Alonzo — a Compiler for Agda

Marcin Benke

Institute of Informatics, Warsaw University,
ben@mimuw.edu.pl

1 Introduction

Agda [Norell, 2007] is an interactive system for developing constructive proofs in a variant of Per Martin-Löf's Type Theory. It can also be seen as a functional programming language with dependent types. To be used as a programming language, it has so far lacked a compiler (or at the very least, a decent interpereter). Alonzo (named in honour of Alonzo Church, as Haskell is named after Haskell B. Curry)[1] is an attempt to fill this gap.

1.1 Goals

Here are the goals we have set up for the project:

1. A compiler for Agda with performance of generated code matching this of GHC.
2. A real dependently typed language with solid type system and typechecker.
3. An environment to try out dependently-typed programming, especially with universes, in practice.

Achieving these goals is far from trivial; A large number of man-years went into development of GHC and to create a compiler producing code with performance matching this of GHC would require similar effort. However, rather than trying to compete with GHC, we can build on it by first translating Agda to Haskell and then compiling the resulting code with GHC.

It seems that the first two goals have been achieved. The performance of compiled programs seems to be quite satisfactory (although still somewhat worse than that of corresponding Haskell programs compiled with GHC).

We cannot honestly claim to have achieved the third goal yet, but it seems that the main hinder here is lack of a reasonably complete Agda library — such library is still under development. Initial results are, however, quite promising: programming with dependent types with Agda/Alonzo turns out to be pleasant (even if sometimes challenging) as well as very different from functional programming without dependent types.

[1] Haskell programs are close to Curry style lambda-calculus. On the other hand, Agda programs are closer to Church style

1.2 Related work

- Cayenne [Augustsson, 1998] — probably the first of the kind, translating a Haskell-like dependently-typed language to untyped LML; sadly: undecidable typechecking, not maintained anymore.
- Epigram — a very interesting functional programming language with dependent types [McBride and McKinna, 2004]. An experimental compiler for an earlier version of Epigram has been written by Edwin Brady [Brady, 2005].
- Agate — also a backend for Agda, embedding untyped lambda calculus in Haskell [Ozaki et al., 2006].

1.3 Example: well-typed printf

One of the standard examples for dependently-typed programming is "well-typed printf" — a function corresponding to *printf* from C, but checking if its argument conform to the specified format string.

As we will be using it in the sequel to illustrate some mechanisms, here is its signature and example use. The inquisitive reader can find the complete code in the Appendix.

```
Printf : String -> Set
printf : (fmt : String) -> Printf fmt -> String

mainS : String
mainS = printf "pi = %f %% %s"
  < 3.14159 | < "Alonzo" | unit > >
```

where the syntax `< a | b >` denotes pairs.

2 Translating Dependent Types into Haskell

At first glance, translating dependent types to Haskell type system is easy: just forget all the dependencies. For example, consider the inductive family of vectors and a function representing their concatenation:

```
data Vec (A : Set) : Nat -> Set where
  nil   : Vec A zero
  cons : {n : Nat} -> A -> Vec A n -> Vec A (suc n)

append : {A:Set} -> {n,m : Nat} -> Vec A n -> Vec A m -> Vec A (n+m)
...
```

One could translate them into following Haskell definitions:[2]

[2] Although we use GHC's extended syntax here, the definitions below can easily be expressed in Haskell98 as well.

```
data List a where
  nil :: List a
  cons  :: List a -> List a -> List a

append : List a -> List a -> List a
```

A similar approach was used by Brady in his Epigram compiler [Brady, 2005].

However, things get more complicated when you get to large elimination. To illustrate the problem, consider the following definitions:

```
Q : Bool -> Set
Q true = Nat
Q false = Bool

f : (b:Bool) -> Q b
f true = pred 3
f false = true

mainS : String
mainS = showBool (f (const false true))
```

Note that the actuial result type of f depends on the value of its arguments. How can such a function be translated into Haskell?

Assume we had a "magic" function `cast :: a -> b`. We could then translate `f` as follows:

```
f :: Bool -> b
f (true) = cast (pred (cast (3)))
f (false) = cast true
```

Luckily, GHC has such a function, albeit with an uglier name: `unsafeCoerce#`. Despite the name however, all coercions (typecasts, actually) we insert are safe, since the program has been already checked by Agda typechecker.

2.1 Translating Types

We need two translations for types:

1. to Haskell types (only data types)
2. to Haskell values

As for the first one, although it could seem that with the casts described above explicit use of Haskell types is not needed, we still need data types for pattern matching and of course as containers for data. But we don't really need to translate all the types, but only the datatypes, introduced with the `data` definitions.

This translation is relatively straightforward. Every Agda datatype is translated to a Haskell datatype and every constructor is translated to a constructor of the same arity.

However, in this translation we sacrifice a bit of (potential) efficiency for safety: GHC could potentially use the (not necessarily correct) knowledge of the types for optimisations. To prevent this,

```
data List (A:Set) : Set where
   nil : List a
   cons : A -> List A -> List A
```

becomes

```
data List a b = C1 | C2 a b
```

2.2 Translating Types to Values

In a dependently-typed system, types can be used as values in expressions. In Haskell no such thing is possible, so we must find a way of translating Agda types to Haskell values. One could of course argue that types cannot influence the result of the computation on the value level, and thus could be erased. However such erasure would alter arity of functions and quite often also strictness properties of the program. This seems to be a sufficient reason to avoid such erasure and go the extra mile of translating types to values as well.

How to translate

```
Q : Bool -> Set
Q true = Nat
Q false = Bool
```

into Haskell?

We use codes (actually, universes, if one prefers a fancier term); all datatypes (as well as primitive types) will have a value of this type assigned at compilation time, e.g.

```
dQ :: PreludeBool.Bool -> Runtime.Code
dQ (PreludeBool.C1) = cast PreludeNat.dNat
dQ (PreludeBool.C2) = cast PreludeBool.dBool
```

2.3 Codes

Since there is no pattern-matching on types (only on values), the encoding doesn't need to be injective. Hence, if we don't care for types at runtime, and want every bit of efficiency, a very simple coding can be used:

```
dNat = ()
dBool = ()
...
```

In this encoding the type of codes is simply the unit type; every type is encoded as unit value. This has the advantage of avoiding type computation at runtime entirely, while still preserving arities and strictness properties.

On the other hand, if we want run-time type information, we can introduce a more refined datatype of codes (e.g. the universe for dependent types described in [Benke et al., 2003]. Then we could have e.g. a (built-in) function

```
showSet : Set -> String
```

at very little cost.

2.4 Pattern matching

Another obstacle is pattern matching, consider an Agda definition

```
printf' : (fmt : List Format) -> Printf' fmt -> String
printf' (stringArg   :: fmt) < s | args > = s ++ printf' fmt args
...
printf' (badFormat _ :: fmt) ()
printf'  []  unit      = ""
```

The patterns are of different types, whereas Haskell demands that in all clauses of a definition, patterns for a given argument be of the same type. Casts don't solve the problem we cannot cast patterns.

We can solve this problem by making every clause into a separate single-clause definition and then gathering them together as local definitions for the main clause; for example the definition above would be translated as

```
  d38 = d38_1
  where d38_1 (PreludeList.C4 (Printf4.C3) v0)
          (AlonzoPrelude.C43 v1 v2)
          = cast
              (PreludeString.d0 (cast v1)
                 (cast (Printf4.d38 (cast v0) (cast v2))))
        d38_1 a b = cast d38_2 a b
  ...
  d38_8 (PreludeList.C3) (Printf4.C1) = cast ("")
```

3 Conclusions and future work

We have built a compiler for Agda, generating efficient code. Initial experiences with the compiler are quite promising: programming with dependent types with Agda/Alonzo turns out to be pleasant (even if sometimes challenging) as well as very different from functional programming without dependent types.

What is still lacking to make it into an environment for "real" dependently-typed programming, is a proper library. It would not be a rewrite of Haskell library, as it turns out that dependently-typed programming begs for different style, constructs and idioms. One particularly interesting area is managing input/output — several approaches have been proposed, although there is, as yet, no general consensus how this issue should be treated.

Another planned area of research fdor the near future is usiong partial evaluation on types to get most of GHC optimizations, thus improving efficiency of generated code.

A more distant, yet seemingly worthwhile plan would be to rewrite Agda in Agda, and prove properties of the code.

Acknowledgments

The author would like to thank the Agda team, especially Ulf Norell, Patrik Jansson, Nils Anders Danielsson, Catarina Coquand and Makoto Takeyama, for many fruitful discussions and help with testing and debugging the system (as well as for having created Agda in the first place).

References

[Augustsson, 1998] Augustsson, L. (1998). Cayenne — a language with dependent types. In *ICFP*, pages 239–250.

[Benke et al., 2003] Benke, M., Dybjer, P., and Jansson, P. (2003). Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289.

[Brady, 2005] Brady, E. C. (2005). *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, Durham University.

[McBride and McKinna, 2004] McBride, C. and McKinna, J. (2004). The view from the left. *Journal of Functional Programing*, 14(1).

[Norell, 2007] Norell, U. (2007). *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

[Ozaki et al., 2006] Ozaki, H., Takeyama, M., and Kinoshita, Y. (2006). Agate — an Agda to Haskell compiler. Technical Report JP: 21129158, Research Center for Verification and Semantics, National Institute of Advanced Industrial Science and Technology.

A Complete printf code

```
data Format : Set where
  stringArg : Format
  natArg    : Format
  intArg    : Format
  floatArg  : Format
  charArg   : Format
  litChar   : Char -> Format
  badFormat : Char -> Format

data BadFormat (c : Char) : Set where

format : String -> List Format
format s = format' (toList s)
  where
    format' : List Char -> List Format
    format' ('%' :: 's' :: fmt) = stringArg   :: format' fmt
    format' ('%' :: 'n' :: fmt) = natArg      :: format' fmt
    format' ('%' :: 'd' :: fmt) = intArg      :: format' fmt
    format' ('%' :: 'f' :: fmt) = floatArg    :: format' fmt
    format' ('%' :: 'c' :: fmt) = charArg     :: format' fmt
    format' ('%' :: '%' :: fmt) = litChar '%' :: format' fmt
    format' ('%' ::  c  :: fmt) = badFormat c :: format' fmt
    format' (c :: fmt) = litChar c   :: format' fmt
    format'  [] = []

Printf' : List Format -> Set
Printf' (stringArg   :: fmt) = String * Printf' fmt
Printf' (natArg      :: fmt) = Nat    * Printf' fmt
Printf' (intArg      :: fmt) = Int    * Printf' fmt
Printf' (floatArg    :: fmt) = Float  * Printf' fmt
Printf' (charArg     :: fmt) = Char   * Printf' fmt
Printf' (badFormat c :: fmt) = BadFormat c
Printf' (litChar _   :: fmt) = Printf' fmt
Printf'  []      = Unit

Printf : String -> Set
Printf fmt = Printf' (format fmt)

printf : (fmt : String) -> Printf fmt -> String
printf fmt = printf' (format fmt)
  where
    printf' : (fmt : List Format) -> Printf' fmt -> String
    printf' (stringArg   :: fmt) < s , args > = s  ++ printf' fmt args
    printf' (natArg      :: fmt) < n , args > = showNat n  ++ printf' fmt args
    printf' (intArg      :: fmt) < n , args > = showInt n  ++ printf' fmt args
    printf' (floatArg    :: fmt) < x , args > = showFloat x  ++ printf' fmt args
    printf' (charArg     :: fmt) < c , args > = showChar c  ++ printf' fmt args
    printf' (litChar c   :: fmt) args     = fromList (c :: []) ++ printf' fmt args
    printf' (badFormat _ :: fmt) ()
    printf'  []  unit      = ""

mainS : String
mainS = printf "Answer is %n, pi = %f %% %s"
< 42 , < 3.14159 , < "Alonzo" , unit > > >
```