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

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

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

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:

data $AgdaDataType\ (A_1 : Kind_1) \cdots (A_n : Kind_n) : Kind_{n+1} \to \cdots \to Kind_m \to Set$ **where** $\ldots$

$$\begin{aligned} &DTD[\![AgdaDataType]\!] \doteq \\ &\quad \textbf{newtype}\ DT[\![AgdaDataType]\!]\ (a_0 :: KT[\![Kind_1]\!]) \cdots (a_m :: KT[\![Kind_m]\!]) \\ &\qquad = DT[\![AgdaRecordType]\!]\ (\forall b_0 \cdots b_k.\ DTMA[\![AgdaDataType]\!]\ b_0\ \cdots\ b_k) \end{aligned}$$

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

2.3 Types

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

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

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

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

2.4 Terms

$Wrap$ 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 Preserving type invariants

Three cases:

1. `newtype` wrappers.

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:

$$\textbf{data}\ Unit : Set\ \textbf{where}$$
$$unit : Unit$$

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.

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. $a\ args\ldots$, 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.