Translation from OCaml to WhyML

All the auxiliary functions and predicates are total definitions. Fig. 1 and Fig. 2 present the definition of the selected OCaml and WhyML subsets, respectively. On the WhyML side, definition of t which stands for the logical subset of WhyML. Cameleer will report a dedicated error message if a user tries to translate an OCaml program that syntactically falls out of the supported fragment.

Expressions. Selected OCaml expressions include variables (x ranges over program variables, while f is used for function names), the conditional `if..then..else`, local bindings of (possibly recursive) expressions, function application, records manipulation (for simplicity, we assume every field to be mutable), treatment of exceptions, loop construction, and finally the `assert false` expression. Values include numerical and Boolean constants, as well as anonymous functions where arguments are annotated with a ghost status. We only consider functions as valid recursive definitions and application is limited to the application of a function name to a list of arguments. The latter is just to ease our presentation; the former is due to recursive definitions in WhyML being limited to functions. Finally, $\mathcal{A}$ notation stands for a (possibly empty) placeholder of OCaml attributes, representing the original place in the expression where

$$
\begin{array}{llll}
e & ::= & x \mid \nu \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{match}\ e\ \texttt{with}\ \overline{\talloblong\, p \Rightarrow e} & \text{Expressions}\\
 & \mid & \texttt{let}\ \mathcal{A}\ x = e\ \mathcal{A}\ \texttt{in}\ e & \\
 & \mid & \texttt{let}\ \mathcal{A}\ \texttt{rec}\ f = e\ \mathcal{A}\ \overline{\texttt{and}\ f = e\ \mathcal{A}}\ \texttt{in}\ e \mid f\ \bar{e} & \\
 & \mid & \{\overline{f = e}\} \mid e.f \mid e.f \leftarrow e & \\
 & \mid & \texttt{raise}\ E\bar{e} \mid \texttt{try}\ e\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} & \\
 & \mid & \texttt{while}\ e\ \texttt{do}\ \mathcal{A}\ e\ \texttt{done} \mid \texttt{assert false} & \\
p & ::= & \_ \mid x \mid \overline{p} \mid C(p) \mid p \talloblong p \mid p\ \texttt{as}\ x \mid p : \tau \mid \texttt{exception}\ p & \text{Patterns}\\
\nu & ::= & n \mid \texttt{true} \mid \texttt{false} \mid \texttt{fun}\ \mathcal{A}\ (x\beta) \rightarrow e & \text{Values}\\
\beta & ::= & \texttt{reg} \mid \texttt{ghost} & \text{Ghost attribute}\\
\tau & ::= & \alpha \mid \tau \rightarrow \tau \mid \overline{\tau} \mid \overline{\tau}\ C & \text{Type expression}\\
\pi & ::= & \beta\tau & \text{Type with ghost status}\\
d & ::= & \texttt{exception}\ E : \overline{\pi} \mid \texttt{type}\ td\ \overline{\texttt{and}\ td} & \text{Top-level declarations}\\
 & \mid & \texttt{let}\ \mathcal{A}\ f = e\ \mathcal{A} \mid \texttt{let}\ \mathcal{A}\ \texttt{rec}\ f = e\ \mathcal{A}\ \overline{\texttt{and}\ f = e\ \mathcal{A}} & \\
 & \mid & \texttt{module}\ \mathcal{M} = m & \\
td & ::= & \overline{\alpha}\ T \mid \overline{\alpha}\ T = \tau \mid \overline{\alpha}\ T = \{\ \overline{f : \pi}\ \}\ \mathcal{A} \mid \overline{\alpha}\ T = \overline{\talloblong\, C\ \texttt{of}\ \overline{\tau}} & \text{Type definition}\\
m & ::= & \texttt{struct}\ \overline{d}\ \texttt{end} \mid \texttt{functor}(\mathcal{X} : mt) \rightarrow m & \text{Modules}\\
mt & ::= & \texttt{sig}\ \overline{s}\ \texttt{end} & \text{Module types}\\
s & ::= & \texttt{val}\ \mathcal{A}\ f : \pi\ \mathcal{A} \mid \texttt{type}\ td\ \overline{\texttt{and}\ td} & \text{Signatures}\\
p & ::= & \overline{d} & \text{Program}
\end{array}
$$

Figure 1: Syntax of core OCaml.

$$
\begin{array}{llll}
e & ::= & x \mid \nu \mid \texttt{if}\, e\, \texttt{then}\, e\, \texttt{else}\, e \mid \texttt{match}\, e\, \texttt{with}\, \overline{\,[\!]\, p \Rightarrow e}\ \texttt{end} & \text{Expressions} \\
 & & \mid \texttt{let}\, \mathcal{K}\, \beta x = e\ \texttt{in}\, e & \\
 & & \mid \texttt{rec}\, \mathcal{K}\, f(\overline{\beta x})\, \mathcal{S} = e\, \overline{\texttt{with}\, \mathcal{K}\, f(\overline{\beta x}) = e\, \mathcal{S}}\ \texttt{in}\, e \mid f\, \overline{e} & \\
 & & \mid \{\overline{f = e}\} \mid e.f \mid e.f \leftarrow e & \\
 & & \mid \texttt{raise}\, E\overline{e} \mid \texttt{try}\, e\, \texttt{with}\, \overline{E\overline{x} \Rightarrow e}\ \texttt{end} & \\
 & & \mid \texttt{while}\, e\, \texttt{do}\, \mathcal{I}\, e\, \texttt{done} \mid \texttt{absurd} \mid \texttt{ghost}\, e & \\
p & ::= & \_ \mid x \mid \overline{p} \mid Cp \mid p\, [\!]\, p \mid p\ \texttt{as}\ x \mid p : \tau \mid \texttt{exception}\, p & \text{Patterns} \\
\nu & ::= & n \mid \texttt{true} \mid \texttt{false} \mid \texttt{fun}\, \mathcal{K}\, (\overline{\beta x})\, \mathcal{S} \rightarrow e & \text{Values} \\
\beta & ::= & \texttt{reg} \mid \texttt{ghost} & \text{Ghost status} \\
\tau & ::= & \alpha \mid \tau \rightarrow \tau \mid \overline{\tau} \mid C\,\overline{\tau} & \text{Type expression} \\
\pi & ::= & \beta\tau & \text{Type with ghost status} \\
\mathcal{K} & ::= & \texttt{reg} \mid \texttt{logic} & \text{Function kind} \\
\mathcal{S} & ::= & \texttt{requires}\, \overline{t}\ \ \texttt{ensures}\, \overline{t}\ \ \texttt{variant}\, \overline{t} & \text{Function specification} \\
\mathcal{I} & ::= & \texttt{invariant}\, \overline{t}\ \ \texttt{variant}\, \overline{t} & \text{Loop specification} \\
d & ::= & \texttt{exception}\, E : \overline{\pi} \mid \texttt{type}\, td\, \overline{\texttt{with}\, td} & \text{Top-level declarations} \\
 & & \mid \texttt{let}\, \mathcal{K}\, f = e & \\
 & & \mid \texttt{let rec}\, \mathcal{K}\, f(\overline{\beta x})\, \mathcal{S} = e\, \overline{\texttt{with}\, \mathcal{K}\, f(\overline{\beta x})\, \mathcal{S} = e} & \\
 & & \mid \texttt{val}\, \mathcal{K}\, \beta f(\overline{x : \pi})\, \mathcal{S} : \pi \mid \texttt{scope}\, \mathcal{M}\, \overline{d}\ \texttt{end} & \\
td & ::= & T\overline{\alpha} \mid T\overline{\alpha} = \tau \mid T\overline{\alpha} = \{\, \overline{f : \pi}\, \}\ \texttt{invariant}\, \overline{t} \mid T\overline{\alpha} = \overline{[\!]\, C\, \overline{\tau}} & \text{Type definition} \\
p & ::= & \texttt{module}\, \mathcal{M}\, \overline{d}\ \texttt{end} & \text{Program}
\end{array}
$$

Figure 2: Syntax of core WhyML.

GOSPEL elements are introduced. For instance, the first $\mathcal{A}$ in a `let..rec` expression can contain the `[@ghost]` and `[@logic]` attribute, while the second one stands for the function specification. We omit the definition of τ and t, respectively from the OCaml and the WhyML sides. The former stands for the grammar of OCaml types, while the latter is the logical subset of WhyML.

Top-level declarations. Selected top-level declarations include exceptions and type declaration, (mutually-recursive) function definition, and introduction of sub-modules. An exception takes a list of π values, types annotated with a ghost status, to account for the possibility of ghost arguments. In Why3 vocabulary, this is the *mask* of an exception. The attribute $\mathcal{A}$ after a record type definition is used to express in GOSPEL a *type invariant*, *i.e.*, a predicate that every inhabitant of such type must satisfy. Type invariants are readily supported by Why3, as depicted in rule (TDRecord) in Fig. 5. Each field of the record type is also annotated with a ghost status.

Modules. The most interesting cases in our translation is how we deal with the modules language from the OCaml side. A WhyML program is a list of modules, a module is a list of top-level declarations, and declarations can be organized within scopes.

$$(\textsc{EAbsurd})\ \frac{}{\texttt{assert false} \overset{\textit{expression}}{\rightsquigarrow} \texttt{absurd}}$$

$$(\textsc{EFun})\ \frac{\texttt{ghost}(\mathcal{A}) = \beta \qquad \mathcal{A} \overset{\textit{function spec}}{\rightsquigarrow} \mathcal{S} \qquad e \overset{\textit{expression}}{\rightsquigarrow} e'}{\texttt{fun}\ \mathcal{A}\ x \to e \overset{\textit{expression}}{\rightsquigarrow} \texttt{fun}\ (\beta x)\ \mathcal{S} = e'} \qquad \frac{\bar{e} \overset{\textit{expression}}{\rightsquigarrow} \overline{e'}}{f\ \bar{e} \overset{\textit{expression}}{\rightsquigarrow} f\ \overline{e'}}\ (\textsc{EApp})$$

$$(\textsc{ERec})\ \frac{\begin{array}{c}\neg \textit{is\_ghost}(\mathcal{A}_0) \qquad \textit{kind}(\mathcal{A}_0) = \mathcal{K} \qquad \mathcal{A}_1 \overset{\textit{function spec}}{\rightsquigarrow} \mathcal{S}_1 \qquad e_0 \overset{\textit{function}}{\rightsquigarrow} \overline{x\beta}, e_0' \\ \overline{e_1} \overset{\textit{function}}{\rightsquigarrow} \overline{\beta y}, \overline{e_1'} \qquad \overline{\mathcal{A}_2} \overset{\textit{function spec}}{\rightsquigarrow} \overline{\mathcal{S}_2} \qquad e_2 \overset{\textit{expression}}{\rightsquigarrow} e_2'\end{array}}{\texttt{let}\ \mathcal{A}_0\ \texttt{rec}\ f_0 = e_0\ \mathcal{A}_1\ \overline{\texttt{and}\ f_1 = e_1\ \mathcal{A}_2}\ \texttt{in}\ e_2 \overset{\textit{expression}}{\rightsquigarrow} \texttt{rec}\ \mathcal{K}\ f(\overline{\beta x})\ \mathcal{S}_1 = e_0'\ \overline{\texttt{with}\ \mathcal{K}\ f_1(\overline{\beta y}) = e_1'\ \mathcal{S}_2}\ \texttt{in}\ e_2'}$$

$$(\textsc{ERecGhost})\ \frac{\begin{array}{c}\textit{is\_ghost}(\mathcal{A}_0) \qquad \textit{kind}(\mathcal{A}_0) = \mathcal{K} \qquad \mathcal{A}_1 \overset{\textit{function spec}}{\rightsquigarrow} \mathcal{S}_1 \qquad e_0 \overset{\textit{function}}{\rightsquigarrow} \overline{x\beta}, e_0' \\ \overline{e_1} \overset{\textit{function}}{\rightsquigarrow} \overline{\beta y}, \overline{e_1'} \qquad \overline{\mathcal{A}_2} \overset{\textit{function spec}}{\rightsquigarrow} \overline{\mathcal{S}_2} \qquad e_2 \overset{\textit{expression}}{\rightsquigarrow} e_2'\end{array}}{\texttt{let}\ \mathcal{A}_0\ \texttt{rec}\ f_0 = e_0\ \mathcal{A}_1\ \overline{\texttt{and}\ f_1 = e_1\ \mathcal{A}_2}\ \texttt{in}\ e_2 \overset{\textit{expression}}{\rightsquigarrow} \texttt{rec}\ \mathcal{K}\ f(\overline{\beta x})\ \mathcal{S}_1 = \texttt{ghost}\ e_0'\ \overline{\texttt{with}\ \mathcal{K}\ f_1(\overline{\beta y}) = e_1'\ \mathcal{S}_2}\ \texttt{in}\ e_2'}$$

$$(\textsc{ELet})\ \frac{\begin{array}{c}\neg \textit{is\_ghost}(\mathcal{A}) \qquad \texttt{kind}(\mathcal{A}) = \mathcal{K} \qquad \neg \textit{is\_functional}(e_0) \\ e_0 \overset{\textit{expression}}{\rightsquigarrow} e_0' \qquad e_1 \overset{\textit{expression}}{\rightsquigarrow} e_1'\end{array}}{\texttt{let}\ \mathcal{A}\ x = e_0\ \mathcal{A}'\ \texttt{in}\ e_1 \overset{\textit{expression}}{\rightsquigarrow} \texttt{let}\ \mathcal{K}\ x = e_0'\ \texttt{in}\ e_1'}$$

$$(\textsc{ELetGhost})\ \frac{\begin{array}{c}\textit{is\_ghost}(\mathcal{A}) \qquad \texttt{kind}(\mathcal{A}) = \mathcal{K} \qquad \neg \textit{is\_functional}(e_0) \\ e_0 \overset{\textit{expression}}{\rightsquigarrow} e_0' \qquad e_1 \overset{\textit{expression}}{\rightsquigarrow} e_1'\end{array}}{\texttt{let}\ \mathcal{A}\ x = e_0\ \mathcal{A}'\ \texttt{in}\ e_1 \overset{\textit{expression}}{\rightsquigarrow} \texttt{let}\ \mathcal{K}\ x = \texttt{ghost}\ e_0'\ \texttt{in}\ e_1'}$$

$$(\textsc{ELetGhostFun})\ \frac{\begin{array}{c}\textit{is\_ghost}(\mathcal{A}) \qquad \textit{kind}(\mathcal{A}) = \mathcal{K} \qquad \textit{is\_functional}(e_0) \\ \mathcal{A}' \overset{\textit{function spec}}{\rightsquigarrow} \mathcal{S} \qquad e_0 \overset{\textit{function}}{\rightsquigarrow} \overline{\beta x}, e_0' \qquad e_1 \overset{\textit{expression}}{\rightsquigarrow} e_1'\end{array}}{\texttt{let}\ \mathcal{A}\ f = e_0\ \mathcal{A}'\ \texttt{in}\ e_1 \overset{\textit{expression}}{\rightsquigarrow} \texttt{let}\ \mathcal{K}\ f = \texttt{fun}(\overline{\beta x})\ \mathcal{S} \to \texttt{ghost}\ e_0'\ \texttt{in}\ e_1'}$$

$$(\textsc{ELetFun})\ \frac{\begin{array}{c}\neg \textit{is\_ghost}(\mathcal{A}) \qquad \textit{kind}(\mathcal{A}) = \mathcal{K} \qquad \textit{is\_functional}(e_0) \\ \mathcal{A}' \overset{\textit{function spec}}{\rightsquigarrow} \mathcal{S} \qquad e_0 \overset{\textit{function}}{\rightsquigarrow} \overline{x\beta}, e_0' \qquad e_1 \overset{\textit{expression}}{\rightsquigarrow} e_1'\end{array}}{\texttt{let}\ \mathcal{A}\ f = e_0\ \mathcal{A}'\ \texttt{in}\ e_1 \overset{\textit{expression}}{\rightsquigarrow} \texttt{let}\ \mathcal{K}\ f = \texttt{fun}(\overline{\beta x})\ \mathcal{S} \to e_0'\ \texttt{in}\ e_1'}$$

$$(\textsc{EWhile})\ \frac{\mathcal{A} \overset{\textit{loop annotation}}{\rightsquigarrow} \mathcal{I} \qquad e_0 \overset{\textit{expression}}{\rightsquigarrow} e_0' \qquad e_1 \overset{\textit{expression}}{\rightsquigarrow} e_1'}{\texttt{while}\ e_0\ \texttt{do}\ \mathcal{A}\ e_1\ \texttt{done} \overset{\textit{expression}}{\rightsquigarrow} \texttt{while}\ e_0'\ \texttt{do}\ \mathcal{I}\ e_1'\ \texttt{done}}$$

$$(\textsc{EIf})\ \frac{e_0 \overset{\textit{expression}}{\rightsquigarrow} e_0' \qquad e_1 \overset{\textit{expression}}{\rightsquigarrow} e_1' \qquad e_2 \overset{\textit{expression}}{\rightsquigarrow} e_2'}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \overset{\textit{expression}}{\rightsquigarrow} \texttt{if}\ e_0'\ \texttt{then}\ e_1'\ \texttt{else}\ e_2'}$$

$$(\textsc{EMatch})\ \frac{e_0 \overset{\textit{expression}}{\rightsquigarrow} e_0' \qquad \overline{e_1} \overset{\textit{expression}}{\rightsquigarrow} \overline{e_1'}}{\texttt{match}\ e_0\ \texttt{with}\ \overline{\mid p \Rightarrow e_1} \overset{\textit{expression}}{\rightsquigarrow} \texttt{match}\ e_0'\ \texttt{with}\ \overline{\mid p \Rightarrow e_1'}\ \texttt{end}}$$

$$(\textsc{ERaise})\ \frac{\bar{e} \overset{\textit{expression}}{\rightsquigarrow} \bar{e}'}{\texttt{raise}\ E\bar{e} \overset{\textit{expression}}{\rightsquigarrow} \texttt{raise}\ E\bar{e}'} \qquad \frac{e \overset{\textit{expression}}{\rightsquigarrow} e' \qquad \overline{E\bar{x} \Rightarrow e} \overset{\textit{expression}}{\rightsquigarrow} \overline{E\bar{x} \Rightarrow e}'}{\texttt{try}\ e\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e} \overset{\textit{expression}}{\rightsquigarrow} \texttt{try}\ e'\ \texttt{with}\ \overline{E\bar{x} \Rightarrow e}'\ \texttt{end}}\ (\textsc{ETry})$$

Figure 3: Translation of OCaml expressions into WhyML.

The first module expression we take into account is the `struct..end` construction. This is translated into a WhyML declarations, as depicted in rule (MStruct) (Appendix C). We note this does not change the structure and code organization of the original program, since a `struct..end` expression follows a `module` declaration. Hence, a declaration of the form $\texttt{module}\,\mathcal{M}\;\bar{d}\;\texttt{end}$ is translated into $\texttt{scope}\,\mathcal{M}\;\bar{d}\;\texttt{end}$.

Functors are a central notion when programming in OCaml, so it is out of question to develop a verification tool for OCaml without a (at least minimal) support for functors. WhyML does not feature a syntactic construction for functors; instead, these are represented as modules containing only abstract symbols [?]. Thus, we propose the following translation rule:

$$(\text{MFunctor})\ \frac{mt \xrightarrow{\textit{module type}} \overline{d} \qquad m \xrightarrow{\textit{module}} \overline{d'}}{\texttt{functor}(\mathcal{X} : mt) \to m \xrightarrow{\textit{module}} \texttt{scope}\ \mathcal{X}\ \overline{d}\ \texttt{end}\ \overline{d'}}$$

Each `functor` expression is translated into a WhyML scope followed by a list of declarations. For instance, the OCaml functor `module Make = functor (X: ...) -> struct ... end`, is translated into the following WhyML excerpt: `scope Make scope X ... end ... end`. The given transformation is the dual of what is actually implemented in the Why3 extraction machinery: every WhyML expression of the form `scope A scope B ...` is translated into `module A (B: ...) ...`, *as long as* scope `B` features only abstract symbols.

Signatures. The argument of a functor is expressed as a *module type*, *i.e.*, a *signature* of the form `sig..end`. This encapsulates a list of declarations belonging to the OCaml signature language, which are translated into a list of WhyML expressions, according to rule (MTSig) (Appendix C). Contrarily to OCaml, WhyML does not impose a separation between signature (interface) and structure (implementation) elements. In particular, the WhyML surface language allows one to include non-defined `val` functions and regular `let` definitions in the same namespace. We give the following translation rule for `val` declarations:

$$(\text{SVal})\ \frac{\neg \textit{is\_ghost}(\mathcal{A}) \qquad \textit{kind}(\mathcal{A}) = \mathcal{K} \qquad \mathcal{A}' \xrightarrow{\textit{function spec}} \mathcal{S} \qquad \pi, \mathcal{A}' \xrightarrow{\textit{function args}} \overline{x : \pi'}, \pi_{res}}{\texttt{val}\ \mathcal{A}\ f : \pi\ \mathcal{A}' \xrightarrow{\textit{signature}} \texttt{val}\ \mathcal{K}\ f(\overline{x : \pi})\ \mathcal{S} : \pi_{res}}$$

The name of the arguments are retrieved from the function specification (Sec. **??** features an example of such case). Non-defined functions can also be declared as ghost and/or logical functions. For brevity, the case of ghost `val` is omitted. The complete set of translation rule for signature items can be found in Appendix D.

Programs. An OCaml program is simply a list of top-level declarations. These are translated into a WhyML module, as follows:

$$(\text{Program})\ \frac{\overline{d} \xrightarrow{\textit{declaration}} \overline{d'}}{\overline{d} \xrightarrow{\textit{program}} \texttt{module}\ \mathcal{M}\ \overline{d'}\ \texttt{end}}$$

The name $\mathcal{M}$ of the generated module is issued from the OCaml file that contains the original program. If file `foo.ml` contains the program p, it gets translated into $\texttt{module Foo}\ p\ \texttt{end}$. In summary, we generate a WhyML program containing a single module, which represents the top-level module of an OCaml file. In turn, each sub-module is translated into a WhyML scope, with a special treatment for functorial definitions.

A Translation of expressions

B Translation of top-level declarations and type definitions

$$(\textsc{DModule})\ \frac{m \overset{\textit{module}}{\rightsquigarrow} \overline{d}}{\texttt{module}\ \mathcal{M} = m \overset{\textit{declaration}}{\rightsquigarrow} \texttt{scope}\ \mathcal{M}\ \overline{d}\ \texttt{end}}$$

$$(\textsc{DLetGhost})\ \frac{\begin{array}{c} \mathit{is\_ghost}(\mathcal{A}) \qquad \mathit{kind}(\mathcal{A}') = \mathcal{K} \qquad \mathit{is\_functional}(e) \\ \mathcal{A} \overset{\textit{function spec}}{\rightsquigarrow} \mathcal{S} \qquad e \overset{\textit{function}}{\rightsquigarrow} \overline{\beta x}, e' \end{array}}{\texttt{let}\ \mathcal{A}\ f = e\ \mathcal{A}' \overset{\textit{declaration}}{\rightsquigarrow} \texttt{let}\ f\ \mathcal{K} = \texttt{fun}(\overline{\beta x})\ \mathcal{S} \rightarrow \texttt{ghost}\ e'}$$

$$(\textsc{DLet})\ \frac{\begin{array}{c} \neg\mathit{is\_ghost}(\mathcal{A}) \qquad \mathit{kind}(\mathcal{A}') = \mathcal{K} \qquad \mathit{is\_functional}(e) \\ \mathcal{A} \overset{\textit{function spec}}{\rightsquigarrow} \mathcal{S} \qquad e \overset{\textit{function}}{\rightsquigarrow} \overline{\beta x}, e' \end{array}}{\texttt{let}\ \mathcal{A}\ f = e\ \mathcal{A}' \overset{\textit{declaration}}{\rightsquigarrow} \texttt{let}\ f\ \mathcal{K} = \texttt{fun}(\overline{\beta x})\ \mathcal{S} \rightarrow\ e'}$$

$$(\textsc{DRec})$$
$$\frac{\begin{array}{c} \neg\mathit{is\_ghost}(\mathcal{A}_0) \qquad \mathit{kind}(\mathcal{A}_0) = \mathcal{K} \qquad \mathcal{A}_1 \overset{\textit{function spec}}{\rightsquigarrow} \mathcal{S}_1 \qquad e_0 \overset{\textit{function}}{\rightsquigarrow} \overline{x\beta}, e'_0 \\ \overline{e_1} \overset{\textit{function}}{\rightsquigarrow} \overline{\overline{y:\pi}, e'_1} \qquad \overline{\mathcal{A}_2} \overset{\textit{function spec}}{\rightsquigarrow} \overline{\mathcal{S}_2} \end{array}}{\texttt{let}\ \mathcal{A}_0\ \texttt{rec}\ f_0 = e_0\ \mathcal{A}_1\ \overline{\texttt{and}\ f_1 = e_1\ \mathcal{A}_2} \overset{\textit{declaration}}{\rightsquigarrow} \texttt{rec}\ \mathcal{K}\ f(\overline{\beta x})\ \mathcal{S}_1 = e'_0\ \overline{\texttt{with}\ \mathcal{K}\ f_1(\overline{y:\pi}) = e'_1\ \mathcal{S}_2}}$$

$$(\textsc{DRecGhost})$$
$$\frac{\begin{array}{c} \mathit{is\_ghost}(\mathcal{A}_0) \qquad \mathit{kind}(\mathcal{A}_0) = \mathcal{K} \qquad \mathcal{A}_1 \overset{\textit{function spec}}{\rightsquigarrow} \mathcal{S}_1 \qquad e_0 \overset{\textit{function}}{\rightsquigarrow} \overline{x\beta}, e'_0 \\ \overline{e_1} \overset{\textit{function}}{\rightsquigarrow} \overline{\overline{y:\pi}, e'_1} \qquad \overline{\mathcal{A}_2} \overset{\textit{function spec}}{\rightsquigarrow} \overline{\mathcal{S}_2} \end{array}}{\texttt{let}\ \mathcal{A}_0\ \texttt{rec}\ f_0 = e_0\ \mathcal{A}_1\ \overline{\texttt{and}\ f_1 = e_1\ \mathcal{A}_2} \overset{\textit{declaration}}{\rightsquigarrow} \texttt{rec}\ \mathcal{K}\ f(\overline{\beta x})\ \mathcal{S}_1 = \texttt{ghost}\ e'_0\ \overline{\texttt{with}\ \mathcal{K}\ f_1(\overline{y:\pi}) = e'_1\ \mathcal{S}_2}}$$

$$(\textsc{DType})\ \frac{td_0 \overset{\textit{type definition}}{\rightsquigarrow} td'_0 \qquad \overline{td_1} \overset{\textit{type definition}}{\rightsquigarrow} \overline{td'_1}}{\texttt{type}\ td_0\ \overline{\texttt{and}\ td_1} \overset{\textit{declaration}}{\rightsquigarrow} \texttt{type}\ td'_0\ \overline{\texttt{with}\ td'_1}}$$

$$(\textsc{DExn})\ \frac{}{\texttt{exception}\ E : \overline{\pi} \overset{\textit{declaration}}{\rightsquigarrow} \texttt{exception}\ E : \overline{\pi}}$$

Figure 4: Translation of OCaml top-level declarations into WhyML.

$$(\textsc{TDAbstract})\ \frac{}{\overline{\alpha}\ T \overset{\textit{type definition}}{\rightsquigarrow} T\overline{\alpha}} \qquad \frac{}{\overline{\alpha}\ T = \tau \overset{\textit{type definition}}{\rightsquigarrow} T\overline{\alpha} = \tau}\ (\textsc{TDAlias})$$

$$(\textsc{TDRecord})\ \frac{\mathcal{A} \overset{\textit{type invariant}}{\rightsquigarrow} \overline{t}}{\overline{\alpha}\ T = \{\ \overline{f:\pi}\ \}\ \mathcal{A} \overset{\textit{declaration}}{\rightsquigarrow} T\overline{\alpha} = \{\ \overline{f:\pi}\ \}\ \texttt{invariant}\ \overline{t}}$$

$$(\textsc{TDVariant})\ \frac{}{\overline{\alpha}\ T = \overline{\mid C\ \texttt{of}\ \overline{\tau}} \overset{\textit{type definition}}{\rightsquigarrow} T\overline{\alpha} = \overline{\mid C\ \overline{\tau}}}$$

Figure 5: Translation of OCaml type definitions into WhyML.

C Translation of module expressions and module types

$$(\textsc{MStruct})\ \frac{\overline{d} \overset{declaration}{\rightsquigarrow} \overline{d'}}{\texttt{struct}\ \overline{d}\ \texttt{end} \overset{module}{\rightsquigarrow} \overline{d'}}$$

$$(\textsc{MFunctor})\ \frac{mt \overset{module\ type}{\rightsquigarrow} \overline{d} \qquad m \overset{module}{\rightsquigarrow} \overline{d'}}{\texttt{functor}(\mathcal{X} : mt) \rightarrow m \overset{module}{\rightsquigarrow} \texttt{scope}\ \mathcal{X}\ \overline{d}\ \texttt{end}\ \overline{d'}}$$

$$(\textsc{MTSig})\ \frac{\overline{s} \overset{signature}{\rightsquigarrow} \overline{d}}{\texttt{sig}\ s\ \texttt{end} \overset{module\ type}{\rightsquigarrow} \overline{d}}$$

Figure 6: Translation of OCaml module expressions and module types into WhyML.

D Translation of signatures

$$(\textsc{SVal})\ \frac{\neg is\_ghost(\mathcal{A}) \qquad kind(\mathcal{A}) = \mathcal{K} \\ \mathcal{A}' \overset{function\ spec}{\rightsquigarrow} \mathcal{S} \qquad \pi, \mathcal{A}' \overset{function\ args}{\rightsquigarrow} \overline{x : \pi'}, \pi_{res}}{\texttt{val}\ \mathcal{A}\ f : \pi\ \mathcal{A}' \overset{signature}{\rightsquigarrow} \texttt{val}\ \mathcal{K}\ f(\overline{x : \pi})\ \mathcal{S} : \pi_{res}}$$

$$(\textsc{SValGhost})\ \frac{\neg is\_ghost(\mathcal{A}) \qquad kind(\mathcal{A}) = \mathcal{K} \\ \mathcal{A}' \overset{function\ spec}{\rightsquigarrow} \mathcal{S} \qquad \pi, \mathcal{A}' \overset{function\ args}{\rightsquigarrow} \overline{x : \pi'}, \pi_{res}}{\texttt{val}\ \mathcal{A}\ f : \pi\ \mathcal{A}' \overset{signature}{\rightsquigarrow} \texttt{val}\ \mathcal{K}\ \texttt{ghost} f(\overline{x : \pi})\ \mathcal{S} : \pi_{res}}$$

$$(\textsc{SType})\ \frac{td_0 \overset{type\ definition}{\rightsquigarrow} td_0' \qquad \overline{td_1} \overset{type\ definition}{\rightsquigarrow} \overline{td_1'}}{\texttt{type}\ td_0\ \overline{\texttt{and}\ td_1} \overset{signature}{\rightsquigarrow} \texttt{type}\ td_0'\ \overline{\texttt{with}\ td_1'}}$$

Figure 7: Translation of OCaml signatures into WhyML.