Translation of *MiniMaple* to Why3ML*

Muhammad Taimoor Khan
Doktoratskolleg Computational Mathematics
and
Research Institute for Symbolic Computation
Johannes Kepler University
Linz, Austria
Muhammad.Taimoor.Khan@risc.jku.at

February 13, 2013

Abstract

In this paper, we give the complete definition of the translation of *MiniMaple* and its specification language to an intermediate language Why3ML of verification calculus Why3. For the verification, we first translate *MiniMaple* annotated program into a semantically equivalent Why3ML program, then verification conditions are generated by using Why3 corresponding verification conditions generator. Finally, the correctness of the generated verification conditions can be proved by various Why3 back-end supported automatic and interactive theorem provers.

*The research was funded by the Austrian Science Fund (FWF): W1214-N15, project DK10.

Contents

1 Introduction

The task of verification of *MiniMaple* annotated program requires the generation of verification conditions to be proved correct later by using some theorem prover. Verification conditions can either be generated by our own as in RISC ProgramExplorer [10] developed at our institute RISC or by using some existing intermediate verification frameworks, e.g. Why3 [3] developed by LRI, France or Boogie [1] developed by Microsoft, USA. Based on our investigations we decided to use Why3 as a verification tool. The goal here is to translate a *MiniMaple* annotated program into a semantically equivalent Why3ML program. Therefore we have defined a translation of *MiniMaple* and its specification language based on the formal semantics of *MiniMaple* [6] and its specification language [7].

Why3 is an intermediate verification tool with Why3ML as a corresponding programming language. Why3 has a verification condition generator as its main component. However, the generated verification conditions are then translated into corresponding specification language called Why. The Why program can later be translated to various back-end theorem provers as provided in [5]. Currently, various front-end tools are using Why3 as a generic intermediate verification platform, e.g. Krakatoa (for Java programs) and Frama-C (for C programs). Why3ML is a functional language influenced by ML that supports pattern matching, inductive predicates, algebraic data types and also supports typical imperative constructs, e.g. loops, sequence and exceptions. Also Why3 support various automated decision procedures (e.g. Z3 and CVC3) and proof assistants (e.g. Coq). The collaborative proof support was one the reasons why we chose Why3, as our main test case, the Maple package *Difference-Differential* [4] developed by Christian Döench, which deals with non-linear arithmetic and requires in general an interactive theorem prover. The existence of a weakest precondition based formal semantics of Why3ML is the other reason for choosing this tool, because one can reasonably argue whether the generated verification conditions are sound with respect to the *MiniMaple* and its specification semantics.

The translation of *MiniMaple* and its specification language to Why3ML has the following main challenges:

- Formalization of *MiniMaple* abstract and other non-standard data types, e.g. union, uneval and symbol etc.
- Support for the formalization of *MiniMaple* **return** statement.
- Axiomatization of numeral quantifiers (**add**, **mul**, **max** and **min**) to compute a certain binary operation for the filtered values (for a given property) of a specification expression as supported in the specification language of *MiniMaple*.
- Also the formalization of a sequence quantifier (**seq**) to represent a sequence of values of a given expression.
- Formalization of semantic domains of values for non-standard types of objects, for example symbol, uneval and union etc.
- Axiomatization of additional functions and predicates as supported by *MiniMaple*, for example type tests i.e. $\mathbf{type}(E, T)$.

The rest of the paper is organized as follows: in Section 2, we discuss the overview of our translation of *MiniMaple* and its specification language. Section 3 presents conclusions and future work. Appendix A and B give the formal syntax of *MiniMaple* and its specification language respectively. Also with the permission of authors, we have reproduced Appendix C which gives the Why3 language reference [2], while Appendix D contains the translation domains, signatures of translation functions, definition of translation functions and the definition of auxiliary functions and predicates.

2 Translation Overview

In this section, we describe the guidelines to read the different sections of Appendix D with the help of some examples. Each of the following subsections presents the corresponding section of the Appendix D. We start by giving the definition of different translation domains.

2.1 Domains

This section declares the different syntactic domains used in the translation. Note, the syntactic domains of *MiniMaple* and its specification language are also included. For the readability of this document, we have also included the corresponding formal syntax of Why3ML and Why in Appendix C. We have used "Why3" as a prefix for the corresponding Why3ML and Why constructs. For example, Why3 specification expression is defined as a disjoint union of corresponding formula and a term as follows:

$$\text{Why3-SExpr} = \text{Why3-Formula} + \text{Why3-Term}$$

2.2 Signatures of Translation Functions

A translation function T is defined which takes a corresponding *MiniMaple* and its specification syntactic domain with some other parameters and returns the corresponding translated Why3 constructs with some additional result.

In this subsection, we declare the signatures of various translation functions for the syntactic domains of annotated *MiniMaple* programs.

2.2.1 Program

The translation function T for the abstract syntax domain annotated program values of P is defined as follows:

$$\mathrm{T}[\![\mathrm{P}]\!]\colon \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \to \textit{Why3-File}$$

The function translates an annotated *MiniMaple* program P to a corresponding Why3 file with a given type environment, Why3 type environment, module (global) declarations and theory as parameters. Here the parameter values of module declarations might be undefined.

2.2.2 Declaration

The translation function T for the abstract syntax domain program declaration values of D is defined as follows:

$$\mathrm{T}[\![\mathrm{D}]\!]: \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{Why3-Theory-DeclU} \times \textit{Why3-Environment}$$

The translation function for program declarations D produces a corresponding Why3 theory declarations and environment with a given type environment, Why3 type environment, module (global) declarations and theory as parameters. Here the result of theory declarations can be empty.

2.2.3 Rule

The translation function T for the abstract syntax domain rule values of R is defined as follows:

$$\mathrm{T}[\![\mathrm{R}]\!]: \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{Why3-SExprU} \times \textit{Why3-Environment} \times \textit{Why3-Theory}$$

The translation function for function definition rules R produces a corresponding Why3 specification expression along-with Why3 environment and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters. Here the resulting specification expression can be undefined.

2.2.4 Procedure Specification

The translation function T for the abstract syntax domain procedure specification values of PS is defined as follows:

$$\mathrm{T}[\![\mathrm{PS}]\!]: \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{Why3-SExpr} \times \textit{Why3-SExpr} \times \textit{Why3-SExpr} \times \textit{Identifier\_Sequence} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory}$$

The translation function for procedure specification PS produces corresponding Why3 pre-condition, post-condition and exceptional behavior expressions along-with an identifier sequence representing assignable global variables, a Why3 environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters. Here the resulting module declarations can be undefined.

2.2.5 Exception Clause

The translation function T for the abstract syntax domain exception clause values of EC is defined as follows:

$$\mathrm{T}[\![\mathrm{EC}]\!]: \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{Why3-Formula} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory}$$

The translation function for exception clause EC produces corresponding Why3 formula, environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters.

2.2.6 Loop Specification

The translation function T for the abstract syntax domain loop specification values of LS is defined as follows:

$$\begin{aligned}\mathrm{T}[\![\mathrm{LS}]\!]: \mathit{Environment} \times \mathit{Why3\text{-}Environment} \times \mathit{Why3\text{-}MDeclU} \times \mathit{Why3\text{-}Theory} \\ \rightarrow \mathit{Why3\text{-}Formula} \times \mathit{Why3\text{-}TermU} \times \\ \mathit{Why3\text{-}Environment} \times \mathit{Why3\text{-}MDeclU} \times \mathit{Why3\text{-}Theory}\end{aligned}$$

The translation function for loop specification LS produces corresponding Why3 invariant formula, termination term, a Why3 environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters. Here the resulting termination term can be empty, which is not required in the case of for-loops.

2.2.7 Assertion

The translation function T for the abstract syntax domain assertion values of A is defined as follows:

$$\begin{aligned}\mathrm{T}[\![\mathrm{A}]\!]: \mathit{Environment} \times \mathit{Why3\text{-}Environment} \times \mathit{Why3\text{-}MDeclU} \times \mathit{Why3\text{-}Theory} \\ \rightarrow \mathit{Why3\text{-}Formula} \times \mathit{Why3\text{-}Environment} \times \mathit{Why3\text{-}MDeclU} \times \mathit{Why3\text{-}Theory}\end{aligned}$$

The translation function for assertion A produces a corresponding Why3 formula, an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters.

2.2.8 Specification Expression Sequence

The translation function T for the abstract syntax domain specification expression values of SEseq is defined as follows:

$$\begin{aligned}\mathrm{T}[\![\mathrm{SEseq}]\!]: \mathit{Environment} \times \mathit{Why3\text{-}Environment} \times \mathit{Why3\text{-}MDeclU} \\ \times \mathit{Why3\text{-}Theory} \rightarrow \\ \mathit{Why3\text{-}SExprU} \times \mathit{Why3\text{-}Environment} \times \mathit{Why3\text{-}MDeclU} \times \mathit{Why3\text{-}Theory}\end{aligned}$$

The translation function for specification expression sequence SEseq produces a corresponding Why3 specification expression, an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters. Here the resulting Why3 specification expression can be empty.

2.2.9 Specification Expression

The translation function T for the abstract syntax domain specification expression values of SEseq is defined as follows:

$$\begin{aligned}\mathrm{T}[\![\mathrm{SE}]\!]: \mathit{Environment} \times \mathit{Why3\text{-}Environment} \times \mathit{Why3\text{-}MDeclU} \times \mathit{Why3\text{-}Theory} \\ \rightarrow \mathit{Why3\text{-}SExpr} \times \mathit{Why3\text{-}Environment} \times \mathit{Why3\text{-}MDeclU} \times \mathit{Why3\text{-}Theory}\end{aligned}$$

The translation function for specification expression SE produces a corresponding Why3 specification expression, an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters.

2.2.10 Command Sequence

The translation function T for the abstract syntax domain command sequence values of Cseq is defined as follows:

$$\mathrm{T}[\![\text{Cseq}]\!]: \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{Why3-ExprU} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory}$$

The translation function for command sequence Cseq produces a corresponding Why3 expression (sequence), an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters. Here the resulting Why3 expression can be empty.

2.2.11 Command

The translation function T for the abstract syntax domain command values of C is defined as follows:

$$\mathrm{T}[\![\text{C}]\!]: \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{Why3-Expr} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory}$$

The translation function for command C produces a corresponding Why3 expression, an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters.

2.2.12 Elif

The translation function T for the abstract syntax domain else-if values of Elif is defined as follows:

$$\mathrm{T}[\![\text{Elif}]\!]: \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{Else-Conditional} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory}$$

The translation function for else-if conditional Elif produces a corresponding Why3 else-conditional construct, an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters.

2.2.13 Catch

The translation function T for the abstract syntax domain catch values of Catch is defined as follows:

$$\mathrm{T}[\![\text{Catch}]\!]: \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{HandlerSeq} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory}$$

The translation function for catch-construct Catch produces a corresponding Why3 handler sequence, an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters.

2.2.14 Expression Sequence

The translation function T for the abstract syntax domain expression sequence values of Eseq is defined as follows:

$$\mathrm{T}[\![\text{Eseq}]\!]: \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{Why3-ExprU} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory}$$

The translation function for expression sequence Eseq produces a corresponding Why3 expression (sequence), an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters. Here the resulting Why3 expression can be empty for empty expression sequence.

2.2.15 Expression

The translation function T for the abstract syntax domain expression values of E is defined as follows:

$$\mathrm{T}[\![\text{E}]\!]: \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{Why3-Expr} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory}$$

The translation function for expression E produces a corresponding Why3 expression, an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters.

2.2.16 Special Expression

The translation function T for the abstract syntax domain special expression (operator) values of Esop is the same as defined for expression values above as follows:

$$\mathrm{T}[\![\text{Esop}]\!]: \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{Why3-Expr} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory}$$

2.2.17 Declaration Sequence

The translation function T for the abstract syntax domain (procedure/module) declaration sequence values of S is defined as follows:

$$\mathrm{T}[\![\text{S}]\!]: \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{Why3-ExprU} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory}$$

The translation function for declaration sequence S produces a corresponding Why3 expression (sequence), an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters. Here the resulting Why3 expression can be empty for empty declarations.

2.2.18 Recurrence

The translation function T for the abstract syntax domain recurrence (procedure/module body) values of R is defined as follows:

$$\mathrm{T}[\![\mathrm{R}]\!]: \mathit{Environment} \times \mathit{Why3\text{-}Environment} \times \mathit{Why3\text{-}MDeclU} \times \mathit{Why3\text{-}Theory} \rightarrow \mathit{Why3\text{-}ExprU} \times \mathit{Why3\text{-}Environment} \times \mathit{Why3\text{-}MDeclU} \times \mathit{Why3\text{-}Theory}$$

The translation function for procedure/module body R produces a corresponding Why3 expression (sequence), an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters.

2.2.19 Parameter Sequence

The translation function T for the abstract syntax domain parameter sequence values of Pseq is defined as follows:

$$\mathrm{T}[\![\mathrm{Pseq}]\!]: \mathit{Environment} \times \mathit{Why3\text{-}Environment} \times \mathit{Why3\text{-}MDeclU} \times \mathit{Why3\text{-}Theory} \rightarrow \mathit{Why3\text{-}ExprU} \times \mathit{Why3\text{-}Environment} \times \mathit{Why3\text{-}MDeclU} \times \mathit{Why3\text{-}Theory}$$

The translation function for parameter sequence Pseq produces a corresponding Why3 expression (sequence), an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters. Here the resulting Why3 expression can be empty for empty parameter sequence.

2.2.20 Parameter

The translation function T for the abstract syntax domain parameter values of P is defined as follows:

$$\mathrm{T}[\![\mathrm{P}]\!]: \mathit{Environment} \times \mathit{Why3\text{-}Environment} \times \mathit{Why3\text{-}MDeclU} \times \mathit{Why3\text{-}Theory} \rightarrow \mathit{Why3\text{-}Expr} \times \mathit{Why3\text{-}Environment} \times \mathit{Why3\text{-}MDeclU} \times \mathit{Why3\text{-}Theory}$$

The translation function for parameter P produces a corresponding Why3 expression, an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters.

2.2.21 Parameter Modifier

The translation function T for the abstract syntax domain parameter modifier values of M is defined as follows:

$$\mathrm{T}[\![\mathrm{M}]\!]: \mathit{Environment} \times \mathit{Why3\text{-}Environment} \times \mathit{Why3\text{-}MDeclU} \times \mathit{Why3\text{-}Theory} \rightarrow \mathit{Why3\text{-}Type} \times \mathit{Why3\text{-}Environment} \times \mathit{Why3\text{-}MDeclU} \times \mathit{Why3\text{-}Theory}$$

The translation function for parameter modifier M produces a corresponding Why3 type, an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters.

2.2.22 Identifier Sequence

The translation function T for the abstract syntax domain identifier sequence values of Iseq is defined as follows:

$$\mathrm{T}[\![\text{Iseq}]\!]: \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory}$$

The translation function for identifier sequence Iseq produces a corresponding Why3 environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters.

2.2.23 Identifier

The translation function T for the abstract syntax domain identifier values of I is defined as follows:

$$\mathrm{T}[\![\text{I}]\!]: \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory}$$

The translation function for identifier I produces a corresponding Why3 environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters.

2.2.24 Identifier Typed Sequence

The translation function T for the abstract syntax domain identifier typed sequence values of Itseq is defined as follows:

$$\mathrm{T}[\![\text{Itseq}]\!]: \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{Why3-ExprU} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory}$$

The translation function for identifier typed sequence Itseq produces a corresponding Why3 expression (sequence), an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters. Here the resulting Why3 expression can be empty for empty identifier typed sequence.

2.2.25 Identifier Typed

The translation function T for the abstract syntax domain identifier typed values of It is defined as follows:

$$\mathrm{T}[\![\text{It}]\!]: \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{Why3-Expr} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory}$$

The translation function for identifier typed It produces a corresponding Why3 expression, an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters.

2.2.26 Type Sequence

The translation function T for the abstract syntax domain type sequence values of Tseq is defined as follows:

$$\mathrm{T}[\![\mathrm{Tseq}]\!]\text{: } \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{Why3-TypeU} \times \textit{Why3-Environment} \times \textit{Why3-MDecl} \times \textit{Why3-Theory}$$

The translation function for parameter sequence Pseq produces a corresponding Why3 type (sequence), an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters. Here the resulting Why3 type can be empty for empty type sequence.

2.2.27 Type

The translation function T for the abstract syntax domain type values of T is defined as follows:

$$\mathrm{T}[\![\mathrm{T}]\!]\text{: } \textit{Environment} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory} \rightarrow \textit{Why3-Type} \times \textit{Why3-Environment} \times \textit{Why3-MDeclU} \times \textit{Why3-Theory}$$

The translation function for type T produces a corresponding Why3 type, an environment, module declarations and a theory with a given type environment, Why3 type environment, module (global) declarations and theory as parameters.

In the following section we give the definition of the auxiliary functions and predicates used in the translation of the associated domains of *MiniMaple* and its specification language.

2.3 Auxiliary Functions and Predicates

This section defines the corresponding auxiliary functions and predicates which are used in the definition of translation functions for various domains of *MiniMaple* and its specification language.

2.4 Definitions of Translation Functions

In this section we give the definition of the translation functions of the various syntactic domains (and associated domains) of *MiniMaple*, e.g. command and expression. The semantics of other domains of the specification language are very simple and can be easily rehearsed.

2.4.1 Command Translation

The translation of a *MiniMaple* command produces a Why3 expression alongwith environment, module declarations and theory.

Assignment The *MiniMaple* assignment command is a simultaneous assignment. The translation of assignment command is defined as nested local bindings (LET-IN constructs of Why3ML) in the same order as appeared in the actual statement.

While Loop *MiniMaple* supports the typical while-loop, which is translated into a corresponding typical Why3ML supported while-loop.

For Loops Also *MiniMaple* supports following four variations of for-loops.

1. The first variation is a for-loop **for** I **in** E **do** $Cseq$ **end do**. First the for-loop condition is translated into an equivalent condition, then an auxiliary variable is introduced which is used as a loop increment. Also in each iteration variable I is assigned the next value of the corresponding translated expression from E.
2. The second variation is a for-while-loop **for** I **in** E_1 **while** E_2 **do** $Cseq$ **end do**. The while-loop conditions is translated first and then combined with the for-loop to form an equivalent Why3ML while-loop.
3. The third variation is another typical for-loop **for** I **from** E_1 **by** E_2 **to** E_3 **do** $Cseq$ **end do**. The corresponding translation is defined by introducing auxiliary variables to represent the corresponding translated expression values from expressions E_1 and E_2.
4. The fourth variation is another for-while-loop **for** I **from** E_1 **by** E_2 **to** E_3 **while** E_4 **do** $Cseq$ **end do**. The corresponding loop conditions are translated and combined in a single while-loop condition. Also auxiliary variables are introduced with local binding to represent the iterated values of the expressions.

Conditionals Why3ML supports both one-sided and two-sided conditionals so *MiniMaple* conditionals are translated into corresponding typical Why3ML conditionals.

Return Why3ML doesn't support return statement, so corresponding translation is defined by exceptional-handling mechanism of Why3ML. We introduce an exception object, and assign a value and raise this exception where-ever the return statement occurs. Also corresponding handler returns the value.

Error The error command raises an exception and is directly translated into a corresponding Why3ML exception-raising statement.

Exception Handler The try-command provides a mechanism for executing commands in a controlled environment. The translation is defined to corresponding WhyMl exception handler construct.

Procedure call and type declaration commands are translated into equivalent Why3ML constructs.

2.4.2 Expression Translation

The translation of a *MiniMaple* expression produces a corresponding Why3ML expression with some additional values of a Why3 environment, module declarations and a theory.

Procedure *MiniMaple* procedure is translated using a LET-IN construct of Why3ML.

Module Also *MiniMaple* module is defined as a corresponding Why3ML module construct.

Binary *MiniMaple* supports typical arithmetic (addition, subtraction, multiplication, division and mod) and logical (less, greater, less equal and greater equal) binary operations. As a special case binary operations of equality and non-equality are defined separately. We define the corresponding binary functions for each type introduced in the program as a Why3ML logic function.

Unary Also *MiniMaple* supports typical arithmetic (plus and minus) and logical (negation) unary operations. The unary functions are also defined for the respective types into equivalent Why3ML logic function.

Logical *MiniMaple* supports logical-and and logical-or binary expressions. These logical expressions are translated into typical Why3ML logical expressions.

Equality Also *MiniMaple* supports binary expressions of equality and non-equality. The corresponding equality and non-equality of types is axiomatized as a corresponding Why3ML function.

Type-Test A type-test is a special feature of *MiniMaple*, which tests if the expression E is of type T. We translate type tests using pattern matching respective match-construct of Why3ML.

Special Expressions *MiniMaple* also supports some special expressions. All of the special expressions of *MiniMaple* are defined as Why3ML functions except for the built-in types of Why3ML.

The translation of all other expressions is a very easy exercise.

3 Conclusions and Future Work

In this paper we gave the definition of translation of *MiniMaple* and its specification language into a corresponding Why3ML program including associated domains, declaration and definitions of translation functions. The challenge here is semantically preserved translation into equivalent Why3ML program. We have formulated the soundness statement of our translation, though the actual proof for the corresponding soundness is planned as a future goal. In the following we formulate the example soundness statement of *MiniMaple* commands and discuss the sketch of corresponding proof:

$$
\begin{aligned}
&\forall\ C \in Command, C_1 \in Exp_w, P, Q \in SpecExp_w \\
&\quad s_1, s_2 \in State_m, s_3, s_4 \in State_w, \\
&\quad e \in Env_m, e_1 \in Env_w ... : \\
&\quad (C_1, e_1, ...) = \mathrm{T}\ [\![C]\!](e, ...) \wedge P = WP(C_1,\ Q)(e_1, ...) \wedge s_1 \equiv s_3 \wedge s_2 \equiv s_4 \\
&\quad \Rightarrow [\![C]\!](e)(s_1, s_2) \wedge [\![P]\!](e_1)(s_3) \\
&\qquad\qquad\qquad\qquad \Rightarrow [\![Q]\!](e_1)(s_4)
\end{aligned}
$$

In the following we discuss the main steps for the soundness proof:

- If we translate a *MiniMaple* command C in an environment e into an equivalent Why3ML expression C_1 and an environment e_1:

$$(C_1, e_1, ...) = \mathrm{T}\ [\![C]\!](e, ...)$$

- and compute, for an arbitrary postcondition Q, the weakest precondition P of the corresponding Why3ML expression C_1 with respect to Q in the environment e_1, where P and Q are Why3 specification expressions:

$$P = WP(C_1,\, Q)(e_1, ...)$$

- and translate the *MiniMaple* command's pre- and post-states s_1 and s_2 into corresponding Why3ML pre- and post-states s_3 and s_4 respectively,
- then if the execution of a *MiniMaple* command C in a pre-state s_1 results in a post-state s_2 and the computed pre-condition P holds in the equivalent pre-state s_3,
- then the corresponding post-condition Q must hold in the equivalent post-state s_4.

In the last step, we assume that the real execution of the translated command $\mathrm{T}[\![C]\!]$ produces s_4' as a post-state in which the post-condition Q does not hold. This implies that the computed weakest precondition P as $WP(C_1, Q)$ can not be the weakest precondition of the translated Why3ML expression C_1 in post-state s_4, which is a contradiction and hence proves the soundness.

Also we have implemented the corresponding translator in Java and have applied its results to some parts of the Maple package *DifferenceDifferential*. As a next step we will translate respective verify a substantial part of our test package. Here, we are mainly interested in detecting violations of methods preconditions.

Acknowledgment

The author cordially thanks Wolfgang Schreiner for his valuable and constructive comments and suggestions throughout this work.

References

[1] Mike Barnett, Boryuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.

[2] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 platform, version 0.72*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.72 edition, May 2012. `https://gforge.inria.fr/docman/view.php/2990/7919/manual-0.72.pdf`.

[3] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.

[4] Christian Dönch. Bivariate Difference-Differential Dimension Polynomials and Their Computation in Maple. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University (JKU), Linz, 2009.

[5] Jean-Christophe Filliâtre. Verifying Two Lines of C with Why3: an Exercise in Program Verification. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, Philadelphia, USA, January 2012.

[6] Muhammad Taimoor Khan. Formal Semantics of *MiniMaple*. DK Technical Report 2012-01, Research Institute for Symbolic Computation, University of Linz, January 2012.

[7] Muhammad Taimoor Khan. On the Formal Semantics of *MiniMaple* and its Specification Language. In *Proceedings of Frontiers of Information Technology*, pages 169–174. IEEE Computer Society, 2012.

[8] Muhammad Taimoor Khan and Wolfgang Schreiner. On Formal Specification of Maple Programs. In Jacques Carette Gabriel Dos Reis Petr Sojka Makarius Wenzel Volker Sorge Johan Jeuring, John A. Campbell, editor, *Intelligent Computer Mathematics*, volume 7362 of *LNCS*, pages 443–447. Springer, 2012.

[9] Muhammad Taimoor Khan and Wolfgang Schreiner. Towards the Formal Specification and Verification of Maple Programs. In Jacques Carette Gabriel Dos Reis Petr Sojka Makarius Wenzel Volker Sorge Johan Jeuring, John A. Campbell, editor, *Intelligent Computer Mathematics*, volume 7362 of *LNCS*, pages 231–247. Springer, 2012.

[10] Wolfgang Schreiner. Computer-Assisted Program Reasoning Based on a Relational Semantics of Programs. In Pedro Quaresma and Ralph-Johan Back, editors, *Proceedings First Workshop on CTP Components for Educational Software (THedu'11)*, number 79 in Electronic Proceedings in Theoretical Computer Science (EPTCS), pages 124–142, Wroclaw, Poland, July 31, 2011, February 2012.

Appendices

Appendix A and Appendix B give the formal abstract syntax (language grammar) for *MiniMaple* and its specification language respectively which is different than the one given in [8, 9]. Appendix C gives the Why3 language reference, which is reproduced with the permission from authors. Appendix D gives the complete definition of translation of the *MiniMaple* and its specification language in addition with auxiliary functions and predicates.

A Formal Syntax of *MiniMaple*

In this appendix we give the formal abstract syntax (language grammar) of *MiniMaple*.

Prog ∈ Program
Cseq ∈ Command Sequence
C ∈ Command
Elif ∈ ElseIf
Catch ∈ Catch
Eseq ∈ Expression Sequence
E ∈ Expression
S ∈ Sequence
R ∈ Recurrence
Pseq ∈ Parameter Sequence
P ∈ Parameter
M ∈ Modifiers
Iseq ∈ Identifier Sequence
I ∈ Identifier
Itseq ∈ Identifier Typed Sequence
It ∈ Identifier Typed
Bop ∈ Boolean Operators
Uop ∈ Unary Operators
Esop ∈ Especial Operators
Tseq ∈ Type Sequence
T ∈ Type
N ∈ Numeral

Prog ::= Cseq
Cseq ::= EMPTY | (C; | E;)Cseq
C ::= **if** E **then** Cseq Elif **end if**; | **if** E **then** Cseq Elif **else** Cseq **end if**;
| **while** E **do** Cseq **end do**;
| **for** I **in** E **do** Cseq **end do**;
| **for** I **in** E **while** E **do** Cseq **end do;**
| **for** I **from** E **by** E **to** E **do** Cseq **end do**;
| **for** I **from** E **by** E **to** E **while** E **do** Cseq **end do;**
| **return** E; | **return**; | **error;** | **error** I,Eseq;
| **try** Cseq Catch **end;** | **try** Cseq Catch **finally** Cseq **end;**
| I,Iseq := E,Eseq; | E(Eseq); | **‘type/I‘** := T; | **print**(e);
Elif ::= EMPTY | **elif** E **then** Cseq;Elif

Catch ::= EMPTY | **catch** “I” **:**Cseq, Catch
Eseq ::= EMPTY | E,Eseq
E ::= I | N | **module()** S;R **end module**;
| **proc**(Pseq) S;R **end proc**;| **proc**(Pseq)::T; S;R **end proc**;
| E1 Bop E2 | Uop E | Esop | E1 **and** E2 | E1 **or** E2 | E(Eseq)
| I1:-I2 | E,E,Eseq | **type(** I,T **)** | E1 = E2 | E1 <> E2
S ::= EMPTY | **local** It,Itseq;S | **global** I,Iseq;S | **uses** I,Iseq;S
| **export** It,Itseq;S
R ::= Cseq | Cseq;E
Pseq ::= EMPTY | P,Pseq
P ::= I | I :: M
M ::= **seq(** T **)** | T
Iseq :: = EMPTY | I, Iseq
I ::= any valid Maple name
Itseq :: = EMPTY | It, Itseq
It ::= I | I :: T | I := E | I::T:=E
Bop ::= +| − | / | ∗ | **mod** | < | > | ≤ | ≥
Uop ::= **not** | − |+
Esop ::= **op(** E1, E2 **)** | **op(** E **)** | **op(** E..E, E **)** | **nops(** E **)**
| **subsop(** E1=E2, E3 **)** | **subs(** I=E1, E2 **)** | “ E “ | [Eseq]
| I[Eseq] | **seq(**E, I = E..E **)** | **seq(**E, I **in** E**)** | **eval(** I,1 **)**
Tseq ::= EMPTY | T,Tseq
T ::= **integer** | **boolean** | **string** | **float** | **rational** | **anything** | **{** T **}**
| **list(** T **)** | [Tseq] | **procedure**[T]**(** Tseq **)**
| I(Tseq) | **Or(** Tseq **)** | **symbol** | **void** | **uneval** | I
N ::= a sequence of decimal digits

B Formal Syntax of a Specification Language for *MiniMaple*

In this appendix we give the formal abstract syntax (language grammar) of the specification language for *MiniMaple*.

decl ∈ Declaration
proc-spec ∈ Procedure_Specification
loop-spec ∈ Loop_Specification
asrt ∈ Assertion
rules ∈ Rules
excep-clause ∈ Exception_Clause
eseq ∈ Specification_Expression_Sequence
spec-expr ∈ Specification_Expression
binding ∈ Binding
Itseq ∈ Identifier_Typed_Sequence
It ∈ Identifier_Typed
Iseq ∈ Identifier_Sequence
I ∈ Identifier
Bop ∈ Binary_Operator
Uop ∈ Unary_Operator
it-op ∈ Iteration_Operator

esop ∈ Especial_Operator
sel-op ∈ Selection_Operator
Tseq ∈ Type_Sequence
T ∈ Type
N ∈ Numeral

decl ::= EMPTY | (**define**(I(Itseq)::T,rules);
| **‘type/**I‘;
| **‘type/**I‘:=T;
| **assume**(spec-expr);) decl
proc-spec ::= **requires** spec-expr; **global** Iseq; **ensures** spec-expr; excep-clause
loop-spec := **invariant** spec-expr; **decreases** spec-expr;
asrt := **ASSERT**(spec-expr, (EMPTY | “I“));
rules ::= EMPTY | I(Itseq) = spec-expr, rules
excep-clause ::= EMPTY | **exceptions** ”I” spec-expr; excep-clause
eseq ::= EMPTY | spec-expr, eseq
spec-expr ::= I **(** eseq **)** | **type**(spec-expr,T) | spec-expr **and** spec-expr | spec-expr **or** spec-expr
| spec-expr **equivalent** spec-expr | spec-expr **implies** spec-expr
| **forall**(Itseq, spec-expr) | **exists**(Itseq, spec-expr)
| **(**spec-expr**)** | spec-expr Bop spec-expr | Uop spec-expr | esop
| it-op(spec-expr, binding, (EMPTY | spec-expr))
| **true** | **false** | **LET** Iseq=eseq **IN** spec-expr | **RESULT**
| **‘if‘**(spec-expr1, spec-expr2, spec-expr3) | I | I1:-I2 | **OLD** I | N
| spec-expr1 = spec-expr2 | spec-expr1 <> spec-expr2
binding ::= I = spec-expr1...spec-expr2 | I **in** spec-expr
Itseq ::= EMPTY | It, Itseq
It ::= I::T
Iseq :: = EMPTY | I, Iseq
I ::= any valid Maple name
Bop ::= +| − | / | ∗ | **mod** | < | > | ≤ | ≥ | = | <>
Uop ::= **not** | − |+
it-op ::= **add** | **mul** | **max** | **min** | **seq**
esop ::= **op(** spec-expr1, spec-expr2 **)** | **op(** spec-expr **)**
| **op(** spec-expr..spec-expr, spec-expr **)** | **nops(** spec-expr **)**
| **subsop(** spec-expr1=spec-expr2, spec-expr3 **)**
| **subs(** I=spec-expr1, spec-expr2 **)** | **“** spec-expr **“**
| I sel-op | [eseq] | { eseq } | I(eseq) | **eval(** I,1 **)**
sel-op ::= EMPTY | [eseq] sel-op
Tseq ::= EMPTY | T,Tseq
T ::= **integer** | **boolean** | **string** | **float** | **rational** | **anything** | **{** T **}**
| **list(** T **)** | [Tseq] | **procedure**[T]**(** Tseq **)**
| I(Tseq) | **Or(** Tseq **)** | **symbol** | **void** | **unevaluated** | I
N ::= a sequence of decimal digits

C Why3 Language Reference Manual

This section gives the formal syntax of Why3ML and Why3 languages [2]. This section is reproduced with the permission of corresponding authors.

Chapter 7

Language Reference

This chapter gives the grammar and semantics for Why3 and Why3ML input files.

7.1 Lexical conventions

Lexical conventions are common to Why3 and Why3ML.

Comments. Comments are enclosed by `(*` and `*)` and can be nested.

Strings. Strings are enclosed in double quotes (`"`). Double quotes can be inserted in strings using the backslash character (\). In the following, strings are referred to with the non-terminal *string*.

Identifiers. The syntax distinguishes lowercase and uppercase identifiers and, similarly, lowercase and uppercase qualified identifiers.

lalpha	::=	`a - z` | `_`
ualpha	::=	`A - Z`
alpha	::=	*lalpha* | *ualpha*
lident	::=	*lalpha* (*alpha* | *digit* | `'`)*
uident	::=	*ualpha* (*alpha* | *digit* | `'`)*
ident	::=	*lident* | *uident*
lqualid	::=	*lident* | *uqualid* `.` *lident*
uqualid	::=	*uident* | *uqualid* `.` *uident*

Constants. The syntax for constants is given in Figure 7.1. Integer and real constants have arbitrary precision. Integer constants may be given in base 16, 10, 8 or 2. Real constants may be given in base 16 or 10.

Operators. Prefix and infix operators are built from characters organized in four categories (*op-char-1* to *op-char-4*).

digit	::=	`0 - 9`	
hex-digit	::=	*digit* | `a - f` | `A - F`	
oct-digit	::=	`0 - 7`	
bin-digit	::=	`0` | `1`	
integer	::=	*digit* (*digit* | `_`)*	decimal
	|	(`0x` | `0X`) *hex-digit* (*hex-digit* | `_`)*	hexadecimal
	|	(`0o` | `0O`) *oct-digit* (*oct-digit* | `_`)*	octal
	|	(`0b` | `0B`) *bin-digit* (*bin-digit* | `_`)*	binary
real	::=	*digit*$^+$ *exponent*	decimal
	|	*digit*$^+$ `.` *digit** *exponent*$^?$	
	|	*digit** `.` *digit*$^+$ *exponent*$^?$	
	|	(`0x` | `0X`) *hex-real* *h-exponent*	hexadecimal
hex-real	::=	*hex-digit*$^+$	
	|	*hex-digit*$^+$ `.` *hex-digit**	
	|	*hex-digit** `.` *hex-digit*$^+$	
exponent	::=	(`e` | `E`) (`-` | `+`)$^?$ *digit*$^+$	
h-exponent	::=	(`p` | `P`) (`-` | `+`)$^?$ *digit*$^+$	

Figure 7.1: Syntax for constants.

op-char-1	::=	`=` | `<` | `>` | `~`
op-char-2	::=	`+` | `-`
op-char-3	::=	`*` | `/` | `%`
op-char-4	::=	`!` | `$` | `&` | `?` | `@` | `^` | `.` | `:` | `|` | `#`
op-char	::=	*op-char*-1 | *op-char*-2 | *op-char*-3 | *op-char*-4
infix-op-1	::=	*op-char** *op-char*-1 *op-char**
infix-op	::=	*op-char*$^+$
prefix-op	::=	*op-char*$^+$
bang-op	::=	`!` *op-char*-4* | `?` *op-char*-4*

Infix operators are classified into 4 categories, according to the characters they are built from:

- level 4: operators containing only characters from *op-char-4*;
- level 3: those containing characters from *op-char-3* or *op-char-4*;
- level 2: those containing characters from *op-char-2*, *op-char-3* or *op-char-4*;
- level 1: all other operators (non-terminal *infix-op-1*).

Labels. Identifiers, terms, formulas, program expressions can all be labeled, either with a string, or with a location tag.

label	::=	*string*	
	|	`#` *filename* *digit*$^+$ *digit*$^+$ *digit*$^+$ `#`	
filename	::=	*string*	

A location tag consists of a file name, a line number, and starting and ending characters.

7.2 Why3 Language

Terms. The syntax for terms is given in Figure 7.2. The various constructs have the following priorities and associativities, from lowest to greatest priority:

construct	associativity
`if then else` / `let in`	–
label	–
cast	–
infix-op level 1	left
infix-op level 2	left
infix-op level 3	left
infix-op level 4	left
prefix-op	–
function application	left
brackets / ternary brackets	–
bang-op	–

Note the curryfied syntax for function application, though partial application is not allowed (rejected at typing).

Type Expressions. The syntax for type expressions is the following:

type	::=	*lqualid* *type**	type symbol
	|	`'` *lident*	type variable
	|	`()`	empty tuple type
	|	`(` *type* (`,` *type*)$^+$ `)`	tuple type
	|	`(` *type* `)`	parentheses

Built-in types are `int`, `real`, and tuple types. Note that the syntax for type expressions notably differs from the usual ML syntax (*e.g.* the type of polymorphic lists is written `list 'a`, not `'a list`).

Formulas. The syntax for formulas is given Figure 7.3. The various constructs have the following priorities and associativities, from lowest to greatest priority:

term	::=	*integer*	integer constant
	|	*real*	real constant
	|	*lqualid*	symbol
	|	*prefix-op* *term*	
	|	*bang-op* *term*	
	|	*term* *infix-op* *term*	
	|	*term* `[` *term* `]`	brackets
	|	*term* `[` *term* `<-` *term* `]`	ternary brackets
	|	*lqualid* $term^+$	function application
	|	`if` *formula* `then` *term* `else` *term*	conditional
	|	`let` *pattern* `=` *term* `in` *term*	local binding
	|	`match` *term* $(\texttt{,}\ term)^*$ `with` $(\texttt{|}\ term\text{-}case)^+$ `end`	pattern matching
	|	`(` *term* $(\texttt{,}\ term)^+$ `)`	tuple
	|	`{|` $field\text{-}value^+$ `|}`	record
	|	*term* `.` *lqualid*	field access
	|	`{|` *term* `with` $field\text{-}value^+$ `|}`	field update
	|	*term* `:` *type*	cast
	|	*label* *term*	label
	|	`'` *uident*	code mark
	|	`(` *term* `)`	parentheses
term-case	::=	*pattern* `->` *term*	
pattern	::=	*pattern* `|` *pattern*	or pattern
	|	*pattern* `,` *pattern*	tuple
	|	`_`	catch-all
	|	*lident*	variable
	|	*uident* $pattern^*$	constructor
	|	`(` *pattern* `)`	parentheses
	|	*pattern* `as` *lident*	binding
field-value	::=	*lqualid* `=` *term* `;`	

Figure 7.2: Syntax for terms.

formula	::=	`true` | `false`	
	|	*formula* `->` *formula*	implication
	|	*formula* `<->` *formula*	equivalence
	|	*formula* `/\` *formula*	conjunction
	|	*formula* `&&` *formula*	asymmetric conj.
	|	*formula* `\/` *formula*	disjunction
	|	*formula* `||` *formula*	asymmetric disj.
	|	`not` *formula*	negation
	|	*lqualid*	symbol
	|	*prefix-op* *term*	
	|	*term* *infix-op* *term*	
	|	*lqualid* $term^+$	predicate application
	|	`if` *formula* `then` *formula*	
		`else` *formula*	conditional
	|	`let` *pattern* `=` *term* `in` *formula*	local binding
	|	`match` *term* (`,` *term*)$^+$ `with`	
		(`|` *formula-case*)$^+$ `end`	pattern matching
	|	*quantifier* *binders* (`,` *binders*)*	
		triggers$^?$ `.` *formula*	quantifier
	|	*label* *formula*	label
	|	`(` *formula* `)`	parentheses
quantifier	::=	`forall` | `exists`	
binders	::=	*lident*$^+$ `:` *type*	
triggers	::=	`[` *trigger* (`|` *trigger*)* `]`	
trigger	::=	*tr-term* (`,` *tr-term*)*	
tr-term	::=	*term* | *formula*	
formula-case	::=	*pattern* `->` *formula*	

Figure 7.3: Syntax for formulas.

construct	associativity
`if then else` / `let in`	–
label	–
`->` / `<->`	right
`\/` / `||`	right
`/\` / `&&`	right
`not`	–
infix level 1	left
infix level 2	left
infix level 3	left
infix level 4	left
prefix	–

Note that infix symbols of level 1 include equality (`=`) and disequality (`<>`).

Notice that there are two symbols for the conjunction: `and` and `&&`, and similarly for disjunction. There are logically equivalent, but may be treated slightly differently by some

theory ::= `theory` *uident* *label** *decl** `end`

decl ::= `type` *type-decl* (`with` *type-decl*)*
| `constant` *constant-decl*
| `function` *function-decl* (`with` *logic-decl*)*
| `predicate` *predicate-decl* (`with` *logic-decl*)*
| `inductive` *inductive-decl* (`with` *inductive-decl*)*
| `axiom` *ident* `:` *formula*
| `lemma` *ident* `:` *formula*
| `goal` *ident* `:` *formula*
| `use` *imp-exp* *tqualid* (`as` *uident-opt*)$^?$
| `clone` *imp-exp* *tqualid* (`as` *uident-opt*)$^?$ *subst*$^?$
| `namespace` `import`$^?$ *uident-opt* *decl** `end`

logic-decl ::= *function-decl*
| *predicate-decl*

constant-decl ::= *lident* *label** `:` *type*
| *lident* *label** `:` *type* `=` *term*

function-decl ::= *lident* *label** *type-param** `:` *type*
| *lident* *label** *type-param** `:` *type* `=` *term*

predicate-decl ::= *lident* *label** *type-param**
| *lident* *label** *type-param** `=` *formula*

inductive-decl ::= *lident* *label** *type-param** `=`
`|`$^?$ *ind-case* (`|` *ind-case*)*

ind-case ::= *ident* *label** `:` *formula*

imp-exp ::= (`import` | `export`)$^?$

uident-opt ::= *uident* | `_`

subst ::= `with` (`,` *subst-elt*)$^+$

subst-elt ::= `type` *lqualid* `=` *lqualid*
| `function` *lqualid* `=` *lqualid*
| `predicate` *lqualid* `=` *lqualid*
| `namespace` (*uqualid* | `.`) `=` (*uqualid* | `.`)
| `lemma` *uqualid*
| `goal` *uqualid*

tqualid ::= *uident* | *ident* (`.` *ident*)* `.` *uident*

type-decl ::= *lident* *label** (`'` *lident* *label**)* *type-defn*

Figure 7.4: Syntax for theories (part 1).

transformation, *e.g.* the `split` transformation transforms A `and` B into subgoals A and B, whereas it transforms A `&&` B into subgoals A and $A \to B$.

Theories. The syntax for theories is given on Figure 7.4 and 7.5.

type-defn	::=		abstract type
	|	`=` *type*	alias type
	|	`=` `|`$^{?}$ *type-case* (`|` *type-case*)*	algebraic type
	|	`=` `{|` *record-field* (`;` *record-field*)* `|}`	record type
type-case	::=	*uident* *label** *type-param**	
record-field	::=	*lident* *label** `:` *type*	
type-param	::=	`'` *lident*	
	|	*lqualid*	
	|	`(` *lident*$^{+}$ `:` *type* `)`	
	|	`(` *type* (`,` *type*)* `)`	
	|	`()`	

Figure 7.5: Syntax for theories (part 2).

Files. A Why3 input file is a (possibly empty) list of theories.

file	::=	*theory**

type-v	::=	*type* | `(` *type-v* `)`	parentheses
	|	*type-v* `->` *type-c*	
	|	*type-v-binder* `->` *type-c*	
type-v-binder	::=	*lident* *label** `:` *type-v*	
type-v-param	::=	`(` *type-v-binder* `)`	
type-c	::=	*type-v*	
	|	*pre* *type-v* *effect* *post*	
effect	::=	*reads*$^?$ *writes*$^?$ *raises*$^?$	
reads	::=	`reads` *tr-term*$^+$	
writes	::=	`writes` *tr-term*$^+$	
raises	::=	`raises` *uqualid*$^+$	
pre	::=	*annotation*	
post	::=	*annotation* *post-exn**	
post-exn	::=	`|` *uqualid* `->` *annotation*	
annotation	::=	`{}` | `{` *formula* `}`	

Figure 7.6: Syntax for program types.

7.3 Why3ML Language

Types. The syntax for program types is given in Figure 7.6.

Expressions. The syntax for program expressions is given in Figure 7.7 and Figure 7.8.

Modules. The syntax for modules is given in Figure 7.9. Any declaration which is accepted in a theory is also accepted in a module. Additionally, modules can introduce record types with mutable fields and declarations which are specific to programs (global variables, functions, exceptions).

Files. A Why3ML input file is a (possibly empty) list of theories and modules.

file	::=	(*theory* | *module*)*

A theory defined in a Why3ML file can only be used within that file. If a theory is supposed to be reused from other files, be they Why3 or Why3ML files, it should be defined in a Why3 file.

expr	::=	*integer*	integer constant
	|	*real*	real constant
	|	*lqualid*	symbol
	|	*prefix-op* *expr*	
	|	*expr* *infix-op* *expr*	
	|	*expr* `[` *expr* `]`	brackets
	|	*expr* `[` *expr* `]` `<-` *expr*	brackets assignment
	|	*expr* `[` *expr* *infix-op*-1 *expr* `]`	ternary brackets
	|	*expr* $expr^+$	function application
	|	`fun` *type-v-param*$^+$ `->` *triple*	lambda abstraction
	|	`let rec` *recfun* (`with` *recfun*)*	recursive functions
	|	`if` *expr* `then` *expr* (`else` *expr*)$^?$	conditional
	|	*expr* `;` *expr*	sequence
	|	`loop` *loop-annot* `end`	infinite loop
	|	`while` *expr* `do` *loop-annot* *expr* `done`	while loop
	|	`for` *lident* `=` *expr* *to-downto* *expr* `do` *loop-inv* *expr* `done`	for loop
	|	*assert* *annotation*	assertion
	|	`absurd`	
	|	`raise` *uqualid*	exception raising
	|	`raise (` *uqualid* *expr* `)`	
	|	`try` *expr* `with` (`|` *handler*)$^+$ `end`	exception catching
	|	`any` *type-c*	
	|	`let` *pattern* `=` *expr* `in` *expr*	local binding
	|	`match` *expr* (`,` *expr*)* `with` (`|` *expr-case*)$^+$ `end`	pattern matching
	|	`(` *expr* (`,` *expr*)$^+$ `)`	tuple
	|	`{|` *field-value*$^+$ `|}`	record
	|	*expr* `.` *lqualid*	field access
	|	*expr* `.` *lqualid* `<-` *expr*	field assignment
	|	`{|` *expr* `with` *field-value*$^+$ `|}`	field update
	|	*expr* `:` *type*	cast
	|	*label* *expr*	label
	|	`'` *uident* `:` *expr*	code mark
	|	`(` *expr* `)`	parentheses
expr-case	::=	*pattern* `->` *expr*	
field-value	::=	*lqualid* `=` *expr* `;`	
triple	::=	*expr*	
	|	*pre* *expr* *post*	Hoare triple

Figure 7.7: Syntax for program expressions (part1).

assert	::=	`assert` | `assume` | `check`	
to-downto	::=	`to` | `downto`	
loop-annot	::=	*loop-inv*$^?$ *variant*$^?$	
loop-inv	::=	`invariant` *annotation*	
variant	::=	`variant {` *term* `}` (`with` *lqualid*)$^?$	

Figure 7.8: Syntax for program expressions (part2).

module	::=	`module` *uident* *label** *mdecl** `end`	
mdecl	::=	*decl*	theory declaration
	|	`type` *mtype-decl* (`with` *mtype-decl*)*	mutable types
	|	`let` *lident* *label** *pgm-defn*	
	|	`let rec` *recfun* (`with` *recfun*)*	
	|	`val` *lident* *label** *pgm-decl*	
	|	`exception` *lident* *label** *type*$^?$	
	|	`use` *imp-exp* `module` *tqualid* (`as` *uident-opt*)$^?$	
	|	`namespace import`$^?$ *uident-opt* *mdecl** `end`	
mtype-decl	::=	*lident* *label** (`'` *lident* *label**)* *mtype-defn*	
mtype-defn	::=		abstract type
	|	`=` *type*	alias type
	|	`=` `|`$^?$ *type-case* (`|` *type-case*)*	algebraic type
	|	`= {|` *mrecord-field* (`;` *mrecord-field*)* `|}`	record type
mrecord-field	::=	`mutable`$^?$ *lident* *label** `:` *type*	
pgm-decl	::=	`:` *type-v*	
	|	*type-v-param*$^+$ `:` *type-c*	
pgm-defn	::=	*type-v-param*$^+$ (`:` *type*)$^?$ `=` *triple*	
	|	`= fun` *type-v-param*$^+$ `->` *triple*	

Figure 7.9: Syntax for modules.

D Translation of *MiniMaple* to Why3ML

This section gives the complete definition of the translation of *MiniMaple* and its specification language to Why3ML. For the completeness of this document, some of the auxiliary and domain functions are just declared. In this case the informal comments are added to give reader our intention. This section presents the preliminary version of the translation, i.e. the implementation of the translation is slightly different than this translation.

2.1 Domains

Also includes the syntactic domains of MiniMaple, its specification language and Why3 domains.

Type-Environment //MiniMaple type environment
File // a why3 file
Why3-Type
Why3-TypeU = (Why3-Type, Why3-TypeU) + Undefined
Why3-Environment // why3 type environment, a partial function from identifier to types
Why3-SExpr = Why3-Formula + Why3-Term // why3 specification expression
Why3-SExprU = Why3-SExpr + Undefined
Why3-TermU = Why3-Term + Undefined
Why3-Theory-DeclU = Why3-Theory-Decl + Undefined

Why3-ExprU = Why3-Expr + Undefined , where Undefined = Unit
Why3-Theory = Why3-Theory-Decl x TypeData

TypeData = Why3-Type x I → I // maps a (expression) type and an operation identifier to a
// corresponding why3 operation identifier

Why3-MDecl // Why3 module declaration
Why3-MDeclU = Why3-MDecl + Undefined

LetInDecls := **let** I = Why3-Expression **in**

HandlerSeq := EMPTY | (| I → Why3-Expr) HandlerSeq

Else-Conditional := EMPTY | **else** Why3-Expr Else-Conditional

Elements of domains

te \in Type-Environment
we \in Why3-Environment
w-expr \in Why3-Expr
wt \in Why3-Theory
w-mdecl \in Why3-MDeclU
w-handler \in HandlerSeq
else-cond \in Else-Conditional
w-type \in Why3-Type
w-type-seq \in Why3-TypeSequence
w-sexpr \in Why3-SExpr
w-f \in Why3-Formula
w-t \in Why3-Term

2.2 Signatures of Translation Functions

2.2.1) For Program:
// arguments:
// Type-Environment – requires MiniMaple type environment
// Why3-Environment – requires Why3 type environment
// Why3-MDeclU – requires why3 module declarations for exceptions and type commands
// Why3-Theory – requires why3 theory for expressions
// returns:
// Why3-File – that contains a theory and a corresponding module
T [[P]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory → Why3-File

2.2.2) For Declaration:
T [[D]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory → Why3-Theory-DeclU x Why3-Environment

2.2.3) For Rules:
T [[R]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory → Why3-SExprU x Why3-Environment x Why3-Theory

2.2.4) For Procedure Specification:
T [[PS]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory → Why3-SExpr x Why3-SExpr x Why3-SExpr x Identifier_Sequence x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.5) For Exception Clause:
T [[EC]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory → Why3-Formula x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.6) For Loop Specification:
T [[LS]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory → Why3-Formula x Why3-TermU x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.7) For Assertion:
T [[A]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory → Why3-Formula x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.8) For Specification Expression Sequence:
T [[SEseq]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory → Why3-SExprU x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.9) For Specification Expression:
T [[SE]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory → Why3-SExpr x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.10) For Command Sequence:

// arguments:
// Type-Environment – requires MiniMaple type environment
// Why3-Environment – requires Why3 type environment
// Why3-MDeclU – requires why3 module declarations for exceptions and type commands
// Why3-Theory – requires why3 theory for expressions
// returns:
// Why3-ExprU – an extended expression sequence introduced by Cseq
// Why3-Environment – an extended why3 environment introduced by Cseq
// Why3-MDeclU – an extended why3 module declarations introduced by Cseq
// Why3-Theory – an extended theory introduced by Cseq

T [[Cseq]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-ExprU x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.11) For Command:

T [[C]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.12) For Elif:

T [[Elif]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Else-Conditional x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.13) For Catch:

T [[Catch]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
HandlerSeq x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.14) For Expression Sequence:

T [[Eseq]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-ExprU x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.15) For Expression:

T [[E]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.16) For Special Operators:

T [[Esop]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.17) For (Declaration) Sequence:

T [[S]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-ExprU x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.18) For Recurrence (procedure/module body):

T [[R]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-ExprU x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.19) For Parameter Sequence:

T [[Pseq]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-ExprU x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.20) For Parameter:
T [[P]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.21) For Parameter Modifiers:
T [[M]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Type x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.22) For Identifier Sequence:
T [[Iseq]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.23) For Identifier:
T [[I]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.24) For Identifier Typed Sequence:
T [[Itseq]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-ExprU x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.25) For Identifier Typed:
T [[It]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

2.2.26) For Type Sequence:
T [[Tseq]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-TypeU x Why3-Environment x Why3-MDecl x Why3-Theory

2.2.27) For Type:
T [[T]] : Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Type x Why3-Environment x Why3-MDecl x Why3-Theory

2.3 Auxiliary Functions and Predicates

// this function returns the type of the given expression in a given environment
getExprType : Why3-Expr x Why3-Environment → Why3-Type

// access function returns the corresponding why3 operation identifier by looking up in the given
// why3 theory for the given type and associated operation
access : I x Type x Why3-Theory → I

// returns the recent anonymous exception identifier
getAnonymousException : Why3-MDeclU → Identifier

```
// this function returns true if the handler sequence is EMPTY
isEmpty : HandlerSeq → Boolean

// this function takes a why3 expression and returns its outer most block (why3 expression) that has
// to be handled by the handler, i.e. procedure/module body
getExprForHandler : Why3-Expr → Why3-Expr

// this function combines the former environment with the latter one.
combine : Why3-MDeclU x Why3-MDeclU → Why3-MDeclU
combine(w-mdecl1, w-mdecl2) = inWhy3-MDeclU(inUndefined())
                            , if w-mdecl1 = w-mdecl2 = inUndefined()
combine(w-mdecl1, w-mdecl2) = inWhy3-MDeclU(w-mdecl1) , if w-mdecl2 = inUndefined()
combine(w-mdecl1, w-mdecl2) = inWhy3-MDeclU(w-mdecl2) , if w-mdecl1 = inUndefined()
combine(w-mdecl1, w-mdecl2) = w-mdecl1 w-mdecl2
                                , if w-mdecl1 = w-mdecl2 <> inUndefined() AND
                                isNotRepeated(w-mdecl1, w-mdecl2)

// this function combines the former handler with the latter one.
combine : HandlerSeq x HandlerSeq → HandlerSeq
combine(EMPTY, EMPTY) = EMPTY
combine(EMPTY, (| I → w-expr1) hseq) = (| I → w-expr1) hseq
combine((| I → w-expr1) hseq, EMPTY) = (| I → w-expr1) hseq
combine((| I0 → w-expr1) hseq1, (| I1 → w-expr2) hseq2) =
        (| I0 → w-expr1) (| I1 → w-expr2) combine(hseq1, hseq2)

// this functions constructs the let-ins and returns the local binded identifier sequence
getLetInDeclsAndIds : Why3-Expr x Integer→ LetInDecls x Identifier_Sequence
getLetInDeclsAndIds(w-expr1, n) = (let In = ref w-expr1 in), In
getLetInDeclsAndIds(w-expr1; w-expr2, n) =
                        LET
                                (let-in0, I0) = getLetInDeclsAndIds(w-expr1, n)
                                (let-in1, Iseq) = getLetInDeclsAndIds(w-expr2, n+1)
                        IN
                                ((let-in0 let-in1), (I0,Iseq))

// this functions returns a why3 expression by the corresponding assignments of
// why3-expression (w-expr2) to the given identifier sequence (Iseq0)
getAssignments : Identifier_Sequence x Why3-Expr → Why3-ExprU
getAssignments(EMPTY, w-expr) = inWhy3-ExprU(inUndefined())
getAssignments(I,Iseq, w-expr1;w-expr2) =
                inWhy3-ExprU(I := !nth(w-expr1,0); getAssignments(Iseq, w-expr2))

// this function returns the nth why3-expression in a given why3-expression sequence
nth: Why3-Expr x Integer → Why3-Expr
nth(w-expr, n) = if n = 0 then w-expr
nth(w-expr1;w-expr2, n) = if n = 0 then w-expr1 else if n > 0 then nth(w-expr2, n)
```

// this function returns the nth declaration in a given Why3-MDecl sequence
nth: Why3-MDecl x Integer → Why3-MDecl
nth(w-mdecl, n) = if n = 0 then w-mdecl
nth(w-mdecl1;w-mdecl2, n) = if n = 0 then w-mdecl1 else if n > 0 then nth(w-mdecl2, n)

// this function returns the return-exception statements (why3 module declarations) which
// are introduced by the given why3 expression
getReturnExceptions : Why3-Expr x Why3-MDeclU → Why3-MDeclU

// this function returns the other-exception (no return) statements (why3 module declarations) which
// are introduced by the given why3 expression
getOtherExceptions : Why3-Expr x Why3-MDeclU → Why3-MDeclU

// this function builds the handler block from the given module declarations (exceptions). The body of
// the corresponding handler will be the given why3 expression
buildFinallyCatch : Why3-MDeclU x Why3-Expr → HandlerSeq

// this function returns an exception identifier and an integer by looking into the given
// why3 environment, theory, expression and module declarations
getExceptionStatements : Why3-Environment x Why3-Theory x Why3-Expr x Why3-MDecl →
Identifier x Integer

// this function returns the corresponding type of the given why3 expression
getType : Why3-Environment x Why3-Expr → Why3-Type

// this function defines the why3 type corresponding to the MiniMaple type. Here an identifier
// represents the corresponding why3 type description, for example for list(integer) as list_integer
mmToWhy3Type :
Identifier x Type-Environment x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Type x Why3-Environment x Why3-MDeclU

// this function defines a corresponding why3 theory for the given type. It introduces all the supported
// operations for the given type, e.g. length, operands, substitutions, sequence, equality and non-equality
defineTheory : Why3-Type x Why3-Environment x Why3-MDeclU x Why3-Theory → Why3-Theory

// this function defines a why3 module with w-expr1 (as declarations) and w-expr2 (as body of module)
defineModule : Whye-ExprU x Why3-ExprU → Why3-Expr

// this function defines a why3 void procedure expression with w-expr1 (as parameters)
// , w-expr2 (as declarations) and w-expr3 (as body of procedure)
defineProcedure : Whye-ExprU x Why3-ExprU x Why3-ExprU→ Why3-Expr

// this function defines a why3 procedure expression (with return type) with w-expr1 (as parameters),
// w-expr2 (as declarations), w-expr3 (as body of procedure) and why3 type (as a return type)
defineProcedure : Whye-ExprU x Why3-ExprU x Why3-ExprU x Why3-Type→ Why3-Expr

// this function defines a binary expression for the given expressions (w-expr1 and w-expr2)
defBinaryExpression :
Why3-Expr x Why3-Expr x Binary_Operator x Why3-Environment x Why3-MDeclU x Why3-Theory
→ Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

// this function defines a unary expression for the given expression (w-expr)
defUnaryExpression :
Why3-Expr x Binary_Operator x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

// this function produces all the import statements for a given module (I) from the .mwl file (file)
getAllImportsOfModule : I x File → Why3-MDecl

// this function returns true if none of the parameters in the given parameter sequence has a MiniMaple
// procedure type, otherwise returns false
hasNoProcedureType : Parameter_Sequence x Type-Environment → Boolean

// this function defines a function to compute operands of the latter why3 expression. It may introduce
// new theory, module declarations and why3 theory
defOp : Why3-Expr x Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

// this function defines a function to compute operands of the given why3 expression. It may introduce
// new theory, module declarations and why3 theory
defOp : Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

// this function defines a function to compute operands of the given why3 expression for the range
// given by the why3 first expression and why3 second expression. It may introduce new theory,
// module declarations and why3 theory
defOp :
Why3-Expr x Why3-Expr x Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

// this function redefines procedure parameters by translating procedure typed parameters into
// equivalent string identifiers and so produces a new why3 environment. It also introduces the
// corresponding apply function for the procedure typed parameters
redefProcedureParameters : Parameter_Sequence x Type-Environment →
Parameter_Sequence x Why3-MDeclU x Why3-Environment

// this function returns the corresponding identifier for the type test of the given type
getTypeTest : Why3-Type x Why3-Theory → Identifier

// this functions gets the previously defined (at type declaration/definition time) equality expression
// for the given two expressions
getEqualityExpression :
Why3-Expr x Why3-Expr x Why3-Environment x Why3-MDelcU x Why3-Theory → Why3-Expr

// this function returns true if none of the expression in the given expression sequence is of type
// MiniMaple procedure, otherwise returns false
hasNoProcedureTypeExpr : Expression_Sequence x Type-Environment → Boolean

// this function redefines procedure parameters by translating procedure typed parameters into
// equivalent string identifiers and so produces a new why3 environment. It also introduces the
// corresponding apply function for the procedure typed parameters
redefCallExprSeq : Expression_Sequence x Type-Environment → Expression_Sequence

// this function gets an identifier for the corresponding why3 expression (as a MiniMaple procedure)
getApplyFunction : Why3-Expr x Why3-Environment x Why3MDeclU x Why3-Theory → Identifier

// this function translates the given expression into an identifier
expr2Identifier : Why3-Expr → Identifier

// this function computes the number of operands (length) of the given
// why3 expression
defNops : Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

// this function computes substitutions operands for the given expressions
defSubsops :
Why3-Expr x Why3-Expr x Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

// this function computes a substitutions for the given expressions and identifier
defSubs :
Why3-Expr x Identifier x Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

// this function defines an unevaluated expression in why3 as a string and its corresponding type
defUnevalExpr : Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

// this function defines a corresponding why3 tuple expression from the given why3 expression
defTuple : Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

// this function defines a corresponding why3 set expression from the given why3 expression
defSet : Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

// this function defines a corresponding select operation for the given identifier and expressions
defSelectOp : Identifier x Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

// this function defines a seq quantifier expression for the given expressions and identifier
defSeq :
Why3-Expr x Identifier x Why3-Expr x Why3-Expr x Why3-Environment
x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

// this function defines a seq quantifier expression for the given expressions and identifier
defSeq :
Why3-Expr x Identifier x Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

// this function defines an evaluation expression for the unevaluated expression
defEvalOp : Identifier x Why3-Environment x Why3-MDeclU x Why3-Theory →
Why3-Expr x Why3-Environment x Why3-MDeclU x Why3-Theory

2.4 Definition of Translation Functions

2.4.1) For Program

D; Cseq

T [[D; Cseq]](te, we, mdecl, wt) = (inWhy3-Theory-DeclU(inUndefined()), we)
, if (inWhy3-SExprU(w-expr1), we1, mdecl1, wt1) = T [[Itseq]](te, we, mdecl, wt)
(w-type, we2, w-mdecl2, wt2) = T [[T]](te, we1, mdecl1, wt1)
(inWhy3-SExprU(inUndefined()), we, wt) = T [[EMPTY]](te, we, mdecl, wt)

2.4.2) For Declaration

EMPTY

T [[EMPTY]](te, we, mdecl, wt) = (inWhy3-Theory-DeclU(inUndefined()), we)

define(I(Itseq)::T, R)

T [[I(Itseq)::T, R]](te, we, mdecl , wt) = (inWhy3-Theory-DeclU(w-t), we1, w-mdecl, wt1)

, if (inWhy3-SExprU(w-expr1), we1, mdecl1, wt1) = T [[Itseq]](te, we, mdecl, wt)
(w-type, we2, w-mdecl2, wt2) = T [[T]](te, we1, mdecl1, wt1)
(inWhy3-SExprU(inUndefined()), we, wt) = T [[EMPTY]](te, we, mdecl, wt)

`**type**/I`

T [[`**type**/I`]](te, we, mdecl , wt) = (inWhy3-Theory-DeclU(**type** I), wt)

**`type**/I`:=T

T [[`**type**/I`:=T]](te, we, mdecl , wt) = (inWhy3-Theory-DeclU(**type** I = w-t), we1)

, if (w-type, we1, w-mdecl1, wt1) = T [[T]](te, we, mdecl, wt)

assume(SE)

T [[SE]](te, we, mdecl , wt) = (inWhy3-Theory-DeclU(**axiom** w-f), we1)

, if (inWhy3-ExprU(w-f), we1, w-mdecl1, wt1) = T [[SE]](te, we, mdecl, wt)

2.4.3) For Rules

EMPTY

T [[EMPTY]](te, we, mdecl , wt) = (inWhy3-SExprU(inUndefined()), we, wt)

I(Itseq) = SE; R

T [[I(Itseq) = SE; R]](te, we, mdecl , wt) = (inWhy3-SExprU(se-1;se-2), we2, wt2)

, if (inWhy3-SExpr(se-1), we1, w-mdecl1, wt1) = T [[SE]](te, we, mdecl , wt)
(inWhy3-SExpr(se-2), we2, wt2) = T [[R]](te, we1, mdecl1, wt1)

2.4.4) For Loop Specification

invariant SE; **decreases** SE;

T [[**invariant** SE1; **decreases** SE2;]](te, we, mdecl , wt) =
(w-f1, inWhy3-TermU(w-t1), we2, w-mdecl2, wt2)

, if (inWhy3-SExpr(w-f1), we1, w-mdecl1, wt1) = T [[SE1]](te, we, mdecl , wt)
(inWhy3-SExpr(w-t1), we2, w-mdecl2, wt2) = T [[SE2]](te, we1, mdecl1, wt1)

2.4.5) For Procedure Specification

requires SE; **global** Iseq; **ensures** SE; EC

T [[**requires** SE1; **global** Iseq; **ensures** SE2; EC]](te, we, mdecl , wt) =
(w-f1, w-f2, w-f3, Iseq, we4, w-mdecl4, wt4)

, if (inWhy3-SExpr(w-f1), we1, w-mdecl1, wt1) = T [[SE1]](te, we, mdecl , wt)
(we2, w-mdecl2, wt2) = T [[Iseq]](te, we1, w-mdecl1, wt1)
(inWhy3-SExpr(w-f2), we3, w-mdecl3, wt3) = T [[SE2]](te, we2, mdecl2, wt2)
(inWhy3-SExpr(w-f3), we4, w-mdecl4, wt4) = T [[EC]](te, we3, mdecl3, wt3)

2.4.6) For Exception Clause

EMPTY

T [[EMPTY]](te, we, mdecl , wt) = (false, we, w-mdecl, wt)

exceptions “I” SE; EC

T [[**exceptions** “I” SE; EC]](te, we, mdecl , wt) =
(w-f1 \/ w-f2, we2, w-mdecl2, wt2)

, if (inWhy3-SExpr(w-f1), we1, w-mdecl1, wt1) = T [[SE]](te, we, mdecl , wt)
(inWhy3-SExpr(w-f2), we2, w-mdecl2, wt2) = T [[EC]](te, we1, mdecl1, wt1)

2.4.7) For Assertion

ASSERT(SE, (EMPTY | “I”))

T [[**ASSERT**(SE, (EMPTY | “I”))]](te, we, mdecl , wt) =
(w-f1, we1, w-mdecl1, wt1)

, if (inWhy3-SExpr(w-f1), we1, w-mdecl1, wt1) = T [[SE1]](te, we, mdecl , wt)

2.4.8) For Specification Expression Sequence

EMPTY

T [[EMPTY]](te, we, mdecl , wt) = (inWhy3-SExprU(inUndefined()), we, w-mdecl, wt)

SE, SEseq

T [[SE, SEseq]](te, we, mdecl , wt) =
(inWhy3-SExprU(w-sexpr1; w-sexpr2), we2, w-mdecl2, wt2)

, if (w-sexpr1, we1, w-mdecl1, wt1) = T [[SE]](te, we, mdecl , wt)
(inWhy3-SExprU(w-sexpr2), we2, w-mdecl2, wt2) = T [[SEseq]](te, we1, mdecl1, wt1)

2.4.9) For Specification Expression

I(SEseq)

T [[I(SEseq)]](te, we, mdecl , wt) =
(w-f1, inWhy3-TermU(w-t1), we2, w-mdecl2, wt2)

, if (inWhy3-SExpr(w-f1), we1, w-mdecl1, wt1) = T [[SE1]](te, we, mdecl , wt)

(inWhy3-SExpr(w-t1), we2, w-mdecl2, wt2) = T [[SE2]](te, we1, mdecl1, wt1)

type(SE, T)

T [[**type**(SE,T)]](te, we, mdecl , wt) =
(inWhy3-SExpr(test-name(w-sexpr, w-type)), we2, w-mdecl2, wt2)

, if (w-sexpr, we1, w-mdecl1, wt1) = T [[SE1]](te, we, mdecl , wt)
(w-type, we2, w-mdecl2, wt2) = T [[T]](te, we1, mdecl1, wt1)
test-name = access(type-test, getType(w-sexpr, we1), wt2)

SE **and** SE2

T [[SE1 **and** SE2]](te, we, mdecl , wt) = (w-sexpr1 /\ w-sexpr2, we2, w-mdecl2, wt2)

, if (w-sexpr1, we1, w-mdecl1, wt1) = T [[SE1]](te, we, mdecl , wt)
(w-sexpr2, we2, w-mdecl2, wt2) = T [[SE2]](te, we1, mdecl1, wt1)

SE **or** SE2

T [[SE1 **or** SE2]](te, we, mdecl , wt) = (w-sexpr1 \/ w-sexpr2, we2, w-mdecl2, wt2)

, if (w-sexpr1, we1, w-mdecl1, wt1) = T [[SE1]](te, we, mdecl , wt)
(w-sexpr2, we2, w-mdecl2, wt2) = T [[SE2]](te, we1, mdecl1, wt1)

SE **implies** SE2

T [[SE1 **implies** SE2]](te, we, mdecl , wt) = (w-sexpr1 -> w-sexpr2, we2, w-mdecl2, wt2)

, if (w-sexpr1, we1, w-mdecl1, wt1) = T [[SE1]](te, we, mdecl , wt)
(w-sexpr2, we2, w-mdecl2, wt2) = T [[SE2]](te, we1, mdecl1, wt1)

SE **equivalent** SE2

T [[SE1 **equivalent** SE2]](te, we, mdecl , wt) = (w-sexpr1 <-> w-sexpr2, we2, w-mdecl2, wt2)

, if (w-sexpr1, we1, w-mdecl1, wt1) = T [[SE1]](te, we, mdecl , wt)
(w-sexpr2, we2, w-mdecl2, wt2) = T [[SE2]](te, we1, mdecl1, wt1)

forall(Itseq, SE)

T [[**forall**(Itseq, SE)]](te, we, mdecl , wt) = (forall binders w-sexpr1, we2, w-mdecl2, wt2)

, if (inWhy3-ExprU(w-expr1), we2, w-mdecl2, wt2) = T [[Itseq]](te, we, mdecl, wt)
(w-sexpr1, we1, w-mdecl1, wt1) = T [[SE1]](te, we, mdecl , wt)
binders = getQuantifierBinders(Itseq, we1)

2.4.10) For Command Sequence

EMPTY

T [[EMPTY]](te, we, mdecl, wt) =
(inWhy3-ExprU(inUndefined()), we, inWhy3-MDeclU (inUndefined()), wt)

C;Cseq

T [[C;Cseq]](te, we, mdecl , wt) = (w-expr1; w-expr2, we2, w-mdecl2, wt2)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[C]](te, we, mdecl , wt)
(inWhy3-ExprU(w-expr2), we2, w-mdecl2, wt2) = T [[Cseq]](te, we1, mdecl 1, wt1)

2.4.11) For Command

if E **then** Cseq Elif **end if**

T [[**if** E **then** Cseq Elif **end if**]](te, we, mdecl, wt) =
(inWhy3-ExprU(**if** w-expr1 **then** w-expr2 else-cond), we3, w-mdecl3, wt3)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E]](te, we, mdecl, wt)
(inWhy3-ExprU(w-expr2), we2, w-mdecl2, wt2) = T [[Cseq]](te, we1, mdecl1, wt1)
(else-cond, we3, w-mdecl3, wt3) = T [[Elif]](te, we2, mdecl2, wt2)

if E **then** Cseq Elif **else** Cseq **end if**

T [[**if** E **then** Cseq1 Elif **else** Cseq2 **end if**]](te, we, mdecl, wt) =
(inWhy3-ExprU(**if** w-expr1 **then** w-expr2 else-cond **else** w-expr4), we4, w-mdecl4, wt4)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E]](te, we, mdecl, wt)
(inWhy3-ExprU(w-expr2), we2, w-mdecl2, wt2) = T [[Cseq1]](te, we1, mdecl1, wt1)
(else-cond, we3, w-mdecl3, wt3) = T [[Elif]](te, we2, mdecl2, wt2)
(inWhy3-ExprU(w-expr4), we4, w-mdecl4, wt4) = T [[Cseq2]](te, we3, mdecl3, wt3)

while E **do** LS Cseq **end do**

T [[**while** E **do** LS Cseq **end do**]](te, we, mdecl, wt) =
(inWhy3-ExprU(**while** w-expr1 **do** w-expr2 **done**), we2, w-mdecl2, wt2)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E]](te, we, mdecl, wt)
(inWhy3-ExprU(w-expr2), we2, w-mdecl2, wt2) = T [[Cseq1]](te, we1, mdecl1, wt1)
(inWhy3-ExprU(w-expr2), we2, w-mdecl2, wt2) = T [[Cseq1]](te, we1, mdecl1, wt1)

for I **in** E **do** Cseq **end do**

T [[**for** I **in** E **do** Cseq **end do**]](te, we, mdecl, wt) =
(inWhy3-ExprU(**for** I0 = 0 **to** op-length(w-expr1) - 1 **do**
let I = op-nth(I0, w-expr1) **in** w-expr2 **done**), we2, w-mdecl2, wt2)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E]](te, we, mdecl, wt)
(inWhy3-ExprU(w-expr2), we2, w-mdecl2, wt2) = T [[Cseq]](te, we1, mdecl1, wt1)
exp-type1 = getExprType(w-expr1, we1)
op-length = access(length, exp-type1, wt1)
op-nth = access(select, exp-type1, wt1)

for I **in** E **while** E **do** Cseq **end do**

T [[**for** I **in** E1 **while** E2 **do** Cseq **end do**]](te, we, mdecl, wt) =
(inWhy3-ExprU(**let** I0 = **ref** 0 **in**
while I0 < op-length(w-expr1) & w-expr2 **do**
let I = op-nth(I0, w-expr1) **in** w-expr3; I0 := !I0 + 1 **done**), we3, w-mdecl3, wt3)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
(w-expr2, we2, w-mdecl2, wt2) = T [[E2]](te, we1, mdecl1, wt1)
(inWhy3-ExprU(w-expr3), we3, w-mdecl3, wt3) = T [[Cseq]](te, we2, mdecl2, wt2)
exp-type1 = getExprType(w-expr1, we1)
op-length = access(length, exp-type1, wt1)
op-nth = access(select, exp-type1, wt1)

for I **from** E **by** E **to** E **do** Cseq **end do**

T [[**for** I **from** E1 **by** E2 **to** E3 **do** Cseq **end do**]](te, we, mdecl, wt) =
(inWhy3-ExprU(**let** I0 = **ref** w-expr1 **in**
for I1 = 0 **to** (w-expr3 / w-expr2) - 1 **do**
let I = !I0 **in** w-expr4; I0 := !I0 + w-expr2 **done**), we4, w-mdecl4, wt4)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
(w-expr2, we2, w-mdecl2, wt2) = T [[E2]](te, we1, mdecl1, wt1) and w-expr2 <> 0
(w-expr3, we3, w-mdecl3, wt3) = T [[E3]](te, we2, mdecl2, wt2)
(inWhy3-ExprU(w-expr4), we4, w-mdecl4, wt4) = T [[Cseq]](te, we3, mdecl3, wt3)

(inWhy3-ExprU(**let** I0 = **ref** w-expr1 **in**
loop let I = !I0 **in** w-expr4; I0 := !I0 + w-expr2 **end**), we4, w-mdecl4, wt4)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
(w-expr2, we2, w-mdecl2, wt2) = T [[E2]](te, we1, mdecl1, wt1) and w-expr2 = 0
(w-expr3, we3, w-mdecl3, wt3) = T [[E3]](te, we2, mdecl2, wt2)
(inWhy3-ExprU(w-expr4), we4, w-mdecl4, wt4) = T [[Cseq]](te, we3, mdecl3, wt3)

for I **from** E **by** E **to** E **while** E **do** Cseq **end do**

T [[**for** I **from** E1 **by** E2 **to** E3 **while** E4 **do** Cseq **end do**]](te, we, mdecl, wt) =
    (inWhy3-ExprU(**let** I0 = **ref** w-expr1 **in**
        for I1 = 0 **to** (w-expr3 / w-expr2) - 1 **do**
            let I = !I0 **in** w-expr4; I0 := !I0 + w-expr2 **done**), we5, w-mdecl5, wt5)

        , if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
        (w-expr2, we2, w-mdecl2, wt2) = T [[E2]](te, we1, mdecl1, wt1) and w-expr2 <> 0
        (w-expr3, we3, w-mdecl3, wt3) = T [[E3]](te, we2, mdecl2, wt2)
        (w-expr4, we4, w-mdecl4, wt4) = T [[E4]](te, we3, mdecl3, wt3)
        (inWhy3-ExprU(w-expr5), we5, w-mdecl5, wt5) = T [[Cseq]](te, we4, mdecl4, wt4)

    (inWhy3-ExprU(**let** I0 = **ref** w-expr1 **in**
        loop let I = !I0 **in** w-expr4; I0 := !I0 + w-expr2 **end**), we5, w-mdecl5, wt5)

        , if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
        (w-expr2, we2, w-mdecl2, wt2) = T [[E2]](te, we1, mdecl1, wt1) and w-expr2 = 0
        (w-expr3, we3, w-mdecl3, wt3) = T [[E3]](te, we2, mdecl2, wt2)
        (w-expr4, we4, w-mdecl4, wt4) = T [[E4]](te, we3, mdecl3, wt3)
        (inWhy3-ExprU(w-expr5), we5, w-mdecl5, wt5) = T [[Cseq]](te, we4, mdecl4, wt4)

return E

T [[**return** E]](te, we, mdecl, wt) =
    (inWhy3-ExprU(**raise** (I(x, w-expr1))), we1,
        inWhy3-MDeclU(combine(w-mdecl1,
                type return_excp_x = (x, w-type) **exception** I return_excp_x)),
                wt1)

    , if (w-expr1, we1, w-mdecl1, wt1) = T [[E]](te, we, mdecl, wt)
    (I, x) = getExceptionStatements(wt1, we1, w-expr1, w-mdecl1)
    w-type = getType(w-expr1, we1)

return

T [[**return**]](te, we, mdecl, wt) =
    (inWhy3-ExprU(**raise** I), we, inWhy3-MDeclU(combine(mdecl, **exception** I)), wt)

error

T [[**error**]](te, we, mdecl, wt) =
        (inWhy3-ExprU(**raise** I), we, inWhy3-MDeclU(combine(mdecl, **exception** I)), wt)

            , where I is an anonymous exception identifier

error I,Eseq

T [[**error** I,Eseq]](te, we, mdecl, wt) =

(inWhy3-ExprU(**raise** (I(x, w-expr1))), we1,
inWhy3-MDeclU(combine(w-mdecl1,
type return_excp_x = (x, w-type) **exception** I return_excp_x)),
wt)

, if (inWhy3-ExprU(w-expr1), we1, w-mdecl1, wt1) = T [[Eseq]](te, we, mdecl, wt)
(I, x) = getExceptionStatements(wt1, we1, w-expr1, w-mdecl1)
w-type = getType(w-expr1, we1)

try Cseq Catch **end**

T [[**try** Cseq Catch **end**]](te, we, mdecl, wt) =
(inWhy3-ExprU(**try** w-expr1 **with** w-handler **end**), we2, w-mdecl2, wt2)

, if (inWhy3-ExprU(w-expr1), we1, w-mdecl1, wt1) = T [[Cseq]](te, we, mdecl, wt)
(w-handler, we2, w-mdecl2, wt2) = T [[Catch]](te, we1, w-mdecl1, wt1)
and NOT isEmpty(w-handler)

(inWhy3-ExpU(**try** w-expr1 **with** | i-an → **absurd end**), we2, w-mdecl2, wt2)

, if (inWhy3-ExpU(w-expr1), we1, w-mdecl1, wt1) = T [[Cseq]](te, we, mdecl, wt)
(w-handler, we2, w-mdecl2, wt2) = T [[Catch]](te, we1, w-mdecl1, wt1)
and isEmpty(w-handler)
i-an = getAnonymousException(mdecl1)

try Cseq Catch **finally** Cseq **end**

T [[**try** Cseq1 Catch **finally** Cseq2 **end**]](te, we, mdecl, wt) =
(inWhy3-ExpU(**try** w-expr1 **with** w-handler1 **end**), we3, w-mdecl3, wt3)

, if (inWhy3-ExpU(w-expr1), we1, w-mdecl1, wt1) = T [[Cseq1]](te, we, mdecl, wt)
(w-handler, we2, w-mdecl2, wt2) = T [[Catch]](te, we1, w-mdecl1, wt1)
excep1 = getReturnExceptions(w-expr1, w-mdecl1)
excep2 = getReturnExceptions(w-expr2, w-mdecl2)
excep3 = getOtherExceptions(w-expr1, w-mdecl1)
excep4 = getOtherExceptions(w-expr2, w-mdecl2)
(inWhy3-ExpU(w-expr3), we3, w-mdecl3, wt3) = T [[Cseq2]](te, we2, w-mdecl2, wt2)
catch1 = combine(buildFinallyCatch(excep1, w-expr3), buildFinallyCatch(excep2, w-expr3))
catch2 = combine(buildFinallyCatch(excep3, w-expr3), buildFinallyCatch(excep4, w-expr3))
catch3 = combine(catch1, catch2)
w-handler1 = combine(catch3, w-handler)

I, Iseq := E, Eseq

T [[I, Iseq := E, Eseq]](te, we, mdecl, wt) =
(inWhy3-ExpU(**let** I0 = **ref** w-expr1 **in** let-indecls I := !I0; w-expr3), we2, w-mdecl2, wt2)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E]](te, we, mdecl, wt)
(inWhy3-ExpU(w-expr2), we2, w-mdecl2, wt2) = T [[Eseq]](te, we1, mdecl1, wt1)
(let-indecls, Iseq0) = getLetInsDeclsAndIds(w-expr2)
inWhy3-ExprU(w-expr3) = getAssignments(let-indecls, Iseq0, w-expr2)

E (Eseq)

T [[E(Eseq)]](te, we, mdecl, wt) = (inWhy3-ExpU(apply(id, w-expr2))) , we2, w-mdecl2, wt2)

, if hasNoProcedureTypeExpr(Eseq, te)
(w-expr1, we1, w-mdecl1, wt1) = T [[E]](te, we, mdecl, wt)
(inWhy3-ExpU(w-expr2), we2, w-mdecl2, wt2) = T [[Eseq]](te, we1, mdecl1, wt1)

, if NOT hasNoProcedureTypeExpr(Eseq, te)
(w-expr1, we1, w-mdecl1, wt1) = T [[E]](te, we, mdecl, wt)
Eseq' = redefineCallExprSeq(Eseq, te)
(inWhy3-ExpU(w-expr2), we2, w-mdecl2, wt2) = T [[Eseq']](te, we1, mdecl1, wt1)
apply = getApplyFunction(w-expr1, we2, w-mdecl2, wt2)
id = expr2Identifier(w-expr1)

`**type**/I` := T

T [[`**type**/I` := T]](te, we, mdecl, wt) =
(inWhy3-ExpU(inUndefined()), we1, inWhy3-MDeclU(combine(w-mdecl1, **type** I = t)), wt1)

, if (te, we1, w-mdecl1, wt1) = T [[T]](te, we, mdecl, wt)

2.4.12) For Expression Sequence

EMPTY

T [[EMPTY]](te, we, mdecl, wt) =
(inWhy3-ExprU(inUndefined()), we, inWhy3-MDeclU (inUndefined()), wt)

E;Eseq

T [[E;Eseq]](te, we, mdecl , wt) = (w-expr1; w-expr2, we2, w-mdecl2, wt2)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E]](te, we, mdecl , wt)
(inWhy3-ExprU(w-expr2), we2, w-mdecl2, wt2) = T [[Eseq]](te, we1, mdecl 1, wt1)

2.4.13) For Expression

module S;R **end**

T [[**module** S;R **end**]](te, we, mdecl, wt) = (module-expr, we2, w-mdecl2, wt2)

, if (inWhy3-ExprU(w-expr1), we1, w-mdecl1, wt1) = T [[S]](te, we, mdecl, wt)
(inWhy3-ExprU(w-expr2), we2, w-mdecl2, wt2) = T [[R]](te, we1, mdecl1, wt1)
module-expr = defineModule(w-expr1, w-expr2)

proc(Pseq) S;R **end**

T [[**proc**(Pseq) S;R **end**]](te, we, mdecl, wt) = (proc-expr, we3, w-mdecl3, wt3)

, if hasNoProcedureType(Pseq, te)
(inWhy3-ExprU(w-expr1), we1, w-mdecl1, wt1) = T [[Pseq]](te, we, mdecl, wt)
(inWhy3-ExprU(w-expr2), we2, w-mdecl2, wt2) = T [[S]](te, we1, mdecl1, wt1)
(inWhy3-ExprU(w-expr3), we3, w-mdecl3, wt3) = T [[R]](te, we2, mdecl2, wt2)
proc-expr = defineProcedure(
inWhy3-ExprU(w-expr1), inWhy3-ExprU(w-expr2),
inWhy3-ExprU(w-expr3))

, if NOT hasNoProcedureType(Pseq, te)
(Pseq', mdecl', we') = redefProcedureParameters(Pseq, te)
(inWhy3-ExprU(w-expr1), we1, w-mdecl1, wt1) = T [[Pseq']](te, we', mdecl', wt)
(inWhy3-ExprU(w-expr2), we2, w-mdecl2, wt2) = T [[S]](te, we1, mdecl1, wt1)
(inWhy3-ExprU(w-expr3), we3, w-mdecl3, wt3) = T [[R]](te, we2, mdecl2, wt2)
proc-expr = defineProcedure(
inWhy3-ExprU(w-expr1), inWhy3-ExprU(w-expr2),
inWhy3-ExprU(w-expr3))

proc(Pseq)::T S;R **end**

T [[**proc**(Pseq)::T S;R **end**]](te, we, mdecl, wt) = (proc-expr, we4, w-mdecl4, wt4)

, if hasNoProcedureType(Pseq, te)
(inWhy3-ExprU(w-expr1), we1, w-mdecl1, wt1) = T [[Pseq]](te, we, mdecl, wt)
(w-type, we1', w-mdecl1', wt1') = T [[T]](te, we1, mdecl1, wt1)
(inWhy3-ExprU(w-expr2), we2, w-mdecl2, wt2) = T [[S]](te, we1', mdecl1', wt1')
(inWhy3-ExprU(w-expr3), we3, w-mdecl3, wt3) = T [[R]](te, we2, mdecl2, wt2)
proc-expr = defineProcedure(
inWhy3-ExprU(w-expr1), inWhy3-ExprU(w-expr2),
inWhy3-ExprU(w-expr3), w-type)

, if NOT hasNoProcedureType(Pseq, te)

(Pseq', mdecl', we') = redefProcedureParameters(Pseq, te)
(inWhy3-ExprU(w-expr1), we1, w-mdecl1, wt1) = T [[Pseq']](te, we', mdecl', wt)
(w-type, we1', w-mdecl1', wt1') = T [[T]](te, we1, mdecl1, wt1)
(inWhy3-ExprU(w-expr2), we2, w-mdecl2, wt2) = T [[S]](te, we1', mdecl1', wt1')
(inWhy3-ExprU(w-expr3), we3, w-mdecl3, wt3) = T [[R]](te, we2, mdecl2, wt2)
proc-expr = defineProcedure(
inWhy3-ExprU(w-expr1), inWhy3-ExprU(w-expr2),
inWhy3-ExprU(w-expr3), w-type)

E1 Bop E2

T [[E1 Bop E2]](te, we, mdecl, wt) = (w-expr3, we3, w-mdecl3, wt3)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
(w-expr2, we2, w-mdecl2, wt2) = T [[E2]](te, we1, mdecl1, wt1)
(w-expr3, we3, mdecl3, wt3) =
defBinaryExpression(w-expr1, w-expr2, Bop, we2, mdecl2, wt2)

Uop E

T [[Uop E]](te, we, mdecl, wt) = (w-expr2, we2, w-mdecl2, wt2)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E]](te, we, mdecl, wt)
(w-expr2, we2, mdecl2, wt2) =
defUnaryExpression(w-expr1, w-expr1, Uop, we1, mdecl1, wt1)

Esop

T [[Esop]](te, we, mdecl, wt) = (w-expr1, we1, w-mdecl1, wt1)

E1 **and** E2

T [[E1 **and** E2]](te, we, mdecl, wt) = ((w-expr1 & w-expr2), we2, w-mdecl2, wt2)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
(w-expr2, we2, w-mdecl2, wt2) = T [[E2]](te, we1, mdecl1, wt1)

E1 **or** E2

T [[E1 **or** E2]](te, we, mdecl, wt) = ((w-expr1 | w-expr2), we2, w-mdecl2, wt2)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
(w-expr2, we2, w-mdecl2, wt2) = T [[E2]](te, we1, mdecl1, wt1)

T [[E(Eseq)]](te, we, mdecl, wt) = (apply(id, w-expr2)) , we2, w-mdecl2, wt2)

, if hasNoProcedureTypeExpr(Eseq, te)
(w-expr1, we1, w-mdecl1, wt1) = T [[E]](te, we, mdecl, wt)
(inWhy3-ExpU(w-expr2), we2, w-mdecl2, wt2) = T [[Eseq]](te, we1, mdecl1, wt1)

, if NOT hasNoProcedureTypeExpr(Eseq, te)
(w-expr1, we1, w-mdecl1, wt1) = T [[E]](te, we, mdecl, wt)
Eseq' = redefineCallExprSeq(Eseq, te)
(inWhy3-ExpU(w-expr2), we2, w-mdecl2, wt2) = T [[Eseq']](te, we1, mdecl1, wt1)
apply = getApplyFunction(w-expr1, we2, w-mdecl2, wt2)
id = expr2Identifier(w-expr1)

I1:-I2

T [[I1:-I2]](te, we, mdecl, wt) = (inWhy3-ExprU(w-expr1 | w-expr2), we2, w-mdecl2, wt2)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
(w-expr2, we2, w-mdecl2, wt2) = T [[E2]](te, we1, mdecl1, wt1)

E, E, Eseq

T [[E1, E2, Eseq]](te, we, mdecl, wt) =
(w-expr1;w-expr2;w-expr3, we3, w-mdecl3, wt3)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
(w-expr2, we2, w-mdecl2, wt2) = T [[E2]](te, we1, mdecl1, wt1)
(inWhy3-ExprU(w-expr3), we3, w-mdecl3, wt3) = T [[E2]](te, we2, mdecl2, wt2)

type(I, T)

T [[**type**(I,T)]](te, we, mdecl, wt) = (test(I, w-type), we2, w-mdecl2, wt2)

, if (we1, w-mdecl1, wt) = T [[I]](te, we, mdecl, wt)
(w-type, we2, w-mdecl2, wt2) = T [[T]](te, we1, mdecl1, wt1)
test = access(type-test, w-type, wt2)

E1 = E2

T [[E1 = E2]](te, we, mdecl, wt) = (eq-expr(w-expr1, w-expr2), we2, w-mdecl2, wt2)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
(w-expr2, we2, w-mdecl2, wt2) = T [[E2]](te, we1, mdecl1, wt1)
eq-expr = access(equal, getExprType(w-expr1, we1), wt2)

T [[E1 <> E2]](te, we, mdecl, wt) = ((~eq-expr(w-expr1, w-expr2)), we2, w-mdecl2, wt2)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
(w-expr2, we2, w-mdecl2, wt2) = T [[E2]](te, we1, mdecl1, wt1)
eq-expr = access(equal, getExprType(w-expr1, we1), wt2)

2.4.14) For Parameter Sequence

EMPTY

T [[EMPTY]](te, we, mdecl, wt) = (inWhy3-ExprU(inUndefined()), we, w-mdecl, wt)

P, Pseq

T [[P, Pseq]](te, we, mdecl, wt) = (inWhy3-ExprU(w-expr1, w-expr2), we2, w-mdecl2, wt2)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[P]](te, we, mdecl, wt)
(inWhy3-ExprU(w-expr2), we2, w-mdecl2, wt2) = T [[Pseq]](te, we1, mdecl1, wt1)

2.4.15) For Parameter

I

T [[I]](te, we, mdecl, wt) = (I, we, w-mdecl, wt)

I :: M

T [[I::M]](te, we, mdecl, wt) = (I:w-type, we1, w-mdecl1, wt1)
, if (w-type, we1, w-mdecl1, wt1) = T [[M]](te, we, mdecl, wt)

2.4.16) For (Parameter) Modifiers

T

T [[T]](te, we, mdecl, wt) = (w-type, we1, w-mdecl1, wt1)
, if (w-type, we1, w-mdecl1, wt1) = T [[T]](te, we, mdecl, wt)

seq(T)

T [[**seq**(T)]](te, we, mdecl, wt) =
(w-type1, we2, w-mdecl2, defineTypeTheory(w-type1, we2, mdecl2, wt1))

, if (w-type, we1, w-mdecl1, wt1) = T [[T]](te, we, mdecl, wt)
(w-type1, we2, w-mdecl2) = mmToWhy3Type(seq_w-type, te, we1, mdecl1, wt1)

2.4.17) For Module/Procedure Declarations (S)

EMPTY

T [[EMPTY]](te, we, mdecl, wt) = (inWhy3-ExprU(inUndefined()), we, w-mdecl, wt)

local It,Itseq;S

T [[**local** It,Itseq;S]](te, we, mdecl, wt) =
(inWhy3-ExprU(w-expr1; w-expr2; w-expr3), we3, w-mdecl3, wt3)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[It]](te, we, mdecl, wt)
(inWhy3-ExprU(w-expr2), we2, w-mdecl2, wt2) = T [[Itseq]](te1, we1, mdecl1, wt1)
(inWhy3-ExprU(w-expr3), we3, w-mdecl3, wt3) = T [[S]](te2, we2, mdecl2, wt2)

global I,Iseq;S

T [[**global** I,Iseq;S]](te, we, mdecl, wt) = (inWhy3-ExprU(w-expr1), we3, w-mdecl3, wt3)

, if (inWhy3-ExprU(inUndefined()), we1, w-mdecl1, wt1) = T [[I]](te, we, mdecl, wt)
(inWhy3-ExprU(inUndefined()), we2, w-mdecl2, wt2) = T [[Iseq]](te1, we1, mdecl1, wt1)
(inWhy3-ExprU(w-expr1), we3, w-mdecl3, wt3) = T [[S]](te2, we2, mdecl2, wt2)

uses I,Iseq;S

T [[**uses** I,Iseq;S]](te, we, mdecl, wt) = (inWhy3-ExprU(w-expr1), we3, m-imports3, wt3)

, if (inWhy3-ExprU(inUndefined()), we1,, wt1) = T [[I]](te, we, mdecl, wt)
module-imports = getAllImportsOfModule(I, file)
(inWhy3-ExprU(inUndefined()), we2, w-mdecl2, wt2) = T [[Iseq]](te1, we1, mdecl1, wt1)
(inWhy3-ExprU(w-expr1), we3, w-mdecl3, wt3) = T [[S]](te2, we2, mdecl2, wt2)
m-imports1 = combine(module-imports, w-mdecl2)
m-imports2 = combine(m-imports1, w-mdecl3)

export I,Iseq;S

T [[**export** I,Iseq;S]](te, we, mdecl, wt) = (inWhy3-ExprU(w-expr1), we3, w-mdecl3, wt3)

, if (inWhy3-ExprU(inUndefined()), we1,, wt1) = T [[I]](te, we, mdecl, wt)
module-imports = getAllImportsOfModule(I, file)
(inWhy3-ExprU(inUndefined()), we2, w-mdecl2, wt2) = T [[Iseq]](te1, we1, mdecl1, wt1)
(inWhy3-ExprU(w-expr1), we3, w-mdecl3, wt3) = T [[S]](te2, we2, mdecl2, wt2)
m-imports1 = combine(module-imports, w-mdecl2)
m-imports2 = combine(m-imports1, w-mdecl3)

2.4.18) For Module/Procedure Body (R)

Cseq

T [[Cseq]](te, we, mdecl, wt) = (inWhy3-ExprU(w-expr1), we1, w-mdecl1, wt1)

Cseq; E

T [[Cseq; E]](te, we, mdecl, wt) = (inWhy3-ExprU(w-expr1; w-expr2), we2, w-mdecl2, wt2)

, if (inWhy3-ExprU(w-expr1), we1, w-mdecl1, wt1) = T [[Cseq]](te, we, mdecl, wt)
(w-expr2, we2, w-mdecl2, wt2) = T [[E]](te1, we1, mdecl1, wt1)

2.4.19) For Identifier Sequence

EMPTY

T [[EMPTY]](te, we, mdecl, wt) = (we, w-mdecl, wt)

I, Iseq

T [[Iseq]](te, we, mdecl, wt) = (we2, w-mdecl3, wt2)

, if (we1, w-mdecl1, wt1) = T [[I]](te, we, mdecl, wt)
(we2, w-mdecl2, wt2) = T [[Iseq]](te, we1, mdecl1, wt1)
w-mdecl3 = combine(w-mdecl1, w-mdecl2)

2.4.20) For Identifier

I

T [[I]](te, we, mdecl, wt) = (we1, combine(**val** I : **ref** w-type, w-mdecl1), wt)

, if (w-type, we1, w-mdecl1) = mmToWhy3Type(anything, te, we, mdecl, wt)

2.4.21) For Identifier Typed Sequence

EMPTY

T [[EMPTY]](te, we, mdecl, wt) = (inWhy3-ExprU(isUndefined()), we, w-mdecl, wt)

It, Itseq

T [[Itseq]](te, we, mdecl, wt) = (inWhy3-ExprU(w-expr1; w-expr2), we2, w-mdecl2, wt2)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[It]](te, we, mdecl, wt)
(inWhy3-ExprU(w-expr2), we2, w-mdecl2, wt2) = T [[Itseq]](te, we1, mdecl1, wt1)

2.4.22) For Identifier Typed

I

T [[I]](te, we, mdecl, wt) = (**let** I = **ref** (**any** mm_symbol) **in**, we, w-mdecl, wt)

I::T

T [[I::T]](te, we, mdecl, wt) = (**let** I = **ref** (**any** w-type) **in**, we1, w-mdecl1, wt1)
, if (w-type, we1, w-mdecl1, wt1) = T [[T]](te, we, mdecl, wt)

I:=E

T [[I::T]](te, we, mdecl, wt) = (**let** I = **ref** w-expr1 **in**, we1, w-mdecl1, wt1)
, if (w-expr1, we1, w-mdecl1, wt1) = T [[E]](te, we, mdecl, wt)

I::T:=E

T [[I::T]](te, we, mdecl, wt) =
(**let** x = **ref** (w-expr **in**
let I = **ref** (w-type) **in** I := !x, we2, w-mdecl2, wt2)

, if (w-type, we1, w-mdecl1, wt1) = T [[T]](te, we, mdecl, wt)
(w-expr1, we2, w-mdecl2, wt2) = T [[E]](te, we1, mdecl1, wt1)

2.4.23) For Special Operators

op(E, E)

T [[**op**(E1,E2)]](te, we, mdecl, wt) = (op-expr(w-expr1, w-expr2), we2, w-mdecl2, wt2)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
(w-expr2, we2, w-mdecl2, wt2) = T [[E2]](te, we1, mdecl1, wt1)
op-expr = access(op2, getExprType(w-expr2, we2), wt2)

op(E)

T [[**op**(E)]](te, we, mdecl, wt) = (op-expr(w-expr1), we1, w-mdecl1, wt1)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E]](te, we, mdecl, wt)
op-expr = access(op1, getExprType(w-expr1, we1), wt1)

op(E..E, E)

T [[**op**(E1..E2, E3)]](te, we, mdecl, wt) = (op-expr(w-expr1, w-expr2, w-expr3), we3, w-mdecl3, wt3)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
(w-expr2, we2, w-mdecl2, wt2) = T [[E2]](te, we1, mdecl1, wt1)

(w-expr3, we3, w-mdecl3, wt3) = T [[E3]](te, we2, mdecl2, wt2)
op-expr = access(op3, getExprType(w-expr3, we3), wt3)

nops(E)

T [[**nops**(E)]](te, we, mdecl, wt) = (nops-expr(w-expr1), we1, w-mdecl1, wt1)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
nops-expr = access(length, getExprType(w-expr1, we1), wt1)

subsop(E=E, E)

T [[**subsop**(E1=E2, E3)]](te, we, mdecl, wt) =
(subs-expr(w-expr1, w-expr2, w-expr3), we3, w-mdecl3, wt3)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
(w-expr2, we2, w-mdecl2, wt2) = T [[E2]](te, we1, mdecl1, wt1)
(w-expr3, we3, w-mdecl3, wt3) = T [[E3]](te, we2, mdecl2, wt2)
subs-expr = access(subs3, getExprType(w-expr3, we3), wt3)

subs(I=E, E)

T [[**subs**(I=E1, E2)]](te, we, mdecl, wt) = (subs-expr(I, w-expr1, w-expr2), we3, w-mdecl3, wt3)

, if (we1, w-mdecl1, wt) = T [[I]](te, we, mdecl, wt)
(w-expr1, we2, w-mdecl2, wt2) = T [[E1]](te, we1, mdecl1, wt1)
(w-expr2, we3, w-mdecl3, wt3) = T [[E2]](te, we2, mdecl2, wt2)
subs-expr = access(length, getExprType(w-expr2, we2), wt2)

" E "

T [[" E "]](te, we, mdecl, wt) = (w-expr2, we2, w-mdecl2, wt2)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E]](te, we, mdecl, wt)
(w-expr2, we2, w-mdecl2, wt2) =
defUnevalExpr(w-expr1, we1, w-mdecl1, wt1)

[Eseq]

T [[[Eseq]]](te, we, mdecl, wt) = (w-expr2, we2, w-mdecl2, wt2)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
(inWhy3-ExprU(w-expr2), we2, w-mdecl2, wt2) =
defTuple(w-expr1, we1, w-mdecl1, wt1)

{Eseq}

T [[{Eseq}]](te, we, mdecl, wt) = (w-expr2, we2, w-mdecl2, wt2)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
(inWhy3-ExprU(w-expr2), we2, w-mdecl2, wt2) =
defSet(w-expr1, we1, w-mdecl1, wt1)

I[Eseq**]**

T [[I[Eseq]]](te, we, mdecl, wt) = (sel-expr(I, w-expr1), we2, w-mdecl2, wt2)

, if (we1, w-mdecl1, wt) = T [[I]](te, we, mdecl, wt)
(inWhy3-ExprU(w-expr1), we2, w-mdecl2, wt2) = T [[Eseq]](te, we1, mdecl1, wt1)
sel-expr = access(select, getExprType(I, we1), wt2)

seq(E, I=E..E)

T [[**seq**(E1, I=E2..E3)]](te, we, mdecl, wt) =
(seq-expr(w-expr1, w-expr2, w-expr3, I), we4, w-mdecl4, wt4)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
(we2, w-mdecl2, wt2) = T [[I]](te, we1, mdecl1, wt1)
(w-expr2, we3, w-mdecl3, wt3) = T [[E2]](te, we2, mdecl2, wt2)
(w-expr3, we4, w-mdecl4, wt4) = T [[E3]](te, we3, mdecl3, wt3)
seq-expr = access(seq3, getExprType(w-expr1, we1), wt4)

seq(E, I **in** E)

T [[**seq**(E1, I **in** E2)]](te, we, mdecl, wt) =
(seq-expr(w-expr1, w-expr2, I), we3, w-mdecl3, wt3)

, if (w-expr1, we1, w-mdecl1, wt1) = T [[E1]](te, we, mdecl, wt)
(we2, w-mdecl2, wt2) = T [[I]](te, we1, mdecl1, wt1)
(w-expr2, we3, w-mdecl3, wt3) = T [[E2]](te, we2, mdecl2, wt2)
seq-expr = access(seq2, getExprType(w-expr1, we1), wt3)

eval(I, 1)

T [[**eval**(I, 1)]](te, we, mdecl, wt) = (w-expr1, we2, w-mdecl2, wt2)

, if (we1, w-mdecl1, wt) = T [[I]](te, we, mdecl, wt)
(w-expr1, we2, w-mdecl2, wt2) = defEval(I, we1, w-mdecl1, wt1)

2.4.24) For Select Operator

EMPTY

T [[EMPTY]](te, we, mdecl, wt) = inWhy3-TypeU(inUndefined()), we, mdecl, wt)

[Eseq] SelOp

T [[[Eseq] SelOp]](te, we, mdecl , wt) =
(inWhy3-TypeU((w-type, w-type-seq)), we2, w-mdecl2, wt2)

, if (w-type, we1, w-mdecl1, wt1) = T [[T]](te, we, mdecl , wt)
(inWhy3-TypeU(w-type-seq), we2, w-mdecl2, wt2) = T [[Tseq]](te, we1, mdecl 1, wt1)

2.4.25) For Type Sequence

EMPTY

T [[EMPTY]](te, we, mdecl, wt) = inWhy3-TypeU(inUndefined()), we, mdecl, wt)

T;Tseq

T [[T,Tseq]](te, we, mdecl , wt) = (inWhy3-TypeU((w-type, w-type-seq)), we2, w-mdecl2, wt2)

, if (w-type, we1, w-mdecl1, wt1) = T [[T]](te, we, mdecl , wt)
(inWhy3-TypeU(w-type-seq), we2, w-mdecl2, wt2) = T [[Tseq]](te, we1, mdecl 1, wt1)

2.4.26) For Type

integer

T [[**integer**]](te, we, mdecl, wt) =
(w-type, we1, w-mdecl1, defineTypeTheory(w-type, we1, mdecl1, wt))

, if (w-type, we1, w-mdecl1) = mmToWhy3Type(integer, te, we, mdecl, wt)

T [[**boolean**]](te, we, mdecl, wt) =
(w-type, we1, w-mdecl1, defineTypeTheory(w-type, we1, mdecl1, wt))

, if (w-type, we1, w-mdecl1) = mmToWhy3Type(boolean, te, we, mdecl, wt)

T [[**float**]](te, we, mdecl, wt) =
(w-type, we1, w-mdecl1, defineTypeTheory(w-type, we1, mdecl1, wt))

, if (w-type, we1, w-mdecl1) = mmToWhy3Type(float, te, we, mdecl, wt)

T [[**string**]](te, we, mdecl, wt) =

(w-type, we1, w-mdecl1, defineTypeTheory(w-type, we1, mdecl1, wt))

, if (w-type, we1, w-mdecl1) = mmToWhy3Type(string, te, we, mdecl, wt)

T [[**rational**]](te, we, mdecl, wt) =
(w-type, we1, w-mdecl1, defineTypeTheory(w-type, we1, mdecl1, wt))

, if (w-type, we1, w-mdecl1) = mmToWhy3Type(rational, te, we, mdecl, wt)

T [[**anything**]](te, we, mdecl, wt) =
(w-type, we1, w-mdecl1, defineTypeTheory(w-type, we1, mdecl1, wt))

, if (w-type, we1, w-mdecl1) = mmToWhy3Type(anything, te, we, mdecl, wt)

T [[**{T}**]](te, we, mdecl, wt) =
(w-type1, we2, w-mdecl2, defineTypeTheory(w-type1, we2, mdecl2, wt1))

, if (w-type, we1, w-mdecl1, wt1) = T [[T]](te, we, mdecl, wt)
(w-type1, we2, w-mdecl2) = mmToWhy3Type(set_w-type, te, we1, mdecl1, wt1)

T [[**list(T)**]](te, we, mdecl, wt) =
(w-type1, we2, w-mdecl2, defineTypeTheory(w-type1, we2, mdecl2, wt1))

, if (w-type, we1, w-mdecl1, wt1) = T [[T]](te, we, mdecl, wt)
(w-type1, we2, w-mdecl2) = mmToWhy3Type(list_w-type, te, we1, mdecl1, wt1)

T [[[Tseq]]](te, we, mdecl, wt) =
(w-type1, we2, w-mdecl2, defineTypeTheory(w-type1, we2, mdecl2, wt1))

, if (w-type-seq, we1, w-mdecl1, wt1) = T [[Tseq]](te, we, mdecl, wt)
(w-type1, we2, w-mdecl2) = mmToWhy3Type(tuple_w-type-seq, te, we1, mdecl1, wt1)

T [[I(Tseq)]](te, we, mdecl, wt) =
(w-type1, we2, w-mdecl2, defineTypeTheory(w-type1, we2, mdecl2, wt1))

, if (w-type-seq, we1, w-mdecl1, wt1) = T [[Tseq]](te, we, mdecl, wt)
(w-type1, we2, w-mdecl2) = mmToWhy3Type(named_tuple_w-type-seq, te, we1, mdecl1, wt1)

T [[Or(Tseq)]](te, we, mdecl, wt) =
(w-type1, we2, w-mdecl2, defineTypeTheory(w-type1, we2, mdecl2, wt1))

, if (w-type-seq, we1, w-mdecl1, wt1) = T [[Tseq]](te, we, mdecl, wt)
(w-type1, we2, w-mdecl2) = mmToWhy3Type(or_w-type-seq, te, we1, mdecl1, wt1)

T [[**symbol**]](te, we, mdecl, wt) =
(w-type1, we1, w-mdecl1, defineTypeTheory(w-type1, we1, mdecl1, wt1))

, if (w-type, we1, w-mdecl1) = mmToWhy3Type(symbol, te, we, mdecl, wt)

T [[**uneval**]](te, we, mdecl, wt) =
(w-type, we2, w-mdecl2, defineTypeTheory(w-type, we1, mdecl1, wt1))

, if (w-type, we1, w-mdecl1) = mmToWhy3Type(uneval, te, we, mdecl, wt)

T [[I]](te, we, mdecl, wt) =
(w-type1, we1, w-mdecl1, defineTypeTheory(w-type, we1, mdecl1, wt1))

, if w-type = I[[I]](te, we, mdecl)
(w-type1, we1, w-mdecl1) = mmToWhy3Type(id_w-type, te, we, mdecl, wt)