

1 Intro

I need to prove that Haskell types and terms that I expose wouldn't break the system. It means two things:

1. Types preserve the same set of invariants
2. Terms have the same interface: any combination of **APPLY** that can be used (ignoring types) to original term must be usable with generated one; and primitives (numbers, strings, ... and their ops) are the same.

2 My transformations

2.1 Kinds

$$KT\llbracket Kind \rrbracket = HaskellKind$$

$$\begin{aligned} KT\llbracket Set_0 \rrbracket &= * \\ KT\llbracket Kind_1 \rightarrow Kind_2 \rrbracket &= KT\llbracket Kind_1 \rrbracket \rightarrow KT\llbracket Kind_2 \rrbracket \end{aligned}$$

2.2 Type declarations

DTMA gives MALonzo generated type name.

DT, *DTD* are defined when `{-# EXPORT AgdaTypeName HaskellTypeName #-}` is specified.

$$\begin{aligned} DTMA\llbracket AgdaTypeName \rrbracket &= HaskellTypeName \\ DT\llbracket AgdaTypeName \rrbracket &= HaskellTypeName \\ DTD\llbracket AgdaTypeName \rrbracket &\doteq HaskellTypeDeclaration \end{aligned}$$

Considering declaration:

data *AgdaDataType* ($A_1 : Kind_1$) \cdots ($A_m : Kind_m$) : $Kind_{m+1} \rightarrow \cdots \rightarrow Kind_n \rightarrow Set$ **where** ...

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

k is an arity of type constructor generated by MALonzo.

It also works for **records**.

2.3 Types

First about primitives. Only those that are used for postulates are allowed. MALonzo gives the following *PTMA* transformation:

$$\begin{aligned} PTMA\llbracket \text{INTEGER} \rrbracket &= Int \\ PTMA\llbracket \text{FLOAT} \rrbracket &= Float \\ PTMA\llbracket \text{CHAR} \rrbracket &= Char \\ PTMA\llbracket \text{STRING} \rrbracket &= String \\ PTMA\llbracket \text{IO} \rrbracket &= IO \end{aligned}$$

$$\begin{aligned} TT\llbracket AgdaType \rrbracket (Context) &= HaskellType \\ Context &= \{ AgdaTypeVarName \mapsto HaskellTypeVarName \} \end{aligned}$$

$$\begin{aligned}
TT\llbracket A \text{ args } \dots \rrbracket(\Gamma) &= a \text{ } TT\llbracket \text{args } \dots \rrbracket(\Gamma), \quad (A \mapsto a) \in \Gamma \\
TT\llbracket CT \text{ args } \dots \rrbracket(\Gamma) &= CT \text{ } TT\llbracket \text{args } \dots \rrbracket(\Gamma), \quad CT \text{ is a } \text{COMPILED_TYPE} \\
TT\llbracket PT \text{ args } \dots \rrbracket(\Gamma) &= PTMA\llbracket PT \rrbracket TT\llbracket \text{args } \dots \rrbracket(\Gamma) \\
TT\llbracket ET \text{ args } \dots \rrbracket(\Gamma) &= DT\llbracket ET \rrbracket TT\llbracket \text{args } \dots \rrbracket(\Gamma) \\
TT\llbracket (A : Kind) \rightarrow T \rrbracket(\Gamma) &= \forall(a :: KT\llbracket Kind \rrbracket). TT\llbracket T \rrbracket(\Gamma \cup \{A \mapsto a\}) \\
TT\llbracket (x : T_1) \rightarrow T_2 \rrbracket(\Gamma) &= TT\llbracket T_1 \rrbracket(\Gamma) \rightarrow TT\llbracket T_2 \rrbracket(\Gamma), \quad x \notin \text{freevars}(T_2) \\
TT\llbracket (x : T_1, T_2) \rrbracket(\Gamma) &= (TT\llbracket T_1 \rrbracket(\Gamma), TT\llbracket T_2 \rrbracket(\Gamma)), \quad x \notin \text{freevars}(T_2)
\end{aligned}$$

2.4 Terms

$Wrap^0\llbracket AgdaType \rrbracket(term)$ is defined only when $TT\llbracket AgdaType \rrbracket(\emptyset)$ is defined.

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

$$\begin{aligned}
Wrap^k\llbracket A \text{ args } \dots \rrbracket(term) &= \text{unsafeCoerce } term \\
Wrap^{2k}\llbracket (A : Kind) \rightarrow T \rrbracket(term) &= Wrap^{2k}\llbracket T \rrbracket(term \text{ }) \\
Wrap^{2k+1}\llbracket (A : Kind) \rightarrow T \rrbracket(term) &= Wrap^{2k+1}\llbracket T \rrbracket(\lambda_. term) \\
Wrap^k\llbracket (x : T_1) \rightarrow T_2 \rrbracket(term) &= \lambda x. Wrap^k\llbracket T_2 \rrbracket(term \text{ } Wrap^{k+1}\llbracket T_1 \rrbracket(x)) \\
Wrap^k\llbracket (x : T_1, T_2) \rrbracket((term_1, term_2)) &= (Wrap^k\llbracket T_1 \rrbracket(term_1), Wrap^k\llbracket T_2 \rrbracket(term_2))
\end{aligned}$$

2.5 Value declarations

$VTMA$ gives $MAlonzo$ generated value name

VT , VTD are defined when $\{-\# \text{EXPORT } AgdaName \text{ HaskellName } \#-\}$ is specified.

$$\begin{aligned}
VTMA\llbracket AgdaName \rrbracket &= HaskellName \\
VT\llbracket AgdaName \rrbracket &= HaskellName \\
VTD\llbracket AgdaName \rrbracket &\doteq HaskellDeclaration
\end{aligned}$$

Considering declaration:

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

$$\begin{aligned}
VTD\llbracket AgdaName \rrbracket &\doteq \\
VT\llbracket AgdaName \rrbracket &:: TT\llbracket AgdaType \rrbracket(\emptyset) \\
VT\llbracket AgdaName \rrbracket &= Wrap^0\llbracket AgdaType \rrbracket(VTMA\llbracket AgdaName \rrbracket)
\end{aligned}$$

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 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.

$$(A : Kind) \rightarrow Type$$

$$(\forall a :: KT \llbracket Kind \rrbracket). HaskellType$$

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 inhabitation of $KT \llbracket Kind \rrbracket$. 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: $Arg_1 \rightarrow \dots \rightarrow Arg_n \rightarrow Set_0$ for $n \geq 0$. Let's define a simple *Unit* type:

```
data Unit : Set where
  unit : Unit
```

We can now construct an A : $A = \lambda arg_1 \dots arg_n. Unit$. 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.

4 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. $a\ args \dots$, where a is a type variable.

Safe because all terms with type $a\ args \dots$ 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 $A\ args \dots$ (via Γ in *TT*) so the compiler will guarantee it too.