Agda コンパイラ—Agate（アガテ）

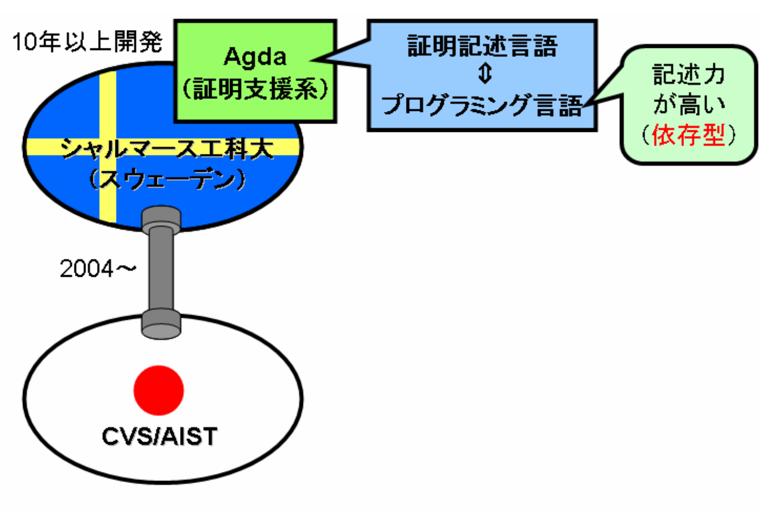
尾崎 弘幸
ozaki@ni.aist.go.jp

2006 年 11 月 27 日
JAIST-AIST Joint Workshop

Table of Contents

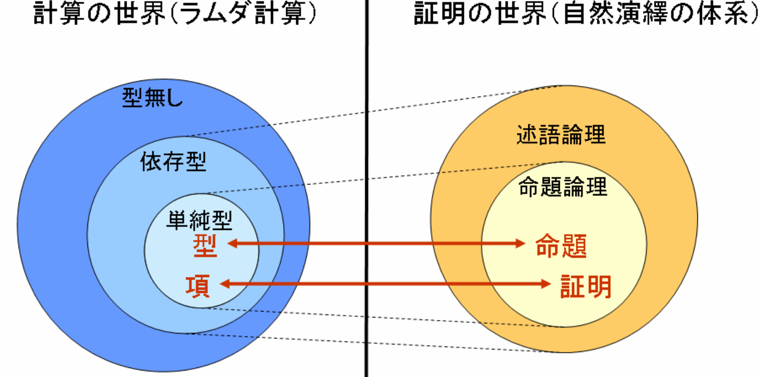
- Agda 言語の紹介
  - Martin-Löf 型理論（今日は紹介しません）に基づく言語
  - ⇒ 「計算」と「証明」を融合できる言語
- Agda コンパイラ Agate（アガテ）の紹介
  - どんなプログラムが書けるの？
    - ⇒ Haskell + α
  - どんな実装なの？
    - ⇒ Higher-Order Abstract Syntax + 型主導 Haskell 関数埋め込み

概要



カリー・ハワード同型対応

型付関数型言語 ⇔ 証明記述言語
型検査 ⇔ 証明検査
型推論 ⇔ 証明支援



コア言語（1/2）

Expressions and Types

e, A	::=	x	(variable)
	|	c	(defined constant)
	|	$\lambda(x : A) . e$	(abstraction)
	|	$e_1 \; e_2$	(application)
	|	$\underline{\text{let}}\; x : A = e_1 \;\underline{\text{in}}\; e_2$	(let binding)
	|	$\underline{\text{struct}}\; \{\ell_1 = e_1; \;\ldots;\; \ell_n = e_n\}$	(dependent record)
	|	$e.\ell$	(projection from structure)
	|	$k \; e_1 \ldots e_n$	(constructor expression)
	|	$\underline{\text{case}}\; e_0 \;\underline{\text{of}}\; \{ (k_1 \; x_{11} \ldots x_{1m_1}) \to e_1; \;\vdots\; (k_n \; x_{n1} \ldots x_{nm_n}) \to e_n\}$	(case expression)
	|	$\underline{\text{Set}}$	(universe of small types)
	|	$(x : A_1) \to A_2[x]$	(dependent function type)
	|	$\underline{\text{sig}}\; \{\ell_1 : A_1; \; \ell_2 : A_2[\ell_1]; \;\ldots;\; \ell_n : A_n[\ell_1, \ldots, \ell_{n-1}]\}$	(dependent record type)

where $n, m_i \geq 0$

コア言語（2/2）

Definitions

def ::= $c : (x_1 : A_1) \to (x_2 : A_2[x_1]) \ldots \to (x_n : A_n[x_1, \ldots, x_{n-1}]) \to A[x_1, \ldots, x_n]$
$c\ x_1 \ldots x_n = e$ (constant definition)
| data $T = k_1\ (x_{11} : A_{11}) \ldots (x_{1m_1} : A_{1m_1})$
$\vdots$
| (data-type definition)
| $k_n\ (x_{n1} : A_{n1}) \ldots (x_{nm_n} : A_{nm_n})$
| postulate $c : (x_1 : A_1) \to (x_2 : A_2[x_1]) \ldots \to (x_n : A_n[x_1, \ldots, x_{n-1}]) \to A[x_1, \ldots, x_n]$
(postulated constants)

where $n, m_i \geq 0$

Propositions as Types

型	命題	証明（項）
`A -> B`	$A \supset B$	A の証明から B の証明を作る関数
`A × B`	$A \wedge B$	A の証明と B の証明のペア
⋮	⋮	
`(x::D) -> P x`	$\forall x \in D.P(x)$	$x \in D$ から $P(x)$ の証明を作る関数
⋮	⋮	

依存型

項に依存する型を**依存型**と呼ぶ．例えば，型 `P x` は項 `x` に依存．

例題

自然数の導入，除去，加算の定義

```
-- Introduction rule for natural numbers
data Nat = zero | succ (n :: Nat)

-- Elimination rule for natural numbers
elimNat :: (P :: Nat -> Set)
        -> P zero
        -> ((m :: Nat) -> P m -> P (succ m))
        -> (n :: Nat) -> P n
elimNat P p_z p_s n
  = case n of
      (zero   )-> p_z
      (succ n')-> p_s n' (elimNat P p_z p_s n')

-- Definition of addition for natural numbers
(+) :: Nat -> Nat -> Nat
m + n = elimNat (\x -> Nat) m (\n' ih -> succ ih) n
```

例題

同値関係

```
idata (==) (A :: Set) :: A -> A -> Set where
    ref (x :: A) :: (==) x x

subst (A :: Set) :: (P :: A -> Set) -> (x, y :: A)
                 -> x == y -> P x -> P y
subst P x y p q = case p of (ref x')-> q
tranId (A :: Set) (x, y, z :: A)
       :: x == y -> y == z -> x == z
tranId p q = subst (\a -> x == a) y z q p
symId (A :: Set) (x, y :: A) :: x == y -> y == x
symId p = subst (\a -> a == x) x y p (ref x)
mapId (A, B :: Set) (x, y :: A)
      :: (f :: A -> B) -> x == y -> f x == f y
mapId f p = subst (\a -> f x == f a) x y p (ref (f x))
```

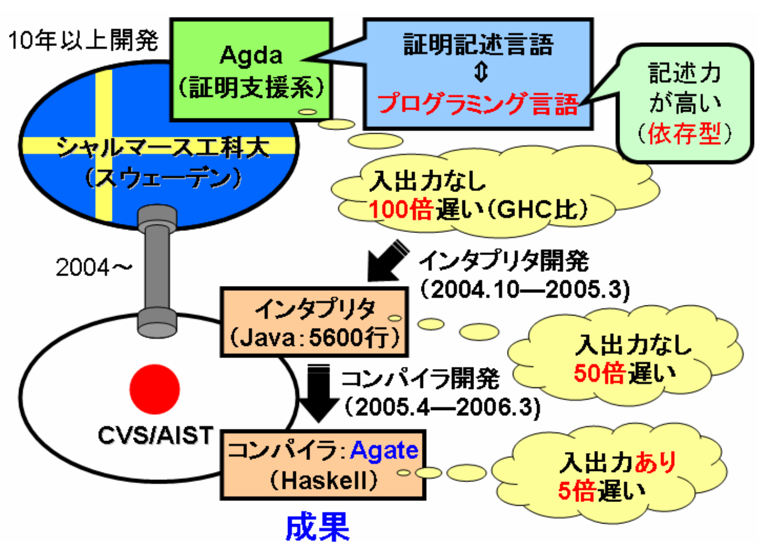
例題

加算の交換律の証明

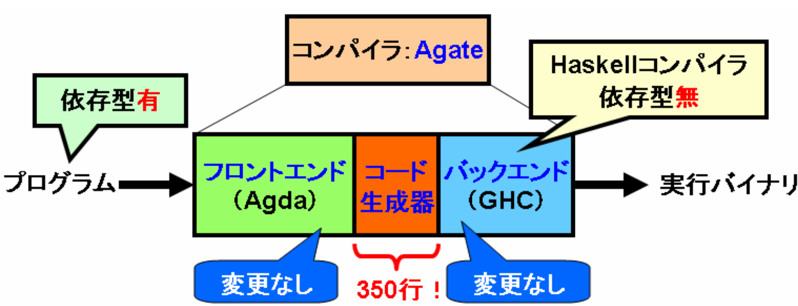
```
lemma :: (m, n :: Nat) -> succ m + n == succ (m + n)
lemma m n = ...

comm :: (m, n :: Nat) -> m + n == n + m
comm m n = elimNat (\x -> m + x == x + m)
                   (elimNat (\y -> y + zero == zero + y)
                            (ref zero)
                            (\m' ih -> mapId succ ih)
                            m)
                   (\m' ih -> tranId
                               (mapId succ ih)
                               (symId (lemma m' m)))
                   n
```

概要



構造



- 理論に基づく実装（Higher-Order Abstract Syntax）
- ソフトウェア工学面（保守性，再利用性，可読性）
- 機能追加（Haskell ライブラリを利用）

（例） 標準入出力，ファイル操作，・・・
ネットワークプログラミングや GUI プログラミング（未実装）

- パフォーマンス

Higher-Order Abstract Syntax

型付関数型言語で型無しラムダ計算をエンコードする方法

```
data Val  =  VAbs (Val -> Val)        -- λx.e
          |  VCon String [Val]        -- k e1 ... en
          |  VStr [(String, Val)]     -- struct{ℓ1 = e1; ... ; ℓn = en}
          |  VType                    -- (any term of type Set)
apply (VAbs f) v = f v
select (VStr bs) x = fromJust(lookup x bs)
```

Comments: $\lambda x.e$ / $k\ e_1\ \ldots\ e_n$ / struct$\{\ell_1 = e_1; \ldots ; \ell_n = e_n\}$ / (any term of type Set)

エンコード例

$\lambda x.x\ x$

→ `VAbs (\x -> apply x x)`

機能追加（1）

Haskell と同じアプローチで IO を扱う（i.e. モナド）

Agda での echo プログラム

```
class Monad (M : Set → Set) exports
  (≫=) (A : Set) (B : Set) : M A → (A → M B) → M B
  return (A : Set) : A → M A

postulate IO : Set → Set
postulate (| ≫= |) (A : Set) (B : Set) : IO A → (A → IO B) → IO B
postulate ret (A : Set) : A → IO A
instance IOMonad : Monad IO where
  (≫=) = (| ≫= |)
  return = ret
postulate Unit : Set
postulate putStr : String → IO Unit
postulate getLine : IO String

main : IO Unit
main = getLine ≫= putStr
```

機能追加（2）

Haskell 関数を埋め込めるようにユニバーサルタイプ `Val` を拡張

```
data Val  =  ...
          |  VIO (IO Val)       -- (value of IO α)
          |  VString String     -- (string literal)
          |  VUnit              -- (value of Unit)
deIO (VIO m)          =  m
deString (VString s)  =  s
deUnit VUnit          =  ()
```

機能追加（3）

postulate 宣言に意味を与える

$[\![\text{postulate } \mathit{putStr} : \mathit{String} \to \mathit{IO}\ \mathit{Unit}]\!]$

$$= \left(\texttt{x_putStr = VAbs(\textbackslash v->}\ \ \texttt{VIO(putStr(deString v) >> return VUnit))} \right)$$

$[\![\text{postulate } \mathit{getLine} : \mathit{IO}\ \mathit{String}]\!]$

$$= \left(\texttt{x_getLine = VIO(fmap VString getLine)} \right)$$

$[\![\text{postulate } (|\ \gg=\ |)\ (A : \underline{\text{Set}})\ (B : \underline{\text{Set}}) : \mathit{IO}\ A \to (A \to \mathit{IO}\ B) \to \mathit{IO}\ B]\!]$

$$= \left(\texttt{(|>>=|) = VAbs(\textbackslash a->VAbs(\textbackslash b->VAbs(\textbackslash m->VAbs(\textbackslash f->}\ \ \texttt{VIO(deIO m >>= deIO . apply f)))))} \right)$$

$[\![\text{postulate } \mathit{ret}\ (A : \underline{\text{Set}}) : A \to \mathit{IO}\ A]\!]$

$$= \left(\texttt{x_ret = VAbs(\textbackslash a->VAbs(\textbackslash v->VIO(return v)))} \right)$$

型主導の Haskell 関数の埋め込み

Haskell のクラス機構を利用した Haskell 関数の埋め込み

- Haskell の型と対応付けに `Val` の拡張が必要
- 基本型の埋め込みは人手による記述が必要

```
data Val   =  ...
           |  VChar Char
           |  VList ([Val])
deChar (VChar c) = c
deList (VList l) = l
```

$$
\begin{aligned}
\uparrow_{\mathtt{Char}}(c) &= \mathtt{VChar}\ c & \downarrow_{\mathtt{Char}}(v) &= \mathtt{deChar}\ v\\
\uparrow_{\mathtt{a->b}}(f) &= \mathtt{VAbs(\backslash v\text{->}}\ \uparrow_{\mathtt{b}}(f(\downarrow_{\mathtt{a}}\mathtt{v})))) & \downarrow_{\mathtt{a->b}}(v) &= \lambda x\ .\ \downarrow_{\mathtt{b}}(\mathtt{apply}\ v\ (\uparrow_{\mathtt{a}}\ x))\\
\uparrow_{\mathtt{[a]}}(l) &= \mathtt{VList(fmap}\ \uparrow_{\mathtt{a}}\ l) & \downarrow_{\mathtt{[a]}}(v) &= \mathtt{fmap}\ \downarrow_{\mathtt{a}}\ (\mathtt{deList}\ v)\\
\uparrow_{\mathtt{(IO\ a)}}(x) &= \mathtt{VIO(fmap}\ \uparrow_{\mathtt{a}}\ x) & \downarrow_{\mathtt{(IO\ a)}}(v) &= \mathtt{fmap}\ \downarrow_{\mathtt{a}}\ (\mathtt{deIO}\ v)\\
\uparrow_{()}(x) &= \mathtt{VUnit} & \downarrow_{()}(v) &= \mathtt{deUnit}\ v
\end{aligned}
$$

$\uparrow_{\mathtt{[Char]->IO\ ()}}$ `putStr =`
`VAbs(\v->VIO(putStr(deString v) >> return VUnit))`

パフォーマンス

GHC と比べ遜色のない性能

- 階乗関数

	fact 8	fact 9	fact 10	fact 11
GHC (sec)	0.047	0.199	1.515	9.952
Agate (sec)	0.063	0.408	3.430	29.746
ratio	1.346	2.055	2.263	2.989

- アッカーマン関数

	ack 3 8	ack 3 9	ack 3 10	ack 3 11
GHC (sec)	0.177	0.564	1.934	7.508
Agate (sec)	0.672	2.384	9.364	39.239
ratio	3.807	4.228	4.841	5.226

依存型プログラミング

同じ長さのリストのみ許す zip 関数（zipVec）

```
zipVec (A,B::Set)
        :: (n::Nat)
        -> Vec A n
        -> Vec B n
        -> Vec (A × B) n
zipVec n as bs
   = case n of
     (zero   )-> unit
     (succ n')-> ((fst as, fst bs),
                  zipVec n' (snd as) (snd bs))
```

⇒ デモにつづく

ダウンロード

Agda

http://agda.sourceforge.net/

Windows XP, MacOSX, Linux x86 **用バイナリパッケージ**

Agate

http://staff.aist.go.jp/hiroyuki.ozaki/

ソースコード．GHC6.4 **が必要．**