

Title	Agdaコンパイラ—Agate(アガテ)
Author(s)	尾崎, 弘幸
Citation	
Issue Date	2006-11-27
Type	Presentation
Text version	publisher
URL	<a href="http://hdl.handle.net/10119/8309">http://hdl.handle.net/10119/8309</a>
Rights	
Description	3rd VERITE : JAIST/TRUST-AIST/CVS joint workshop on VERification TEchnologyでの発表資料, 開催 : 2006年11月27日 ~ 28日, 開催場所 : JAIST 知識科学研究科講義棟・中講義室

# Agda コンパイラーAgate (アガテ)

尾崎 弘幸

**ozaki@ni.aist.go.jp**

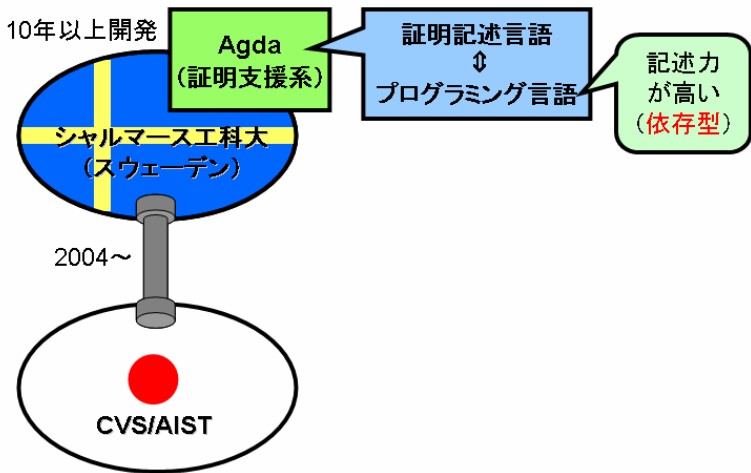
 独立行政法人 産業技術総合研究所 システム検証研究センター 

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 +  $\alpha$
  - どんな実装なの？
    - ⇒ Higher-Order Abstract Syntax + 型主導 Haskell 関数埋め込み

## 概要



# カーリー・ハワード同型対応

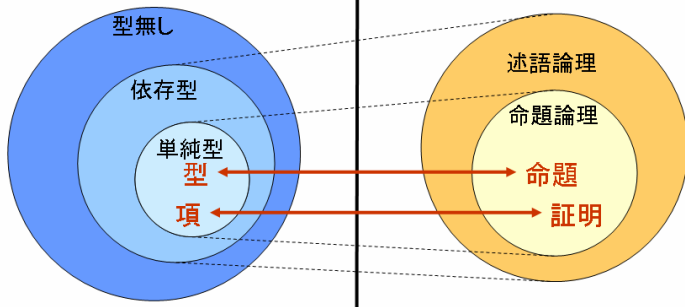
型付関数型言語  $\leftrightarrow$  証明記述言語

型検査  $\leftrightarrow$  証明検査

型推論  $\leftrightarrow$  証明支援

計算の世界(ラムダ計算)

証明の世界(自然演繹の体系)



## 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)
$\text{let } x: A = e_1 \text{ in } e_2$	(let binding)
$\text{struct } \{ \ell_1 = e_1; \dots; \ell_n = e_n \}$	(dependent record)
$e.l$	(projection from structure)
$k e_1 \dots e_n$	(constructor expression)
$\text{case } e_0 \text{ of } \{$	(case expression)
$(k_1 x_{11} \dots x_{1m_1}) \rightarrow e_1;$	
$\vdots$	
$(k_n x_{n1} \dots x_{nm_n}) \rightarrow e_n \}$	
$\text{Set}$	(universe of small types)
$(x: A_1) \rightarrow A_2[x]$	(dependent function type)
$\text{sig } \{ \ell_1: A_1; \ell_2: A_2[\ell_1]; \dots; \ell_n: A_n[\ell_1, \dots, \ell_{n-1}] \}$	(dependent record type)

where  $n, m_i \geq 0$

## コア言語 ( 2 / 2 )

## Definitions

```

def ::= c : (x1 : A1) → (x2 : A2[x1]) ... → (xn : An[x1, ..., xn-1]) → A[x1, ..., xn]
      c x1 ... xn = e                                     (constant definition)
|     data T = k1 (x11 : A11) ... (x1m1 : A1m1)
      |
      |           ⋮
      |           | kn (xn1 : An1) ... (xnmn : Anmn)           (data-type definition)
|     postulate c : (x1 : A1) → (x2 : A2[x1]) ... → (xn : An[x1, ..., xn-1]) → A[x1, ..., xn]
      (postulated constants)

```

where  $n, m_i \geq 0$

# Propositions as Types

型	命題	証明 (項)
$A \rightarrow B$	$A \supset B$	$A$ の証明から $B$ の証明を作る関数
$A \times B$	$A \wedge B$	$A$ の証明と $B$ の証明のペア
$\vdots$	$\vdots$	
$(x :: D) \rightarrow P\ x$	$\forall x \in D. P(x)$	$x \in D$ から $P(x)$ の証明を作る関数
$\vdots$	$\vdots$	

## 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))

```

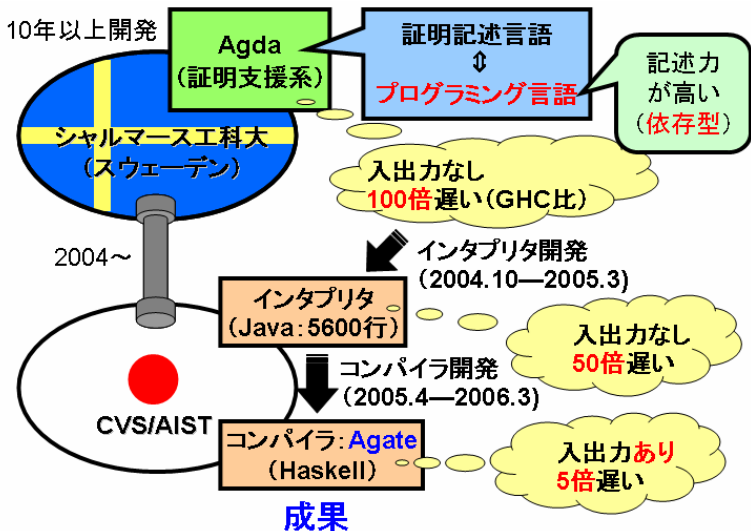
## 例題

## 加算の交換律の証明

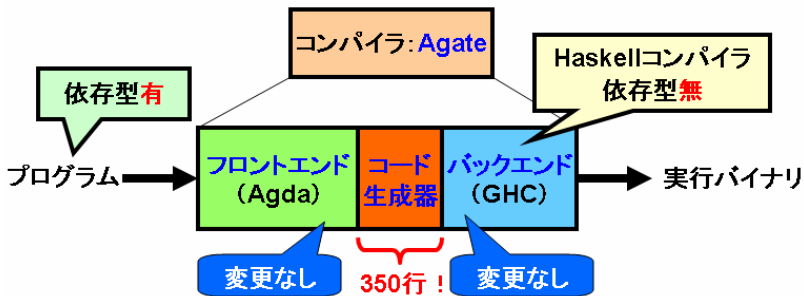
```
lemma :: (m, n :: Nat) -> succ m + n == succ (m + n)
lemma m n = ...
```

```
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)      --  $\lambda x.e$ 
        | VCon String [Val]      --  $k e_1 \dots e_n$ 
        | VStr [(String, Val)]   -- struct{ $\ell_1 = e_1; \dots; \ell_n = e_n$ }
        | VType                  -- (any term of type Set)

apply (VAbs f) v = f v
select (VStr bs) x = fromJust(lookup x bs)

```

## 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  $\alpha$ )
         | 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

$$\overline{\llbracket \text{postulate } \textit{putStr} : \textit{String} \rightarrow \textit{IO Unit} \rrbracket}} = \left( \begin{array}{l} \text{x\_putStr} = \text{VAbs}(\backslash \text{v} \rightarrow \\ \text{VIO}(\text{putStr}(\text{deString } \text{v}) \gg \text{return } \text{VUnit})) \end{array} \right)$$

$$\overline{\llbracket \text{postulate } \textit{getLine} : \textit{IO String} \rrbracket}} = \left( \text{x\_getLine} = \text{VIO}(\text{fmap } \text{VString } \text{getLine}) \right)$$

$$\overline{\llbracket \text{postulate } (| \gg= |) (A : \textit{Set}) (B : \textit{Set}) : \textit{IO A} \rightarrow (A \rightarrow \textit{IO B}) \rightarrow \textit{IO B} \rrbracket}} = \left( \begin{array}{l} (| \gg= |) = \text{VAbs}(\backslash \text{a} \rightarrow \text{VAbs}(\backslash \text{b} \rightarrow \text{VAbs}(\backslash \text{m} \rightarrow \text{VAbs}(\backslash \text{f} \rightarrow \\ \text{VIO}(\text{deIO } \text{m} \gg= \text{deIO } . \text{apply } \text{f})))))) \end{array} \right)$$

$$\overline{\llbracket \text{postulate } \textit{ret} (A : \textit{Set}) : A \rightarrow \textit{IO A} \rrbracket}} = \left( \text{x\_ret} = \text{VAbs}(\backslash \text{a} \rightarrow \text{VAbs}(\backslash \text{v} \rightarrow \text{VIO}(\text{return } \text{v}))) \right)$$

# 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
```

$\uparrow_{\text{Char}} (c)$	$=$	$\text{VChar } c$	$\downarrow_{\text{Char}} (v)$	$=$	$\text{deChar } v$
$\uparrow_{a \rightarrow b} (f)$	$=$	$\text{VAbs}(\backslash v \rightarrow \uparrow_b (f(\downarrow_a v)))$	$\downarrow_{a \rightarrow b} (v)$	$=$	$\lambda x . \downarrow_b (\text{apply } v (\uparrow_a x))$
$\uparrow_{[a]} (l)$	$=$	$\text{VList}(\text{fmap } \uparrow_a l)$	$\downarrow_{[a]} (v)$	$=$	$\text{fmap } \downarrow_a (\text{deList } v)$
$\uparrow_{(\text{IO } a)} (x)$	$=$	$\text{VIO}(\text{fmap } \uparrow_a x)$	$\downarrow_{(\text{IO } a)} (v)$	$=$	$\text{fmap } \downarrow_a (\text{deIO } v)$
$\uparrow_{()} (x)$	$=$	$\text{VUnit}$	$\downarrow_{()} (v)$	$=$	$\text{deUnit } v$

```
 $\uparrow_{[\text{Char}] \rightarrow \text{IO } ()}$  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
<b>ratio</b>	<b>1.346</b>	<b>2.055</b>	<b>2.263</b>	<b>2.989</b>

- 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
<b>ratio</b>	<b>3.807</b>	<b>4.228</b>	<b>4.841</b>	<b>5.226</b>

## 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 .