Agate –an Agda-to-Haskell compiler

Hiroyuki Ozaki,　Makoto Takeyama,

Yoshiki Kinoshita

Research Center for Verification and Semantics (CVS),
National Institute of Advanced Industrial Science and
Technology (AIST)

Agate –an Agda-to-Haskell compiler

Hiroyuki Ozaki, Makoto Takeyama and Yoshiki Kinoshita

Research Center for Verification and Semantics(CVS),
National Institute of Advanced Industrial Science and Technology (AIST)
{ozaki@ni., makoto.takeyama@, yoshiki@ni.}aist.go.jp

Abstract. We report some features of Agate, a compiler for the dependently typed functional language of Agda proof-assistant. Agate is developed to be an experimental platform for practice of dependently typed programming and it extends Agda language with I/O facilities and calls to Haskell functions. The first feature is the use of higher order abstract syntax to translate terms of Agda language into untyped λ-calculus encoded in Haskell. The second is the application of Haskell class mechanism to embed typed Haskell terms in the universal type for untyped λ-calculus. This approach makes Agate very lightweight. Performance report of some codes emitted by Agate is given.

1 Introduction

Agda[1] is an interactive proof assistant based on Martin-Löf Type Theory (MLTT)[2]. Its input language, *Agda language* (also called Agda for short), extends MLTT with several programming language features such as record types, classes and hidden arguments. Some aspects of the language features was reported by Coquand and Coquand[3].

We developed *Agate*, a compiler that translates Agda into Haskell. Its aim is to be a basis for experiments on the practice of dependently typed programming. For this, we need execution speed much higher than that of the interpreter built in Agda proof-assistant. Agda language also needs to be extended for practical programming, for example, with Input/Output facilities.

This paper reports some features of Agate. The first is the use of higher order abstract syntax to translate terms of MLTT into terms of untyped λ-calculus. This solves the problem that dependently typed Agda terms are in general not well-typed in Haskell type system. The second is the application of Haskell class mechanism to systematically embed Haskell typed terms into the universal type of untyped λ-terms. This makes it automatic to extend Agate by calling out to Haskell functions.

Agate is very lightweight: it is implemented as ca. 350 lines of Haskell code. A lot of thought was put into making the code short. Compactness of course contributes to the higher maintainability of the compiler.

The paper is organised as follows: Section 2 introduces the source language Agda, explains the representation of untyped λ-calculus in Haskell, and gives the compilation scheme from Agda to untyped λ-calculus. Section 3 shows how

Agate extends Agda language with I/O primitives taken from Haskell. Section 4 explains the use of Haskell class mechanism to automatically give such extensions and its limitation. We show some performance figure for Agate comparing it with GHC in Section 5. Section 6 concludes with a summary and some directions for future work.

Availability Agate works on Linux, Windows, MacOSX. More precisely, it is available on any operating-systems supported by GHC. Agate is open source and its source code is available at

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

1.1 The structure of Agate implementation

One of our goals was to cut down costs of implementation and maintenance for the development of Agate. For that purpose, we decided to reuse the parser and the type-checker of Agda proof-assistant as the front-end of Agate and Glasgow Haskell Compiler(GHC)[4] as the back-end of Agate as shown in Figure 1. Agate use both without any modification.

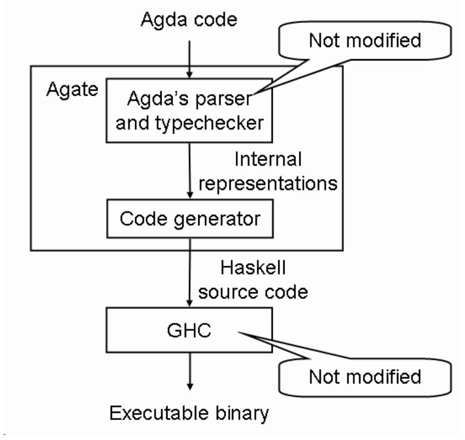

Fig. 1. The overall structure of Agate

So, the middle part translates (internal representation of) Agda code to Haskell source. This part must transform dependently typed Agda code to one that is well-typed in Haskell. As explained in the next section, the transformation is basically from dependently typed λ-calculus to untyped λ-calculus, and we use the technique of Higher-Order Abstract Syntax (HOAS)[5] to embed untyped λ-calculus in Haskell.

1.2 Related work

The main advantage of Agate is its lightweight implementation. We compare the compilation scheme of Agate with some more sophisticate compilers in literature for dependently typed languages.

Cayenne[6] is a compiler of the dependently typed functional language Cayenne, which is quite similar to Agda. The older version used Lazy ML as the intermediate language, which is compiled by LML compiler with the type-checking turned off. The latest version does effectively the same with GHC using `unsafeCoerce` feature of GHC Haskell. This strategy cannot expect type-based optimisation by the back-ends to work well and also cannot be extended to deal with open-reduction (reduction of terms with free variables). Cayenne's translations from Cayenne to Haskell is more sophisticated than Agate's in that it translates away certain type arguments not relevant for computations. However, Cayenne type system's universe hierarchy that supports this is incompatible with that of Agda.

Coq[7] has commands for program extraction[8] that strips off from dependently typed terms the part that only concerns proofs and translates the result to Objective Caml, Haskell, or Scheme. This again uses a feature of Coq's type system not present in Agda (distinction of set and prop). In Coq, dependently typed terms are also internally compiled to byte code that is run on Objective Caml abstract machine modified for open reduction[9].

Epigram[10] is dependently typed language based on MLTT with a novel pattern-matching high-level notation for function definitions. Edwin Brady's thesis[11] details its elaboration to conventional Type Theory that is compiled to abstract machine code through several intermediate languages. The compilation scheme output highly optimised code utilising both the detailed type information in source programs and the low level nature of target programs. However its implementation is a large undertaking and a full version seems to be still under development.

DML[12] is an extension of ML with a form of dependent types where indexing types are restricted to some constraint domains. Type-checking requires (automatic) theorem-proving. The compiler for a dialect of DML, de Caml, essentially is a extension of Caml-light compiler which supports the modified front-end[13]. But, it has no documentation.

Our compilation approach using the universal type of untyped λ-terms makes the implementation easy; in particular, there is no need to change back-end. Another advantage, though unimplemented, is that open reduction can be naturally supported by adding neutral terms in the universal type.

2 Translation to untyped λ-calculus

In this section, we briefly introduce the sub-language of Agda for exposition and describe Agate's compilation scheme from the sub-language to Haskell, though the actual Agate implementation compiles full Agda language.

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

$$
\begin{array}{lll}
\mathit{def} ::= & c\colon (x_1\colon A_1) \to (x_2\colon A_2[x_1]) \ldots \to (x_n\colon A_n[x_1, \ldots, x_{n-1}]) \to A & \\
 & c\ x_1 \ldots x_n = e & \text{(constant definition)} \\
 & \mid \underline{\mathsf{data}}\ T = k_1\ (x_{11}\colon A_{11}) \ldots (x_{1m_1}\colon A_{1m_1}) & \\
 & \qquad\quad \underline{\mid}\quad \vdots & \text{(data-type definition)} \\
 & \qquad\quad \underline{\mid}\ k_n\ (x_{n1}\colon A_{n1}) \ldots (x_{nm_n}\colon A_{nm_n}) & \\
 & \mid \underline{\mathsf{postulate}}\ c\colon (x_1\colon A_1) \to (x_2\colon A_2[x_1]) \ldots \to (x_n\colon A_n[x_1, \ldots, x_{n-1}]) \to A & \\
 & & \text{(postulated constants)}
\end{array}
$$

where $n, m_i \geq 0$

Fig. 2. Agda language

2.1 Agda language

Agda language is a version of MLTT extended with some programming language features. It has dependent product types, dependent record types[14] that generalises dependent sum types, inductively defined types and the type universe Set. There are forms of terms according to these kinds of types. These logical language features are extended with packages (a module mechanism), classes and hidden arguments. Classes and hidden arguments are briefly described in Section 3.1. Although Agate supports translations of packages, we do not explain them in this paper.

Figure 2 describes the sub-language we consider. Agda programs are sequences of definitions. Postulated constants are newly created (primitive) values of the given types, without actual definitions.

2.2 The universal type Val

The universal type `Val` in Haskell implements the untyped λ-calculus. λ-terms are the Haskell terms of type `Val`. Functions `apply` and `select` manipulate λ-terms.

```
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)
```

A value `VAbs (\x -> e)` represents a λ-abstraction $\lambda x.e$. The Haskell function inside `VAbs` corresponds to the mapping $t \mapsto e[t/x]$ on λ-terms t. For instance, the untyped λ-term $\lambda x.x\ x$ is represented by the following value of `Val`.

```
VAbs (\x -> apply x x)
```

`VCon` tag indicates data constructors. The value `VType` represents any Agda term of type Set. Distinctions among those Agda terms, including dependent types, are all lost in the translation.

Values with `VStr` tag are struct$\{\ldots\}$ terms. A pair `(`ℓ`, `e`)` in `VStr [...,(`ℓ`, `e`), ...]` represents a field $\ell = e$ in a record. The projection operator (dot operator) searches for a given label from a given list and returns its value. The use of `fromJust` is justified since Agate translates only well-typed Agda terms, for which looking up a field never fails.

2.3 Compilation

We now show how programs in our sub-language would be compiled. Figure 3 explains how Agate compiles Agda expressions to Haskell. We write the translation algorithm as the map $[\![-]\!]$ sending Agda terms to Haskell terms.

For variables and constants, $[\![y]\!]$ adds the prefix `x_` to create a Haskell identifier; i.e., for Agda identifier *Foo*, $[\![Foo]\!]$ is `x_Foo`. (This is needed since Agda allows variable identifiers starting with capital letters while Haskell does not.)

Abstractions are compiled using the feature of HOAS. To compile $\lambda(x\colon A)\ .\ e$, we first compile e to get $[\![e]\!]$. In it, occurrences of the variable x is translated to $[\![x]\!]$, so the mapping $t \mapsto e[t/x]$ is given by the Haskell function `\`$[\![x]\!]$ `->` $[\![e]\!]$. The constructor `VAbs` makes this a value in `Val`. The compilation of Agda let expressions uses the same feature of representing Agda variables by Haskell variables.

The translation scheme for dependent record struct$\{x_1 = e_1; \ldots\ x_n = e_n\}$ also uses that feature, this time representing Agda field labels by Haskell variables. The translation should produce the list of pairs of field label strings and values, but the simple construction `VStr[("`x_1`",`$[\![e_1]\!]$`),...,("`x_n`",`$[\![e_n]\!]$`)]` does not suffice. This is because, in Agda's scoping rule for dependent records, the

$$
\begin{array}{ll}
\llbracket y \rrbracket & = \texttt{x\_}y \\
\llbracket c \rrbracket & = \texttt{x\_}c \\
\llbracket \lambda x{:}\, A \,.\, e \rrbracket & = \texttt{VAbs(\textbackslash}\llbracket x \rrbracket\ \texttt{->}\ \llbracket e \rrbracket\texttt{)} \\
\llbracket e_1\ e_2 \rrbracket & = \texttt{apply}\ \llbracket e_1 \rrbracket\ \llbracket e_2 \rrbracket \\
\llbracket \underline{\mathsf{let}}\ x{:}\, A = e_1\ \underline{\mathsf{in}}\ e_2 \rrbracket & = \texttt{let}\ \llbracket x \rrbracket\texttt{=}\llbracket e_1 \rrbracket\ \texttt{in}\ \llbracket e_2 \rrbracket \\
\llbracket \underline{\mathsf{struct}}\ \{x_1 = e_1; \ldots; x_n = e_n\} \rrbracket & = \texttt{let}\ \{\llbracket x_1 \rrbracket\texttt{=}\llbracket e_1 \rrbracket\texttt{;}\ \ldots\ \texttt{;}\llbracket x_n \rrbracket\texttt{=}\llbracket e_n \rrbracket\} \\
 & \quad \texttt{in VStr[("}x_1\texttt{",}\llbracket x_1 \rrbracket\texttt{)},\ldots\texttt{,("}x_n\texttt{",}\llbracket x_n \rrbracket\texttt{)]} \\
\llbracket k\ e_1 \ldots e_n \rrbracket & = \texttt{VCon "}k\texttt{" [}\llbracket e_1 \rrbracket\texttt{,}\ldots\texttt{,}\llbracket e_n \rrbracket\texttt{]} \\
\llbracket e.\ell \rrbracket & = \texttt{select}\ \llbracket e \rrbracket\ \texttt{"}\ell\texttt{"} \\
\left\llbracket \begin{array}{l} \underline{\mathsf{case}}\ e_0\ \underline{\mathsf{of}}\ \{ \\ \quad (k_1\ x_{11} \ldots x_{1m_1}) \to e_1; \\ \qquad\quad \vdots \\ \quad (k_n\ x_{n1} \ldots x_{nm_n}) \to e_n\} \end{array} \right\rrbracket & = \begin{array}{l} \texttt{case}\ \llbracket e_0 \rrbracket\ \texttt{of \{} \\ \quad \texttt{VCon "}k_1\texttt{" [}\llbracket x_{11} \rrbracket\texttt{,}\ldots\texttt{,}\llbracket x_{1m_1} \rrbracket\texttt{]->}\llbracket e_1 \rrbracket\texttt{;} \\ \qquad\quad \vdots \\ \quad \texttt{VCon "}k_n\texttt{" [}\llbracket x_{n1} \rrbracket\texttt{,}\ldots\texttt{,}\llbracket x_{nm_n} \rrbracket\texttt{]->}\llbracket e_n \rrbracket\texttt{\}} \end{array} \\
\llbracket \underline{\mathsf{Set}} \rrbracket & = \texttt{VType} \\
\llbracket (x{:}\, A_1) \to A_2[x] \rrbracket & = \texttt{VType} \\
\llbracket \underline{\mathsf{sig}}\ \{\ell_1{:}\, A_1; \ldots; \ell_n{:}\, A_n\} \rrbracket & = \texttt{VType}
\end{array}
$$

Fig. 3. The compilation of expressions

label x_i of a preceding field can occur in the expression e_j in a later field $(i < j)$. For example, *inc* in the second field of

$$\underline{\mathsf{struct}}\ \{\ inc = \lambda(x{:}\, Nat)\ .\ succ\ x;\ n = inc\ zero\}$$

refers to the function given in the first field. Translating this naively results in non-well-scoped Haskell term

```
VStr [("inc", VAbs (\x_x -> apply x_succ x_x)), ("n", apply x_inc x_zero)]
```

where variable `x_inc` is unbound. To avoid this, the translation first binds each field label $\llbracket x_i \rrbracket$ to $\llbracket e_i \rrbracket$ in Haskell `let` declarations and uses these to give pairs of field label strings and values.

Agda type expressions are all compiled to just `VType`. Computation of translated Agda programs does not depend on type values, though it still needs to be there as dummy arguments for explicitly polymorphic functions.

Secondly, we explain how to compile definitions of Agda language. Figure 4 describes the compilation scheme of definitions. Defined constants in Agda are simply translated to corresponding declarations in Haskell. (We do not allow postulated constants in Agda programs, which cannot be computed with in general, except for certain specific ones explained in Section 3) For data-type definitions, the type itself is translated to `VType`. For constructors, Agate adds definitions for the corresponding functions that return fully applied constructor expressions.

For example, Figure 5 shows the definitions of the data type of natural numbers and addition operator on it in Agda language and its compiled Haskell code (we write $A_1 \to A_2$ for $(x{:}\, A_1) \to A_2[x]$ in Agda when $A_2[x]$ has no free occurrences of x).

$$\left[\!\!\left[\begin{array}{l} c\colon (x_1\colon A_1) \to (x_2\colon A_2[x_1]) \to \ldots \to (x_n\colon A_n[x_1,\ldots x_{n-1}]) \to A \\ c\,x_1\ldots x_n = e \end{array}\right]\!\!\right]$$

$$= \left([\![c]\!] \ \texttt{=}\ [\![\lambda(x_1\colon A_1)\ldots\lambda(x_n\colon A_n)\ .\ e]\!] \right)$$

$$\left[\!\!\left[\begin{array}{l} \underline{\textsf{data}}\ T = k_1\ (x_{11}\colon A_{11})\ldots(x_{1m_1}\colon A_{1m_1}) \\ \qquad\quad \underline{|}\quad \vdots \\ \qquad\quad \underline{|}\ \ k_n\ (x_{n1}\colon A_{n1})\ldots(x_{nm_n}\colon A_{nm_n}) \end{array}\right]\!\!\right]$$

$$= \left(\begin{array}{l} [\![T]\!] \ \texttt{= VType} \\ [\![k_1]\!] \ \texttt{=}\ [\![\lambda(x_{11}\colon A_{11})\ldots\lambda(x_{1m_1}\colon A_{1m_1})\ .\ k_1\ x_{11}\ldots x_{1m_1}]\!] \\ \quad\vdots \\ [\![k_n]\!] \ \texttt{=}\ [\![\lambda(x_{n1}\colon A_{11})\ldots\lambda(x_{nm_n}\colon A_{nm_n})\ .\ k_1\ x_{n1}\ldots x_{nm_n}]\!] \end{array} \right)$$

Fig. 4. The compilation of definitions

3 Agda language extension by Haskell I/O primitives

We extend Agda language so that it can deal with I/O facilities. The way we do so is through IO monad, similarly to Haskell. In this section, we first explain Agda's class mechanism for implementing IO monad in Agda and how class and instance declarations are reduced to the sub-language. Secondly, we extend the semantics of certain specific postulated constants in order to deal with I/O facilities.

3.1 Classes and instances

Classes and *instances* in Agda serve the same purpose as in Haskell, that is, they allow expressing ad-hoc polymorphism. The internal mechanism realising it is also similar to dictionary passing in Haskell. So, a class declaration is internally treated as a record type definition (and selector functions), and an instance declaration as a record definition. Writing $\{\![-]\!\}$ for the map that reduces class and instance declarations in the full Agda language to the corresponding ones in the sub-language, Figure 6 shows an example reduction of *Show* class and its instance for natural numbers.

We briefly remark that the first two arguments to the function *show*, i.e., type D and dictionary m, are *hidden arguments*. This is a feature of Agda language that applies to arguments bound to the left of :, which allows them to be omitted in application expressions. So when using *show* function, one can write *show* d instead of *show* D m d and let D and m be inferred at type-checking time.

The above translation of class and instance declarations is actually done by the Agda front-end. Agate simply compiles the resulting record / record type definitions as explained in the previous section.

```
data Nat = zero | succ (n: Nat)
plus: Nat → Nat → Nat
plus m n = case m of {(zero) → n; (succ m′) → succ (plus m′ n)}

x_Nat  = VType
x_zero = VCon "zero" []
x_succ = VAbs(\x_n -> VCon "succ" [x_n])
x_plus = VAbs(\x_m -> VAbs(\x_n ->
           case x_m of {
             VCon "zero" [] -> x_n;
             VCon "succ" [x_m']->
               apply x_succ (apply (apply x_plus x_m') x_n)}))
```

Fig. 5. Agda code and its compiled Haskell code

3.2 Extensions with I/O facilities

The declaration of the *Monad* class using Agda's class mechanism is the same as in Haskell.

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

The IO monad instance is merely postulated in Agda. More precisely, we introduce the names of IO type constructor, bind (| ≫= |) and return (*ret*) operations using Agda's postulate declaration with appropriate typing but without definition bodies; the IO instance is defined to be those postulated constants. Other names for primitive I/O operations such as *putStr*, *getLine*, etc. are similarly postulated.

```
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
```

With these declarations, one can write and type-check Agda programs with I/O facilities. For instance, the following Agda program, which is an echo program, is type-correct.

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

$$\left\{\left\lceil \begin{array}{l} \underline{\mathsf{class}}\ \mathit{Show}\ (D\colon \underline{\mathsf{Set}})\ \underline{\mathsf{exports}} \\ \quad \mathit{show}\colon D \to \mathit{String} \end{array} \right\rceil\right\}$$

$$= \left(\begin{array}{l} \mathit{Show}\colon \underline{\mathsf{Set}} \to \underline{\mathsf{Set}} \\ \mathit{Show}\ D = \underline{\mathsf{sig}}\ \{\mathit{show}\colon D \to \mathit{String}\} \\ \mathit{show}\ (D\colon \underline{\mathsf{Set}})\ (m\colon \mathit{Show}\ D) : D \to \mathit{String} \\ \mathit{show}\ d = (m.\mathit{show})\ d \end{array} \right)$$

$$\left\{\left\lceil \begin{array}{l} \underline{\mathsf{instance}}\ \mathit{showNat}\colon \mathit{Show}\ \mathit{Nat}\ \underline{\mathsf{where}} \\ \quad \mathit{show}\ n = \underline{\mathsf{case}}\ n\ \underline{\mathsf{of}}\ \{ \\ \qquad\qquad\qquad (\mathit{zero}) \to \text{"}\mathit{zero}\text{"}; \\ \qquad\qquad\qquad (\mathit{succ}\ n') \to \text{"}\mathit{succ(}\text{"} +\!\!+ \mathit{show}\ n' +\!\!+ \text{"}\mathit{)}\text{"}\} \end{array} \right\rceil\right\}$$

$$= \left(\begin{array}{l} \mathit{showNat}\colon \mathit{Show}\ \mathit{Nat} \\ \mathit{showNat} = \underline{\mathsf{struct}}\ \{ \\ \qquad\qquad\qquad \mathit{show} = \lambda(n\colon \mathit{Nat})\ . \\ \qquad\qquad\qquad\qquad\qquad \underline{\mathsf{case}}\ n\ \underline{\mathsf{of}}\ \{ \\ \qquad\qquad\qquad\qquad\qquad\quad (\mathit{zero}) \to \text{"}\mathit{zero}\text{"}; \\ \qquad\qquad\qquad\qquad\qquad\quad (\mathit{succ}\ n') \to \text{"}\mathit{succ(}\text{"} +\!\!+ \mathit{show}\ n' +\!\!+ \text{"}\mathit{)}\text{"}\}\} \end{array} \right)$$

Fig. 6. Internal treatment of classes and instances

Of course, we cannot actually run the *main* program defined with postulated constants. In Haskell, the semantics of these I/O primitives is given by predefined library and runtime system. For Agda, our idea is to give meaning to those names of Agda I/O primitives by translating them to corresponding Haskell I/O primitives.

So we extend the universal type `Val` in Section 2.2 to include IO computations that return `Val`-type value when performed. We also need terms representing string literals and the unit value `()` for the explanation of *putStr* and *getLine*. The new constructors for `Val` and corresponding projection functions are shown below.

```
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         = ()
```

Now the postulated constant names can be given the actual Haskell definitions as shown in Figure 7. For instance, the translation of *putStr* is the untyped λ-term sending a value that should represent a string to the IO computation that performs Haskell `putStr` on the string and returns the value `VUnit` representing `()`.

$$[\![\underline{\mathsf{postulate}}\ \mathit{putStr}\colon \mathit{String} \to \mathit{IO\ Unit}]\!]$$

```
        = ( x_putStr = VAbs(\v->
                            VIO(putStr(deString v) >> return VUnit)) )
```

$$[\![\underline{\mathsf{postulate}}\ \mathit{getLine}\colon \mathit{IO\ String}]\!]$$

```
        = ( x_getLine = VIO(fmap VString getLine) )
```

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

```
        = ( (|>>=|) = VAbs(\a->VAbs(\b->VAbs(\m->VAbs(\f->
                            VIO(deIO m >>= deIO . apply f))))) )
```

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

```
        = ( x_ret = VAbs(\a->VAbs(\v->VIO(return v))) )
```

Fig. 7. The semantics of the postulated constants for I/O facilities

The echo program *main* is translated as previously explained, resulting in the Haskell function `x_main` of type `Val`. To let Haskell run time system execute the program, we extract IO computation and define it to be the Haskell `main` function.

```
main = deIO x_main
```

4 Systematic embedding to and projection from `Val`

The translations of Agda I/O primitives in the last section are embedding of corresponding Haskell I/O primitives as untyped λ-terms, that is, `Val` type values. This section describes a mechanism to systematically generate this embedding as well as projection of `Val` type values to the original Haskell values.

4.1 Type-directed `Val` embedding and projection

Our embedding, together with corresponding projection, of Haskell values to `Val` type values is type-directed. They are automatically generated by Haskell's class mechanism, when appropriate embedding to / projection from `Val` are given for each base type and type constructor. Here we consider the base types (type constructors) that are relevant when embedding `putStr`: `Char`, `a -> b`, `[a]`, `IO a`, and `()`.

First, we add to `Val` the data constructors and projection functions that correspond to those base types / type constructors. The extended `Val` and the projection functions are as follows.

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

(We scratch `VString` as we consider it now as the type synonym for `[Char]`.)

We write $\uparrow_{\texttt{a}}$ `: a -> Val` for the embedding from type `a` and $\downarrow_{\texttt{a}}$ `: Val -> a` for the projection. The defining clauses for them are as follows, each corresponding to the base type or type constructor. Together, these clauses generate the whole family $(\uparrow_{\texttt{a}}, \downarrow_{\texttt{a}})_{\texttt{a}}$ where `a` ranges over types constructed from those base types and constructors.

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

For example, $\uparrow_{\texttt{[Char]->IO ()}}$ `putStr` becomes the exactly same as the semantics of *putStr* in Figure 7.

With the base types and constructors considered here, many other Haskell I/O functions can be embedded to `Val`. Indeed, the most of Haskell IO library functions can be easily covered by just adding a few more base types.

The family of functions $\uparrow_{\texttt{a}}$ and $\downarrow_{\texttt{a}}$ can be considered to have a form of ad-hoc polymorphism in `a`. They can be implemented using the Haskell class-mechanism.

```
class Embeddable a where
  toVal     :: a -> Val
  fromVal   :: Val -> a
instance Embeddable Char where
  toVal     = VChar
  fromVal   = deChar
instance (Embeddable a, Embeddable b) => Embeddable (a -> b) where
  toVal   f = VAbs $ toVal . f . fromVal
  fromVal v = fromVal . (apply v) . toVal
instance (Embeddable a) => Embeddable [a] where
  toVal   l = VList (fmap toVal l)
  fromVal v = fmap fromVal (deList v)
instance (Embeddable a) => Embeddable (IO a) where
  toVal   x = VIO (fmap toVal x)
  fromVal v = fmap fromVal (deIO v)
instance Embeddable () where
  toVal   x = VUnit
  fromVal v = deUnit v
```

```
instance Embeddable Val where
  toVal   x = x
  fromVal x = x
```

Now the semantics of *putStr* in the previous section can be generated by simply writing `x_putStr = toVal putStr`.

We can systematise the generation further. In the above, the clauses for `[a]` and `IO a` are treated separately. But the functors `[]` and `IO` are instances of functor `f` equipped with $\Uparrow_{\mathtt{f}}$ `: f Val -> Val` and $\Downarrow_{\mathtt{f}}$ `: Val -> f Val`. With these, $\uparrow_{\mathtt{f\ a}}$ and $\downarrow_{\mathtt{f\ a}}$ can be defined for general `f` as follows:

$$\begin{array}{llll}
\uparrow_{\mathtt{f\ a}}(x) & = \Uparrow_{\mathtt{f}} (\mathtt{fmap}\ \uparrow_{\mathtt{a}}\ x) & \downarrow_{\mathtt{f\ a}}(v) & = \mathtt{fmap}\ \downarrow_{\mathtt{a}}\ (\Downarrow_{\mathtt{f}}\ v) \\
\Uparrow_{[]} & = \mathtt{VList} & \Downarrow_{[]} & = \mathtt{deList} \\
\Uparrow_{\mathtt{IO}} & = \mathtt{VIO} & \Downarrow_{\mathtt{IO}} & = \mathtt{deIO}
\end{array}$$

A limitation of the embedding / projection using Haskell class mechanism is that it needs manual intervention when embedding polymorphic Haskell functions. Let us consider Haskell function `return :: a -> IO a`. Unfortunately, the definition `x_ret = toVal return` does not work.

Firstly, `a` in general is not in `Embeddable` class, obviously, and which monad `return` refers to is not inferable for that matter. So we need to give a type signature to `return` by hand. The appropriate one to give here is

```
x_ret = toVal (return :: Val -> IO Val)
```

This is justified by the fact that `(fromVal x_ret) :: a -> IO a` for `a` in `Embeddable` class is then equivalent to `return :: a -> IO a`. Secondly, since Agda is explicitly polymorphic ('monomorphic' in type theory parlance), the translated `return` must take a (hidden) type argument. So the correct translation of `return`, which is `x_ret` in Figure 7, must be given as

```
x_ret = VAbs (\a -> toVal (return :: Val -> IO Val))
```

with further addition of `VAbs` and dummy abstraction by hand.

With this embedding of Haskell values into `Val`, one can write Agda programs that manipulate those embedded values through functions imported from Haskell. However, we need further consideration to analyse or construct those values with functions defined in Agda , for example, the modified echo program *main* below, as we explain next.

$$\begin{array}{l}
\mathit{main}\colon \mathit{IO\ Unit} \\
\mathit{main} = \mathit{getLine}\ \gg\!=\ \lambda(s\colon \mathit{String})\ .\ \mathit{putStr}\,(\text{"}\mathit{echo}:\text{"} +\!\!+\ s)
\end{array}$$

4.2 Converting embedded Haskell values to Agda terms

This section explains the conversion between values in `Val` that are embedded Haskell values and ones that represents Agda terms. With the translation of the last subsection, the Agda program *main* causes runtime error when it is compiled by Agate and run. Its cause is that embedded Haskell values are not constructed by `VCon` constructor.

On the one hand, the `Val` value that embed Haskell string `"ab"` is `VList [VChar 'a', VChar 'b']`. On the other hand, the translation from the same string in Agda to `Val` is `VCon ":" [VChar 'a', VCon ":" [VChar 'b', VCon "Nil" []]]`. When ++ does the case analysis expecting the latter form, run time error occurs. So we need a conversion between the two in order to use `getLine` or `putStr` mixed with functions defined in Agda.

For the conversion between lists given by $\uparrow_{[a]}$ and ones by the translation clause $[\![k\ e_1 \ldots e_n]\!]$ we replace `VList` and `deList` in the definition of $\Uparrow_{[]}$ and $\Downarrow_{[]}$ as follows:

```
⇑[] []     = VCon "Nil" []        | ⇓[] (VCon "Nil" [])  = []
⇑[] (x:xs) = VCon ":" [x, ⇑[] xs] | ⇓[] (VCon ":" [h,t]) = h: ⇓[] t
```

With this change, Agate generates Haskell source code that runs correctly.

Also with this change, there is no longer a need for the `Val` constructor `VList`, since lists are always embedded with `VCon ":" [...]` and `VCon "Nil" []`. Similarly, we can eliminate the constructor `VUnit` by replacing it by, say, `VCon "unit" []`. Generally, any Haskell data type can be embedded to `Val` without changing the latter.

5 Performance

The performance of Agate was compared[1] with GHC. We chose two programs factorial and Ackermann function, which are in the Haskell-fragment of Agda language shown in Figure 8, and compiled them with Agate on one hand, and directly with GHC on the other. So what we measured is the penalty of the encoding using `Val`.

The programs operate on the unary representation of natural numbers as a user-defined data type. The measurements reflect the performance of data construction, case-analysis, and function calls. To avoid for garbage-collection to take up most of execution time, we let the computation result to be consumed by the constant-zero function `go` in both Agate and Haskell version:

```
go (Succ n)       = go n
go Zero           = Zero
```

The Agate compiled versions show the penalty of about 2 to 6 times slower speed against the original GHC compiled versions as in Table 1[2]. Some obvious causes of inefficiencies in Haskell sources output by Agate are:

[1] Test environment: Cygwin on Windows XP SP2, Pentium M 1.2GHz, 512MB, 64KB L1 cache, 1MB L2 cache, ThinkPad X40, GHC 6.4.1 with optimisation package disabled.

[2] We used `time` command for the measurements. Both of the Agate compiled versions and GHC compiled versions ran with the runtime option `+RTS -A128m` except for the program "fact 11". It ran with the option `+RTS -M256m` since a runtime error `Heap exhausted` occurred with the previous option.

$$\begin{array}{l}
\underline{\mathsf{data}}\ \mathit{Nat} = \mathit{zero} \mid \mathit{succ}\ (n \colon \mathit{Nat}) \\
\mathit{plus} \colon \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\
\mathit{plus}\ m\ n = \underline{\mathsf{case}}\ m\ \underline{\mathsf{of}}\ \{(\mathit{zero}) \to n; (\mathit{succ}\ m') \to \mathit{succ}\ (\mathit{plus}\ m'\ n)\} \\
\mathit{mult} \colon \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\
\mathit{mult}\ m\ n = \underline{\mathsf{case}}\ m\ \underline{\mathsf{of}}\ \{(\mathit{zero}) \to \mathit{zero}; (\mathit{succ}\ m') \to \mathit{plus}\ n\ (\mathit{mult}\ m'\ n)\} \\
\mathit{fact} \colon \mathit{Nat} \to \mathit{Nat} \\
\mathit{fact}\ n = \underline{\mathsf{case}}\ n\ \underline{\mathsf{of}}\ \{(\mathit{zero}) \to \mathit{succ}\ \mathit{zero}; (\mathit{succ}\ n') \to \mathit{mult}\ n\ (\mathit{fact}\ n')\} \\
\mathit{ack} \colon \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\
\mathit{ack}\ m\ n = \underline{\mathsf{case}}\ m\ \underline{\mathsf{of}}\ \{(\mathit{zero}) \to \mathit{succ}\ n; \\
\qquad\qquad\qquad\qquad (\mathit{succ}\ m') \to \underline{\mathsf{case}}\ n\ \underline{\mathsf{of}}\ \{(\mathit{zero}) \to \mathit{ack}\ m'\ (\mathit{succ}\ \mathit{zero}); \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{succ}\ n') \to \mathit{ack}\ m'\ (\mathit{ack}\ (\mathit{succ}\ m')\ n')\}\}
\end{array}$$

Fig. 8. Factorial and Ackermann functions for benchmark

- Function calls are mediated by `VAbs` and `apply`.
- Case analysis requires string comparison.
- Constructor arguments are stored in Haskell lists.

Some of these are optimised away when Agate outputs are compiled by GHC. For example, for a declaration output by Agate `x_f = VAbs(\x -> ...)`, GHC gives a name to the function inside `VAbs`. An Agate application `apply f e` is compiled to a direct call to that function.

Table 1. Performance comparison of Agate and GHC

	fact 8	fact 9	fact 10	fact 11	ack 3 8	ack 3 9	ack 3 10	ack 3 11
GHC (sec)	0.047	0.199	1.515	9.952	0.177	0.564	1.934	7.508
Agate (sec)	0.063	0.408	3.430	29.746	0.672	2.384	9.364	39.239
ratio	1.346	2.055	2.263	2.989	3.807	4.228	4.841	5.226

We believe that this level of performance is sufficient for experiments on the practice of dependently typed programming.

6 Conclusion and Future work

We conclude that using higher order encoding of untyped λ-calculus was an effective way to obtain a compiler for dependently typed language with moderate performance in a small implementation effort.

For the future work, a few obvious and some more elaborate improvements can be made.

- More efficient data structure for the type `Val`: for example, `Val` can be extended for a given Agda program so that each constructor k used in it have its own `Val` constructor `VCon`$_k$, avoiding string representation `VCon "`k" ....

– Avoiding untyped encoding when possible: for example, list append `(++)` can be translated to have the type `[Val] -> [Val] -> [Val]` rather than just `Val`, when it is not used as an argument.
– Type-directed optimisation

Also, using Agate, we plan to experiment on various styles of dependently typed programming. For example, coding well-typed interpreter[15, 16] can be a good exercise with dependently typed programming. A longer term goal is Agda proof-assistant written in Agda language, or "Agda in Agda" (cf. [17])

References

[1] Coquand, C., et al.: Agda: An interactive proof editor (2006) Available at http://agda.sourceforge.net/.
[2] Martin-Löf, P.: Intuitionistic type theory (1984)
[3] Coquand, C., Coquand, T.: Structured type theory. In: Proc. of Workshop on Logical Frameworks and Meta-languages. (1999)
[4] Jones, S.L.P., Hall, C.V., Hammond, K., Partain, W., Wadler, P.: The glasgow haskell compiler: a technical overview. In: Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference. (1993) 249–257
[5] Pfenning, F., Elliot, C.: Higher-order abstract syntax. In: PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, New York, NY, USA, ACM Press (1988) 199–208
[6] Augustsson, L.: Cayenne - a language with dependent types. In: International Conference on Functional Programming. (1998) 239–250
[7] The Coq Development Team: The coq proof assistant reference manual – version v8.0 (2004)
[8] Paulin-Mohring, C., Werner, B.: Synthesis of ML programs in the system Coq. Journal of Symbolic Computation **15** (1993) 607–640
[9] Grégoire, B., Leroy, C.: A compiled implementation of strong reduction. In: Proc. of International Conference on Functional Programming, ACM Press (2002) 235–246
[10] McBride, C., McKinna, J.: The view from the left. Journal of Functional Programming **14**(1) (2004) 69–111
[11] Brady, E.: Practical Implementation of a Dependently Typed Functional Programming Language. PhD thesis, University of Durham (2005)
[12] Xi, H., Pfenning, F.: Dependent types in practical programming. In: Proceedings of the 26th ACM SIGPLAN Symbosium on Principles of Programming Languages. (1999) 214–227
[13] Xi, H.: de Caml (1999) Available at http://www.cs.bu.edu/ hwxi/DML/DML.html.
[14] Coquand, T., Pollack, R., Takeyama, M.: A logical framework with dependently typed records. In: Typed Lambda Calculus and Applications, TLCA'03. Volume 2701 of LNCS., Springer-Verlag (2003)
[15] Augustsson, L., Carlsson, M.: An exercise in dependent types: A well-typed interpreter (1999)
[16] Brady, E., Hammond, K.: Dependently typed meta-programming. In: Proc. of 7th Symposium on Trends in Functional Programming. (2006) Available at http://www.cs.nott.ac.uk/˜nhn/TFP2006/Papers/30-BradyHammond-DependentlyTypedMetaProgramming.pdf.
[17] Barras, B., Werner, B.: Coq in coq (1997)

Agate –Agda から Haskell へのコンパイラ
(in English)
(算譜科学研究速報)
発行日：2006 年 10 月 5 日
編集・発行：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
同連絡先：〒563-8577 大阪府池田市緑丘 1-8-31
e-mail：informatics-inquiry@m.aist.go.jp
本掲載記事の無断転載を禁じます

Agate –an Agda-to-Haskell compiler
(Programming Science Technical Report)
Oct. 5, 2006
Research Center for Verification and Semantics (CVS)
National Institute of Advanced Industrial Science and Technology (AIST)
1-8-31 Midorigaoka, Ikeda, Osaka, 563-8577, Japan
e-mail: informatics- inquiry@m.aist.go.jp