1 Exporting from Agda to Haskell

1.1 Goal

As of Agda 2.3.2.2 there is a way to call Haskell code from Agda but no way to call Agda code from Haskell: i.e. to compile Agda code into a library usable from Haskell.

The feature is desirable because one can create a formally verified library in Agda, create a simple API around it and then use it in Haskell for real world applications.

1.2 Current state of things in Agda

1.2.1 As of Agda 2.3.2.2

There is a compiler from Agda to Haskell called MAlonzo[1](which is a rewrite of Alonzo[2] compiler) - it translates Agda code into a Haskell code so that observable behaviour of running an Agda interpreter and generated Haskell code is the same.

MAlonzo focuses on generating executables and therefore it has some undesirable characteristics:

- Names for functions and datatypes are of the form: a letter + a magic number
- Type signatures for functions are not written
- Generated datatypes omit all type parameters

On the other hand, apart from datatype parameters nothing is omitted: even arguments that represent proofs are carried(but are replaced with () type)

Still, at this state using generated code from Haskell is not pleasant.

1.2.2 Since Agda 2.3.4

New `{-# COMPILED_EXPORT` *AgdaName HaskellName* `#-}` pragma is added. It will force the MAlonzo compiler to use $HaskellName$ instead of a generated name for *AgdaName* function and will also generate an explicit type signature for $HaskellName$ according to the rules:

$$\begin{aligned}
T[\![Set\ A]\!] &= Unit \\
T[\![x\ As]\!] &= x\ T[\![As]\!] \\
T[\![(x : A) \to B]\!] &= \begin{cases} \forall x.\ T[\![A]\!] \to T[\![B]\!] & x \in freevars(B) \\ T[\![A]\!] \to T[\![B]\!] & otherwise \end{cases} \\
T[\![k\ As]\!] &= \begin{cases} T\ T[\![As]\!] & \texttt{COMPILED\_TYPE}\ k\ T \\ Unit & \texttt{COMPILED}\ k\ E \end{cases}
\end{aligned}$$

Also MAlonzo gained the ability to generate a Haskell library - not only an executable.

While improving the experience it still lacks better datatype generation.

1.3 The scope of this work

This work was developed on Agda 2.3.2.2 and features of 2.3.4 were not available.

The `{-# EXPORT` *AgdaName HaskellName* `#-}` pragma is introduced which will create a wrapper named $HaskellName$ for MAlonzo generated code for *AgdaName* function or datatype.

The primary motivation is exposing functions with such types that using them in any legitimate way from Haskell(i.e. no `unsafeCoerce` and such) will not break any internal invariant in Agda code. It automatically means that only functions with types that can be expressed in Haskell can be exposed - otherwise we lose information and invariants as a consequence.

Overview of features:

1. Functions with types expressed in Haskell can be exported and will undergo Church to Curry polymorphism transformation: instead of(as can be seen in Agda 2.3.4)

   $$TT[\![(A : Set) \to B]\!] = \forall a.\ () \to TT[\![B]\!]$$

   this holds

   $$TT[\![(A : Set) \to B]\!] = \forall a.\ TT[\![B]\!].$$

2. Datatypes can be exported as abstract datatypes when their type parameters are of types equivalent to kinds in Haskell(a combination of Set_0 and arrows).
   Constructors of this datatypes can be exported as Haskell functions (i.e. loosing the ability to pattern match on them) via feature 1.

3. Some Agda builtins: `INTEGER`, `FLOAT`, `CHAR`, `STRING`, `IO` will be exported as Int, $Float$, $Char$, $String$, IO respectively.

4. `COMPILED_TYPE` $AgdaType$ $HaskellType$ will be exported as $HaskellType$.

Section 2 gives formal definition of wrappers generated, section 3 gives proofs that exposed interface will not break the system.

2 Wrapper generation

2.1 Kinds

$$KT[\![Kind]\!] = HaskellKind$$

$$\begin{aligned} KT[\![Set_0]\!] &= * \\ KT[\![Kind_1 \to Kind_2]\!] &= KT[\![Kind_1]\!] \to KT[\![Kind_2]\!] \end{aligned}$$

2.2 Type declarations

$DTMA$ gives MAlonzo generated type name.
DT, DTD are defined when {`-# EXPORT` $AgdaTypeName$ $HaskellTypeName$ `#-`} is specified.

$$\begin{aligned} &DTMA[\![AgdaTypeName]\!] = HaskellTypeName \\ &DT[\![AgdaTypeName]\!] = HaskellTypeName \\ &DTD[\![AgdaTypeName]\!] \doteq HaskellTypeDeclaration \end{aligned}$$

Considering declaration:

$$\textbf{data}\ AgdaDataType\ (A_1 : Kind_1) \cdots (A_m : Kind_m) : Kind_{m+1} \to \cdots \to Kind_n \to Set\ \textbf{where}\ \ldots$$

$$\begin{aligned} &DTD[\![AgdaDataType]\!] \doteq \\ &\quad \textbf{newtype}\ DT[\![AgdaDataType]\!]\ (a_0 :: KT[\![Kind_1]\!]) \cdots (a_n :: KT[\![Kind_n]\!]) \\ &\qquad = DT[\![AgdaDataType]\!]\ (\forall b_1 \cdots b_k.\ DTMA[\![AgdaDataType]\!]\ b_1\ \cdots\ b_k) \end{aligned}$$

k is an arity of type constructor generated by MAlonzo.
It also works for **record**s.

2.3 Types

2.3.1 Primitive transformation

Only primitives that are used for postulates are allowed. MAlonzo defines:

$$PTMA[\![PrimitiveType]\!] = HaskellType$$

$$\begin{aligned} PTMA[\![\texttt{INTEGER}]\!] &= Int \\ PTMA[\![\texttt{FLOAT}]\!] &= Float \\ PTMA[\![\texttt{CHAR}]\!] &= Char \\ PTMA[\![\texttt{STRING}]\!] &= String \\ PTMA[\![\texttt{IO}]\!] &= IO \end{aligned}$$

2.3.2 Type transformation

$$\begin{aligned} TT[\![AgdaType]\!](Context) &= HaskellType \\ Context &= \{AgdaTypeVarName \mapsto HaskellTypeVarName\} \end{aligned}$$

$$\begin{aligned}
TT[\![T\ args\ldots]\!](\Gamma) &= \begin{cases} a\ TT[\![args\ldots]\!](\Gamma) & (T \mapsto a) \in \Gamma \\ CT\ TT[\![args\ldots]\!](\Gamma) & \texttt{\{-\# COMPILED\_TYPE}\ T\ \ CT\ \texttt{\#-\}} \text{ is specified} \\ PTMA[\![P]\!]\ TT[\![args\ldots]\!](\Gamma) & PTMA[\![P]\!] \text{ is defined and } \texttt{\{-\# BUILTIN}\ P\ \ T\ \texttt{\#-\}} \text{ is specified} \\ DT[\![T]\!]\ TT[\![args\ldots]\!](\Gamma) & DT[\![T]\!] \text{ is defined} \end{cases} \\
TT[\![(A : Kind) \to T]\!](\Gamma) &= \forall (a :: KT[\![Kind]\!]).\ TT[\![T]\!](\Gamma \cup \{A \mapsto a\}) \\
TT[\![(x : T_1) \to T_2]\!](\Gamma) &= TT[\![T_1]\!](\Gamma) \to TT[\![T_2]\!](\Gamma) \quad x \notin freevars(T_2) \\
TT[\![(x : T_1,\ T_2)]\!](\Gamma) &= (TT[\![T_1]\!](\Gamma),\ TT[\![T_2]\!](\Gamma)) \quad x \notin freevars(T_2)
\end{aligned}$$

2.4 Terms

$Wrap^0[\![AgdaType]\!](term)$ is defined only when $TT[\![AgdaType]\!](\varnothing)$ is defined.

$$\begin{aligned} Wrap^{2k}[\![AgdaType]\!](MAlonzoTerm) &= MyTerm \\ Wrap^{2k+1}[\![AgdaType]\!](MyTerm) &= MAlonzoTerm \end{aligned}$$

$$\begin{aligned}
Wrap^{k}[\![A\ args\ldots]\!](term) &= \texttt{unsafeCoerce}\ term \\
Wrap^{2k}[\![(A : Kind) \to T]\!](term) &= Wrap^{2k}[\![T]\!](term\ ()) \\
Wrap^{2k+1}[\![(A : Kind) \to T]\!](term) &= Wrap^{2k+1}[\![T]\!](\lambda\_.\ term) \\
Wrap^{k}[\![(x : T_1) \to T_2]\!](term) &= \lambda x.\ Wrap^{k}[\![T_2]\!](term\ Wrap^{k+1}[\![T_1]\!](x)) \\
Wrap^{k}[\![(x : T_1,\ T_2)]\!]((term_1,\ term_2)) &= (Wrap^{k}[\![T_1]\!](term_1),\ Wrap^{k}[\![T_2]\!](term_2))
\end{aligned}$$

2.5 Value declarations

$VTMA$ gives MAlonzo generated value name
$VT,\ VTD$ are defined when {-# EXPORT $AgdaName\ HaskellName$ #-} is specified.

$$\begin{aligned} VTMA[\![AgdaName]\!] &= HaskellName \\ VT[\![AgdaName]\!] &= HaskellName \\ VTD[\![AgdaName]\!] &\doteq HaskellDeclaration \end{aligned}$$

Considering declaration:

$$\begin{aligned} &AgdaName : AgdaType \\ &AgdaName = \ldots \end{aligned}$$

$$\begin{aligned} &VTD[\![AgdaName]\!] \doteq \\ &\quad VT[\![AgdaName]\!] :: TT[\![AgdaType]\!](\varnothing) \\ &\quad VT[\![AgdaName]\!] = Wrap^0[\![AgdaType]\!](VTMA[\![AgdaName]\!]) \end{aligned}$$

It works in the same way for constructors(it exports them as Haskell functions - not Haskell constructors). It also works seamlessly with parametrized modules and, consequently, with record functions.

3 Proofs

3.1 What does it mean not to break the system

1. Types describe a set of invariants. Therefore types exported from Agda to Haskell should refer to the same set.

2. There is a lot of `unsafeCoerce` which means there is a potential to use `APPLY` not on a function, to pass a wrong datatype to a function(and thus failing during pattern matching) and to use some primitive operation on a wrong builtin type(e.g. on `STRING` instead of `INTEGER`). Therefore all functions and its arguments should preserve the arity during the export; datatypes should be exported to some type that contains exactly the term Agda code produced; no Agda type can be exported into two different types(automatically solves the builtin problem).

3.2 Preserving type invariants

Three cases:

1. **newtype** wrappers.

   A datatype can be viewed as a logical statement and its constructors — as proofs of this statement. **newtype** wrapping gives us a statement but hides proofs. Acquiring an instance of this datatype in Haskell can only be done when some Agda function returns it. Therefore it was constructed with all the invariants checked by Agda. We can only use this datatype with functions exported from Agda that used the original version of it. Therefore all the internal invariants are safely passed from Agda to Agda invisibly through Haskell.

2. Transformation from Church polymorphism to Curry polymorphism.

   $$(A : Kind) \to Type$$
   $$(\forall a :: KT[\![Kind]\!]).\ HaskellType$$

   They both mean the same thing but the first one always requires a proof that $Kind$ is inhabited:

   - If $A \notin freevars(Type)$ and, by construction(TT), $a \notin freevars(HaskellType)$.
   - A (and consequently a) is a phantom type(i.e. only used as a type parameter).

   In both cases Haskell will completely ignore the inhabitance of $KT[\![Kind]\!]$. Agda however will require you to provide an evidence that $Kind$ can be constructed. Now, $Kind$ is defined as a combination of Set_0 and arrows. Therefore some $Kind\ A$ can be viewed as follows: $Arg_1 \to \ldots \to Arg_n \to Set_0$ for $n \geq 0$. Let's define a simple $Unit$ type:

   $$\begin{array}{l}\textbf{data}\ Unit : Set\ \textbf{where}\\ \quad unit : Unit\end{array}$$

   We can now construct an A: $A = \lambda arg_1 \ldots arg_n.\ Unit$. Therefore, each $Kind$ is inhabited and we can safely omit this proof in our transformation.

3. In every other case type is exactly the same.

   So invariants are clearly the same.

3.3 Preserving term interface

$Wrap$ clearly deals with the issue of passing and skipping type parameters with MAlonzo-generated code. A thing to watch for is `unsafeCoerce`. There are three cases for a coerced type:

1. a **newtype** wrapper around an MAlonzo-generated datatype.

   Safe because **newtype** is required to have the same internal structure as its wrapped type.

2. a primitive as defined by $PTMA$.

   Safe because type of MAlonzo-generated code is the same as ours by construction.

3. $a\ args\ldots$, where a is a type variable.

   Safe because all terms with type $a\ args\ldots$ will have the same internal structure. That's because from Haskell side compiler will guarantee that and from Agda side terms will have a corresponding type $A\ args\ldots$ (via Γ in TT) so the compiler will guarantee it too.

References

[1] Makoto Takeyama, *a new compiler MAlonzo*. `http://thread.gmane.org/gmane.comp.lang.agda/62`, 2008.

[2] Marcin Benke, *Alonzo — a compiler for Agda*. TYPES2007, 2007.