

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 ($A_n : Kind_n$) : $Kind_{n+1} \rightarrow \cdots \rightarrow Kind_m \rightarrow Set$ **where** ...

$$\begin{aligned} DTD\llbracket AgdaDataType \rrbracket &\doteq \\ \mathbf{newtype} \quad DT\llbracket AgdaDataType \rrbracket \quad (a_0 :: KT\llbracket Kind_1 \rrbracket) \cdots (a_m :: KT\llbracket Kind_m \rrbracket) \\ &= DT\llbracket AgdaRecordType \rrbracket \quad (\forall b_0 \cdots b_k. DTMA\llbracket AgdaDataType \rrbracket \quad b_0 \cdots b_k) \end{aligned}$$

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:

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

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

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

2.4 Terms

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

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

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

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 also works seamlessly with parametrized modules and, consequently, with record functions.

3 Preserving type invariants

Two things to watch for:

- **newtype** wrappers preserve internal invariants of underlying Agda datatype.
- Transformation from Church polymorphism to Curry polymorphism.

In every other case type is exactly 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`. It is legal because it's either:

- The same term (when its type is a type variable)
- A newtype around MAlonzo generated type
- A primitive