形式手法を学び始めて思うことと，形式手法を広めるには

比　嘉　健　太[†1]　河　野　真　治[†1]

大学の講義や卒業研究を通して一年ほど形式手法を学んだ過程やつまづき，それらに基づいた形式手法を広めるための意見を述べます．

Yasutaka Higa [†1] and Shinji Kono[†1]

1. 現在行なっている研究

プログラムの信頼性の向上は自動で行なわれるべきだと思っています．形式化された範囲で作成されたプログラムは，プログラムのみで高い信頼性を持つのが理想だと考えています．

現在私はプログラムの変更を形式化する研究を行なっています．形式化された変更によりプログラムを変更する際，プログラムが完成に近づいているかどうか判断することが目的です．現在は特定の二点間での信頼性に注目し，2 つのバージョン間でのトレースの相違を指摘しようと考えています．

私の研究においてプログラムの変更は Monad を用いて表現します．Monad とは，圏においては Monad 則を満たす自然変換 μ と η と Functor T です[1)]．つまり，ある法則を満たす 3 つの要素として，プログラムの変更を表します．

例題の記述にはプログラミング言語 Haskell を使用しています．まず，プログラムの変更を表すようなデータ構造 Delta を定義します．Delta は全ての変更時点のプログラムの値を保持します．プログラムが変更されても変更前の値を持ち続けることにより，プログラムの変更を表す試みです．

Delta を用いたプログラムでは，変数は必ず Delta で記述し，関数は必ず Delta を返すようにします．プログラムへの変更は，関数か変数のバージョンアップとして表現し，Delta への新しい値の追加として記述します．その結果，任意の変数もしくは関数は必ず 1 つ以上のバージョンを持ちます．

次にバージョンを持った関数の適用を考えます．関数の適用は変数と関数のバージョンの違いを考慮したルールに基づいて適用しなくてはなりません．例えば，バージョン数が 1 つの変数に対してバージョン数が 3 つの関数と 2 つの関数を適用する場合，結果の変数のバージョン数を決めなくては適用することができません (図 1)．

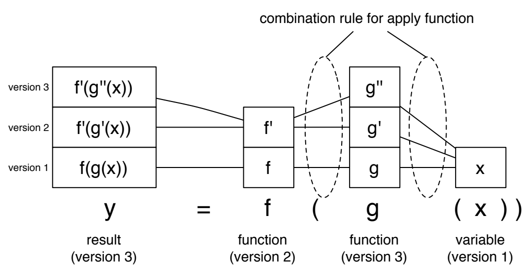

図 1　関数適用時のバージョンの組み合せ例

変数と関数のバージョンの組み合せのルールは Monad を用いて記述しました．Haskell における Monad はデータ構造とメタな計算との対応付けです[2)]．メタ計算としてバージョンの組み合せの解決を μ と η により定義します．関数を適用する際に必ず μ と η を用いて適用することにより，関数や値の変更時にバージョンの組み合せを考える必要が無くなります．メタ計算としてバージョンの組み合せを分離することで，任意の関数や変数に対し変更を行なっても変更後のプログラムが実行できます．

また，バージョンの組み合せの解決はプログラム全

†1 琉球大学工学部情報工学科
Department of Information Engineering, University of the Ryukyus.

体に対して一意でなくてはなりません．一意で無い場合，特定の箇所のみを変更することでプログラム全体のバージョンの組み合せが変わってしまいます．プログラム全体における組み合せが一意であることは Monad 則を満たすことで確認できます．定義した μ と η が Monad 則を満たすことは証明支援系言語 Agda[3] によって証明しました．

現在行なったのは Delta に対する Monad 則の証明までです．これからプログラムのトレースの取得と，2 つのプログラムのトレースの比較をしようと思っています．

2. Agda を学んだ流れとつまづき

形式化や Agda は三年次向けの講義で知りました．もともと形式手法を知らず，プログラムのバグを自動で検出したかった私には関心の高い講義でした．その講義では Curry-Howard 対応や Agda における証明を行ないました．

型や自然演繹は直感的に理解できました．型の変換は値の変換に応じて型が変わるため，関数によって値を変えることだと思えたからです．自然演繹もルールが数個しかなく，理解するのにそれほど時間はかかりませんでした．しかし，Agda による証明記述はすぐには理解できませんでした．依存型を持つ Agda では，証明は型で記述されます．証明を満たす型を持つ値，というものが直感的に理解できませんでした．

四年次になり研究室に配属された私は，プログラムのバグを自動で検出したいと希望しました．そこで，Proofs and Types[4] を読むことになりました．Proofs and Types ではもう一度 Curry-Howard 対応や自然演繹，それに加えて System T や System F などを学びました．Agda の特徴を取らえられたのは System T や Sytem F を Agda で記述していた時です．

Agda における証明の記述は Emacs の agda-mode で行なわれます．証明を記述し，その証明が成り立っているか何度も対話的に繰り返します．また，等式を変形していく際に前後の文脈からどのような規則により変換されるのか提示することもできます．Agda の理解が深まったと思えたのは，System T における自然数の加法の結合法則を記述した時です．結合法則を上手く記述できなかった私は，1 つの等式を思い付く限り変形していました．変形を繰り返していくうち，加法の交換法則によって変形が可能だと提示されました．その時，Agda による証明が理解できたように思えました．規則とそれから導出される証明も規則であり，それらによって等式を変形することで証明を記述しているのだと理解しました．

何度も等式の変形を試すことによって Agda を理解した私は，高速に何度も試行することが重要だと思っています．特に，Agda が対話的に実行するようなものであったため，対話的に実行することは多くのメリットがあると考えます．例えば Agda では，規則を仮定してどのような等式変形が可能かを何度も試すことができました．等式変形を何度も試すことにより必要な証明を明確にすることができます．明確化した式をさらに定義することで，証明すべき大きな式は見失わずに必要な小さな式に分割することができます．このように，問題の変換や詳細の把握，問題の分割などが行なえるため対話的実行は有用だと思っています．

3. 形式手法を広めるには

形式手法を広めるには，手軽に何度も試行可能にするのが重要だと思っています．手軽に何度も試行することができれば，形式手法の学習にも役立ちます．学習者は理解できない部分を含めて何度も実行することで，結果から理解に必要な情報を明確化していきます．必要になった部分のみ学習することで，学習コストを最小限に抑えることができると思われます．

何度も実行することで必要な情報を把握する方法のメリットは学習コストの面以外にもあると思います．例えば，検証の実行コストがあります．検証する範囲の指定とコストを対話的に何度も提示可能なら，払うコストに対して最大の効果が得られる範囲を探ることができます．企業のようにコストや利益の見積りを考えた上で製品を開発しなくてはならない場合，コスト対効果が明確であることは検討の際の指針になると思えます．さらに，検証範囲を対話的に指定できれば，必要以上の検証を防ぐことや検証の必要性を判断することも可能かもしれません．よって形式手法を広めるには，手軽に何度も試行可能であり，必要なコストや必要な情報を提示可能であることが重要だと思います．

参 考 文 献

1) Michael Barr and Charles Wells: Category Theory for Computing Science(1989)
2) Eugenio Moggi: Notion of Computation and Monads(1991)
3) The Agda Wiki - Agda: `http://wiki.portal.chalmers.se/agda`
4) Jean-Yves Girard, Paulr Taylor, Yves Lafont: Proofs and Types(1990)