Agda による各種コンパイラ最適化プログラムの定式化

門脇香子　(指導教官: 浅井健一)

1　はじめに

Agda はマルティンレフの型理論に基づいた定理証明支援器であり，その一方で依存型をもつ関数型プログラミング言語でもある．その両方の特徴をもちあわせていることから，何かを実装しつつそのプログラムの正当性を証明するのに適している．そこで本稿では，K 正規化をはじめ，ML におけるコンパイル (高水準言語と低水準言語のギャップを埋めること) の際の変換を Agda で実装しつつ，その正当性の証明を試みた．本稿での変換はすべて MinCaml コンパイラ [1] に基づくものである．

2　依存型 (Dependant Types)

Agda は依存型理論 [2] に基づいている．依存型とは，値に依存する型を作ることのできる型のことである．例えば，値に依存する型を用いると，型レベルでサイズが与えられているリストを作ることなどができる．

3　Parametric Higher-Order Abstract Syntax

本稿では，Syntax として Parametric Higher-Order Abstract Syntax (PHOAS)[3] を採用した．PHOAS は Higher-Order Abstract Syntax(HOAS) を変数の表現でパラメータ化したものである．HOAS ではターゲット言語の binding をメタ言語の binding で表しており，例えば Let 文などで自分で変数名を用意する必要がなくなるなどの利点がある．以下に K 正規化の source language と target language を示す．

```
-- Source language
data exp (var : Type → Set) : Type → Set
    where
  SVar : ∀ {A} → var A → exp var A
  SNum : N → exp var Int
  SOp : exp var Int → Operator  → exp var
      Int → exp var Int
  SLam : ∀ {A B} → (var A → exp var B) →
       exp var (Fun A B)
  SApp : ∀ {A B} → exp var (Fun A B) →
      exp var A → exp var B
Exp : Type → Set1
Exp A = ∀ {var} → exp var A

-- Target language
data expt (var : Type → Set) : Type → Set
    where
  TVar : ∀ {A} → var A → expt var A
  TNum : N → expt var Int
  TOp : var Int → Operator → var Int →
      expt var Int
  TLam : ∀ {A B} → (var A → expt var B)
      → expt var (Fun A B)
  TLet : ∀ {A B} → expt var A → (var A ->
       expt var B) → expt var B
  TApp : ∀ {A B} → var (Fun A B) → var A
      → expt var B
Expt : Type → Set1
Expt A = ∀ {var} → expt var A
```

var をパラメータ化することで，論理的な矛盾を防ぎ，Agda で定義可能にしている．

4　K 正規形変換

例えば $a+b-c-d$ のようなネストした式を ML などの言語は一発で計算できるが，アセンブリ等の低水準言語にはそのような命令はない．このような場合のギャップを埋めるのが K 正規形変換である．まず source language と target language を先のように定義する．そして関数 g (K 正規形変換本体) は以下の通りである．

```
g : ∀ {A} → exp var A → expt var A
g (SVar x) = TVar x
g (SNum x) = TNum x
g (SOp b op b1) = TLet (g b) (\x1 → TLet (g
     b1)
    (\x2 → TOp x1 x x2))
g (SLam x) = TLam (\ a → g (x a))
g (SApp b b1) = TLet (g b1) (\x → TLet (g b
    ) (\x1 → TApp x1 x))
```

次に，プログラムの意味を計算する関数を source, target それぞれについて定義する．

```
expdenotet : {A : Type} → expt Value A →
    Value A
expdenotet (TVar x) = x
expdenotet (TNum x) = x
expdenotet (TOp x Plus x2) = x + x2
expdenotet (TOp x Mult x2) = x * x2
expdenotet (TLam f) = \v → expdenotet (f v)
expdenotet (TLet b x) = (\a → expdenotet (x
     a)) (expdenotet b)
expdenotet (TApp x x2) = x x2
Expdenotet : ∀ {A} → Expt A → Value A
Expdenotet E = expdenotet E
```

次に，変換 g の前後でプログラムが変わらないことを関数 lem-k によって示す．ここでは lem-k は Lam 文の同値性の証明に関数の外延性 (任意の a において，f a = g a ならば f = g であること) を必要としている．証明の一部を以下に示す．

```
-- translation correct
postulate
  lem-ext : ∀ {A B : Set} {f g : A → B} →
       (∀ (a : A) → f a ≡ g a) → f ≡ g

lem-k : ∀ {A} → (B : exp Value A) → (
    expdenote B ≡ expdenotet (g B))
lem-k (SVar x) = refl
lem-k (SLam f) = lem-ext (\a → lem-k (f a))
lem-k (SApp f a) =
  begin
    expdenote f (expdenote a)
  ≡⟨ cong (\x → expdenote f x) (lem-k a) ⟩
  (expdenote f) (expdenotet (g a))
  ≡⟨ cong (\x → x (expdenotet (g a))) (lem
      -k f) ⟩
   expdenotet(g f) (expdenotet (g a))
  \qed.
```

例として，Application 文の部分ではまず変換前の denotation である expdenote f (expdenote a) が変換後の denotation である expdenotet(g f) (expdenotet (g a)) と同値であることを示す際，まず expdenote a と expdenotet (g a) は lem-k により同値であることを示し，次に expdenote f と expdenote (g f) が同値であ

ることを順に示していく方式をとった．lem-k が Agda のチェックを通れば，正当性が証明されたことになる．本稿では以降，この方式で他の変換の定式化も行う．

5 β 変換

たとえば let x = y in x + y のような式において，x と y が等しいことは明らかなので，x + y を y + y と置き換えることができる．このような変換を β 変換と言う． MinCaml における β 変換は，一般の β 変換の特殊形であり，以下のように定義される．

$$let\ x = y\ in\ M \quad \triangleright \quad M\,[x := y]$$

β 変換以降は source ，target 言語ともに Expt で書かれる．変換と証明の一部を以下に示す．

```
b : expt var A → expt var A
b (TVar x) = TVar x
b (TLet (TVar x) M) = M x
b (TLet x M) = TLet x M
...
lem-b : ∀ {A} → (B : expt Value A) → (
    expdenotet B ≡ expdenotet (b B))
lem-b (TVar x) = refl
lem-b (TLet (TVar x) y) = refl
...
```

6 η 簡約

MinCaml における η 簡約は，(λx.M) x のような式を M と簡約するものである．一般に η 簡約を行うためには，外延性が成り立つことが必要条件であることが知られている．η 簡約が可能かどうか (その式に対して外延性が成り立つかどうか) を判定する関数 canEta を以下に示す．

```
caneta : ∀ {A}  → expt (\x → Bool) A →
    Bool
caneta (TVar x) = x
caneta (TNum x) = true
caneta (TOp x op x2) = x ∧ x2
caneta (TLam x) = caneta (x true)
caneta (TLet x x2) = caneta (x2 true)
caneta (TApp x x2) = x ∧ x2

canEta :  ∀ {A}  → expt (\x → Bool) A →
    Bool
canEta (TLam x) with x false
canEta (TLam x) | TApp x1 false = x1
canEta (TLam x) | _ = false
canEta _ = false
```

現段階では， η 簡約本体は未実装であり，Expt A 型と expt var A 型との間の同値性が記述できていないことが問題となっている．これは PHOAS を Agda で扱う際の問題点の一つである．

7 不要定義削除

一般に，e1 が自明であり，x が e2 に出現していなければ、let x = e1 in e2 という式を単なる e2 に変換することができる．PHOAS の syntax における let M f の M の部分を捨てることができるということである． この変換を不要定義削除という．この「自明である」という条件の判定を実装したのが以下の関数 t である．ただし,「自明である」かどうかは実際には決定不能なので，この判定は Application 文が含まれるものはすべて非自明にしているなど，暫定的なものである．

```
t : expt var A → Bool
t (TVar x) = true
t (TNum x) = true
t (TOp x op y) = true
t (TLam x) = true
t (TLet x M) = false
t (TApp e1 e2) = false
```

さらに「変数が式に出現する」という条件の判定を実装したものが以下の関数 vars である．

```
vars : expt (\x → Bool) A → Bool
vars (TVar x) = x
vars (TNum x) = false
vars (TOp x op y) = x ∨ y
vars (TLam x) = vars (x false)
vars (TLet x M) = vars x ∨ vars (M false)
vars (TApp e1 e2) = e1 ∨ e2
```

実装の段階において， let M f から M を捨てる際に，f に受け取らせる dummy の値として，PHOAS の var パラメータと任意の A 型から expt var A 型の値を作成する必要がある． しかしながら，Agda の PHOAS において，関数のパラメータ型が任意の場合に整合する式を作るのは容易ではなく， dummy は未実装となっている．例として不要定義削除の関数 R' の補助関数 R'Let を以下に示す．

```
dummy : ∀{A} →  {var : Type → Set} → var
     A
dummy {A} {var} = ?
R'Let : ∀ {A} →  (var1 : Type → Set) → (
    Bool → expt (\x →Bool) A) →
   expt (\x → Bool) A → Expt A →  expt
      var1 A
R'Let var1 f M E with E {var1}
R'Let var1 f M E | TLet M' f' = if t M then
    if vars (f true) then f' (dummy) else E
    else E
R'Let var1 f M E | _ = E
```

8 まとめと今後の課題

本研究では PHOAS の各種コンパイラ最適化変換について Agda で実装と定式化を試みた．PHOAS では束縛変数をメタ言語の binding で表すことによって，変数名の決定や衝突について考慮せずに済む．その結果，K 正規系変換，β 変換などは比較的平易に記述できることがわかった．しかし一方で，PHOAS ではメタ言語の binder の中に直接入っていくことができない．そのため，η 簡約，不要定義削除における定義の一般化が容易ではなく，ここで PHOAS を使う際の問題点が発生している．今後はそのような問題点を解決する手法を見つけ，さらに様々な MinCaml の各種コンパイラ最適化プログラムの定式化を試みていく．

参考文献

[1] Eijiro Sumii, "MinCaml: A Simple and Efficient Compiler for a Minimal Functional Language," *FDPE '05*, 2005.

[2] Ana Bove and Peter Dybjer, "Dependent Types at Work," *International LerNet ALFA Summer School '08*, 2008, Revised Tutorial Lectures.

[3] Adam Chlipala, "Parametric Higher-Order Abstract Syntax for Mechanized Semantics," *ICFP '08*, p143–156, 2008.