを用いるプログラムでこのようなことが起こりやすい。
- 値が巨大ないし複雑になってしまった結果、テストが実質的に不可能、ないしは明らかに非効率となってしまう例がある（詳細は 6.1 節参照）。

本研究では既存研究とは異なる方針に基づき、多相関数を単相化しテストする手法を与える。本研究は特に、**出力結果の正しさがどのように確認されるか**に注目し、その確認に必要な情報をもつ入力を構成することを目指す。例として再び *reverse* を考える。*reverse* に長さ n のリストを入力し実行したとき、出力結果が正しくない場合は、以下の 2 通りに大別できる。

- 出力結果のリストの長さが n でない。
- 出力結果のリストの i 番目（$0 \leq i < n$）の要素が、入力リストの $n-i-1$ 番目の要素でない。

前者の場合、*reverse* 関数の動作はリストの要素によらないため、どんな要素でも同様の誤りが生じるはずである。つまり、() 型で試しても十分である。また、後者の場合、「その値が入力リストの $n-i-1$ 番目の要素と等しいか？」という点だけが問題となっている。つまり、$n-i-1$ 番目の要素が *True*、それ以外が *False* のリストを入力したとして、出力の i 番目の要素が *True* であることを確認すれば十分である。以上をふまえると、*reverse* 関数の誤りは *Bool* 型のリストを十分テストすれば必ず発見できるはずである。

以上の例からもわかるとおり、本研究では、**テスト結果の値が正しくないことを発見する観察**を入力値へと伝播することで、入力としてどのような値を与えれば十分にテストできるかを調べる。特に、多相型変数を有限値域の型に単相化してテストすることで、テスト対象の関数が所望の性質を満たすことを確認できる（本稿ではこれを**有限単相化テスト**と呼ぶ）ための十分条件を発見することを目指す。

本稿ではまず、上記の直感を定式化し、構造に関する性質が () 型への単相化で、要素に関する性質が *Bool* 型への単相化で、それぞれ有限単相化テストできるための十分条件を与える（3 節）。次に、典型的ないくつかの例を通して、提案手法の利用を実演する（4 節）。その中でも特に、提案手法が直接は適用でき

ない多相関数に対して、上記変換が適用できる形式へと変換する前処理を与える（4.2 節）。そして、提案手法が実用的なプログラムに対してどの程度有用であるかの評価の第一歩として、関数型言語 Haskell [17] の Prelude ライブラリの関数に対して網羅的に適用した結果を報告する（5 節）。6 節では提案手法を既存手法と比較し、長短や今後の課題について論じる。

2 準備

本稿では関数型言語 Haskell [17][†1] の記法を用いる。が、読みやすさのため、明らかに意図が理解できる範囲で、Haskell と多少異なる記号を用いる部分がある。例えば、ラムダ式には $\backslash x- > e$ ではなく $\lambda x.\ e$ を用いる。さらに、$\lambda x.\ \lambda y.\ e$ は $\lambda x, y.\ e$ と略記する。また、等値演算子は $\equiv$ を、非等値演算子は $\neq$ を、大小比較演算子は $\leq$ 等を、それぞれ用いる。真偽値演算には否定に $\neg$、論理積に $\wedge$、論理和に $\vee$ を用いる。さらに、関数合成演算子としては $\circ$ を用いる。

型についても、読みやすさのため、Haskell と多少異なる名称を用いる。要素を 1 つのみ含む型（Haskell の () 型）は $\mathbf{1}$ で表す。真偽値型（Haskell の *Bool* 型）は $\mathbf{2}$ で表す。整数の集合に対応する方は $\mathbb{Z}$ で、自然数の集合に対応する型は $\mathbb{N}$ で表す。なお、$\mathbb{N}$ はゼロと後者構成子からなる再帰データ型ではなく、$\mathbb{Z}$ の部分集合とする。

本稿では、α, β, γ は型変数を表すために用いる。また、型や型構成子を表すメタ変数としては A, B, T などの英大文字を用いる。プログラム中の変数名や関数名は f や x などの英小文字で表す。

本稿に現れる Haskell の関数のうち代表的なものの定義を図 1 に示す。

多相型をもつ式 $e :: \forall\alpha.\ T(\alpha)$ に対し、型 A による e の単相化を $e^{[\alpha:=A]} :: T(A)$ と表す。特に、どの型変数に対する単相化であるかが文脈から明らかな場合、$e^{[\alpha:=A]}$ を単に e^A と略記する。なお、この添字は文脈から明らかであれば省略することもある。また、$e :: \forall\alpha.\ T(\alpha)$ に対し、「α 型の値」という言葉を「適当な型 A による単相化 e^A における A 型の値」を表すために用いる。

本稿の議論では圏論における関手の概念を援用する。（共変）関手 F とは、本稿においては、任意の型 A を型 $\mathsf{F}A$ に変換する型構成子であり、さらに任意の関数 $f :: A \to B$ に対し関数 $\mathsf{F}f :: \mathsf{F}A \to \mathsf{F}B$ を伴うものである。特に、関手のもたらす関数は $\mathsf{F}id^A = id^{\mathsf{F}A}$ および $\mathsf{F}f \circ \mathsf{F}g = \mathsf{F}(f \circ g)$ を満たさなければならない。関手の典型例はリスト型構成子 $\mathsf{L}X = [X]$ であり、*map* 関数が関手のもたらす関数となる。また、積・和・定数・関数・関手の組合せで構成される型 $T(\alpha)$ について、α がその共変位置にのみ現れる場合、T は関手である。関手を表すメタ変数としては $\mathsf{F}, \mathsf{G}, \mathsf{H}$ 等を用いる。

以降では、定式化を単純にするため、「関手 F および G を用いて $\mathsf{F}\alpha \to \mathsf{G}\alpha$ と表せる型」は「関手 G を用いて $\mathsf{G}\alpha$ と表せる型」を含むものとする。これは $\mathsf{G}\alpha$ 型と $\mathbf{1} \to \mathsf{G}\alpha$ 型が同型であることに基づいている。

本稿ではパラメトリック多相型に対するパラメトリシティ原理 [18] [24] [23] に基づく “fast and loose but morally correct” [5] な議論を行う。具体的には、評価戦略によらず、有限ステップでエラーなく終了するような実行のみを考え、その範囲ではパラメトリシティ原理が成り立つことを仮定する。例えば、リストの先頭要素を取り出す関数 $head :: \forall\alpha.\ [\alpha] \to \alpha$ については、1 つ以上の要素を含むリストを入力した際の動作のみを議論の対象とする。そのため、Haskell の遅延評価や非正格性は議論に影響しない。

本稿の議論では以下の補題を用いる。この補題は多相型のパラメトリシティ [18] [24] [23] から容易に証明することができる。

補題 1. 多相関数 $g :: \forall\alpha.(\mathsf{F}_0\alpha \to \mathsf{G}_0\alpha) \to (\mathsf{F}_1\alpha \to \mathsf{G}_1\alpha) \to \cdots \to (\mathsf{F}_{n-1}\alpha \to \mathsf{G}_{n-1}\alpha) \to \mathsf{H}(\alpha)$（ただし各 $\mathsf{F}_i, \mathsf{G}_i$（$0 \leq i \leq n-1$）および H は関手）、項 $f_i :: \mathsf{F}_iA \to \mathsf{G}_iA$ と $f'_i :: \mathsf{F}_iB \to \mathsf{G}_iB$ $(0 \leq i \leq n-1)$、および関数 $h :: A \to B$ について、以下の条件が満たされるとする。

条件 各 $0 \leq i \leq n-1$ について、$\mathsf{G}_ih \circ f_i = f'_i \circ \mathsf{F}_ih$ が成り立つ。

このとき、次の等式が成り立つ。

†1 特に明示しない限り Haskell 98 を考える。

$$
\begin{array}{ll}
id :: \forall\alpha.\alpha \to \alpha & (!!) :: \forall\alpha.[\alpha] \to \alpha \\
id\ x \quad = \quad x & (a : x)\ !!\ n \quad = \quad \textbf{if}\ n \equiv\ 0\ \textbf{then}\ a\ \textbf{else}\ x\ !!\ (n-1) \\
\\
fst :: \forall\alpha,\beta.(\alpha,\beta) \to \alpha & snd :: \forall\alpha,\beta.(\alpha,\beta) \to \alpha \\
fst\ (x,y) \quad = \quad x & snd\ (x,y) \quad = \quad x \\
\\
min :: \forall\alpha.Ord\ \alpha \Rightarrow \alpha \to \alpha \to \alpha & max :: \forall\alpha.Ord\ \alpha \Rightarrow \alpha \to \alpha \to \alpha \\
min\ x\ y \quad = \quad \textbf{if}\ x \leq y\ \textbf{then}\ x\ \textbf{else}\ y & max\ x\ y \quad = \quad \textbf{if}\ x \leq y\ \textbf{then}\ y\ \textbf{else}\ x \\
\\
(+\!\!+) :: \forall\alpha.[\alpha] \to [\alpha] \to [\alpha] & concat :: \forall\alpha.[[\alpha]] \to [\alpha] \\
[\,] +\!\!+ y \quad = \quad y & concat\ [\,] \quad = \quad [\,] \\
(a : x) +\!\!+ y \quad = \quad a : (x +\!\!+ y) & concat\ (x : xs) \quad = \quad x +\!\!+ concat\ xs \\
\\
length :: \forall\alpha.[\alpha] \to \mathbb{N} & reverse :: \forall\alpha.[\alpha] \to [\alpha] \\
length\ [\,] \quad = \quad 0 & reverse\ [\,] \quad = \quad 0 \\
length\ (a : x) \quad = \quad 1 + length\ x & reverse\ (a : x) \quad = \quad reverse\ x +\!\!+ [a] \\
\\
map :: \forall\alpha,\beta.(\alpha \to \beta) \to [\alpha] \to [\beta] & foldr :: \forall\alpha,\beta.(\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\
map\ f\ [\,] \quad = \quad [\,] & foldr\ f\ e\ [\,] \quad = \quad e \\
map\ f\ (a : x) \quad = \quad f\ a : map\ f\ x & foldr\ f\ e\ (a : x) \quad = \quad f\ a\ (foldr\ f\ e\ x)
\end{array}
$$

図 1　本稿で用いる代表的な Haskell 関数

$$\mathsf{H}h\ (g^A\ f_0 \cdots f_{n-1}) = g_B\ f'_0 \cdots f'_{n-1} \qquad \square$$

3　有限単相化テスト可能条件

いま、多相関数 $g :: \forall\alpha.T_1(\alpha) \to T_2(\alpha)$ をテストすることを考える。このためには、適当な型 A で単相化し、g^A に対して適当な値 $v :: T_1(A)$ を渡し、結果 $g^A\ v :: T_2(A)$ が所望の性質を満たす—典型的には意図通りの値に等しい—ことを確認することになる。このようなテストのための一般的な理論の構築が本研究の目的である。しかし実際には、T_2 がどのような型かによってテストの状況はずいぶん変わってくる。

このことを確認するため、「出力結果が特定の値と等しいかどうか」を調べることを考えよう。仮に $T_2(X) = \mathbb{N} \to X$ であれば、出力結果が関数なので「特定の値と等しいかどうか」は調べられない †2。また、平衡木によって集合を表している場合、平衡木の形はどうでも良く、要素集合として同じ物が含まれていれば「等価である」とするようなテストをしたい可能性もある。この場合、「特定の値に等しいかどうか」を調べる処理は構造に対する再帰を伴う複雑な処理となってしまう。しかも、平衡木に対する「等価性」のテストはこれに限らない。例えば、要素集合だけでなく、平衡木の高さまで含めてテストしたい可能性もある。

このように、T_2 の型によって、またどのような性質をテストしたいかによって、テストの実態は様々に変わる。この状況を統一的に扱おうとすると、等価性の確認まで含めた処理をテストするという方針が自然に考えられる。例えば、上記 g であれば、これの参照実装 G を用意し、

$$\lambda v, eq.\ eq\ (\mathrm{G}\ v)\ (g\ v)$$
$$:: \forall\alpha.T_1(\alpha) \to (T_2(\alpha) \to T_2(\alpha) \to \mathbf{2}) \to \mathbf{2}$$

という式を考え、これに引数 v に加えて等価性確認関数 $(\equiv)$ を入力として渡すという方針である。この式であれば、結果が v によらず $True$ であることを確認すれば良いため、「性質の確認方法」に悩むことはない。

しかし残念ながら、このようにテスト対象の型が変わってしまうと、十分なテストのために必要な単相化も変わってしまう。例として以下の型の関数を考える。

$$thd :: \forall\alpha.(\alpha,\alpha,\alpha) \to \alpha$$

実は、この関数が常に組の 3 つ目の要素を返すことを確認するには、$\mathbf{2}$ 型で単相化し、

$$thd^{\mathbf{2}} :: (\mathbf{2},\mathbf{2},\mathbf{2}) \to \mathbf{2}$$

をテストすれば十分である。しかし上記方針に従い、

$$thd' :: \forall\alpha.(\alpha,\alpha,\alpha) \to (\alpha \to \alpha \to \mathbf{2}) \to \mathbf{2}$$

という型の式を考えてしまうと、$\mathbf{2}$ 型への単相化では

†2　このケースでは、出力結果の引数 $\mathbb{N}$ も g の引数だとみなし、この部分も含めてテストすれば良いと考えるかも知れない。しかし、このアプローチは $T_2 = [\mathbb{N} \to X]$ のような場合にスケールしない。

バグを発見できない可能性が残ってしまう。具体的には、以下の式が表すような間違った実装 $wrong\_thd'$ は、正しい実装を表していないにもかかわらず、**2** 型への単相化では常に $True$ を返してしまう。

$$\begin{aligned} & wrong\_thd'\ (a,b,c)\ eq \\ = \quad & eq\ c\ (\textbf{if}\ \neg(eq\ a\ b) \wedge \neg(eq\ b\ c) \wedge \neg(eq\ c\ a) \\ & \quad \textbf{then}\ a \\ & \quad \textbf{else}\ c) \end{aligned}$$

$wrong\_thd'$ は、a、b、c の値が全て異なる場合には a を返し、そうでなければ c を返すような実装を表している。入力が **2** 型の組の場合、a、b、c の値が全て異なることはあり得ないので、常に c を返す。しかし、一般には a を返しうるために正しくない。

以上の事実は、**値をどのように観察するかは可能な単相化を判断する上で本質的**であることを示唆する。thd は α 型の値の等価性を確認するすべを持たなかった。一方、$wrong\_thd'$ は α 型の値の等価性をプログラム中で確認することができるようになってしまったため、**2** 型への単相化では不十分となった。言い換えると、テスト対象のプログラムの実行結果に対して調べたい性質は、そのプログラムとは別個に決めなければならない。

そこで本研究では、$T(\alpha)$ 型要素の性質を**構造的性質**と**要素的性質**に分けて考える。構造的性質は α 型の要素に依存しないものであり、**要素的性質**は原則として T に依存しない性質である。$\forall\alpha.\ T(\alpha)$ 型という多相型を持つ式をテストする上で、両者の区別は本質的である。構造的性質は α をどの型に単相化するかと無関係であり、どの型に単相化してテストすれば良いかには主に要素的性質が関わるはずである。

定義 2. 関手 H に対し、多相関数 $sp :: \forall\alpha.\ \mathsf{H}\alpha \to \mathbf{2}$ を H に対する**構造的性質**と呼ぶ。

定義 3. 関手 H に対し、$acc :: \forall\alpha.\mathsf{H}\alpha \to \alpha$ と $p :: A \to \mathbf{2}$ を用いて $ep = p \circ acc$ と定義される関数 $ep :: \mathsf{H}A \to \mathbf{2}$ を H に対する (acc, p) による**要素的性質**と呼ぶ。

構造的性質の多相性は α 型の要素について関知しないことを表している。また、要素的性質は、acc によって要素のみを取り出した上で性質を調べている。

構造的性質・要素的性質に基づいて $reverse$ 関数のテストをすることを考える。いま、参照実装（つまり入力リストを正しく逆順に並べ替える関数）を REVERSE とすると、調べたい性質は、任意のリスト x に対して

$$reverse\ x = \text{REVERSE}\ x$$

が成り立つことである。これを確かめるためには、まず構造的性質としては、出力されるリストの「形」が等しいこと、すなわち

$$length\ (reverse\ x) = length\ (\text{REVERSE}\ x)$$

を確認すればよい。また、要素的性質としては、出力される「要素」がそれぞれ等しいことを示せば良い。つまり、$0 \le i < length\ x$ なる任意の自然数 i に対して以下が真となることを確認できれば良い。

$$\begin{aligned} & p\ (acc\ (reverse\ x)) \\ & \quad \textbf{where}\ acc\ x = x\ !!\ i \\ & \qquad p\ v = v \equiv (\text{REVERSE}\ x\ !!\ i) \end{aligned}$$

出力が単純なリストでない場合でも同様にしてテストを行うことができる。例えば、計算結果がネストしたリスト $r :: [[A]]$ であり、参照実装の実行結果が r^* であれば、構造的性質を $map\ length\ r = map\ length\ r^*$ で確認し、要素的性質は、総要素数を n として、各 $0 \le i < n$ について、$acc\ x = concat\ x\ !!\ i$ と $p\ v = v \equiv (concat\ r^*\ !!\ i)$ を用いて調べれば良い。

構造的性質・要素的性質は以下の性質を満たす。

補題 4. 関手 H に対する構造的性質 $sp :: \forall\alpha.\ \mathsf{H}\alpha \to \mathbf{2}$ について以下の等式が成り立つ。ここで A は任意の型、$erase :: \forall\alpha.\ \alpha \to \mathbf{1}$ は $erase\ a = ()$ で定義される関数である。

$$sp^A = sp^{\mathbf{1}} \circ \mathsf{H}erase$$

Proof. sp の多相型に対するパラメトリシティ原理 [18] [24] [23] から直裁である。 □

補題 5. 関手 H に対する (acc, p) による要素的性質 $ep :: \mathsf{H}A \to \mathbf{2}$ について、以下の等式が成り立つ

$$ep = acc^{\mathbf{2}} \circ \mathsf{H}p$$

Proof. acc の多相型に対するパラメトリシティ原理 [18] [24] [23] から直裁である。 □

補題 4・5 を補題 1 と併せることで、以下の単相化

テスト手法が得られる。これらは、一定の条件の下で、構造的性質の確認には $\mathbf{1}$ 型への単相化で、要素的性質の確認には $\mathbf{2}$ 型への単相化で十分であることを示している。

定理 6. 多相関数 $g :: \forall\alpha.(\mathsf{F}_0\alpha \to \mathsf{G}_0\alpha) \to (\mathsf{F}_1\alpha \to \mathsf{G}_1\alpha) \to \cdots \to (\mathsf{F}_{n-1}\alpha \to \mathsf{G}_{n-1}\alpha) \to \mathsf{H}(\alpha)$（ただし各 $\mathsf{F}_i, \mathsf{G}_i$（$0 \le i \le n-1$）および H は関手）について、以下の条件が満たされるとする。

条件 任意の $f_i :: \mathsf{F}_iA \to \mathsf{G}_iA$（$0 \le i \le n-1$）について、$\mathsf{G}_i erase \circ f_i = f_i' \circ \mathsf{F}_i erase$ を満たす関数 $f_i' :: \mathsf{F}_i\mathbf{1} \to \mathsf{G}_i\mathbf{1}$ が存在する。

このとき、H に対する構造的性質 sp が g^A のあらゆる出力結果で満たされるかどうかの確認には、$g^{\mathbf{1}}$ の出力結果に対して構造的性質 $sp^{\mathbf{1}}$ をテストすれば十分である。

Proof.

$$
\begin{aligned}
& sp^A\ (g^A\ f_0 \cdots f_{n-1}) \\
= \quad & \{\ \text{補題 4}\ \} \\
& sp^{\mathbf{1}}\ (\mathsf{H}erase\ (g^A\ f_0 \cdots f_{n-1})) \\
= \quad & \{\ \text{補題 1}\ \} \\
& sp^{\mathbf{1}}\ (g^{\mathbf{1}}\ f_0' \cdots f_{n-1}')
\end{aligned}
$$

□

定理 7. 多相関数 $g :: \forall\alpha.(\mathsf{F}_0\alpha \to \mathsf{G}_0\alpha) \to (\mathsf{F}_1\alpha \to \mathsf{G}_1\alpha) \to \cdots \to (\mathsf{F}_{n-1}\alpha \to \mathsf{G}_{n-1}\alpha) \to \mathsf{H}(\alpha)$（ただし各 $\mathsf{F}_i, \mathsf{G}_i$（$0 \le i \le n-1$）および H は関手）、および $p :: A \to \mathbf{2}$ について、以下の条件が満たされるとする。

条件 任意の $f_i :: \mathsf{F}_iA \to \mathsf{G}_iA$（$0 \le i \le n-1$）に対し、$\mathsf{G}_ip \circ f_i = f_i' \circ \mathsf{F}_ip$ を満たす関数 $f_i' :: \mathsf{F}_i\mathbf{2} \to \mathsf{G}_i\mathbf{2}$ が存在する。

このとき、H に対する (acc, p) による要素的性質 ep が g^A のあらゆる出力結果で満たされるかどうかの確認には、$g^{\mathbf{2}}$ の出力結果に対して (acc, id) による要素的性質をテストすれば十分である。

Proof.

$$
\begin{aligned}
& ep\ (g^A\ f_0 \cdots f_{n-1}) \\
= \quad & \{\ \text{補題 5}\ \} \\
& acc\ (\mathsf{H}p\ (g^A\ f_0 \cdots f_{n-1})) \\
= \quad & \{\ \text{補題 1}\ \} \\
& acc\ (g^{\mathbf{2}}\ f_0' \cdots f_{n-1}') \\
= \quad & \{\ ep' \text{ を } (acc, id) \text{ による要素的性質とする}\ \} \\
& ep'\ (g^{\mathbf{2}}\ f_0' \cdots f_{n-1}')
\end{aligned}
$$

□

定理 7・定理 6 の利用にあたっての補足

定理 6・定理 7 は、関数 f_i に対して適切な条件を満たす関数 f_i' の**存在**のみを要求している。この点は極めて重要である。

例として関数 $length :: \forall\alpha.[\alpha] \to \mathbb{N}$ に定理 6 を適用することを考える。仮に、$length$ に対してある値 $v :: [A]$ を入力すればバグが見つかるとしよう。定理 6 は、$map\ erase\ v = v'$ となる値 $v' :: [\mathbf{1}]$（この v' はどんな v に対しても必ず**存在**する）を入力しても同様にバグが見つかることを示している。このとき、我々は具体的な v' の値を知る必要はない。$[\mathbf{1}]$ 型の値のどれかが v' であることは分かるので、$[\mathbf{1}]$ 型の値をテストしてゆけばそのうち v' に遭遇するからだ。

よって、定理 6・定理 7 を使うにあたっては、適切な条件を満たす関数 f_i' の**存在が型のみからわかる**と非常に望ましい。以下にそのような場合を挙げる。

- 関手 $\mathsf{F}_i(X)$ が定数関手、すなわち X の出現を伴わない場合。この場合、任意の関数 h に対して、$\mathsf{F}_ih = id$ なので、$f_i' = \mathsf{G}_ih \circ f_i$ とすれば $\mathsf{G}_ih \circ f_i = f_i' \circ \mathsf{F}_ih$ を満たす。特に典型的な場合は、f_i が定数関数（値と同一視できる関数）である場合である。
- $f_i :: \forall\beta.\mathsf{F}_i\beta \to \mathsf{G}_i\beta$ の場合。この場合には、任意の関数 $h :: A \to B$ に対して、$\mathsf{G}_ih \circ f_i^A = f_i^B \circ \mathsf{F}_ih$ であることが f_i の多相型に関するパラメトリシティ原理から直裁に導かれる。

4 有限単相化テストの例

本節では、具体例を通して定理 6・定理 7 の有用性と利用方法を論じる。

4.1 単純なリスト処理関数

まず、1 節で議論した *reverse* 関数を再訪する。*reverse* 関数の型は $\forall\alpha.[\alpha] \to [\alpha]$ であり、リスト型構成子が関手であることをふまえると定理 6・定理 7 が即座に適用できる。よって、構造的性質は **1** 型へ単相化、要素的性質は **2** 型へ単相化した上でテストすれば十分である。

様々な関数のテストが同様に可能である。例えば、$concat :: \forall\alpha.[[\alpha]] \to [\alpha]$ や、リストの先頭から指定された個数の要素を取り出す関数 $take :: \forall\alpha.\mathbb{N} \to [\alpha] \to [\alpha]$ についても、構造的性質は **1** 型へ単相化、要素的性質は **2** 型へ単相化した上でテストすれば十分である。また、$length :: \forall\alpha.[\alpha] \to \mathbb{N}$ や、リストが空であるかどうかを調べる関数 $null :: \forall\alpha.[\alpha] \to \mathbf{2}$ であれば、結果が α 型の要素を含まないため、構造的性質しか必要ない。よって、**1** 型への単相化で十分である。

4.2 要素を消費する関数を伴う場合の前処理

テストしたい関数の引数の型が（共変）関手であれば、定理 6・定理 7 を即座に適用できる。しかし、この条件が成り立たないものも多い。例として、与えられた述語を満たす要素のみをリストから抽出する *filter* 関数を考える。

$$filter :: \forall\alpha.(\alpha \to \mathbf{2}) \to [\alpha] \to [\alpha]$$

この関数は引数として $\alpha \to \mathbf{2}$ 型の述語をとるが、$T(X) = X \to \mathbf{2}$ とすると T は関手ではない。このように、多相型変数に対応する要素を消費・観察するような関数を引数とする関数の場合、定理 6・定理 7 はそのままでは適用できない。

しかしよく見ると、第 1 引数の述語に対する入力として α 型の値を提供できるのは第 2 引数のリストしかない。このことをふまえると、*filter* p の代わりに以下の関数をテストすれば十分であると思われる。この関数は、まず最初にリストの全ての要素に p を適用しておき、*filter* 関数はその結果を参照するものである。

$$\lambda x.map\ fst\ (filter\ snd\ (map\ (\lambda a.(a, p\ a))\ x))$$

このことは、そもそも $map\ (\lambda a.(a, p\ a))\ x$ が入力リストであったとみなせば、*filter* 関数のテストは代わりに $filter' = map\ fst \circ filter\ snd$ をテストすれば十分であることを意味する。*filter*′ 関数の型は $filter' :: \forall\alpha.[(\alpha, \mathbf{2})] \to [\alpha]$ なので、定理 6・定理 7 を適用できる。

以上の議論は、次の定理として定式化できる。

定理 8. 多相関数 $g :: \forall\alpha.T_0(\alpha) \to T_1(\alpha) \to \cdots \to T_{n-1}(\alpha) \to \mathsf{H}(\alpha)$（ただし H は関手）について、以下の条件が満たされるとする。

条件 1 $T_0(\alpha) = \alpha \to \mathsf{F}_0(\alpha) \to \mathsf{G}_0$（$\mathsf{F}_0$ と G_0 は関手。特に G_0 は α を含まない）である。

条件 2 各 $1 \le i \le n-1$ について、$T_i(\alpha) = \mathsf{F}_i\alpha \to \mathsf{G}_i\alpha$（$\mathsf{F}_i$ と G_i は関手）である。

このとき、g^A に対するテストは、$B = (A, \mathsf{F}_0(A) \to \mathsf{G}_0)$ および $f_0'\ x\ y = snd\ x\ (\mathsf{F}_0 fst\ y)$ として、代わりに

$$\lambda f_1, \ldots, f_{n-1}.\mathsf{H}fst\ (g^B\ f_0'\ f_1 \cdots f_{n-1}) :: \forall\alpha.T_1(B) \to \cdots \to T_{n-1}(B) \to \mathsf{H}(B)$$

をテストすれば十分である。

Proof. 各 $f_i :: T_i(A)$（$0 \le i \le n-1$）が与えられたとき、$mkpair_{f_0}\ a = (a, f_0\ a)$ とし、これを用いて各 f_i'（$1 \le i \le n-1$）を以下で定義する。

$$f_i' = \mathsf{G}_i mkpair_{f_0} \circ f_i \circ \mathsf{F}_i fst$$

このとき以下の性質が成り立つ。

- $f_0\ x\ y = f_0'\ (mkpair_{f_0}\ x)\ (\mathsf{F}_0 mkpair_{f_0}\ y)$
- $\mathsf{G}_i mkpair_{f_0} \circ f_i = f_i' \circ \mathsf{F}_i mkpair_{f_0}$ $(0 \le i \le n-1)$

後者については $fst \circ mkpair_{f_0} = id$ および関手の性質から簡単に導くことができる。前者については以下のように証明できる。

$$\begin{aligned}
& f_0'\ (mkpair_{f_0}\ x)\ (\mathsf{F}_0 mkpair_{f_0}\ y) \\
= \quad & \{\ f_0' \text{ の定義 }\} \\
& snd\ (mkpair_{f_0}\ x)\ (\mathsf{F}_0 fst\ (\mathsf{F}_0 mkpair_{f_0}\ y)) \\
= \quad & \{\ \mathsf{F}_0 \text{ は関手 }\} \\
& snd\ (mkpair_{f_0}\ x)\ (\mathsf{F}_0(fst \circ mkpair_{f_0})\ y) \\
= \quad & \{\ fst \circ mkpair_{f_0} = id \text{ および } \mathsf{F}_0 id = id\ \} \\
& snd\ (mkpair_{f_0}\ x)\ y \\
= \quad & \{\ mkpair_{f_0} \text{ と } snd \text{ の定義 }\} \\
& f_0\ x\ y
\end{aligned}$$

以上をふまえ、任意の $f_i :: T_i(A)$（$0 \le i \le n-1$）

に対して、等式

$$g^A\ f_0 \cdots f_{n-1} = \mathsf{H}fst\ (g^B\ f'_0\ f'_1 \cdots f'_{n-1})$$

が成り立つことを示す。

$$\begin{aligned}
& \mathsf{H}fst\ (g^B\ f'_0\ f'_1 \cdots f'_{n-1}) \\
= \quad & \{\ \text{補題 1 および上記の性質}\ \} \\
& \mathsf{H}fst\ (\mathsf{H}mkpair_{f_0}\ (g^A\ f_0\ f_1 \cdots f_{n-1})) \\
= \quad & \{\ fst \circ mkpair_{f_0} = id\ \text{および関手 H の性質}\ \} \\
& g^A\ f_0\ f_1 \cdots f_{n-1} \qquad \square
\end{aligned}$$

定理 8 の効果をまず $filter :: \forall\alpha.(\alpha \to \mathbf{2}) \to [\alpha] \to [\alpha]$ で確認する。$T_0(\alpha) = \alpha \to \mathbf{2}$ なので、定理 8 から、*filter* の代わりに

$$\lambda x.\ map\ fst\ (filter\ snd\ x) :: \forall\alpha.[(\alpha, \mathbf{2})] \to [\alpha]$$

をテストすれば十分であることが導かれる。これは、我々がまさに観察してきたとおりの結果である。全く同様にして、リストの全ての要素が与えられた条件を満たすかどうかを確認する関数 $all :: \forall\alpha.(\alpha \to \mathbf{2}) \to [\alpha] \to \mathbf{2}$ やリストの要素を先頭から与えられた条件を満たす要素まで取り出す関数 $takeWhile :: \forall\alpha.(\alpha \to \mathbf{2}) \to [\alpha] \to [\alpha]$ なども扱うことができる。

次の例として、$map :: \forall\alpha,\beta.(\alpha \to \beta) \to [\alpha] \to [\beta]$ を考える。これに対しては、α を多相型変数、β を定数とみなし、定理 8 を適用すると、代わりに

$$map\ snd :: \forall\alpha,\beta.[(\alpha,\beta)] \to [\beta]$$

をテストすれば良いことが分かる。この結果をよく見ると、返値には α が現れないため、α について注目するならば構造的性質のみを確認すれば良いことが分かる。よって、定理 6 より、さらに

$$map^{[\alpha:=\mathbf{1}]}\ snd :: \forall\beta.[(\mathbf{1},\beta)] \to [\beta]$$

をテストすれば十分であるとわかる。$(\mathbf{1},\beta)$ が β と同型であることをふまえると、「*map* をテストするには *map id* をテストすれば十分である」というよく知られた事実と等価な結果 [3] [8] を系統的に導出できたことになる。なお、型変数 β については、構造的性質であれば $\mathbf{1}$ 型へ、要素的性質であれば $\mathbf{2}$ 型へ単相化してテストすれば十分である。

4.3 型クラス制約を伴う場合

パラメトリック多相性は、型変数のあらゆる具体化に対して、その項が同じように動作することを要求する。これは実用上は非常に強い制約となる。そこで、型変数を具体化できる型を一定範囲に制限するような枠組みが多く提案されている。その中でも型クラス [16] [25] は特に広く用いられている。

型クラスを伴う典型的な関数として 2 要素のうち大きい方を返す関数 *max* を考える。

$$max :: \forall\alpha.\ Ord\ \alpha \Rightarrow \alpha \to \alpha \to \alpha$$

この型において、*Ord* α は「α を具体化できるのは *Ord* のインスタンスである型に限る」という制約を表している。*Ord* のインスタンスでは、大小比較を行う関数 $(\leq) :: \alpha \to \alpha \to \mathbf{2}$ が定義されていなければならない †3。

型クラスを伴う多相関数を提案手法で扱う場合、2 通りの方針が考えられる。本小節では以降、そのうちの一つである、型クラスが要求する関数を追加の引数とする関数を考え †4、これに対するテストを行うという方針について論じる。もう一つの方針については次小節で扱う。

例えば、*max* 関数であれば、$(\leq) :: \alpha \to \alpha \to \mathbf{2}$ が追加の引数として与えられると考え、代わりに型クラス制約を伴わない関数

$$max_1 :: \forall\alpha.\ (\alpha \to \alpha \to \mathbf{2}) \to \alpha \to \alpha \to \alpha$$

をテストすれば十分である。定理 8 を適用すると、max_1 の代わりに

$$\begin{aligned}
max_2 &= \lambda x, y.\ fst\ (max'\ (\lambda v, w.\ snd\ v\ (fst\ w))\ x\ y) \\
&:: \forall\alpha.(\alpha, \alpha \to \mathbf{2}) \to (\alpha, \alpha \to \mathbf{2}) \to \alpha
\end{aligned}$$

をテストすれば良いことが分かる。さらに、カリー化によって max_2 と等価な関数

$$max_3 :: \forall\alpha.\alpha \to (\alpha \to \mathbf{2}) \to \alpha \to (\alpha \to \mathbf{2}) \to \alpha$$

が得られるので、更に定理 8 を 2 度適用することで、

$$\begin{aligned}
&\lambda x, y.\ fst\ (fst\ (max_3\ x\ snd\ y\ (snd \circ fst))) \\
&:: \forall\alpha.((\alpha, \mathbf{2}), \mathbf{2}) \to ((\alpha, \mathbf{2}), \mathbf{2}) \to \alpha
\end{aligned}$$

をテストすれば良いことが分かる。これに対しては定理 7 が適用でき、$\mathbf{2}$ 型への単相化でテストが可能で

†3 実際にはより多くの関数が定義されているが、本稿では一貫して必要最小限の定義を考える。

†4 なお、これは型クラスの標準的な実装方法 [16] でもある。

ある。

この方針は *Ord* クラス以外でも利用できる。例えば、第 1 引数の値から第 2 引数の値までを列挙したリストを得る関数 $enumFromTo :: \forall\alpha.Enum\ a \Rightarrow \alpha \to \alpha \to [\alpha]$ であれば、型クラス *Enum* が要求する関数が $fromEnum :: \alpha \to \mathbb{Z}$ および $toEnum :: \mathbb{Z} \to \alpha$ なので、これらを引数とした

$$enumFromTo_1 :: \forall\alpha.\ (\alpha \to \mathbb{Z}) \to (\mathbb{Z} \to \alpha) \to \alpha \to \alpha \to [\alpha]$$

を考えることになる。定理 8 により、以下を代わりにテストすれば良いことが分かる。

$$\lambda f, x, y.\ map\ fst\ (enumFromTo_1\ snd\ f\ x\ y) :: \forall\alpha.\ (\mathbb{Z} \to (\alpha, \mathbb{Z})) \to (\alpha, \mathbb{Z}) \to (\alpha, \mathbb{Z}) \to [\alpha]$$

この式にも定理 6・定理 7 が適用できる。

また、Haskell 2010 [13] の *length* 関数は、リストだけでなく様々な構造の大きさを計算できるように、型クラスを用いて定義されている。

$$length :: \forall\alpha, \tau.\ Foldable\ \tau \Rightarrow \tau\alpha \to \mathbb{N}$$

ここで、*Foldable* クラスは型コンストラクタを表す変数 τ への制約となっており、その構造に含まれる要素を 1 列に並べて処理できることを表している。より具体的には、$toList :: \tau\alpha \to [\alpha]$ の定義を要求している †5。よって、これを追加の引数とすると、

$$length_1 :: \forall\alpha, \tau.\ (\tau\alpha \to [\alpha]) \to \tau\alpha \to \mathbb{N}$$

となる。この定義であれば $\tau\alpha$ を新しい変数 β だとみなしても問題ないため、これをふまえて定理 8 を適用すると、

$$\lambda x.\ length_1\ snd :: \forall\alpha, \beta.\ (\beta, [\alpha]) \to \mathbb{N}$$

をテストすれば良いことが分かる。型変数 α, β はどちらも計算結果に表れないため、定理 6 から **1** 型への単相化でテストできることが分かる。

4.4 順序集合上の束を伴う場合

4.3 節では、型クラスが定義を要求する関数については、適切な型であればどんな関数でもよいとみなしていた。しかし、特に型クラスが複数の関数を要求する場合、それらの間には一定の性質が成り立つと期待することがほとんどである。この性質を活用すればより精緻なテストができるのではないか、と予想するのは自然である。

しかし、残念ながら、著者が調べた範囲では、提案手法でのテストに直接役立つような良い性質を伴う型クラスは見つかっていない。以下では、このようなアプローチの可能性を示唆する例として、束（lattice）を取り上げる。

集合 L の上の二項関係 $\preceq$ が任意の $a, b, c \in L$ に対して以下の 3 つの演算性質を満たすとき、$(L, \preceq)$ を半順序集合とよぶ。

- 反射律：$a \preceq a$
- 推移律：$a \preceq b$ かつ $b \preceq c$ ならば $a \preceq c$
- 反対称律：$a \preceq b$ かつ $b \preceq a$ ならば $a = b$

半順序集合 $(L, \preceq)$ が、任意の $a, b, c, d \in L$ に対して以下の 6 つの性質を満たす演算 $\vee^L$（結び、join）と $\wedge^L$（交わり、meet）を伴うとき、束と呼ぶ。

- $a \vee^L b = b \vee^L a$
- $a \preceq (a \vee^L b)$
- $a \preceq b$ かつ $c \preceq d$ ならば $(a \vee^L c) \preceq (b \vee^L d)$
- $a \wedge^L b = b \wedge^L a$
- $(a \wedge^L b) \preceq a$
- $a \preceq b$ かつ $c \preceq d$ ならば $(a \wedge^L c) \preceq (b \wedge^L d)$

束の典型的な例としては、全順序集合に対して *max* と *min* を演算とした場合や、べき集合に対して包含関係を順序とし和集合と積集合を演算とした場合が挙げられる。また、真偽値に対して論理和・論理積演算を用いる場合も束になる。特に真偽値に対する論理和・論理積を明示的に表すため、本小節ではこれらを $\vee^{\mathbf{2}}, \wedge^{\mathbf{2}}$ で表す。

いま、束をなす 2 つの演算 $(\vee) :: \alpha \to \alpha \to \alpha$ と $(\wedge) :: \alpha \to \alpha \to \alpha$ によって定義される型クラス *Lattice* を考える。そして、この型クラスに基づき定義される関数 $g :: \forall\alpha.\ Lattice\ \alpha \Rightarrow T(\alpha)$ の要素的性質をテストすることを考える。述語としては、「ある値以上」かどうかを調べるもの、すなわち $\lambda x.v \preceq x$ の形式のものを考える。このとき、以下の性質が束の性質から直裁に導かれる。

- $v \preceq (a \vee b) = (v \preceq a) \vee^{\mathbf{2}} (v \preceq b)$
- $v \preceq (a \wedge b) = (v \preceq a) \wedge^{\mathbf{2}} (v \preceq b)$

†5 標準的な定義では、*toList* ではなく *foldr* の定義を要求する。*toList* の定義を与えるのは *foldr* を与えるのと等価であり、かつテストに関する分析には明らかに *toList* を与える定義の方が適しているため、本稿では *toList* を引数としている。

これは、任意の束について、$\vee$ と $\wedge$ が定理 7 の前提条件を満たすことを表す。以上の観察から以下の系が導かれる。

系 9. 多相関数 $g :: \forall\alpha.Lattice\ \alpha \Rightarrow (\mathsf{F}_0\alpha \to \mathsf{G}_0\alpha) \to (\mathsf{F}_1\alpha \to \mathsf{G}_1\alpha) \to \cdots \to (\mathsf{F}_{n-1}\alpha \to \mathsf{G}_{n-1}\alpha) \to \mathsf{H}(\alpha)$（ただし各 $\mathsf{F}_i, \mathsf{G}_i$（$0 \le i \le n-1$）および H は関手）、および $v :: A$ にによって $p\ x = v \preceq x$ と定義される関数 $p :: A \to \mathbf{2}$ について、以下の条件が満たされるとする。

条件 任意の $f_i :: \mathsf{F}_iA \to \mathsf{G}_iA$（$0 \le i \le n-1$）に対し、$\mathsf{G}_ip \circ f_i = f_i' \circ \mathsf{F}_ip$ を満たす関数 $f_i' :: \mathsf{F}_i\mathbf{2} \to \mathsf{G}_i\mathbf{2}$ が存在する。

このとき、H に対する (acc, p) による要素的性質 ep が g^A のあらゆる出力結果で満たされるかどうかの確認には、$g^{\mathbf{2}}$ をテストすれば十分である。 □

系 9 は、テストしたい実装の実行結果の各要素がある値以下であると確認するには、$\mathbf{2}$ 型への単相化で十分であることを主張する。この際、束の演算子である $\vee$ と $\wedge$ もそれに従って単相化され、$\vee^{\mathbf{2}}$ と $\wedge^{\mathbf{2}}$ が代わりに使われることに注意せよ。なお、これだけでは、要素がある値と一致することは確認できていない。これについては、以下の 2 通りの方法がある。

- $\preceq$ が全順序（すなわち任意の $a, b \in L$ に対して $a \preceq b$ または $b \preceq a$）の場合、$a \preceq b$ かつ $a \neq b$ を $\prec$ で表すとすると、$\lambda x.v \prec x$ が真でありかつ $\lambda x.v \preceq x$ が偽であることを調べれば v との一致を確認できる。しかも、$\lambda x.v \prec x$ の形式の述語に対しても、$\vee$ と $\wedge$ は定理 7 の前提条件を満たす。このことは、結局のところ、$\mathbf{2}$ 型への単相化が十分であることを意味する。
- $\preceq$ が全順序でない場合、$\lambda x.v \preceq x$ と $\lambda x.x \preceq v$ が共に真であることを調べれば v との一致を確認できる。全ての順序を逆にし、結びと交わりを逆にした構造もまた束をなすことから、後者の形式の述語も $\mathbf{2}$ 型への単相化によってテストすることができる。ただしこのテストでは $\vee^{\mathbf{2}}$ と $\wedge^{\mathbf{2}}$ を入れ替えて使う必要があることに注意が必要である。

系 9 は、型クラスが要求する関数（結びと交わり）が順序関係に対して適切な条件を満たすことから導かれる。そのため、完全な束ではなく、結び半束（束から交わりを除いたもの）や交わり半束（束から結びを除いたもの）に対しても全く同様に成り立つ。

系 9 は古典的な Knuth の 0-1 原理 [9] を導く。Knuth の 0-1 原理は、値の交換による整列手順の正しさは真偽値の整列のみによって確認できることを主張している。値の交換が min と max で実現できることをふまえれば、このような整列は

$$sort :: \forall\alpha.Lattice\ \alpha \Rightarrow [\alpha] \to [\alpha]$$

という型を持つ関数で表すことができる †6。系 9 はこれが $\mathbf{2}$ 型への単相化でテストできることを即座に示す。

全く同様の観察は他の関数にも適用できる。例えば、リストから最大値を取り出す関数

$$maximum :: \forall\alpha.Ord\ \alpha \Rightarrow [\alpha] \to \alpha$$

を考える。この関数はしばしば条件分岐ではなく max を用いて実装される。この場合、α は Ord ではなく $Lattice$ クラスのインスタンスでよい。そして、以下の型の関数

$$maximum_1 :: \forall\alpha.Lattice\ \alpha \Rightarrow [\alpha] \to \alpha$$

であれば、系 9 から $\mathbf{2}$ 型への単相化でテストできることがわかる。

また、Morihata [15] は、交換則・結合則・冪等則を全て満たす二項演算子を引数とする接頭辞和であれば、真偽値リストに対するテストのみで正しさが保証できることを示した。この証明は以下の 2 ステップからなる。

1. 既存の単相化手法 [3] [4] [8] によって、この関数のテストが二項演算子として和集合演算のみを考えれば良いことを示す。
2. 和集合演算の代わりに論理和を使っても十分なテストができることを示す。

系 9 はこの証明に異なる道筋を与える。なぜなら、「あらゆる結び半束演算（これは和集合演算を含む）をテストすればよい」とわかれば即座に、$\mathbf{2}$ 型への単相化で十分であることが導かれるからである。

†6 Knuth の 0-1 原理が *min-max* ネットワークに拡張できることは folklore である。また、*min-max* ネットワークは値の交換によるネットワークより真に高い表現力を持つ。*min-max* ネットワークについての詳しい議論は Levi と Litman [10] を参照されたい。

Lattice **クラスについての補足**

Lattice クラスは背景に半順序があることを前提としていおり、従って同値関係も定義されている。よって、*Lattice* クラスは *Eq* クラスのサブクラスでなければならないように思えるかもしれない。しかし、本稿ではそのように定義していない。これは本質的である。Haskell において、ある型が *Eq* クラスのインスタンスであることは、その型の上に**同値関係が定義されることではなく、その型の値を $\equiv$ で比較するプログラムが書けること**を表す。本研究で導入した *Lattice* クラスは、値を半順序や同値関係で比較することは許さない。むしろ、$\vee$ と $\wedge$ のみで値を計算するからこそ、定理 9 が成り立っている。

4.5 有限単相化テストができない例

提案手法は、必ずしも全ての多相関数に適用できるわけではない。

まず、定理 6・定理 7 は引数の各型に対して条件を課しており、特に引数の型に多相型変数を含む関数が引数の場合には適用できない。さらに、定理 8 も多相型の要素を消費・観察する関数が返値の型に多相型変数を含まないことを要求している。以上をふまえると、引数と返値の両方に多相型変数を持つような関数を引数とする関数に対しては、原則として提案手法は適用できない †7。

典型的な例は、$\mathit{foldr} :: \forall\alpha,\beta.(\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$ や、所定の条件が満たされるまで指定された関数を繰り返し適用する $\mathit{until} :: \forall\alpha.(\alpha \to \mathbf{2}) \to (\alpha \to \alpha) \to \alpha \to \alpha$ などの、繰り返し構造を表すコンビネータが挙げられる。これは本質的な限界である。これらは、引数として与えられた関数を何度も適用し、次々と新しい値を作り出してゆくため、有限個の値を持つ型ではテストができないのは自然である。例えば、*foldr* は本質的にリストのチャーチ表現に対応するため、これをテストするにはリストと同程度の表現力をもつ型を用いる必要がある。

また、定理 8 では、多相型の要素を消費・観察する関数が有限種類であり、可能性を静的に列挙できることに依存している。そのため、例えば $\forall\alpha.[\alpha \to \mathbf{2}] \to [\alpha] \to \mathbf{2}$ ような型をもつ関数には適用できない。このような型は、実用上はそれほど現れないようにみえるかも知れない。しかし、しばしば定理 8 の適用によって得られてしまう。

例として、与えられた関数を等価関係としたときに定義される同値類の代表元のみを残す関数 $\mathit{nubBy} :: \forall\alpha.(\alpha \to \alpha \to \mathbf{2}) \to [\alpha] \to [\alpha]$ を考える。これは第 1 引数が引数に型変数 α を含むために定理 6・定理 7 を直接適用できない。そこで定理 8 を適用すると、代わりに $\mathit{nubBy}' :: \forall\alpha.[(\alpha, \alpha \to \mathbf{2})] \to [\alpha]$ という型の関数をテストすれば良いと分かる。しかし、これは依然として定理 6・定理 7 が適用できず、しかも定理 8 も適用できない。*nubBy* 関数について言えば、これは本質的な限界である。仮に α を有限種類の値（仮に n 種類とする）しかない型に単相化してテストすると、*nubBy* は長さ n 以下のリストしか返せない。これは、テストとしては不十分である。長さ $n+1$ 以上の時におかしな結果を返す、以下の *wrong_nubBy* を定義できてしまう（nubBy は *nubBy* の正しい実装だとする）。

$$\begin{aligned}&\mathit{wrong\_nubBy} :: \forall\alpha.(\alpha \to \alpha \to \mathbf{2}) \to [\alpha] \to [\alpha]\\&\mathit{wrong\_nubBy}\ \mathit{eq}\ x\\&\quad = \mathbf{let}\ r = \textsc{nubBy}\ \mathit{eq}\ x\\&\qquad \mathbf{in\ if}\ \mathit{length}\ r \geq n\ \mathbf{then}\ [\,]\ \mathbf{else}\ r\end{aligned}$$

同様の状況はしばしば現れる。例えば、ある要素がリストに含まれるかどうかを調べる関数 $\mathit{elem} :: \forall\alpha.\ \mathit{Eq}\ \alpha \Rightarrow \alpha \to [\alpha] \to \mathbf{2}$ も、型クラス *Eq* が同値関係をパラメタとしていることをふまえて 4.3 節の方針を適用すると、$\mathit{elem}' :: \forall\alpha.\ (\alpha \to \alpha \to \mathbf{2}) \to \alpha \to [\alpha] \to \mathbf{2}$ を考えることになる。しかし、この関数には *nubBy* と同じ理由で提案手法を適用できない。

しかし、その関数の意図を注意深く分析しインタフェースを修正すれば提案手法を適用できるようになるような関数も多い。例えば *elem* であれば、上記は任意の α 型の要素が比較できそうな型だったが、本来は入力リストの要素と与えられた要素を比較したいだけのはずである。つまり、上記の型は

†7 4.2 節で見たように、*map* 関数のように、引数と返値に異なる多相型変数を含む場合、片方を定数型とみなすことで扱える場合もある。

過剰に一般的である。代わりに、ある条件を満たす要素がリストに含まれるかどうかを調べる関数 $contain :: \forall\alpha.\ (\alpha \to \mathbf{2}) \to [\alpha] \to \mathbf{2}$ を用意し、これを用いて $elem\ a\ x = contain\ (\lambda b.\ a \equiv b)\ x$ と定義すれば十分だったはずである。そして、$contain$ であれば提案手法でテストできる [†8]。

$elem$ 関数の例は、効率的なテストのためには、プログラムの意図や挙動を適切に表したインタフェース（すなわち型）が必要だということを示唆する。これは自明な主張ではあるが、プログラミングの際にはしばしば無視される。特にライブラリ関数はできる限りシンプルで一般的なインタフェースが良いとされがちである。本研究で示した理論は、テストを効率化するためにはインタフェースをどのように制限すれば良いか、に対する一定の指針を示唆していると解することもできる。しかし、より系統的・自動的な手法を与えるにはさらなる調査が必要であろう。

5 評価

提案手法がどの程度の範囲の関数に対して適用可能であるかを確認するため、Haskell 98 の Prelude ライブラリの関数に対する評価実験を行った。

まず、全 196 個の関数（定数を含む）の中から以下に合致する物を除き、79 個の関数を抽出した。

- 型変数を含まず、本質的に単相である関数（25 個）、例えば $and :: [\mathbf{2}] \to \mathbf{2}$。
- 型変数が全て Num 型などの数値に関する型クラスのインスタンスである関数（66 個）、例えば $(+) :: Num\ a \Rightarrow a \to a \to a$。これらは数値の計算を伴うため、**2** 型や **1** 型への単相化は絶望的だと考えるのが自然である。
- その型をもつ定義が 1 通り（ないしは 0 通り）しかないもの（15 個）、例えば $fst :: \forall\alpha,\beta.(\alpha,\beta) \to \alpha$。これらは型のみから正しさが示せるため、テストは不要である。
- それが型クラスの最小限の定義に含まれる物（11 個）、例えば Eq クラスに対する $(\equiv)$。これらは型クラスの各インスタンスで個別に定義されるため、本質的は単相型だとみなすべきである。

これら 79 個の関数に対し、網羅的に提案手法の適用を試みた。なお、IO モナドを伴う関数については、IO が関手であり、構造的性質を確認でき、かつ要素抽出のための関数 acc が適切に定義できることを仮定した。IO モナドの引き起こしうるあらゆる副作用を考えるとこの仮定は成り立たない。しかし、テストをしたいという前提であれば、副作用としては画面やファイル等への出力のようなものを想定するのが自然であり、そうであればこの仮定は成り立つ。

なお、以上をすべて手作業で行った。提案手法の実装・自動化は今後の課題である。

結果を表 1・2 に示す。定理 6・定理 7 が適用可能だった物には Yes、そうでないものは空欄を記載している。また「前処理なし」の列は定理 8 を用いない場合、「前処理あり」は定理 8 を適用した場合を表す。なお、– は前処理の必要が無かったことを表す。前処理無しで適用可能だったのは 24 個であった。前処理を行うことでさらに 33 個の関数が適用可能となった。適用できなかったのは 22 個であった。

提案手法が適用できなかった 22 個は、以下の 2 類型に分類することができる。

一つは、4.5 節で議論したように、繰り返し構造を表す高階関数である。提案手法が適用できなかった関数の大部分（17 個）はこちらの類型に属しており、特に型クラスを伴わない関数はすべて当てはまった。また、型クラスを伴う関数のうち、モナドクラスに関する関数もこの類型だとみなすことができる。モナドクラスは $(>>=) :: \forall m,a,b.Monad\ m \Rightarrow m\,a \to (a \to m\,b) \to m\,b$ をパラメタとしており、これを用いて計算を繰り返すことができる型となっている。

もう一つは、Eq や Ord に関する一部の関数群である。4.5 節で議論したように、これらについては、リスト等を引数に含むと定理 8 の適用が行き詰まってしまう。この類型に属していたのは以下の 5 個の関数だが、その全てについて、下記のようにインタフェースを修正することができれば提案手法が適用できることも分かった。

- $maximum$ と $minimum$：これらは、4.4 節で議論したように、比較演算子の代わりに max また

†8 なお、全く同じ状況は、$lookup :: \forall\alpha,\beta.\ Eq\ \alpha \Rightarrow \alpha \to [(\alpha,\beta)] \to Maybe\ \beta$ でも現れる。

表 1　Prelude 関数への適用結果（型クラスを伴わないもの）

関数名	型	前処理なし	前処理あり
catch	$IO\ a \to (IO\ Error \to IO\ a) \to IO\ a$	Yes	–
concat	$[[a]] \to [a]$	Yes	–
maybe	$b \to (a \to b) \to Maybe\ a \to b$		Yes
zipWith	$(a \to b \to c) \to [a] \to [b] \to [c]$		Yes
(!!)	$[a] \to Int \to a$	Yes	–
(++)	$[a] \to [a] \to [a]$	Yes	–
all	$(a \to \mathbf{2}) \to [a] \to \mathbf{2}$		Yes
any	$(a \to \mathbf{2}) \to [a] \to \mathbf{2}$		Yes
asTypeOf	$a \to a \to a$	Yes	–
break	$(a \to \mathbf{2}) \to [a] \to ([a], [a])$		Yes
concatMap	$(a \to [b]) \to [a] \to [b]$		Yes
cycle	$[a] \to [a]$	Yes	–
drop	$Int \to [a] \to [a]$	Yes	–
dropWhile	$(a \to \mathbf{2}) \to [a] \to [a]$		Yes
filter	$(a \to \mathbf{2}) \to [a] \to [a]$		Yes
foldl	$(a \to b \to a) \to a \to [b] \to a$		
foldl1	$(a \to a \to a) \to [a] \to a$		
foldr	$(a \to b \to b) \to b \to [a] \to b$		
foldr1	$(a \to a \to a) \to [a] \to a$		
head	$[a] \to a$	Yes	–
init	$[a] \to [a]$	Yes	–
iterate	$(a \to a) \to a \to [a]$		
last	$[a] \to a$	Yes	–
length	$[a] \to Int$	Yes	–
map	$(a \to b) \to [a] \to [b]$		Yes
null	$[a] \to \mathbf{2}$	Yes	–
readParen	$\mathbf{2} \to ReadS\ a \to ReadS\ a$	Yes	–
repeat	$a \to [a]$	Yes	–
replicate	$Int \to a \to [a]$	Yes	–
reverse	$[a] \to [a]$	Yes	–
scanl	$(a \to b \to a) \to a \to [b] \to [a]$		
scanl1	$(a \to a \to a) \to [a] \to [a]$		
scanr	$(a \to b \to b) \to b \to [a] \to [b]$		
scanr1	$(a \to a \to a) \to [a] \to [a]$		
span	$(a \to \mathbf{2}) \to [a] \to ([a], [a])$		Yes
splitAt	$Int \to [a] \to ([a], [a])$	Yes	–
tail	$[a] \to [a]$	Yes	–
take	$Int \to [a] \to [a]$	Yes	–
takeWhile	$(a \to \mathbf{2}) \to [a] \to [a]$		Yes
until	$(a \to \mathbf{2}) \to (a \to a) \to a \to a$		
unzip	$[(a, b)] \to ([a], [b])$		Yes
unzip3	$[(a, b, c)] \to ([a], [b], [c])$		Yes
zip	$[a] \to [b] \to [(a, b)]$		Yes
zip3	$[a] \to [b] \to [c] \to [(a, b, c)]$		Yes
zipWith3	$(a \to b \to c \to d) \to [a] \to [b] \to [c] \to [d]$		Yes

は *min* 演算子を引数として与えればよい。

- *elem*、*lookup* と *notElem*：これらは、4.5 節で議論した通り、等価演算子の代わりに、見つけたい要素を特定する単項述語（型 $a \to \mathbf{2}$）を引数として与えればよい。

以上を見てきたとおり、Prelude ライブラリに現れる関数群に関しては、提案手法はかなりの部分を扱う能力を持っていた。また、適用できなかったものは、繰り返しを表しており有限値域の型への単相化では本質的にテストが不十分となる関数群がほとんどであった。この結果は提案手法の可能性を示していると言えよう。

表 2　Prelude 関数への適用結果（型クラスを伴うもの）

関数名	型	前処理なし	前処理あり
enumFrom	$Enum\ a \Rightarrow a \to [a]$		Yes
enumFromThen	$Enum\ a \Rightarrow a \to a \to [a]$		Yes
enumFromThenTo	$Enum\ a \Rightarrow a \to a \to a \to [a]$		Yes
enumFromTo	$Enum\ a \Rightarrow a \to a \to [a]$		Yes
pred	$Enum\ a \Rightarrow a \to a$		Yes
succ	$Enum\ a \Rightarrow a \to a$		Yes
$(\neq)$	$Eq\ a \Rightarrow a \to a \to \mathbf{2}$		Yes
elem	$Eq\ a \Rightarrow a \to [a] \to \mathbf{2}$		
lookup	$Eq\ a \Rightarrow a \to [(a,b)] \to Maybe\ b$		
notElem	$Eq\ a \Rightarrow a \to [a] \to \mathbf{2}$		
$(=<<)$	$Monad\ m \Rightarrow (a \to m\ b) \to m\ a \to m\ b$		
$(>>)$	$Monad\ m \Rightarrow m\ a \to m\ b \to m\ b$		
applyM	$Monad\ m \Rightarrow (a \to m\ b) \to m\ a \to m\ b$		
sequence	$Monad\ m \Rightarrow [m\ a] \to m\ [a]$		
*sequence_*	$Monad\ m \Rightarrow [m\ a] \to m\ \mathbf{1}$		
mapM	$Monad\ m \Rightarrow (a \to m\ b) \to [a] \to m\ [b]$		
*mapM_*	$Monad\ m \Rightarrow (a \to m\ b) \to [a] \to m\ \mathbf{1}$		
$(<)$	$Ord\ a \Rightarrow a \to a \to \mathbf{2}$		Yes
$(>)$	$Ord\ a \Rightarrow a \to a \to \mathbf{2}$		Yes
$(\geq)$	$Ord\ a \Rightarrow a \to a \to \mathbf{2}$		Yes
compare	$Ord\ a \Rightarrow a \to a \to Ordering$		Yes
max	$Ord\ a \Rightarrow a \to a \to a$		Yes
maximum	$Ord\ a \Rightarrow [a] \to a$		
min	$Ord\ a \Rightarrow a \to a \to a$		Yes
minimum	$Ord\ a \Rightarrow [a] \to a$		
read	$Read\ a \Rightarrow String \to a$	Yes	–
readIO	$Read\ a \Rightarrow String \to IO\ a$	Yes	–
readList	$Read\ a \Rightarrow ReadS\ [a]$	Yes	–
readLn	$Read\ a \Rightarrow IO\ a$	Yes	–
reads	$Read\ a \Rightarrow ReadS\ a$	Yes	–
print	$Show\ a \Rightarrow a \to IO\ \mathbf{1}$		Yes
showList	$Show\ a \Rightarrow [a] \to ShowS$		Yes
shows	$Show\ a \Rightarrow a \to ShowS$		Yes
showsPrec	$Show\ a \Rightarrow Int \to a \to ShowS$		Yes

6　議論

6.1　既存の単相化手法との比較

パラメトリック多相型関数の単相化によるテスト手法は、既存の複数の研究が提案している。著者の知る限り、この問題を始めて取り扱ったのは Bernardy ら [3] である。ほとんど同時期に、Christiansen と Seidel [4] は正格性の分析の文脈で非常によく似た議論を行っている。Hou と Wang [8] はこれらの議論に型理論の観点から精緻かつより一般的な定式化を与えている。1 節で論じたとおり、これらの研究はいずれも「多相型変数に対応する値がどう得られるか」に注目する。例えば、$reverse :: \forall\alpha.[\alpha] \to [\alpha]$ 関数を長さ n のリストに対してテストするならば、「入力リストの何番目の値か」を区別することで、つまり入力として $[0, 1, \ldots, n-1]$ をテストすることを求める。

以降、これら既存手法と提案手法を比較する。なお、既存手法にはそれぞれ細部に技術的な差があるため、以降では原則として Hou と Wang による手法と比較する。

テストによって多相関数の正しさを完全に確かめようとした場合、多くの場合において、提案手法の方が「より小さなサイズのデータを、より多くの回数」テストすることを要求する。例えば先の *reverse* 関数であれば、既存手法は整数のリスト（$n \log n$ ビット）を 1 回テストすれば十分だと主張するのに対し、提

案手法は真偽値のリスト（n ビット）を網羅的に（2^n 回 †9）テストすることを求める。よって、この例に関しては、既存手法の方が効率が良いと言える。なお、提案手法にも、インクリメンタルにテストができる、テストの並列化ができる、等のメリットがないわけではない。

しかし、「より少ない回数のテストで十分」という主張がどの程度実用上の意味を持つかは微妙ではある。*reverse* 関数であっても、正しさを完全に確かめようとした場合、任意の長さのリストについてテストしなければならず、有限のテストでは完遂できない。この状況は引数が増えるとたちまち悪化する。例えば、$take :: \forall\alpha.\mathbb{N} \to [\alpha] \to [\alpha]$ 関数であれば、リストに加えて自然数を与えてテストする必要があり、自然数も有限種類ではない。

型がより複雑になると、効率の比較もさらに微妙となる。例として $g_1 :: \forall\alpha.([\mathbf{2}] \to \alpha) \to \mathbb{N} \to \alpha$ という型の関数を考える。既存手法であれば、第 1 引数の関数からどのように α 型の値を得たかに注目するので、引数である関数の引数の型、すなわち [**2**] 型で型変数 α を単相化する。より具体的には、第 1 引数を id 関数とした $g_1^{[\mathbf{2}]}\ id :: \mathbb{N} \to [\mathbf{2}]$ のテストを要求する。この場合、出力結果はリストとなるので、その要素的性質のテストを O(1) 時間で行うのは絶望的に見える。一方で、提案手法であれば、この代わりに $g_1^{\mathbf{2}} :: ([\mathbf{2}] \to \mathbf{2}) \to \mathbb{N} \to \mathbf{2}$ のテストを要求する。第 1 引数を id 関数に固定できていないというデメリットはあるが、すくなくとも出力結果の確認は O(1) 時間で行うことができる。

さらに、$g_2 :: \forall\alpha.((\mathbb{N} \to \mathbb{N}) \to \alpha) \to [\alpha]$ という型の関数を考えると、既存手法では $\mathbb{N} \to \mathbb{N}$ 型の関数で型変数 α を単相化しようとする。しかし、これは望ましいことではない。この関数の実行結果を参照実装と比較しようとしても、「$\mathbb{N} \to \mathbb{N}$ 型の関数同士が等しい」ことを確かめることはできない。提案手法であれば、やはり **2** 型への単相化が十分であることが分かるので、このような問題は起こらない。

手法の適用範囲は、原則として既存手法の方が広い。例えば、既存手法であれば、提案手法では扱えない $foldr :: \forall\alpha,\beta.(\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$ 関数に対しても、$foldr\ (:)\ [\,]\ x = x$ が成り立つことを確認すれば十分である、ということを示すことができる。ただし、このテストは実質的には *foldr* の実行トレース（第 1 引数・第 2 引数をどのように使って計算結果を構築したか）を完全に覚えていることになるので、効率的なテストだと言えるかどうかは微妙である。逆に提案手法は、単相化によってある程度効率的なテストが可能になるための条件を示していると解することもできる。また、例えば整列に対する Knuth の 0-1 原理 [9] に関して言えば、既存手法 [3] よりも提案手法の方がより直接的な導出に成功している。

以上の状況をふまえると、既存手法と提案手法はどちらが優れているともいえないと考えられる。上手く組み合わせる方法があればより望ましいが、これは今後の課題である。

6.2 値の観察への注目について

前述の通り、既存手法は値が得られる手順に注目する。圏論の言葉を使うならば、既存手法は値が得られる手順を表す始代数を構築し、その始代数で単相化を行う。これに対し、本研究では、「値をどのように観察するか」に注目して理論を構築した。

Xia [26] は他の既存手法の圏論的双対として、終代数で値が壊される手順を表すことで単相化を行う手法を提案した。提案手法は一見すると Xia の手法と非常に関連が深そうだが、現時点では両者の関係性はよく分かっていない。既存手法と Xia らの手法もどちらも、最終的には多相関数と等価な単相関数を得る。よって両者の単相化結果は等価である。既に見てきたとおり、本研究では、最終的にどのような述語等を用いてテストを行うかを活用することで、単相化前の多相関数とは本質的に等価でない関数をテストする。

本研究では、構造的性質と要素的性質の区別によって、どのような単相化が可能かを論じている。この方針は、Abbott らの container type [1] [2] の影響を受けている。Container type はデータを格納するよう

†9 実際には、入力が全て異なるリストをテストすれば十分であることが既存手法から分かるので、これをふまえれば、提案手法でも 1 つだけ *True* をふくむ n 通りの真偽値リストを試せば十分である。

な構造に対し、その「形（shape）」と「アクセス方法（positions）」によって捉えることで、一般的かつ理論的にシンプルな定式化を与えている。

Container type に基づいていることは、実用上も多少のメリットがある。既存手法では、値の構成手法を完全に追うために、多相型関数がデータ型に関する計算を行っていた場合、そのデータ型が代数的データ型であること、そしてそのデータ型の完全な定義が手に入ることを要求している。しかし、この要求は、そのデータ型が（特にライブラリ等で定義された）抽象データである場合や、型クラスで抽象化されている場合などには満たされない。提案手法では、そのデータ型が関手であること、およびそのデータ型の要素がアクセスできることを求めるにとどめており、データ型の詳しい定義を要求せずに済んでいる。

6.3 Knuth の 0-1 原理とその亜種

本研究の動機の一つは、Knuth の 0-1 原理 [9] のようなテスト手法を与える一般的な方法を構築することである。

著者の知る限り、Knuth の 0-1 原理と多相型パラメトリシティとの関係を初めて示したのは Day ら [6] である。Day らは整列に特化した議論を行っていたが、本研究では 4.4 節においてこの議論を束に一般化した。また、Knuth の 0-1 原理の亜種としては、ソート済み列の併合 [14]、上位と下位の分離 [11]、二分割 [7] などが提案されている。本研究の成果はこれらをほとんど即座に導出できる。

Voigtländer [21] は、結合的二項演算子を用いた接頭辞和計算において、真偽値（2 種類の値）ではなく 3 種類の値でテストすれば十分であるという、「0-1-2 原理」を示した。Morihata [15] はこの結果をふまえ、二項演算子が結合的・交換的であるだけでは真偽値のテストは不十分であり、結合的・交換的・冪等的であれば十分となることを示した。本研究では、4.4 節において、Morihata の結果と Knuth の 0-1 原理との関係を示した。Voigtländer の 0-1-2 原理についても同様に理解できるかは、今後の課題である。

Maric ら [12] は分散合意形成アルゴリズムを記述するための言語を設計し、この言語で記述できるアルゴリズムについては 0-1 原理が成り立つことを示した。この 0-1 原理を本研究の枠組みで捉えることができるかについても今後の課題である。

6.4 今後の課題

本研究に残された直接的な課題は提案手法の自動化である。特に、定理 8 を型同型（例えばカリー化・反カリー化）をふまえつつ繰り返し適用してテスト可能な式を導出するのは、原理的にはそれほど難しくないが実際にはかなり煩雑である。これを自動化するシステムは、より大規模な実験等には不可欠だと考える。

本研究では、構造的性質と要素的性質についての議論を行った。しかし、プログラムの仕様の中には、これらどちらとも言えないようなものも少なからずある。例えば整列アルゴリズムについて考えると「値が小さい順に並ぶ」「入力に含まれる値と出力に含まれる値は集合としては一致する」といったことが最も自然な仕様だが、これらはどちらも構造的性質・要素的性質ではないように見える。これに対し、本研究では「結果の i 番目の要素が参照実装の i 番目の要素と一致する」というような形で要素的性質に翻訳し、間接的に正しさを確認している。このような翻訳は多くの場合可能だと思われるが、翻訳方法自体が非自明な場合もあるかも知れない。より系統的な方法を模索する余地は残っている。

評価実験 5 からも見て取れるように、実用性の観点からは型クラスの扱いは不可欠である。型パラメトリシティの観点から型クラスを扱う研究はいくつかある [22] [20] ので、これらも参照しつつ、より広い範囲の型クラスを扱う方法や、テストに適した型クラス・適さない型クラスを判別する手法などを与えることも今後の課題である。

謝辞 本研究は JSPS 科研費 23K11044 の助成を受けている。

参 考 文 献

[1] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In Andrew D. Gordon, editor, *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as*

Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings, volume 2620 of *Lecture Notes in Computer Science*, pages 23–38. Springer, 2003.

[2] Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theor. Comput. Sci.*, 342(1):3–27, 2005.

[3] Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. Testing polymorphic properties. In Andrew D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 125–144, Berlin, Germany, 2010. Springer.

[4] Jan Christiansen and Daniel Seidel. Minimally strict polymorphic functions. In Peter Schneider-Kamp and Michael Hanus, editors, *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, pages 53–64. ACM, 2011.

[5] Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and loose reasoning is morally correct. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 206–217. ACM, 2006.

[6] Nancy A. Day, John Launchbury, and Jeff Lewis. Logical abstractions in Haskell. In *Proceedings of the 1999 Haskell Workshop*, Utrecht, Netherlands, October 1999. Utrecht University Department of Computer Science, Technical Report UU-CS-1999-28.

[7] Guy Even, Tamir Levi, and Ami Litman. Optimal conclusive sets for comparator networks. *Theor. Comput. Sci.*, 410(14):1369–1376, 2009.

[8] Kuen-Bang Hou and Zhuyang Wang. Logarithm and program testing. *Proc. ACM Program. Lang.*, 6(POPL):1–26, 2022.

[9] Donald Knuth. *The Art of Computer Programming*, volume 3. Addison Wesley Longman, Boston, MA, USA, second edition, 1998.

[10] Tamir Levi and Ami Litman. The strongest model of computation obeying 0-1 principles. *Theory Comput. Syst.*, 48(2):374–388, 2011.

[11] Shuo-Yen Robert Li. Unified algebraic theory of sorting, routing, multicasting, and concentration networks. *IEEE Trans. Commun.*, 58(1):247–256, 2010.

[12] Ognjen Maric, Christoph Sprenger, and David A. Basin. Cutoff bounds for consensus algorithms. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 217–237. Springer, 2017.

[13] Simon Marlow. Haskell 2010 language report. Available from `https://www.haskell.org/onlinereport/haskell2010/`, 2009.

[14] Peter Bro Miltersen, Mike Paterson, and Jun Tarui. The asymptotic complexity of merging networks. *J. ACM*, 43(1):147–165, 1996.

[15] Akimasa Morihata. When does 0-1 principle hold for prefix sums? *New Gener. Comput.*, 41(3):523–531, 2023.

[16] John Peterson and Mark P. Jones. Implementing type classes. In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 227–236. ACM, 1993.

[17] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, UK, 2003.

[18] John C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing*, 83:513–523, 1983.

[19] Donald Sannella, Michael Fourman, Haoran Peng, and Philip Wadler. *Introduction to Computation - Haskell, Logic and Automata*. Undergraduate Topics in Computer Science. Springer, 2021.

[20] Daniel Seidel and Janis Voigtländer. Proving properties about functions on lists involving element tests. In Till Mossakowski and Hans-Jörg Kreowski, editors, *Recent Trends in Algebraic Development Techniques - 20th International Workshop, WADT 2010, Etelsen, Germany, July 1-4, 2010, Revised Selected Papers*, volume 7137 of *Lecture Notes in Computer Science*, pages 270–286. Springer, 2010.

[21] Janis Voigtländer. Much ado about two (pearl): a pearl on parallel prefix computation. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 29–35, New York, NY, USA, 2008. ACM.

[22] Janis Voigtländer. Free theorems involving type constructor classes: functional pearl. In Graham Hutton and Andrew P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 173–184. ACM, 2009.

[23] Janis Voigtländer. Free theorems simply, via dinaturality. In Petra Hofstedt, Salvador Abreu, Ulrich John, Herbert Kuchen, and Dietmar Seipel, editors, *Declarative Programming and Knowledge*

Management - Conference on Declarative Programming, DECLARE 2019, Unifying INAP, WLP, and WFLP, Cottbus, Germany, September 9-12, 2019, Revised Selected Papers, volume 12057 of *Lecture Notes in Computer Science*, pages 247–267. Springer, 2019.

[24] Philip Wadler. Theorems for free! In *FPCA'89 Conference on Functional Programming Languages and Computer Architecture. Imperial College, London, England, 11-13 September 1989*, pages 347–359. ACM, New York, 1989.

[25] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press, 1989.

[26] Li-yao Xia. A terminal view of testing polymorphic functions. Blog post, `https://blog.poisson.chat/posts/2017-06-29-terminal-monomorphization.html`, 2017.