Agda の Reflection API を用いた自動証明に向けて

石尾 千晶, 山本 充子, 浅井 健一

お茶の水女子大学

{ishio, jukoyama}@pllab.is.ocha.ac.jp, asai@is.ocha.ac.jp

概 要　定理証明支援系言語で証明を書くとき、場合分けが多くなることや、定義を少し変更しただけで残りの証明の大部分を変更しなければいけないことがある。これを解決するために、Coq などの言語では自動証明のための環境が整っており、Agda でもそのような自動証明ができるのが望ましい。現在、Agda では、Idris の Elaborator Reflection に影響を受けた新しい Reflection API が提供され始めている。この Reflection API はまだ開発途中の段階だが、証明の型チェック中に証明の型の情報を読み取ったり、書き換えたりすることができる。本稿では、単純型付き λ 計算に対する CPS 変換の正当性の証明を例にとって、Agda の Reflection API で自動証明をおこなった模様を報告する。具体的には、λ 計算の代入と簡約を自動でおこなう tactic を作成することで、単純な証明の部分を自動化することができた。また、Selective CPS 変換の正当性の証明のような大規模な証明の一部を自動化する。

1　はじめに

プログラムの品質を保証するには様々な方法がある。その中でも、定理証明系言語を使ったプログラムの正当性の証明は強力である。最近では、コンパイラや OS などの大規模なプログラムの持つ性質を保証するために定理証明系が使われ始めている。

このような証明にはかなりの労力がかかるが、その労力は「少量の複雑な作業」と「多量の単純な作業」の 2 つに分類できる。証明に必要なデータ型や、新しい補題の文言を定義する作業は複雑だが、これらの定義が証明全体に占める割合は小さい。一方、場合分けなどによって生じる繰り返しの作業は比較的単純で、これが証明のほとんどを占める上に、定義の文言を少し変更しただけで多量の作業をやり直さなければならないこともある。実際、これまでに著者らがおこなった Selective CPS 変換 [3] の Agda による正当性の証明 [12] では、全体の実装 約 7800 行のうち、約 7000 行が「多量の単純な作業」に分類できるものだった。

このような問題を解決するために、定理証明系言語では「多量の単純な作業」を自動化する仕組みの開発が進んでいる。例えば、Coq では自動証明の機能が豊富で、Ltac [6] などを使えばユーザが各自の証明に特化した tactic を書くことができる。一方、Agda には自動証明機能 [16] があるが、ビルトインの機能のためユーザの証明にカスタマイズするのが難しい。

本稿では、「ユーザが自分の証明に特化した tactic を書いて、比較的大規模な証明の負担を軽減する」ということを Agda でも実現するためのケーススタディをおこなう。Agda の証明での「単純な作業」を自動化するには、Agda の Reflection API [1] を使うことができる。Reflection API の最近のアップデートで、型チェックのための機能の多くが「型チェックモナド」として API の形でユーザに提供され始めた。これは現在も開発中の機能のため、Reflection API を使ってモナディックに Agda を書く方法はあまり広く知られていない。そこで、まずは Reflection API とその使い方を説明する (2 節)。また、現在の Reflection API でどの程度のことができるのかを見るために、単純型付き λ 計算に対する CPS 変換を例にとって、自動証明の方法を説明する (3 節)。さらに、Selective CPS 変換の定式化について 3 節の方法を適用した結果を述べた上で、現在の Reflection API の問題点を考える (4 節)。

2 Agda の Reflection API

Agda では、メタプログラミングの道具として Reflection API [1] が用意されている。本節では、新しくなった Reflection API と、その使い方を紹介する。

2.1 概要と経緯

Agda の Reflection API では、Agda のコードと Agda の内部表現の構文木を行き来することで、型の情報を読み取ったりコードを生成したりできる。コードと内部表現の行き来には quote と unquote が使われる。コードを quote すると内部表現に変換されるので、内部表現から型の情報を得ることができる。このような情報をもとに新たに内部表現を構築したら、内部表現を unquote することでコードに変換できる。

Agda のバージョン 2.5.1 までの Reflection API はこのような quote と unquote が中心だったが、バージョン 2.5.2 からは新たに型チェックのための機能群も API に追加され始めた。それにより、これまで Agda に組み込まれていた型チェック機能が「型チェックモナド (Type Checking Monad; 以下、TC モナド)」として API の形で呼び出せるようになった。TC モナドを使うと、自在に Agda の内部表現を書き換えて、それを型チェックに通したものをコードとして生成できる。TC モナドを使ったプログラムを書くには、基本的に Haskell でのモナディックプログラミングの記法にしたがって書けばよい。

2.2 内部表現

TC モナドを使って内部表現を直接操作するために、まずは内部表現を知る必要がある。

内部表現は、主に `Term`, `Name`, `Arg` の 3 つの要素からなる。内部表現は、`Term` 型を持つ構文木として表現される。構文木の中で識別子の役割を担うのが `Name` 型である。また、構文木中に現れる引数の情報を付け加えるのが `Arg` 型である。

`Term` 型の定義を以下に示す。

```
1  data Term : Set
2  Type = Term
3
4  data Term where
5    var : (x : Nat) (args : List (Arg Term)) → Term                          変数
6    con : (c : Name) (args : List (Arg Term)) → Term                         コンストラクタ
7    def : (f : Name) (args : List (Arg Term)) → Term                         関数定義･データ型定義
8    lam : (v : Visibility) (t : Abs Term) → Term                             ラムダ抽象
9    pat-lam : (cs : List Clause) (args : List (Arg Term)) → Term             パターンマッチングラムダ
10   pi : (a : Arg Type) (b : Abs Type) → Term                                関数型
11   agda-sort : (s : Sort) → Term                                            Set 型
12   lit : (l : Literal) → Term                                               リテラル (数字や文字列)
13   meta : (x : Meta) → List (Arg Term) → Term                               メタ変数
14   unknown : Term                                                           不明･型情報なし
```

2 行目には、`Type` 型は `Term` 型の型シノニムだと書かれている。これは、Agda のような依存型付き言語では、型で項レベルの計算をおこなうため、型も項も同じ構文木であらわすからである。

5 〜 14 行目がそれぞれ `Term` 型を持つコンストラクタとなっている。このうち一部を抜粋して説明する。変数 `var` は、de Bruijn index で管理されるため自然数を受け取る。変数管理とコンテキストについては 3.4 節で述べる。コンストラクタ `con` や関数定義 `def` では、`Name` 型の識別子と、`(Arg Term)` 型のリストとして表された引数を受け取る。例えば、Agda では自然数型 $\mathbb{N}$ はデータ型として定義され、それに対応する内部表現は `(def (quote ℕ) [])` と書ける。ここで、`(quote ℕ)` のように書くと、自然数型 $\mathbb{N}$ の識別子をあらわす `Name` 型を得ることができる。

また、`lam` はラムダ抽象をあらわし、`pi` は関数型をあらわしている。例として、自然数を受け取って真偽値を返す次のような関数を見てみよう。

```
example0 : ℕ → Bool
example0 = { }0
```

この 0 番 hole (プログラム中で `{ }0` とあらわされた部分) の型は関数型である。その型の内部表現は以下のようにあらわされる。

```
pi (arg (arg-info visible relevant) (def (quote ℕ) []))
(abs "_" (def (quote Bool) []))
```

上の内部表現では、0 番 hole が変数名が未定 (`"_"`) の ℕ 型の引数を受け取って、`Bool` 型を返す関数型 `pi` であることを意味している。この内部表現にあらわれた `arg` では、自然数型の引数が「explicit であること (`visible`)」と「型チェックに通った後も証明項を保持して計算に利用すること (`relevant`)」の 2 つの情報を引数に付加している。

また、Agda では未完成の証明も扱うため、`Term` 型のコンストラクタとして `meta` と `unknown` が用意されている。Agda のコードで、未完成の証明部分を `?` と書いて証明をロードすると、ユーザがインタラクティブに証明を書くことができる部分として穴 (hole) が生成される。この未完成の証明部分をあらわすのが `unknown` である。一方、メタ変数 `meta` は、Agda の内部で型チェック中の状態のものをあらわしている。型推論のための情報が足りず `var, def, con` などの具体的な型に決定するには至らないものが `meta` に分類されると思われる。

2.3 TC モナド

次に、内部表現の操作をおこなう TC モナドを紹介する。以下に TC モナドで提供されている関数の一部を示した[1]。

```
1  postulate
2    TC : ∀ {a} → Set a → Set a
3    returnTC : ∀ {a}{A : Set a} → A → TC A                    モナドを返す
4    bindTC : ∀ {a b}{A : Set a}{B : Set b} → TC A →           モナドを連結
5             (A → TC B) → TC B
6    unify : Term → Term → TC ⊤                                項を単一化し構文木を書き換える
7    typeError : ∀ {a}{A : Set a} → List ErrorPart → TC A      指定した形式の型エラーを出す
8    inferType : Term → TC Type                                項の型を推論
9    checkType : Term → Type → TC Term                         項が指定した型に一致するか確認
10   catchTC : ∀ {a}{A : Set a} → TC A → TC A → TC A           型エラーを捕捉
11   quoteTC : ∀ {a}{A : Set a} → A → TC Term                  コードから内部表現へ
12   unquoteTC : ∀ {a}{A : Set a} → Term → TC A                内部表現からコードへ
13   getContext : TC (List (Arg Type))                         現在使用可能な変数一覧を取得
14   extendContext : ∀ {a}{A : Set a} → Arg Type → TC A → TC A 一時的に変数一覧を拡張
15   getType : Name → TC Type                                  指定した識別子の型を取得
```

TC モナドは、「型チェックが成功しているかどうか」という状態を常に持ち歩くモナドで、型チェックに失敗すると即座に型エラーを出す (6 行目の `typeError`)。TC モナドを返すのが 3 行目の `returnTC` で、TC モナドを連結するのが 4 行目の `bindTC` である。この中でも、項の型チェック (単一化) をおこなうのが 5 行目の `unify` である。`unify` は、2 つの構文木を受け取ると、それらの most general unifier で構文木を書き換えるという副作用を起こすため、`TC ⊤` 型 (⊤ 型は unit 型に相当する) を返す。7 行目以降の関数は、これ以降で例とともに使い方を説明していく。

2.4 Reflection API を使ってみる

Reflection API を使って、コードの型情報を読み取ったり、簡単なコードを生成したりする方法を紹介する。以下の操作では、Emacs でインタラクティブに証明を書くための emacs-mode [2] を使用していることを前提とする。

[1] `Term`, `Name`, `Arg` 型や TC モナドの定義を含むモジュール `Agda.Builtin.Reflection` は、次の URL から閲覧可能である。`http://www.cse.chalmers.se/~nad/listings/equality/Agda.Builtin.Reflection.html`

[2] `https://agda.readthedocs.io/en/latest/tools/emacs-mode.html`

2.4.1 準備

Reflection API を使うには、必要なモジュールを先に読み込んでおく必要がある。API が定義されているモジュール Agda.Builtin.Reflection と、API を使ったライブラリのモジュール Reflection [2] を読み込む。TC モナドの bindTC 関数を _>>=_ に読み替え、さらに _>>_ 関数の定義を追加することで、do 記法が使えるようになる。この記法は、Haskell での do 記法と同じである。

```
open import Reflection renaming (bindTC to _>>=_)
open import Agda.Builtin.Reflection
_>>_ : ∀ {a b} {A : Set a} {B : Set b} → TC A → TC B → TC B
m >> m' = m >>= λ _ → m'
```

2.4.2 型情報を読み取る

TC モナドを使うと、「マクロ」というものを書くことができる。マクロは、未完成の証明部分の内部表現の構文木を受け取ると、それを書き換えて Agda のコードを生成するものである。マクロの最初の例として、hole の型情報を読み取る showGoal を考える。

```
1 macro
2   showGoal : Term → TC ⊤
3   showGoal hole = do
4     type ← inferType hole
5     t ← quoteTC type
6     typeError (termErr t :: [])
7
8 example1 : ℕ → Bool
9 example1 = { showGoal }1
```

まず、マクロの記法と実行方法について説明する。マクロを定義するには、1 行目のように macro と書き、定義したいマクロをそのコードブロックの中に入れる。マクロの関数の型は、必ず Term → TC ⊤ 型で終わらなければならない。マクロを実行するには、example1 の 1 番 hole に関数名を書き、hole の中にカーソルを合わせて C-c C-m というコマンド[3]を実行する。すると、hole の型をあらわす内部表現が Term 型としてマクロに渡される。マクロの実行後は、その結果が unquote されて内部表現から Agda のコードに変換された上で hole にコードが生成される。

次に、showGoal マクロの説明に移る。上に示したコードの 4 行目で、hole の型を inferType という API を使って推論し、その結果を type という変数に格納する。このとき、変数 type は 2.2 節で紹介したような、Term 型を持つ構文木になっている。一旦 5 行目をとばして 6 行目を見ると、型エラーを起こすことによって、エラーメッセージの一部として構文木を表示させようとしている。Agda のような依存型付き言語では、一般的なプログラミング言語のような I/O をおこなうのが難しいため、型エラーの機能を利用して Emacs の別バッファにエラーメッセージが表示されるようにするのが便利である。もし、5 行目の動作を書かなければ、この構文木はエラーメッセージの中に $(\mathbb{N}_1 : \mathbb{N}) \to$ Bool と表示される。これは、マクロの実行結果が unquote されてしまったことにより、表示したかった構文木が内部表現から Agda のコードに変換されたからである。そのため、5 行目では quoteTC 関数を使って構文木をさらに一段階 quote してから 6 行目で表示させている。

このマクロを 1 番 hole で実行した結果、次のようなエラーメッセージが表示される。メッセージの最初の 2 行が、1 番 hole の型の内部表現である。

```
pi (arg (arg-info visible relevant) (def (quote ℕ) []))
(abs "_" (def (quote Bool) []))
when checking that the expression unquote showGoal has type
ℕ → Bool
```

[3] これは、「Control キーと c を一緒に押し、一旦手を離してから、Control キーと m を一緒に押す」ことを意味する。

2.4.3 簡単なコード生成

Reflection API を使って、簡単なコード生成をおこなっていく。ここでは、自然数を生成する 2 つのマクロ unifyZero と unifySuc を考える。

```
1  macro
2    unifyZero : Term → TC ⊤
3    unifyZero hole = do
4      (def (quote ℕ) []) ← inferType hole
5        where t → typeError (strErr "not a number!" :: [])
6      unify hole (con (quote zero) [])
7
8  example2 : ℕ
9  example2 = { unifyZero }2
10
11 example3 : Bool
12 example3 = { unifyZero }3
```

上に示したコードの 4 行目で、hole の型推論をおこない、その結果が (def (quote ℕ) []) の形に一致するかどうかを確かめている。ここで一致した場合のみ、6 行目に進むことができる。一致しなければ、5 行目で typeError 関数を使って型エラーを起こし、マクロの実行を中断する。6 行目では、自然数型をもつとわかっている hole の型に対して、自然数 0 をあらわすコンストラクタ zero と単一化をおこなう。

マクロを実行してみよう。example2 の 2 番 hole に unifyZero を書き入れて C-c C-m でマクロを実行すると、2 番 hole が消えて、zero というコンストラクタが正しくあらわれる。一方、example3 でマクロを実行しようとすると、型推論に失敗するため、指定した通りの型エラーが出てくる。

```
not a number!
when checking that the expression unquote unifyZero has type Bool
```

続けて、1 以上の自然数をあらわすコンストラクタ suc のコード生成をおこなう。

```
1  vArg : {A : Set} → A → Arg A
2  vArg = arg (arg-info visible relevant)
3
4  macro
5    unifySuc : Term → TC ⊤
6    unifySuc hole = do
7      (def (quote ℕ) []) ← inferType hole
8        where t → typeError (strErr "not a number!" :: [])
9      m ← newMeta unknown
10     unify hole (con (quote suc) (vArg m :: []))
```

マクロ unifySuc は、基本的には unifyZero と似た動作をおこなうが、9, 10 行目が少し異なる。今回生成したい suc というコンストラクタは自然数型の引数を 1 つ受け取って自然数を返すので、手で証明を書くなら (suc ?) としたいところである。これに対応しているのが 9 行目で、newMeta という関数 [2] を使って、Type 型を与えるとその型を持つ新しいメタ変数を生成している。今回は簡単のために、型情報のない (unknown) 新しいメタ変数を生成した。10 行目では、コンストラクタの引数に (Arg Term) 型のメタ変数を追加している。その際、毎回 (arg (arg-info visible relevant) m) と書くのは面倒なので、vArg という関数を使っている。このマクロを先ほどの example2 で実行すると次のようになり、メタ変数の部分が新しい hole になっているのがわかる。

```
example2 : ℕ
example2 = suc { }4
```

3 CPS 変換の自動証明

Agda では、ユーザが hole に証明項を書き込んで型チェックする、という作業を繰り返すことで証明を完成させていく。本節では、Agda で証明を書く際に主に必要になる作業として、コンスト

ラクタの適用・コンテキストの拡張・関数の適用・再帰的な証明 の 4 つを取り上げ、これらの各作業を Reflection API で置き換え、繰り返すことで証明の自動化を目指す。

説明のために、単純型付き λ 計算に対する CPS 変換 [10] の正当性の証明を例にとり、この証明にカスタマイズしたマクロを書く方法を示す。著者らが以前おこなった CPS 変換の Agda による実装 [13] をもとにおこなっていく[4]。CPS 変換の実装は、大きく分けて、対象言語の定義・CPS 変換の定義・必要な補題の定義・正当性の証明の 4 つのパートからなる。本節では、このうち最後の正当性の証明を Reflection API を利用して自動生成する方法を説明する。その正当性の証明に必要な定義はユーザが先に準備しておくものとする。まずは、必要な定義について 3.1 節で説明する。その後 3.2 節から 3.6 節で正当性の証明を順を追って自動生成していく。

3.1 CPS 変換の実装の背景

CPS 変換の対象となる言語は単相の単純型付き λ 計算であり、その変数束縛には PHOAS (Parameterized Higher-Order Abstract Syntax) [5] を利用している。PHOAS を利用すると、λ 計算の変数管理をユーザが実装しなくても、実装に使った言語 (今回は Agda のこと) の変数管理機能を使えるため、証明を簡潔に書くことができる。以下に、対象言語を Agda で実装したものを示す。

```
1  data typ : Set where
2    Nat : typ
3    _⇒_ : typ → typ → typ
4
5  mutual
6    data value[_] (myvar : typ → Set) : typ → Set where
7      Var : {τ₁ : typ} → myvar τ₁ → value[ myvar ] τ₁
8      Num : (n : ℕ) → value[ myvar ] Nat
9      Fun : {τ₁ τ₂ : typ} (e₁ : myvar τ₂ → term[ myvar ] τ₁) → value[ myvar ] (τ₂ ⇒ τ₁)
10
11   data term[_] (myvar : typ → Set) : typ → Set where
12     Val : {τ₁ : typ} → value[ myvar ] τ₁ → term[ myvar ] τ₁
13     App : {τ₁ τ₂ : typ} (e₁ : term[ myvar ] (τ₂ ⇒ τ₁)) (e₂ : term[ myvar ] τ₂) →
14           term[ myvar ] τ₁
```

対象言語の型は typ で定義されていて、自然数型か矢印型のいずれかである。また、値を value[_] というデータ型で、項を term[_] というデータ型で定義している。値と項はいずれも typ → Set 型の変数を受け取るようになっており、ここが PHOAS でパラメータ化されている部分である。

3 節のこれ以降の証明では代入規則や簡約規則などの名前が多く登場するため、簡単に紹介する。

代入規則には、値のための規則 SubstVal と、項のための規則 Subst がある。例えば、項の代入規則 Subst の型は以下のようになっており、関数ではなく関係として定義されている。

```
data Subst {myvar : typ → Set} : {τ τ₁ : typ} → (myvar τ → term[ myvar ] τ₁) →
          value[ myvar ] τ → term[ myvar ] τ₁ → Set where
```

Subst の型は、「代入する場所が τ 型の変数であるような τ_1 型の項」と、τ 型の値を受け取ったら、代入結果として τ_1 型の項を返す、と読むことができる。この関係は sVal や sApp という 2 つの規則 (コンストラクタ) によって再帰的に定義されている。値の代入規則 SubstVal も同様である。簡約規則には、1 ステップ簡約の規則 Reduce と、複数ステップ簡約の規則 Reduce* がある。Reduce には、β 変換をおこなう RBeta と、フレームを使って再帰的に簡約する RFrame がある。Reduce* には、0 ステップの簡約規則 R*Id と、1 ステップ以上の簡約をおこなう規則 R*Trans がある。

CPS 変換の定義では、Danvy/Filinski の CPS 変換 [10] の定義がそのまま Agda の関数として実装されている。対象言語で型・値・項を定義したので、それぞれについて CPS 変換 (cpsT, cpsEtaV, cpsEta) を用意する。さらに、余分な η 簡約基を残さないための補助的な CPS 変換 cpsEta′ も用意する。必要な補題は主に CPS 変換と代入規則の可換性を示すものになっている。これらはそれぞれ eSubst′, kSubst′, kSubst と呼ばれている。

[4] 手動でおこなった証明は http://pllab.is.ocha.ac.jp/~asai/jpapers/ppl/18/mono.agda から閲覧可能。

```
correctEta : {myvar : typ → Set} {τ : typ} {e e′ : term[ myvar ∘ cpsT ] τ} →
             (κ : value[ myvar ] cpsT τ → term[ myvar ] Nat) →
             Reduce e e′ → schematic κ →
             Reduce* (cpsEta e κ) (cpsEta e′ κ)
correctEta {myvar} {τ} {e′ = e′} (RBeta {e = e} {v₂} sub) k sche = { }0
correctEta {myvar} {τ} (RFrame {e = e} {e′} (App₁ e₂) red) k sche = { }1
correctEta {myvar} {τ} (RFrame {e = e} {e′} (App₂ v₁) red) k sche = { }2
```

図 1. 示すべき定理 (CPS 変換の正当性)

3.2 Reflection API を使って証明を進める

この節では、Reflection API を使って実際に CPS 変換の正当性を証明する方法を示す。

示すべき定理は図 1 のようになっている。このコードは、「簡約関係 Reduce が成り立つような τ 型を持つ項 e, e' と、schematic な継続 κ があるとき、e を CPS 変換したものを複数回簡約すると e' を CPS 変換したものになる」と読むことができる。

これ以降の説明では、0 番 hole を Reflection API を用いて解く方法を考える。(0 番 hole の証明を Reflection API を使わずに手動で解く方法の詳細は、付録 A を参照されたい。)

3.2.1 手動の作業を Reflection API に置き換える

証明したい関数 correctEta の 0 番 hole の型は、以下のようになっている。

```
1 Goal: Reduce*
2       (App
3        (App
4         (Val (Fun (λ x₂ → Val (Fun (λ k₁ → cpsEta′ (e x₂) (Var k₁))))))
5         (Val (cpsEtaV v₂)))
6        (Val (Fun (λ v → k (Var v)))))
7       (cpsEta e′ k)
```

示したい場所の型が Reduce* で始まり、複数回の簡約が必要なことから、答えとして R*Trans が該当することがわかる。また、R*Trans は引数を 5 つ受け取るため、手動で答えを書く場合は (R*Trans ? ? ? ? ?) となる。このことをマクロで表現すると次のようになる。

```
1  macro
2    runTC : Term → TC ⊤
3    runTC hole = do
4      m1 ← newMeta unknown
5      m2 ← newMeta unknown
6      m3 ← newMeta unknown
7      m4 ← newMeta unknown
8      m5 ← newMeta unknown
9      unify hole (con (quote R*Trans)
10       (vArg m1 :: vArg m2 :: vArg m3 :: vArg m4 :: vArg m5 :: []))
```

この runTC というマクロを correctEta 関数の前に定義し、0 番 hole の中に runTC と書いて C-c C-m コマンドを実行すると、以下のようになる。R*Trans の 5 つの引数のうち、1 つ目と 3 つ目は R*Trans で unify した段階で型が一意に定まるため、自動的に答えが埋まっている。

```
correctEta {myvar} {τ} {e′ = e′} (RBeta {e = e} {v₂} sub) k sche =
  R*Trans
    (App
      (App (Val (Fun (λ x → Val (Fun (λ k₁ → cpsEta′ (e x) (Var k₁))))))
       (Val (cpsEtaV v₂)))
      (Val (Fun (λ v → k (Var v)))))
    { }4 (cpsEta e′ k) { }6 { }7
```

ここで、6 番の型は 4 番の型に依存しているため、6 番が解ければ連動して 4 番の型が制約解消されるようになっている。次の節では、6 番を解く方法を考える。

3.2.2 他の hole に依存している hole での問題

3.2.1 節と同様の方法を用いて証明を続けたいところだが、6 番が 4 番に依存した型を持つため、6 番でマクロを実行できなくなってしまう。まずは、その様子を観察する。

6 番 hole に入れるべき答えの型は以下のようになっている。?4 は 4 番 hole を意味する。

```
Goal: Reduce
      (App
       (App (Val (Fun (λ x → Val (Fun (λ k₁ → cpsEta′ (e x) (Var k₁))))))
        (Val (cpsEtaV v₂)))
       (Val (Fun (λ v → k (Var v)))))
      ?4
```

Reduce の 1 つ目の引数の冒頭で 2 つの App が現れることから、すぐには β 簡約することはできず、RFrame と App_1 を使って関数適用の関数部分を先に評価することがわかる。手動で答えを埋めるには (RFrame (App_1 ?) ?) と書けばよく、同じ意味を持つマクロを作成すると次のようになる。

```
macro
  runTC : Term → TC ⊤
  runTC hole = do
    m1 ← newMeta unknown
    m2 ← newMeta unknown
    unify hole
      (con (quote RFrame)
        (vArg (con (quote App₁) (vArg m1 :: [])) :: vArg m2 :: []))
```

6 番でこのマクロを実行しようとすると、以下のようなエラーが出てしまう。

```
((sub₁ : Subst _e_397 _v₂_398 _e′_399)
 (k₁ : value[ _myvar_395 ] (cpsT _τ_394) → term[ _myvar_395 ] Nat)
 (sche₁ : schematic k₁) →
 term[ _myvar_395 ] Nat)
!=< (term[ (λ v → myvar v) ] Nat) of type Set
when checking that the expression ?4 has type
term[ (λ v → myvar v) ] Nat
```

このエラーメッセージは通常のものとかなり異なる形をしており、著者らは、Agda 内部のエラーがそのまま表示されてしまっているのではないかと考えている[5]。いずれにしても、このエラーメッセージは「6 番では runTC を実行しようとしたが、4 番の型でエラーが起きたためマクロの実行には失敗した」ことを意味している。もし先に 4 番 hole が解いてあれば、6 番で runTC を実行することはできる。しかし、今回のように比較的単純な例でも 4 番を正確に自動生成させることは難しく、6 番を解いてから Agda の制約解消によって 4 番を苦労せずに解く方法を探りたい。そこで、R*Trans を適用したところに戻ってマクロを書いてみる。

3.2.3 問題を回避する

3.2.2 節での問題は、他の hole に依存する hole でマクロを実行したことが原因のようだ。そこで、3.2.1 節でのマクロを実行する際に、R*Trans と同時に RFrame と App_1 を答えとして与えるようにする。3.2.1 節のマクロを図 2 のように変更する。

図 2 のマクロでは、unify が 2 回使われている。1 回目では m1 から m5 までのメタ変数を生成し、2 回目でそれらのうち m4 についてさらに unify をおこなった。これを一般化すると、1 つのメタ変数 (これを goal と呼ぶことにする) を解くために unify をおこなうと、新しい複数のメタ変数 (これを subgoal と呼ぶことにする) が作られ、その操作を繰り返すことで自動的に Agda の証明を書くことができる、と考えられる。3.3 節では、goal と subgoal を使って様々なコンストラクタを繰り返し unify させる方法を考える。

[5] Agda Bug Tracker (https://github.com/agda/agda/issues/4143) に報告済み。4 番を解いていない場合、6 番ではどんなマクロも C-c C-m で実行することはできなかった。6 番 hole は 4 番 hole の型に依存してはいるが、PHOAS でない状況ではこのようなエラーが起きないので、今回のような現象は PHOAS 特有のものである可能性はある。

```
1  macro
2    runTC : Term → TC ⊤
3    runTC hole = do
4      m1 ← newMeta unknown
5      m2 ← newMeta unknown
6      m3 ← newMeta unknown
7      m4 ← newMeta unknown
8      m5 ← newMeta unknown
9      unify hole (con (quote R*Trans)
10       (vArg m1 :: vArg m2 :: vArg m3 :: vArg m4 :: vArg m5 :: []))
11     m6 ← newMeta unknown
12     m7 ← newMeta unknown
13     unify m4
14       (con (quote RFrame)
15         (vArg (con (quote App₁) (vArg m6 :: [])) :: vArg m7 :: []))
```

新しいメタ変数を 5 つ作成

解きたい hole と R*Trans を単一化 5つの explicitな引数と共に

メタ変数 m4 で単一化 コンストラクタ RFrame と App_1と引数 2 つと共に

図 2. 3.2.3 節でのエラーを回避するマクロ

3.3 Goal を使って繰り返す

この節では、goal を使って繰り返し証明項を生成することを目指す。

3.3.1 作業を一般化する

まずは、Goal というデータ型を定義する。Goal は、解くべきメタ変数をあらわす gHole と、メタ変数を解くときの「現在のコンテキスト」をあらわす gCtx の 2 つのフィールドからなる。gCtx は 3.4 節で重要な役割をはたす。

```
record Goal : Set where
  constructor mkGoal
  field
    gHole : Term
    gCtx : List (Arg Type)
```

また、3.2 節では、unify したいコンストラクタによって異なる数のメタ変数をいちいち用意していた。これを、下に示す unifyGoalCons という関数で一般化する。

```
1  unifyGoalCons : (conName : Name) → (g : Goal) → TC (List Goal)
2  unifyGoalCons conName (mkGoal hole ctx) = do
3    metas ← nameToMetas conName
4    let goals = metaToGoal (map unArg metas) ctx
5    unify hole (con conName metas)
6    returnTC goals
```

型情報をメタ変数に置換 メタ変数を Goal 型に変形 コンストラクタで単一化 新たな subgoal を返す

この関数では、unify したいコンストラクタ名と、unify 先の goal を受け取ってきたら、unify をおこなって新しく生成された subgoal のリストを返している。例えば、R*Trans を unify したいときは unifyGoalCons (quote R*Trans) g のように使うことができる。これを詳しく見ていく。

3 行目では、下に示す R*Trans の型にしたがってメタ変数のリストを作成している。R*Trans の型は、implicit な引数 τ_1 が 1 つと、explicit な引数が 5 つあるので、それに対応して hidden なメタ変数を 1 つと visible なメタ変数を 5 つ作成する。4 行目では、metaToGoal という関数でメタ変数のリストを Goal 型のリストになるように変形している。5 行目でコンストラクタとメタ変数のリストを、現在解きたい goal である hole と unify する。最後に、6 行目で subgoal のリストを返す。

```
R*Trans : {τ₁ : typ} (e₁ e₂ e₃ : term[ myvar ] τ₁) →
          Reduce e₁ e₂ → Reduce* e₂ e₃ → Reduce* e₁ e₃
```

```
1  substTac : (g : Goal) → (args : List (Arg Term)) → TC (List Goal)
2  substTac g args with findName args
3  substTac g args | just (quote Val) = unifyGoalCons (quote sVal) g
4  substTac g args | just (quote App) = unifyGoalCons (quote sApp) g
5  substTac g args | _ = print "no idea what to do with Subst"
6
7  reduceTac : (g : Goal) → (args : List (Arg Term)) → TC (List Goal)
8  reduceTac (mkGoal gHole gCtx) args = do
9    catchTC
10     (unifyGoalCons (quote RBeta) (mkGoal gHole gCtx))
11     do
12       m1 ← newMeta unknown
13       m2 ← newMeta unknown
14       unify gHole
15         (con (quote RFrame)
16           (vArg (con (quote App1) (vArg m2 :: [])) :: vArg m1 :: []))
17       returnTC (mkGoal m1 gCtx :: [])
18
19 baseTac : (g : Goal) → TC (List Goal)
20 baseTac g =
21   inferType (Goal.gHole g) >>=
22     λ{ (def (quote SubstVal) args) → substValTac g args
23      ; (def (quote Subst) args) → substTac g args
24      ; (def (quote Reduce) args) → reduceTac g args
25      ; (def (quote Reduce*) args) → reduce*Tac g args
26      ; (pi a b) → introTac g
27      ; term → typeError (termErr term :: [])
28      }
29
30 recTac : ℕ → List Goal → TC ⊤
31 recTac zero g = returnTC tt
32 recTac (suc n) [] = returnTC tt
33 recTac (suc n) (mkGoal gHole gCtx :: rest) =
34   catchTC
35     (do
36       subgoals ← extendCtx (reverse gCtx) (baseTac (mkGoal gHole gCtx))
37       recTac n (subgoals ++ rest))
38     (recTac n rest)
39
40 macro
41   runTC : Term → TC ⊤
42   runTC hole = do
43     recTac 100 ((mkGoal hole []) :: [])
```

[代入規則の場合]
構文木を走査
値の規則のとき
関数適用のとき
それ以外

[簡約規則の場合]

β 簡約を試して
失敗したら
メタ変数を生成

解きたい hole と
RFrame で単一化
App_1 も一緒に
m1 が subgoalに

各規則を 1 回適用

hole の型を推論
値の代入規則
項の代入規則
簡約規則(1 回)
簡約規則(複数回)
関数型
それ以外はエラー

繰り返し呼び出す
燃料切れ
証明完成

gCtxで拡張して
先頭のgoalを解く
subgoalを追加

失敗したら
先頭はスキップ

解くgoalは hole
燃料量は 100

図 3. unifyGoalCons を使った作業を繰り返す

3.3.2 作業を繰り返す

3.3.1 節で一般化した作業を繰り返しおこなうためのコードを、図 3 に示す。ユーザは 40 〜 43 行目に定義された runTC を hole の中に書いて C-c C-m のコマンドを実行してマクロを呼び出すことで、これらの一連のコードを実行することができる。

runTC では、recTac が数字と現在の goal と共に呼び出されている。3.3.1 節からもわかるように、subgoal のリストの長さは必ずしも短くなるとは限らない。そのため recTac では呼び出し回数 (燃料の量) をユーザが指定し、その自然数について再帰させることによって recTac の停止性を保証している。31 行目は燃料が尽きたため何もしないことを意味し、32 行目はたとえ燃料が残っていてもすべての goal が解けたのでこれ以上何もしないことを意味する。35 〜 38 行目は、baseTac を呼び出して goal を解こうと試みている。もし goal を解くことに成功したら、37 行目で新しく生成された subgoal を解くべき goal のリストに加えて再帰呼び出しをする。失敗した場合は、38 行目で現在の goal には何もせずに無視して、残りの goal を解くよう再帰呼び出しをおこなう。失敗した goal を無視するのは、解けるところから解いて残りは Agda の制約解消で自動生成したいと考えるためである。

次に、baseTac を見る。21 行目で解きたい goal の型を推論し、22 〜 27 行目でその結果をパター

ンマッチしている。3.1 節でも述べたが、λ 計算の代入規則や簡約規則をすべてデータ型として定義したので、どれも (def (quote 規則名) 引数) の形をとっている。各規則に対する動作 (substTac, reduceTac など) を呼び出し、その結果をそのまま返している。また、推論結果が (pi a b) になるのは、Agda レベルの関数型のときである。関数型については 3.4 節で詳しく述べる。

最後に、各規則に対する動作を見る。関数 baseTac で goal の型の推論結果が (def (quote Subst) args) だった場合は substTac が呼び出される。このとき、引数のリスト args の構文木の形によって、適用できるコンストラクタが異なる。例えば、goal の型の推論結果が次に示すような形をしていたとする。このとき、出だしの Subst (λ x → Val ... を見るだけで Val のための代入規則 sVal を適用すればよいと判断できる。

```
Goal: Subst (λ x → Val (Fun (λ k₁ → cpsEta′ (e x) (Var k₁))))
      (cpsEtaV v₂) ?7
```

そこで、図 3 の 2 行目のように、構文木の中で一番最初に現れるコンストラクタ名を探す findName という関数を用いて場合分けをおこなう。コードの詳細は省略するが、substValTac や reduce*Tac も同様の形式で定義することができる。

また、reduceTac では、findName が使われていない。これは、1 ステップの簡約の際は一番最初にあらわれたコンストラクタ名を見るだけではどの規則を適用すべきか判定できないため、愚直に RBeta を試し、失敗したら RFrame を試すようになっている。

3.4 コンテキストを拡張する

この節では、解きたい goal が Agda の関数型 (pi 型) の場合について考える。

3.4.1 型とコンテキストの内部表現を調べる

コンテキストを拡張するには、hole の型とコンテキストの構造を知る必要がある。まずは、以下の簡単な例を調べてみる。

```
introTest : {a : Set} → ℕ → Bool → String → ℕ
introTest {a} n b = { }0
```

マクロを使って hole の型やコンテキストの内部表現を調べてみる。introTest の 0 番 hole で、2.4.2 節で定義した showGoal を実行すると以下のようになり、これは「String 型の変数を受け取ると、ℕ 型を返す」ことを意味する。

```
pi (arg (arg-info visible relevant) (def (quote String) []))
(abs "_" (def (quote ℕ) []))
when checking that the expression unquote showGoal has type
String → ℕ
```

また、0 番 hole 内の現在のコンテキストを調べるには、次に示す showCtx を使う。

```
1 macro
2   showCtx : Term → TC ⊤
3   showCtx hole = do
4     ctx ← getContext
5     t ← quoteTC ctx
6     typeError (termErr t :: [])
```

4 行目の getContext では、この関数が呼び出された時点のコンテキストを取得している。0 番 hole のコンテキストは次のようになり、現在 3 つの変数が含まれていることがわかる。

```
arg (arg-info visible relevant) (def (quote Bool) []) ::
arg (arg-info visible relevant) (def (quote ℕ) []) ::
arg (arg-info hidden relevant) (sort (lit 0)) :: []
when checking that the expression unquote showCtx has type
String → ℕ
```

これらの引数は hole に近い順に表示されている。ここで、Reflection API の機能を使わずに、Emacs の C-c C-, コマンドで 0 番 hole の型とコンテキストを確かめると以下のようになる。showGoal や showCtx の結果と対応していることがわかる。

```
Goal: String → ℕ
————————————————————————————————————————————————————
b : Bool
n : ℕ
a : Set
```

これまでの知識を踏まえて、以下のようなマクロ showExtendedCtx を考える。一見 showCtx に似たマクロだが、4 行目に着目すると、「String 型で拡張されたコンテキストのもとで、『その時点のコンテキスト』を取ってくる」ということをしている。

```
1 macro
2   showExtendedCtx : Term → TC ⊤
3   showExtendedCtx hole = do
4     newCtx ← extendContext (vArg (def (quote String) [])) getContext
5     t ← quoteTC newCtx
6     typeError (termErr t :: [])
```

実際に introTest の 0 番 hole で実行してみると、以下の結果が得られる。showCtx の結果に加えて、一番最初に String 型の変数が現れている。

```
arg (arg-info visible relevant) (def (quote String) []) ::
arg (arg-info visible relevant) (def (quote Bool) []) ::
arg (arg-info visible relevant) (def (quote ℕ) []) ::
arg (arg-info hidden relevant) (sort (lit 0)) :: []
when checking that the expression unquote boring has type
String → ℕ
```

このように、extendContext という Reflection API の 1 つを利用すると、指定した型で拡張したコンテキストのもとで動作をおこなうことができる。

3.4.2 繰り返しコンテキストを拡張する

コンテキストの拡張方法がわかったので、図 3 で一緒に使われる一般的な関数を定義する。

```
1 introTac : (g : Goal) → TC (List Goal)
2 introTac (mkGoal gHole gCtx) = do
3   (pi (arg (arg-info v r) a) (abs s b)) ← inferType gHole
4     where t → print "not a function type!"
5   body ← extendContext (arg (arg-info v r) a) (newMeta b)
6   unify gHole (lam v (abs s body))
7   returnTC ((mkGoal body (arg (arg-info v r) a :: gCtx) :: []))
```

この関数 introTac は、図 3 での関数と同様に、goal を受け取ったらそれを解いて新たな subgoal を返すものである。3,4 行目では、受け取ってきた goal の型が関数型であることを確認し、そうでなければエラーを出している。このとき、型は (pi (arg (arg-info v r) a) (abs s b)) の形をしていて、a 型を受け取って b 型のものを返す関数型であるといっている。そこで、5 行目では、a 型で拡張したコンテキストのもとで b 型のメタ変数を新たに作成している。6 行目で、5 行目で作成したメタ変数 body をラムダ抽象の body 部分として与えている。最後に、メタ変数 body と一緒に、現在のコンテキスト gCtx を a 型で拡張したものを新たな subgoal として返している。

このように goal の中に登録されたコンテキストの情報は、図 3 の recTac で利用されている。recTac の 36 行目を見ると、各 goal を baseTac を使って解く前に、extendCtx という関数で一時的にコンテキストを拡張している。extendCtx は、extendContext を何度も呼び出すことで、複数の型でコンテキストを拡張できる。もし、これをおこなわないと、λ 抽象の body 部分でマクロを実行しようとした時点で型チェックに失敗し、「スコープ外の de Bruijn index にアクセスできない」という内容のエラーが出てしまう。

ここまでの内容を使って図 3 のコードを実行すると、下のような結果が得られる。以下の 9,10 行目で RFrame, App$_1$, RBeta, sVal, sFun のコンストラクタを与えることができており、さらに sFun の後に (λ x → ?) を表示できている。

```
1  correctEta {myvar} {τ} {e′ = e′} (RBeta {e = e} {v₂} sub) k sche =
2    R*Trans
3      (App
4       (App (Val (Fun (λ x → Val (Fun (λ k₁ → cpsEta′ (e x) (Var k₁))))))
5        (Val (cpsEtaV v₂)))
6       (Val (Fun (λ v → k (Var v)))))
7      (App (Val (Fun (λ z → { }0))) (Val (Fun (λ v → k (Var v)))))
8      (cpsEta e′ k)
9      (RFrame (App₁ (Val (Fun (λ v → k (Var v)))))
10      (RBeta (sVal (sFun (λ x → { }1)))))
11     (R*Trans (App (Val (Fun (λ z → _))) (Val (Fun (λ v → k (Var v)))))
12      (cpsEta e′ k) (cpsEta e′ k) (RBeta { }2) (R*Id (cpsEta e′ k)))
```

この状態からさらに進んで、10 行目に残っている 1 番 hole を解くには、ユーザが定義した補題を使う必要がある。ここで想定されている答えは、(eSubst′ x sub) というもので、該当する補題の関数 eSubst′ を unify し、さらに補題の引数を一緒に与えることができればよい。

3.5 補題を適用する

補題を適用する方法は、コンストラクタの適用方法と基本的には同じで、3.3.1 節の unifyGoalCons の作り方に従えばよい。つまり、適用したい補題の型を取得し、補題の引数の数と同じだけのメタ変数のリストを新たに作成して unify すればよい。ただし、今回の証明では、補題の引数には現在のコンテキストに含まれる変数が該当することがあるため、メタ変数のリストの各要素が現在のコンテキスト内の変数と unify できるか試している。(この手順の詳細は付録 B を参照されたい。)

3.6 再帰を書く

ここまでの方法を利用すると図 1 の 0 番を埋められるが、1,2 番には再帰的な証明が必要となる。

関数の再帰を Reflection API で表現する方法は、公式 [1] からは提供されていない。そこで、induction principle [19] を利用する方法が考えられる。一見、この方法を使えば correctEta を示すのに必要な証明項をすべて自動で生成できるように思えるが、現時点での Reflection API の機能ではすべてを自動化することは難しい。本節では、induction principle を手動で定義し、それを適用する際の作業の一部を Reflection API によって自動化することを目指す。

3.6.1 Induction principle を作る

示したい定理 correctEta では、導出による帰納法、つまり簡約規則 Reduce の場合分けによって証明をおこなうことにしていた。そこで、Reduce の規則に従って、次のような induction principle を定義する。この関数 foldReduce では、元のデータ型 Reduce の各規則を base, ind$_1$, ind$_2$ として書き下すような形で定義できている。

```
foldReduce : ∀ {τ : typ} →
       {myvar : typ → Set} {e : term[ myvar ] τ} {e′ : term[ myvar ] τ} →
       (P : ∀ {τ₀ : typ} → term[ myvar ] τ₀ → term[ myvar ] τ₀ → Set) →
       (base : ∀ {τ₁ τ₂ : typ} {e : myvar τ₂ → term[ myvar ] τ₁}
                 {v₂ : value[ myvar ] τ₂} {e′ : term[ myvar ] τ₁} →
               Subst e v₂ e′ →
               P {τ₀ = τ₁} (App (Val (Fun e)) (Val v₂)) e′)
       (ind₁ : ∀ {τ₃ τ₄ : typ} (e₂ : term[ myvar ] τ₄)
             (e e′ : term[ myvar ] (τ₄ ⇒ τ₃)) →
             P {τ₀ = τ₄ ⇒ τ₃} e e′ →
             P {τ₀ = τ₃} (frame-plug (App₁ e₂) e) (frame-plug (App₁ e₂) e′)) →
       (ind₂ : ∀ {τ₃ τ₄ : typ} (v₁ : value[ myvar ] (τ₄ ⇒ τ₃))
             (e e′ : term[ myvar ] τ₄) →
```

```
              P {τ₀ = τ₄} e e′ →
              P {τ₀ = τ₃} (frame-plug (App₂ v₁) e) (frame-plug (App₂ v₁) e′)) →
        Reduce e e′ → P {τ₀ = τ} e e′
foldReduce P base ind₁ ind₂ (RBeta sub) = base sub
foldReduce P base ind₁ ind₂ (RFrame {e = e} {e′} (App₁ e₂) red) =
  ind₁ e₂ e e′ (foldReduce P base ind₁ ind₂ red)
foldReduce P base ind₁ ind₂ (RFrame {e = e} {e′} (App₂ v₁) red) =
  ind₂ v₁ e e′ (foldReduce P base ind₁ ind₂ red)
```

もし、Agda の Reflection API の機能として再帰をサポートするとなると、すでに Coq でおこなわれている [19] ように、ユーザがデータ型を定義するたびにこのような induction principle を自動生成させる必要があるだろう。

3.6.2 Induction principle を適用する

この節では、induction principle の foldReduce を証明の中で使う方法を示す。図 1 では、これまで 3 つのケースにあらかじめ場合分けしてから証明項を与えていた。しかし、induction principle が場合分けの役割も担うため、図 1 での場合分けの操作は必要なくなる。場合分け前の状態で foldReduce を手動で適用したものを以下に示す。

```
1  correctEta : {myvar : typ → Set} {τ : typ} {e e′ : term[ myvar ∘ cpsT ] τ} →
2               (κ : value[ myvar ] cpsT τ → term[ myvar ] Nat) →
3               Reduce e e′ → schematic κ →
4               Reduce* (cpsEta e κ) (cpsEta e′ κ)
5  correctEta {myvar} {τ} {e = e} {e′} red =
6    foldReduce
7      (λ {τ} e₀ e′₀ →
8        (k : value[ myvar ] (cpsT τ) → term[ myvar ] Nat) →
9        schematic k →
10       Reduce* (cpsEta e₀ k) (cpsEta e′₀ k))
11     { }0 { }1 { }2 red
```

Induction principle を使うには、7 〜 10 行目のように、correctEta の型の一部を示したい性質として foldReduce の 1 つ目の引数に与える必要がある。これは Agda の制約解消では自動生成することができない上に、Reflection API を使って生成することも難しい。Reflection で自動生成しようとすると、3.5 節の補題のときのように、マクロ runTC を定義している時点で関数の型が判明していることが前提になってくる。しかし、runTC は関数 correctEta よりも先に定義したいので、Reflection を使って生成することは難しい[6]。

上に示したような foldReduce の適用の操作を手動でおこなえば、11 行目に残っている 0 〜 2 番 hole の証明はほとんど自動化できる。0 番 hole はこれまでの RBeta のケースに対応しているため、図 3 の一連のマクロを 0 番 hole の中で実行すれば完成させられる。2 番 hole は、3.5 節の方法を利用して補題を適用すると証明を完成させることができる。1 番 hole も 2 番と同様の方法を試みたが、必要な補題 (kSubst) の中で手動で引数を与えざるをえない場面があるため、完全には自動化できていない。ここまでの方法を使って、示したい定理 correctEta の証明約 30 行分を生成できた。

3.7 マクロに汎用性はあるか

3 節では、Reflection API を用いて、ユーザが定義したデータ型を用いた証明にカスタマイズしたマクロを書き、コンストラクタ・ラムダ抽象・関数・再帰 のコードを自動生成する方法を示した。

[6] あまり推奨したくない方法ではあるが、示したい定理 correctEta と一連のマクロを相互再帰の形で定義してしまえば、無理矢理マクロに定理の型を取得させることができる。さらにいえば、この方法を使うと induction principle を使わなくても correctEta を補題のように取り扱える。マクロの実行前は、示すべき定理は未完成な証明を含むため、これを補題として扱うのは安全な方法とはいえない。ただ、仮にこのような方法を取ったとしても、Reflection API の unify を実行するときにいずれ停止性チェックが行われるため、無限ループを起こすようなコードは生成されないだろう。

3 節で利用したマクロは全体で約 250 行あり、これによって約 30 行の正当性の証明を自動化している。マクロの内訳は、ユーザが定義したデータ型によらず利用できる関数 (コンストラクタを適用する `unifyGoalCons` などが相当) が約 100 行程度と、ユーザが定義したデータ型にカスタマイズしたマクロ (図 3 の 1 〜 28 行目などが相当) が約 150 行程度となっている。後者については、ユーザが定義したデータ型を Reflection API で使える形に素直に書き直した形になっている。例えば、3.1 節で名前を紹介した `Subst` は図 3 の `substTac` という関数に、また `Reduce` は `reduceTac` に対応している。そのため、もしユーザが定義したデータ型などを読み込んで、対応するマクロや関数を自動生成することができれば、CPS 変換以外の証明にも利用しやすくなるだろう。

3.2 節から 3.5 節までで生成した証明項は、変数名や改行位置の違いなどはあるが、手動で書く証明項とほぼ同じ内容を出力できている。さらに読みやすさを追求するには、カスタマイズしたマクロを equational reasoning を出力するように変更するのがよいだろう。一方で、3.6 節では、手動の証明とは異なり、induction principle を利用したため、手動の証明よりも読みにくくなっている。

4 より大規模な証明でマクロを利用する

本節では、3 節で紹介したマクロの書き方を比較的大規模な証明で応用する様子を述べる。

例として、限定継続命令 `shift/reset` のための Selective CPS 変換 [3] の正当性の証明を Agda で実装したもの [12] を用いる。この実装は 3 節の実装を拡張して作られており、実装は全体で約 7800 行ある。その内訳は、項や規則の定義 (800 行)・補題 (5000 行)・正当性の証明 (2000 行) で、補題と正当性の証明が実装の多くを占める。正当性の証明や補題の一部は、簡約規則や代入規則を繰り返し適用することよって証明できるという点で 3 節と同じである。ただし、項の定義が 3 節よりも複雑なため、それに伴って簡約規則や代入規則も複雑になり、規則数も増えている。

現時点では、Selective CPS 変換の正当性の証明の手動での実装 約 2000 行分に対して、3 節で紹介した方法を適用することで、手動での実装 約 1300 行分のコードを生成できている。このとき、実際に生成されたコードは再帰的な部分を除けば手動の証明とほとんど同じ内容だが、改行が頻繁に行われたため、約 1500 行が出力された。このために利用したマクロは全部で約 350 行あり、その内訳は、ユーザが定義したデータ型によらず利用できるマクロ (約 100 行) と、項や規則の定義にカスタマイズしたマクロ (約 250 行) である。前者は 3 節と同一の実装を利用し、後者は項や規則の定義に合わせて新たに作成した。同様の方法は、正当性の証明部分だけでなく、補題の一部にも利用できるだろう。

Selective CPS 変換の正当性の証明では、マクロだけでは生成できていない部分もある。まず、正当性の証明内で相互再帰が起きるので、より複雑な induction principle を適用しなければならない。また、3 節では順方向のみの簡約で証明を完成させることができていたが、本節の証明では順方向と逆方向の簡約関係がおこる場合がある。さらに、簡約関係には似た規則が複数あるため、内部表現の構造の情報だけでは適用すべき規則が一意に定まらない場合もある。もし適切でない規則が `unify` できてしまうと、型エラーは起きないので `catchTC` を用いても捕捉できず、バックトラックができない。適切にバックトラックするには、非決定性モナドを導入して複数の規則を試すのがよいだろう。

また、本稿のケーススタディによって、現時点の Reflection API には、主に 2 つの課題があることがわかってきた。1 つ目の課題は、再帰についてである。再帰を書くために induction principle を使ったとしても、3.6 節で述べたように、マクロのみで完全に自動化することは難しい。2 つ目の課題は、ユーザビリティについてである。証明の自動生成中に `unify` に成功しても、内部表現からは Agda のコードが一意には定まらないために、エラーを出さずにコード生成に失敗することがある。これは unquote のための情報が足りなかっただけであり、型エラーではないので、`catchTC` を使っても捕捉することができない。また、3.2.2 節で述べた現象についても改善が期待される。

5 関連研究

Agda の Reflection API 本稿で紹介した Reflection API は、定理証明系言語 Idris [4] の Elaborator Reflection [7] に影響を受けて作られ、バージョン 2.5.2 から提供され始めた。Agda と Idris の Reflection についての基本的なアイディアは共通していて「型チェック機能を API として提供する」というものだが、両者の言語実装が異なるため、それぞれの API の詳細は異なっている。Reflection API の単一化機能は、Cockx らによる unification [9] をもとにしている。

Agda での自動証明の取り組み Agda には、ビルトインの自動証明機能がある。これは、Lindblad らによる Agsy [16] をもとにしている。Agsy は Haskell で実装されており、Agda の機能の一部としてユーザに提供される。そのため、Agsy をユーザがカスタマイズすることは難しい。小規模で一般的な証明であれば Agsy で証明項を自動生成できるが、本稿で取り上げたような PHOAS で書かれた証明には未対応だった。

また、Agda のバージョン 2.5.1 以前に提供されていた Reflection 機能 [20] では、主に quote と unquote が中心となっており、型チェック機能やメタ変数を取り扱う機能は含まれていなかった。Kokke らが提案した Auto in Agda [15] というライブラリでは、これを利用して自動証明をおこなう。ユーザが証明に使いたいコンストラクタ名などをヒントとして (n 個) 指定すると、ヒントから証明探索木 (n 分木) を作成し証明探索をおこなう。このライブラリでは、first-order unification [18] を Agda で実装することで、生成された証明項の正しさを保証している。本研究は、単一化したい項の内部表現の構造によって適用するヒントの数を絞っているという点において [15] と異なる。新しい Reflection API では、依然として、古い Reflection 機能のコマンドもいくつか残っている。しかしこれらはマクロの実装には推奨されていない [1]。

バージョン 2.5.2 以降の Reflection API を用いたものに Ataca [8] がある。Ataca では Coq にある `intro` や `destruct` などの tactic を実装した。また、Vivekanandan は、Agda の Reflection API を使って higher inductive type の実装を自動化する過程で、再帰的なデータ型に対する証明を自動化した [21]。

Agda 以外の言語の自動証明の取り組み 定理証明支援系言語の多くで、メタプログラミングの方法が提案されている。特に Coq では、自動証明のための環境がよく整備されている。自動証明 tactic を書くための言語として、Ltac [6,11], Mtac [14,22], Rtac [17] などが開発されている。

6 まとめと今後の課題

本稿では、Agda の Reflection API を紹介し、CPS 変換の正当性の証明を例にとってその使い方を説明した。また、Reflection が比較的大規模な証明においてもユーザの負担を軽減できることを、Selective CPS 変換の Agda 実装を使って示した。

今後は、ユーザが定義したデータ型によらず利用できるマクロのライブラリの作成と、ユーザが定義したデータ型にカスタマイズしたマクロそのものを自動生成する仕組みの作成によって、Agda によるさまざまな証明の負担を軽減する方法を模索したい。

謝辞

有益なコメントをくださった査読者の皆様に感謝申し上げます。また、本研究は一部 JSPS 科研費 18H03218 の助成を受けたものです。

参考文献

[1] Agda Development Team. Reflection – Agda 2.6.0.1 documentation. https://agda.readthedocs.io/en/v2.6.0.1/language/reflection.html, 2019. [Online; accessed 29-Dec-2019].

[2] Agda Development Team. The Agda standard library. https://github.com/agda/agda-stdlib/blob/master/src/Reflection.agda, 2019. [Online; accessed 29-Dec-2019].

[3] K. Asai and C. Uehara. Selective CPS Transformation for Shift and Reset. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2018, pp. 40–52, 2018.

[4] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming*, Vol. 23, No. 5, pp. 552–593, 2013.

[5] A. Chlipala. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, ICFP 2008, pp. 143–156, September 2008.

[6] A. Chlipala. *Certified Programming with Dependent Types*. Cambridge: MIT Press, 2013.

[7] D. Christiansen and E. Brady. Elaborator Reflection: Extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pp. 284–297, New York, NY, USA, 2016. ACM.

[8] J. Cockx. ataca: A TACtic library for Agda. https://github.com/jespercockx/ataca, 2019. [Online; accessed 29-Dec-2019].

[9] J. Cockx, D. Devriese, and F. Piessens. Unifiers as equivalences: proof-relevant unification of dependently typed data. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pp. 270–283. ACM, 2016.

[10] O. Danvy and A. Filinski. Representing Control: a Study of the CPS Transformation. *Mathematical Structures in Computer Science*, Vol. 2, No. 4, pp. 361–391, 1992.

[11] D. Delahaye. A proof dedicated meta-language. *Electronic Notes in Theoretical Computer Science*, Vol. 70, No. 2, pp. 96–109, 2002.

[12] C. Ishio and K. Asai. Verifying Selective CPS Transformation for shift/reset. In *Proceedings of the symposium on Trends in Functional Programming 2019 (TFP'19)*, 21 pages. Post proceeding version to appear in EPTCS.

[13] 石尾千晶, 山田麗, 浅井健一. Agda による PHOAS を用いた CPS 変換の正当性の証明. 第 20 回プログラミングおよびプログラミング言語ワークショップ論文集, 2018.

[14] J. Kaiser, B. Ziliani, R. Krebbers, Y. Régis-Gianas, and D. Dreyer. Mtac2: typed tactics for backward reasoning in Coq. *Proceedings of the ACM on Programming Languages*, Vol. 2, No. ICFP, p. 78, 2018.

[15] P. Kokke and W. Swierstra. Auto in Agda. In *Mathematics of Program Construction*, pp. 276–301. Springer International Publishing, 2015.

[16] F. Lindblad and M. Benke. A Tool for Automated Theorem Proving in Agda. In *Types for Proofs and Programs*, pp. 154–169, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[17] G. Malecha and J. Bengtson. Rtac: A fully reflective tactic language. In *International Workshop on Coq for PL (CoqPL)*. Citeseer, 2015.

[18] C. McBride. First-order unification by structural recursion. *Journal of Functional Programming*, Vol. 13, pp. 1061–1075, 2003.

[19] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hritcu, V. Sjöberg, and B. Yorgey. Logical Foundations. Software Foundations series. https://softwarefoundations.cis.upenn.edu/lf-current/IndPrinciples.html, May 2018. [Online; Version 5.5; accessed 29-Dec-2019].

[20] P. Van Der Walt and W. Swierstra. Engineering proof by reflection in Agda. In *Symposium on Implementation and Application of Functional Languages*, pp. 157–173. Springer, 2012.

[21] P. Vivekanandan. Code generation for higher inductive types. In *International Workshop on Functional and Constraint Logic Programming*, pp. 18–35. Springer, 2018.

[22] B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: a monad for typed tactic programming in Coq. *Journal of Functional Programming*, Vol. 25, 2015.

A　手動で Agda の証明を書く方法

本節では、ユーザが手動で証明を書くときにどのような作業をおこなうかを明らかにする。
図 1 を証明するためには、主に以下のような 3 つの手順が必要となる。

1. 帰納法をかけるものについて場合分けをおこなう
   今回の証明では、簡約関係 Reduce の導出による帰納法で証明できるよう、C-c C-c で場合分けをおこなう。必要に応じて、それ以外の引数もイコールの左辺に書いておく。

```
correctEta : {myvar : typ → Set} {τ : typ} {e e′ : term[ myvar ∘ cpsT ] τ} →
             (κ : value[ myvar ] cpsT τ → term[ myvar ] Nat) →
             Reduce e e′ → schematic κ →
             Reduce* (cpsEta e κ) (cpsEta e′ κ)
correctEta {myvar} {τ} {e′ = e′} (RBeta {e = e} {v₂} sub) k sche = { }0
correctEta {myvar} {τ} (RFrame {e = e} {e′} (App₁ e₂) red) k sche = { }1
correctEta {myvar} {τ} (RFrame {e = e} {e′} (App₂ v₁) red) k sche = { }2
```

2. hole の型と現在のコンテキストの情報を得る
   まずは、ベースケースとなる RBeta の証明の 0 番 hole を見ていく。hole の中にカーソルを置いて C-c C-, というコマンドを実行すると、Emacs の別バッファで、hole の中に入れるべき型が表示される。また、点線より下には現在の hole の中で使用可能な変数一覧を見ることができる。このような変数一覧のことを「現在のコンテキスト」と呼ぶ。

```
Goal: Reduce*
      (App
       (App (Val (Fun (λ x → Val (Fun (λ k₁ → cpsEta′ (e x) (Var k₁))))))
        (Val (cpsEtaV v₂)))
       (Val (Fun (λ v → k (Var v)))))
      (cpsEta e′ k)
————————————————————————————————————————————————————————————
sche : schematic k
k : value[ myvar ] (cpsT τ) → term[ myvar ] Nat
sub : Subst e v₂ e′
e′ : term[ (λ {x} → myvar) ∘ cpsT ] τ
v₂ : value[ (λ {x} → myvar) ∘ cpsT ] τ₂
e : ((λ {x} → myvar) ∘ cpsT) τ₂ →
      term[ (λ {x} → myvar) ∘ cpsT ] τ
τ₂ : typ (not in scope)
myvar : typ → Set
τ : typ
```

3. 手順 2 で得た型の情報をもとに、予想される答えを書き、型チェックをおこなう
   手順 2 の情報から、複数回の簡約が必要であるため、Reduce* のコンストラクタのひとつである R*Trans が該当することがわかる。R*Trans というコンストラクタは引数を明示的に 5 つ受け取ることがわかるので、ユーザは hole の中に R*Trans ? ? ? ? ? を書き、C-c C-r のコマンドを使ってユーザの答えが型チェックに通るかどうかを確かめることができる。型チェックの結果、問題がなければ下のように新たな hole が 5 つ生成される。

```
correctEta {myvar} {τ} {e′ = e′} (RBeta {e = e} {v₂} sub) k sche =
  R*Trans { }3 { }4 { }5 { }6 { }7
```

   このように新たに生成された hole のうちいくつかは、すでに Agda 内部で型の制約解消が起きて答えが明らかになっているものや、他の hole の型が定まることで制約解消が起きるものがある。例えば、定義から 3 番と 5 番の型はすでに Agda 内部での制約解消によって一意に定まっている。これらの hole の中で C-c C-s コマンドを実行すると、以下のようになる。

```
correctEta {myvar} {τ} {e′ = e′} (RBeta {e = e} {v₂} sub) k sche =
  R*Trans
    (App
```

```
  (App (Val (Fun (λ x → Val (Fun (λ k₁ → cpsEta′ (e x) (Var k₁))))))
   (Val (cpsEtaV v₂)))
  (Val (Fun (λ v → k (Var v)))))
{ }4
(cpsEta e′ k)
{ }6
{ }7
```

さらに、6 番の型は 3 番と 4 番の間の 1 ステップ簡約の関係をあらわすので、6 番がわかると連動して 4 番が制約解消できるようになる。そのため、6 番を先に解くのが簡単である。

残っている hole に対しても、手順 2 と手順 3 を繰り返すことで手動で証明を完成させることができる。

B 補題を適用する

本節では、Reflection API を使って補題を適用する方法を述べる。基本的な考え方は 3.3.1 節でのコンストラクタの適用と同じだ。ただし、少し複雑な補題を与える際には引数の依存関係に注意が必要だ。

まずは、補題を適用するための以下の関数を見てみよう。

```
1 unifyGoalDef : (defName : Name) → (g : Goal) → TC (List Goal)
2 unifyGoalDef defName (mkGoal hole ctx) = do
3   metas ← nameToMetas defName
4   let goals = metaToGoal (map unArg metas) ctx
5   unify hole (def defName metas)
6   repeat assumption goals
7   returnTC goals
```

3 〜 5 行目は、3.3.1 節とほぼ同じ操作をおこなっている。3.3.1 節と異なるのは 6 行目で、補題の引数として与えるメタ変数のリストの全ての要素に対して、現在のコンテキスト内の変数があてはまるか調べる assumption をおこなっている。

assumption と補助関数 tryVar の定義を以下に示した。

```
1  tryVar : (varIndex : ℕ) → (g : Goal) → TC ⊤
2  tryVar zero (mkGoal gHole gCtx) =
3    catchTC (unify gHole ((var 0) [])) (returnTC tt)
4  tryVar (suc i) (mkGoal gHole gCtx) =
5    catchTC (unify gHole (var (suc i) [])) (tryVar i (mkGoal gHole gCtx))
6
7  assumption : (g : Goal) → TC (List Goal)
8  assumption g = do
9    ctx ← getContext
10   tryVar (length ctx) g
11   returnTC []
```

関数 assumption では、9 行目で現在のコンテキスト一覧を取得して、10 行目でコンテキストのリストの長さと解きたい goal を tryVar に渡している。Reflection の内部表現では、変数管理に de Bruijn index を利用しているので、コンテキストに含まれる i 番目の変数にアクセスするには、(var i []) と書けばよい。そのため、tryVar は現在のコンテキストのリストの長さの情報さえあれば、与えられた goal と単一化ができるかどうか順に試すことができている。

もしこのような assumption の操作をしないと、何が起きるだろうか。例えば、補題の eSubst′ の文言は次のようになっており、補題のほとんどの引数が型パラメータの myvar に依存している。

```
eSubst′ : {myvar : typ → Set} {τ₁ τ₂ : typ}
        {e : myvar (cpsT τ₂) → term[ myvar ∘ cpsT ] τ₁}
        {e′ : term[ myvar ∘ cpsT ] τ₁} {v : value[ myvar ∘ cpsT ] τ₂}
        (k : myvar (cpsT τ₁ ⇒ Nat)) → Subst e v e′ →
        Subst (λ y → cpsEta′ {myvar} (e y) (Var k)) (cpsEtaV v) (cpsEta′ e′ (Var k))
```

このような場合、補題の適用時に引数 myvar を明確に与えてやらないと、補題の文言が型パラメータを含んだ状態のまま確定しないため、仮に補題の型が unify できたとしても、その結果を unquote しようとした際に Agda のコードが一通りに定まらなくなってしまう。

tryVar や assumption では、現在のコンテキストに含まれる変数がメタ変数と unify できるかどうか愚直に試している。そのため、現在のコンテキスト内に同じ型を持つ複数の変数があるときは、どれか 1 つの変数で先に unify されてしまうと他の変数を試すことができず、もし不適切な変数を選んでしまうと証明が行き詰まる可能性がある。今回の証明では、現在のコンテキスト内に同じ型を持つ複数の変数がそれほど現れないためうまく証明できているが、型の情報だけでは適切な変数かどうか判定できない場合に対応するには、バックトラックの機能を実装する必要があるだろう。

本節で定義した unifyGoalDef を図 3 に組み込むには、substTac 関数を次のように変更して、補題を適用できるようにすればよい。以下の 5 〜 7 行目が新たに追加された部分である。

```
1 substTac : Goal → (args : List (Arg Term)) → TC (List Goal)
2 substTac g args with findName args
3 substTac g args | just (quote Val) = unifyGoalCons (quote sVal) g
4 substTac g args | just (quote App) = unifyGoalCons (quote sApp) g
5 substTac g args | just (quote cpsEta′) =
6   catchTC (unifyGoalDef (quote eSubst′) g)
7           (unifyGoalDef (quote kSubst′) g)
8 substTac g args | _ = print "no idea what to do with Subst"
```

同様のことを reduce*Tac についてもおこなうと、図 1 に示した correctEta の証明のうち、これまで取り組んできた RBeta のケース (0 番) を完成させることができる。