Advanced Documentation and Testing Tool (Adatt)

Thomas Tuerk

May 2021

Git Revision: fd83087
Thu May 6 22:22:46 2021
Thomas Tuerk

Contents

1. Motivation

Advanced **D**ocumentation **a**nd **T**esting **T**ool (Adatt) is a tool for writing formal, executable specifications. It is intended to be used by both domain and formal-method experts. Different users with different backgrounds can contribute different parts of a specification. One core idea is using the same Adatt documents for multiple purposes and allowing different users with different backgrounds to contribute to different parts of a specification.

Superficially, Adatt looks like a functional programming language with special support for writing documentation and specifications. Adatt specifications can be translated to

- high quality human readable documentation (Latex, HTML)
- executable code (Ocaml, SML, Haskell, Javascript)
- formal specifications for interactive theorem provers (HOL 4, Isabelle/HOL, Coq)

One can start with writing human readable documentation without any formal content. Declaring types, constants (including functions), which are referred to in the natural language text, can improve the documentation by e.g. ensuring there are no spelling mistakes in these or easily finding all places, a type or constant is referred to. The documentation can be further improved by adding some type definitions, constant signatures or simple test-cases. Perhaps it is even enough for already generating a little bit of executable code that can be used as a test-oracle. By adding more test cases and simple executable specifications the generated executable code becomes more complete and thereby useful. Even if not all types and constants have definitions, one can already use formal methods to reason about the existing specifications and test-cases. However, by adding non-executable properties and having some theorem prover experts add more complicated specifications a good formal specification can be written that maps well to theories already present in various interactive theorem provers.

2. Language of Adatt

Adatt's specification language is essentially higher order logic with a few simple extensions. One core extension is that certain types and functions can be marked as non-executable. This allows carving out the subset of the specification language that can be executed, i. e. that can be exported to a programming language. This extension also allows to distinguish between `Bool` and `Prop`, i. e. between executable and non-executable boolean values. This simplifies the export to backends like Coq significantly.

While Adatt's specification language has to provide nice syntax and powerful features to be convenient and user-friendly, it is also problematic to stray too far from the features directly supported by the target languages, especially the theorem prover backends. A close connection between the specifications exported by Adatt to interactive theorem provers and the input provided by users is essential. Without such a close connection, it is difficult to transfer the users intuition about the specification to proofs about the exported specification. Moreover, convincing oneself that the exported specification represents the input faithfully becomes tricky. Due to the logics used by common interactive theorem provers, this means that all features of Adatt's specification languages need to be compilable in a straightforward way to higher order logic.

The syntax of Adatt's specification language is resembling Haskell-syntax. However, in contrast to Haskell, the specification language does not rely on indentation and is more verbose. One design goal was to have an easily readable, intuitive language. This readability comes at the price of being more verbose than e. g. Haskell. However, a good user-interface mitigates this drawback by automatically generating some of the code for the user. Adatt comes with a build-in *Language Server Protocol* (LSP)[1] server. This provides i. a. rapid feedback, auto-completion and code generation. LSP is a protocol supported by many IDEs and editors. Adatt comes with an extension for *Visual Studio Code* (VS Code) [2] that uses the LSP server. Commonly used editors and IDEs come with LSP support and it is therefore hopefully straightforward to use with your favourite editor.

A typical example for the verbosity of Adatt's language is that Adatt expects (but does not strictly requires) functions to be declared. This improves readability, but seems superfluous since Adatt can infer the type of functions. However, the LSP interface can automatically generate declarations and fix type-mismatches in declarations. Another example is that Adatt requires all types and functions to state explicitly, whether they are (mutually) recursive. Again, the LSP interface can be used to add and fix the recursivity declarations.

[1] `https://microsoft.github.io/language-server-protocol/`

[2] https://code.visualstudio.com/

2.1. Noteworthy features of Adatt

2.1.1. Module System

Adatt organises specifications in *modules*. They are similar to Haskell or SML modules in that they group types, functions, tests, type-classes etc.. Adatt tries to give users maximum flexibility in how to organise specifications. One file can contain multiple modules or parts of modules and modules can consists of multiple parts defined in multiple files. The order in which statements like function and type definitions and declarations appear in a module part or in which part they occur mostly does not matter.

The envisioned usage is that different users own different files which operate on different levels of abstraction. One user can write high-level documentation and just add a few type and function names inlined in this human readable documentation using literal programming. In another file, these types could be made more concrete and some type signatures of functions could be added (e. g. as part of a technical documentation). A third file could contain test cases and yet another one executable specifications. These executable specifications could already be exported to programming languages and theorem provers. Experts for these backends could tweak the output in even more files for the same module by mapping the defined concepts to existing libraries and theories for the backends.

While this is the envisioned usage for large, complex specifications written by multiple users, the classical approach of defining one module per file is to be preferred in simpler situations. The main point is that the module system tries to be as flexible as possible to accommodate the workflow preferred by the user and suitable for a concrete project.

2.1.2. Partial and extensible definitions

An important part of the flexibility Adatt aims for is that even functions and types can be defined in multiple places.

One can start with just declaring a type. At another place it could be defined as e. g. an datatype (variant type) or a recordtype. Not all constructors or fields have to be provided at the same place. In one module part some of the constructors or fields can be defined, while another user can add more constructors or fields in another part in another file. Similarly, functions can be defined in multiple places.

Extending types with additional fields or constructors is one of the very few places in Adatt where the order in which statements appear in a module matters. When automatically generating code for the type via templates, the order, in which the fields or constructors are defined matters. For example the automatically generate code for an instance of the *Ord* typeclass for comparing values uses the order in which fields are defined to determine their precedence. While this minor dependency on order is still acceptable for auto-generated code, such dependencies would be rather confusing for manually written code. Therefore, Adatt enforces multiple definitions of a function to non-overlap. This means that none of the multiple definitions (partly) shadows another one. Therefore their order does not matter.

2.1.3. Type Classes

Adatt supports type-classes with default implementations, multiple arguments with simple functional dependencies, unification constraints and higher kinded arguments. Moreover, it is possible to attach tests and properties to type classes that all instances need to satisfy.

Type classes are a very convenient and powerful feature. Unluckily, they are not supported by some of the backends. Moreover, the supported features of type-classes differ significantly even for backends that support them. Therefore, Adatt has to be able to remove type-classes during compilation. A classical approach for removing type-classes is the introduction of type-class dictionaries. A record-type for a type-class is defined, type-class instances become values of this new type and functions using the type class get another implicit argument of this dictionary-type. When using higher order logic this approach has some restrictions due to the type-system of higher order logic: this translation cannot support higher kinded type variables and no polymorphism. Unluckily, these features are very common. Examples are the type-classes for folding over some datastructure or for monads:

```
class Monad ('M :: * -> *) where
  declare (>>=)  :: 'M 'a -> ('a -> 'M 'b) -> 'M 'b
  declare return :: 'a -> 'M 'a
end-class

class Foldable 'F where
  declare toList :: 'F 'e -> ['e]
  declare foldr :: ('e -> 'a -> 'a) -> 'a -> 'F 'e -> 'a
  declare foldl :: ('a -> 'e -> 'a) -> 'a -> 'F 'e -> 'a
  declare elem :: Eq 'e => 'e -> 'F 'e  -> Bool
  declare size :: 'F 'e -> Natural
  declare null :: 'F 'e -> Bool
end-class
```

Adatt supports such higher kinded type-classes by allowing them to be declared as *statically resolved*. With this annotation Adatt requires all usages of functions of this type class to be resolve at usage point. Because the needed instance it known statically, inlining can be used and the need for dictionaries disappears. Similarly, constants or types using higher-kinded type variables must not be present after compilation. This means that they need to be inlined. Static resolving and inlining functions are good workarounds, but can be inconvenient. A typical example is the definition of auxiliary functions like `mapM` (see section 2.1.4).

Sometimes the issue with higher kinded type-class arguments can be mitigated by using multi-parameter type-classes. In this case, functional dependencies are often vital.

They allow functions that do not use all the arguments of the type-class, because functional dependencies allow to derive the missing ones and thereby the instance to use. An example is `null` in the `Foldable` example below. By using multiple parameters, only the folding-functions, which are polymorphic in the type of the accumulator `'a`, need to be statically resolved, while other functions can be used via dictionaries:

```
class Foldable 't 'e | 't -> 'e where

  declare toList :: 't -> ['e]

  declare foldr :: ('e -> 'a -> 'a) -> 'a -> 't -> 'a
  {-# static resolve foldr #-}

  declare foldl :: ('a -> 'e -> 'a) -> 'a -> 't -> 'a
  {-# static resolve foldl #-}

  declare elem :: Eq 'e => 'e -> 't  -> Bool
  declare size :: 't -> Natural
  declare null :: 't -> Bool
end-class
```

While the main purpose of functional dependencies is to support type-class constants that do not use all argument types, they can also simplify typechecking. By providing additional information they mitigate the need for explicit type annotations. Only the non-derived arguments are used to determine the instance. Derived arguments provide extra information for the typechecker. In the example of `Foldable 't 'e` above, only the argument `'t` is used to determine the instance. The argument `'e` is used as extra information for the typechecker. For the term `elem x [y]` (without any type information for `x` or `y`) the system can for example figure out, that `[y]` is of type `List 'a` (where `'a` is a fresh type variable). With this information, the instance for lists is found, which contains the information that the argument `'t` of the type-class is instantiated to `'a`. Therefore the type of `x` is derived to be `'a`. Without the functional dependency `'t -> 'e`, the term `elem x [y]` could not be type-checked. Adatt would search for an instance `Foldable ['a] 'b`, which cannot be found, since the list instance has type `Foldable ['a] 'a`. To make the term typecheck, extra type-annotations would be needed: `elem (x :: 'a) [y :: 'a]`.

Functional dependencies often provide sufficient extra information for comfortable typechecking. In the rare cases when this is not sufficient, unification constraints can be used with type-class instances. As an example, consider the type-class `Dot`. It is a purely syntactic type-class for the inline operator `.`, which is used for e. g. function composition:

```
class Dot 'a 'b 'c | 'a 'b -> 'c where
  declare (.) :: 'a -> 'b -> 'c
end-class
```

As you can see, the type of the input have to be sufficient to derive the output type. However, otherwise, there are no restrictions. To instantiate this type-class for function composition, one could try:

```
instance Dot ('b -> 'c) ('a -> 'b) ('a -> 'c) where
  define (.) f1 f2 x := f1 (f2 x)
end-instance
```

However, if the instance is defined like this, usually many type-annotations are needed. The instance only fires if the first two arguments are functions such that the output type of the second is the input type of the first one. Often this needs to be enforced by explicit type annotations. For example `Just .  Just` cannot be typechecked without any additional type information. If type-wildcards are used to ensure that the input and output type match this looks like `(Just ::  _1 -> _) .  (Just ::  _-> _1)`.

It is usually much more convenient that the instance should fire, when `.` is applied to two functions. The type-checker should then try to unify the input and output types automatically. If they cannot be unified, this should be reported as an error. One can achieve this by abbreviating certain parts of the instance arguments with fresh type variables and add constraints for these variables:

```
instance Dot ('b -> 'c) ('a -> 'b2) ('a -> 'c) | 'b2 -> 'b where
  define (.) f1 f2 x := f1 (f2 x)
end-instance
```

One could even go further and force the usage of this instance whenever the first argument is a function. Essentially, one can choose between flexibility with instances and additional type-checking information.

```
instance Dot ('b -> 'c) 'f ('a -> 'c) | 'f -> ('a -> 'b) where
  define (.) f1 f2 x := f1 (f2 x)
end-instance
```

Adatt's unification constraints take the same role as Haskell's type equality constraints `ty1   ty2`. However, due to simpler typechecker of Adatt, the left type of Adatt's unification constraints are always type variables. This simpler form suffices for Adatt's purposes.

2.1.4. Monads

Adatt supports monads via the type class shown above. There is do-notation for monads, which uses this type-class. However, there are restrictions compared to languages like Haskell. The monad type-class uses a higher kinded type-argument and therefore needs to be statically resolved. This means that wherever monads and do-notation is used, it needs to be possible to statically figure out during type-checking which monad is used. This makes it a bit awkward to define auxiliary monad functions, which are very common in languages like Haskell.

A typical example is `mapM`, a map function for monads. To define such a map function, one could try the following (broken) definition:

```
declare mapM :: Monad 'M => ('a -> 'M 'b) -> ['a] -> 'M ['b]
define rec mapM _ [] := return []
         | mapM f (x:xs) :=
              do
                fx <- f x;
                fxs <- mapM f xs;
                return $ fx : fxs
              end
```

In this code-fragment, the Monad `'M` cannot be resolved statically (which is e. g. required by the do-notation). Therefore this code is invalid. Trying to declare `mapM` as inline and thereby move the static resolving of the monad to the usage points of `mapM` fails as well, because `mapM` is recursive. However, we can easily remove this recursion by using a standard fold function. This results in:

```
declare mapM :: !Monad 'M => ('a -> 'M 'b) -> ['a] -> 'M ['b]
define mapM f l :=
  List.foldr (fn p q ->
    bind p (fn x -> bind q (fn xs -> return (x:xs)))
  ) (return []) (map f l)
{-# inline mapM #-}
```

This code defines a general `mapM` function that can be used conveniently inside Adatt. However, there is still a problem when exporting code using this function to a theorem prover or a programming language backend. The body of the function gets inlined at all usage points. This obfuscates the generated code and makes it harder to reason about it. This can be mitigated by introducing an auxiliary function to keep the structure of `mapM` as much as possible while inlining the monad functions. A first, broken attempt to achieve this might look like this:

```
declare mapM' ::
     ('M 'b -> ('b -> 'M ['b]) -> 'M ['b])
  -> ('M ['b] -> (['b] -> 'M ['b]) -> 'M ['b])
  -> (['b] -> 'M ['b])
  -> ('a -> 'M 'b)
  -> ['a] -> 'M ['b]
define mapM' bind1 bindl ret f l :=
  List.foldr (fn p q ->
    bind1 p (fn x -> bindl q (fn xs -> ret (x:xs)))
  ) (ret []) (map f l)
```

```
declare mapM :: !Monad 'M => ('a -> 'M 'b) -> ['a] -> 'M ['b]
define mapM :=  mapM' bind bind return
{-# inline mapM #-}
```

Unluckily, now the auxiliary function `mapM'` uses the higher-kinded type variable `M'` and can therefore not be exported to higher-order-logic. We could declare `mapM'` as inlined, but this would defy the purpose of introducing it in the first place. More sensibly, we could relax the type of `mapM'` by replacing `'M 'b` and `'M ['b]` with fresh type variables.

```
declare mapM' ::
     ('mb -> ('b -> 'mbs) -> 'mbs)
  -> ('mbs -> (['b] -> 'mbs) -> 'mbs)
  ->  (['b] -> 'mbs)
  -> ('a -> 'mb)
  -> ['a] -> 'mbs
```

So, there is some support for monads. Due to the restrictions of the target languages this support is limited compared with languages like Haskell however. Using some tricks and workarounds, it's nevertheless possible to use monads without too much inconvenience.

2.1.5. Templates

The core function of Adatt is to translate specifications to text-based outputs for various backends. To support this core function Adatt comes with a simple template language inspired by languages like Liquid[3] or Jinja[4]. Simple templates are just text with special tags that are replaced with values, when the template is *rendered*. This is sufficient for just mapping a constant or type to some target representation. However, Adatt's template language features a full programming language. It is a simple, imperative, statically typed language with special support for text generation. There are variables and common control structures like conditional execution or loops. Templates can call other templates. This includes recursive calls. Since templates can also return data values, these template-calls can take the role of function calls in other languages. There are plenty a special build-in functions for text-output like functions for generating unique names.

Besides rendering backend output, templates can also be used to generate Adatt input. Parsing, basic typechecking and rendering code-templates are interleaved. Templates can generate additional Adatt input which is parsed and typechecked in turn. This generated code can contain arbitrary Adatt statements, including template definitions and code-generation statements. The code-generation templates can access basic information

[3]`https://shopify.github.io/liquid`
[4]`https://jinja.palletsprojects.com`

about types, constants and classes. However, there are limitations: if a type or constant is defined by code-generation it cannot extend or refine definitions outside the generated code. So it is for example not possible to define some type constructors manually and add further constructors via code generation. Moreover, code-generation happens at an early stage of processing an Adatt module. This means that only basic information about types, classes and constants defined within this module is available. Slightly simplified, everything that requires type inference, is not available. It is for example possible to access the names of the constructors of a datatype and the number of arguments of these constructors. It is however not possible to access the types of the constructor arguments. For normal constants, i. e. not fields and constructors, it is not even possible to get the number of their arguments, because their type might be (partially) inferred. A positive effect of generating code at an early stage of processing is that types and constants defined via code-generation can be used in the same module outside the generated code.

The main usage of code-generation templates is automatically generating instances for common type-classes. They are for example used to automatically derive instances of the `Eq` type-class[5] for newly defined data- or record-types via derive statements. However, one can use code-generation for arbitrary purposes. A trivial example is defining abbreviation types for tuples whose elements all have the same type:

```
template BuildTupleAbbrev (n :: Nat) :=
  '''type Tuple{{= n =}} \'a := {{=!BuildTuple(replicate(n, "'a"))=}}'''

generate-code '''
  {{% for var i := 2 to 9 %}}
  {{! BuildTupleAbbrev(i) !}}
  {{% end-for %}}
'''
```

In the code above, a template `BuildTupleAbbrev` is defined, which given a number `n` emits code for defining an abbreviation type. It uses a build-in function `replicate` to generate a list containing n-times `"'a"`. Then a user-defined template `BuildTuple` is called to turn this list into a tuple-representation as a String. In the generate-code statement the template `BuildTupleAbbrev` is called multiple times inside a loop. These statements generate the code:

```
type Tuple2 'a := ('a, 'a)
type Tuple3 'a := ('a, 'a, 'a)
type Tuple4 'a := ('a, 'a, 'a, 'a)
type Tuple5 'a := ('a, 'a, 'a, 'a, 'a)
type Tuple6 'a := ('a, 'a, 'a, 'a, 'a, 'a)
type Tuple7 'a := ('a, 'a, 'a, 'a, 'a, 'a, 'a)
type Tuple8 'a := ('a, 'a, 'a, 'a, 'a, 'a, 'a, 'a)
type Tuple9 'a := ('a, 'a, 'a, 'a, 'a, 'a, 'a, 'a, 'a)
```

[5]which provides boolean equality

Templates can be registered to be used for generating type class instances (see Section 4). This allows to use special syntax for generating instances. Examples are:

```
derive Maybe instances (Eq, EqP, Ord)
deriving instance Eq 'a => Eq (Maybe 'a)
```

2.1.6. List and Set-Comprehensions

Currently Adatt does not have any list- or set-comprehensions. However, there are monad instances for lists and sets which allow to write something very close to list-comprehensions in do notation. The Haskell list-comprehension

```
[(x, y, z) | x <- l1, let z = f x, y <- l2, x > y]
```

for example becomes in Adatt's do notation:

```
do
  x <- l1;
  z := f x;
  y <- l2;
  guard (x > y);
  return (x,y,z)
end
```

For the moment there are no plans to add list- or set-comprehensions to Adatt. This would however be easy, because there is such a straightforward mapping to the existing list- and set-monads.

2.1.7. Literals

Adatt supports char, string, natural number and decimal literals. These are implemented via statically resolved type-classes. There are type-classes for each type of literal, which contain a single function turning an internal representation of the literal into the target type. By instantiating these type classes, users can add support for their own types. Templates have special support for the internal representation. This allows the definition of proper backend representations of all user-defined literal types.

2.1.8. Default values

Each type in Adatt needs to be inhabited. When defining a new type, Adatt tries to compute a default-value for this type. If this computation fails, an error message is produced. This automatically defined value, which is present for all types is available via the special construct `???`.

2.1.9. Language Server Protocol

3. Configuration

Adatt can be configured via system-wide, project-wide and directory-wide configuration files as well as configuration-pragmas within source files. ...

4. Templates

Adatt contains a template language for producing backend output, but also for generating input for Adatt that gets processed again. Adatt's template language is inspired by languages like Liquid[1] or Jinja[2]. It features a simple, statically typed imperative language aimed at producing text.

Simple templates are text with special tags. Such simple templates might be used for e. g. for the backend representation of an Adatt constant. The template is used to generate text by replacing the tags with text. This process is called *rendering*. As very simple template example is `'''2 + 5 = {{= 2 + 5 =}}'''`. It uses a single tag `{{= 2 + 5 =}}`. This tag is replaced during rendering with the computed value `7` thereby resulting in the output `2 + 5 = 7`. Templates also feature control structures, various formatting functions, generation of fresh names, (recursive) template calls . . . .

4.1. types of tags and templates

statement tags

The most basic and most general type of tag are *statement tags*. All other tag-types can be simulated using statement tags. They are enclosed in double curly brackets (i. e. `{{ ... }}`), where the brackets are followed by a space. Template statements are usually used to change the state somehow. One can e. g. declare variables or assign values to them. However, there are also print-statements that output text. Moreover, there are control structures like conditional-execution or loops. One can also render other templates.

print expression tags

One very common operation is inserting values into the text. *Print expression tags* are tags enclosed in double curly brackets qualified with `=` (i. e. `{{=` *e* `=}}`). They evaluate the expression `e` and print it. If there are newlines produced by the expression, they are automatically indented. This means that all lines start in the same column the print expression tag starts. The print expression tag `{{=` *e* `=}}` is short for the statement tag `{{ printWithIndent(`*e*`) }}`.

[1] `https://shopify.github.io/liquid`
[2] `https://jinja.palletsprojects.com`

block tags

Many statements require whole blocks of templates as arguments. Typical examples are control structures like conditional execution or loops. In order to avoid the need to express the whole templates in this block with statements, one can use block tags. One very common operation is inserting values into the text. *Block tags* are tags enclosed in double curly brackets qualified with `%` (i. e. `{{% ... %}}`). There is at least an opening and a matching closing block tag. Some tags allow addition tags (e. g. else-tags branches for conditional execution tags). And example for a template using block tags is:

```
{{% if (cond) %}}
true-case
{{= e =}}
{{% else %}}
false-case
{{% end-if %}}
```

Using a statement tag, this example template could be expressed as

```
{{ if (cond) {
     println("true-case");
     printlnWithIndent(e);
   } else {
     println("false-case");
   } }}
```

Statement tags usually open new variable scopes, i. e. template variables defined within such a block are not accessible outside the block. The only exception are name-scopes (see below).

call template tags

To render another template and include its result text, call template tags can be used. These are tags qualified by `!` (i. e. `{{!`*`templateName(args)`*`!}}`)). This can be simulated by statement templates, that evaluate the called template, throw away all return values except the produced text and print this text.

template types

Templates can call (render) other templates. As call template tags show, this is frequently used to insert the text produced by another template. However, templates can also used like functions in common programming languages. Such templates don't produce any text, but just return values. Such templates consist of just a list of statements, which may not include print statements.

4.2. template expressions

Template statements like variable assignments or prints use expressions as arguments. As usual, these expressions are used to compute values. They are mostly side-effect free. The only exception is marking names as used when generating fresh names (see below). However, there is nothing like e. g. an increment operator that changes the value of a variable as side-effect. Adatt templates are statically typed. To explain template expressions, we need to discuss template types first.

4.2.1. template types

Bool
: boolean value, i. e. `True` or `False`

Char
: single unicode character

String
: unicode string, in many respects this behaves like a list of characters

Nat
: Natural numbers of arbitrary size, i. e. whole numbers starting at 0. There are no negative numbers. When using counters in for-loops this sometimes needs to be taken into consideration.

The `Nat` type corresponds to the `BuildInNum` type in Adatt's specification language. It can be used to format literals of this type for various backends. For this purpose, the default encoding base is also available for numbers of type `Nat`. If the number comes from some literal input, the base is the base of this encoding (e. g. `0x0A` returns base 16 while input `11` returns base 10). If the number was created via some computation, the default base is always 10.

Decimal
: Positive rational number. The name *`Decimal`* is slightly misleading, because internally quotients are used. So in the internal representation numbers like 1/3 are represented exactly. However, input for numbers of type `Decimal` are decimal literals (e. g. 10.3212).

The `Decimal` type corresponds to the `BuildInDecimal` type in Adatt's specification language. It can be used to format literals of this type for various backends.

ConstID
: reference to some constant from an Adatt module. The name can be pretty-printed taking the local namespace of the location it is printed in consideration. Perhaps more interestingly, some information can be looked up (e. g. the number of arguments of the constant).

TypeID
: reference to some type from an Adatt module. The name can be pretty-printed taking the local namespace of the location it is printed in consideration. Perhaps more interestingly, some information can be looked up (e. g. the constructors of a datatype).

ClassID
: reference to some type from an Adatt module. The name can be pretty-printed taking the local namespace of the location it is printed in consideration.

TemplateID(*template-signature*)
: reference to a template. These can be used to call the template. That way they act like function pointers. Because templates are statically typed, the signature of the template needs to be provided. Signatures are given in the form *`arg type 1-> ... -> arg type n -> result type`*. The result type is special. It can be any normal template type (including tuple types). This indicates a templates producing values of the given type but not producing text. To indicate templates producing text there is a special type `Text`, which means not returning any other results, and a special tuple type, whose first component is `Text`.

Type
: a full Adatt type. It can be pretty-printed. Notice that `TypeID` is a reference to a type (e. g. `Maybe`). It can be used to lookup information about the type. In contrast `Type` is a concrete type with all arguments applied (e. g. `Maybe 'a`). It can only be pretty-printed.

Term
: a full Adatt term. It can be pretty-printed. Notice that `ConstID` is a reference to a constant. It can be used to lookup information about the type. In contrast `Term` is a term. This means it can be a constant, a constant applied to arguments or even things like case-expressions. It can only be pretty-printed.

()
: unit value, used in rare places for technical reasons

(ty1, . . . , tyn)
: tuple type. Two or more types can be combined to form a type for tuples of the given element number and element types.

[ty]
: list of elements of given type

4.2.2. Type Coercion

Template types can be coerced into other types. This often happens automatically, when e. g. assigning values of mismatching types to variables or when calling a build-in template function. Moreover coercions are possible using the syntax `coerce(target-type,` `expression)`. Values of type Nat can be coerced to `Decimal` and a value of any type can be coerced to `String`. A tuple or list can be coerced to another one, if all the elements can be coerced.

4.2.3. Literals

`Bool` literals
: `True`, `False`

`Nat` literals
: Natural number literals can be encoded base 10, base 16 (prefix 0x), base 8 (prefix 0o) or base 2 (prefix 0b). Examples all representing the statement value are `12`, `0x0C`, `0o14`, `0b1100`.

`Decimal` literals
: Decimal number literals may only be given in base 10 using dots as decimal separator. An example is `12.5`. Notice that `12.0` is a `Decimal` literal, whereas `12` is a `Num` literal.

`Char` literals
: Char literals are enclosed in single quotes. Examples are `'a'` or `'b'`. Common escape sequences can be used. Examples are `'\n'`, `'\''` or `'\t'`. By using the syntax `'\nnnn'` it is also possible to give chars by using their unicode number. `'\120'` is for example representing the char `'x'`.

`String` literals
: String literals are enclosed in double quotes. Examples are `"abcdef"` or `"Hello world"`. Common escape sequences can be used. These are the same escape sequences as for character literals.

unit literals
: `()`

tuple literals
: tuples can be written enclosed in parenthesis with the elements separating by commata. Examples are `(1)` or `(1, "a", 'b', 12.5)`.

list literals
: lists can be written enclosed in brackets with the elements separating by commata. Examples are `[1]` or `[1, 2, 3]`. The empty list can be represented by `[] :: `*`type`*. Notice that an explicit type annotation is needed for the empty list.

4.2.4. Functions and Attributes

Template expressions can use infix, prefix and postfix functions. These functions can be overloaded to have different meanings for different argument types. Prefix functions take a list of arguments in parenthesis. There might be default values for certain arguments. Postfix functions are written as attributes, i. e. directly after an expression separated by `.`.

Arithmetic Functions

+
: `True`, `False`

A. Grammar

Character Classes

```
⟨opChar⟩            ::= ‘:’ | ‘!’ | ‘#’ | ‘$’ | ‘%’ | ‘&’ | ‘*’ | ‘+’ | ‘/’ | ‘\’ | ‘<’ | ‘=’ | ‘>’ | ‘?’ |
                        ‘+’ | ‘|’ | ‘-’ | ‘~’

⟨letterChar⟩        ::= ‘a’ | ... | ‘z’ | ‘A’ | ... | ‘Z’

⟨digitChar⟩         ::= ‘0’ | ... | ‘9’

⟨identChar⟩         ::= ⟨letterChar⟩ | ⟨digitChar⟩ | ‘_’ | ‘'’
```

Identifiers and Operators

```
⟨ident⟩             ::= ⟨letterChar⟩ ⟨identChar⟩*

⟨identList⟩         ::= ⟨ident⟩ | ⟨ident⟩ ‘,’ ⟨identList⟩

⟨ident2⟩            ::= ⟨ident⟩ | ⟨ident⟩ ‘.’ ⟨ident⟩

⟨ident2List⟩        ::= ⟨ident2⟩ | ⟨ident2⟩ ‘,’ ⟨ident2List⟩

⟨idents⟩            ::= ⟨ident⟩ | ⟨ident⟩ ‘.’ ⟨idents⟩

⟨oper⟩              ::= ⟨opChar⟩+

⟨identOrOper⟩       ::= ⟨ident⟩ | ‘(’ ⟨oper⟩ ‘)’

⟨identOrOperList⟩ ::= ⟨identOrOper⟩ | ⟨identOrOper⟩ ‘,’ ⟨identOrOperList⟩

⟨identifier⟩        ::= ⟨ident2⟩ | ‘(’ ⟨oper⟩ ‘)’

⟨operator⟩          ::= ⟨oper⟩ | ‘`’ ⟨ident2⟩ ‘`’

⟨namespaceIdent⟩ ::= [ (‘const’ | ‘type’ | ‘class’ | ‘template’) ] ⟨identOrOper⟩
```

Literals

⟨*unitLiteral*⟩	::=	‘`()`’
⟨*charLiteral*⟩	::=	‘`'`’ `char` ‘`'`’
⟨*stringLiteral*⟩	::=	‘`"`’ `string` ‘`"`’
⟨*natLiteral*⟩	::=	‘`0b`’ `binary number`
	|	‘`0o`’ `octal number`
	|	`decimal number`
	|	‘`0x`’ `hex number`
⟨*decimalLiteral*⟩	::=	`decimal number` ‘`.`’ `decimal number`
⟨*literal*⟩	::=	⟨*stringLiteral*⟩
	|	⟨*charLiteral*⟩
	|	⟨*unitLiteral*⟩
	|	⟨*natLiteral*⟩
	|	⟨*decimalLiteral*⟩

Types

⟨*typeVar*⟩	::=	‘’’ ⟨*ident*⟩
⟨*typeVarList*⟩	::=	⟨*typeVar*⟩ | ⟨*typeVar*⟩ ‘`,`’ ⟨*typeVarList*⟩
⟨*typeWildcard*⟩	::=	‘`_`’
⟨*typeUnit*⟩	::=	‘`()`’
⟨*typeList*⟩	::=	⟨*type*⟩ | ⟨*type*⟩ ‘`,`’ ⟨*typeList*⟩
⟨*type*⟩	::=	⟨*typeVar*⟩
	|	⟨*identifier*⟩ ⟨*type*⟩*
	|	⟨*type*⟩ ‘`->`’ ⟨*type*⟩
	|	‘`[`’ ⟨*type*⟩ ‘`]`’
	|	‘`(`’ ⟨*typeList*⟩ ‘`)`’
⟨*typeConstraint*⟩	::=	[‘`!`’] ⟨*identifier*⟩ ⟨*typeVar*⟩*
⟨*typeConstraintList*⟩	::=	⟨*typeConstraint*⟩ | ⟨*typeConstraint*⟩ ‘`,`’ ⟨*typeConstraintList*⟩
⟨*typeConstraints*⟩	::	⟨*typeConstraint*⟩ | ‘`(`’ ⟨*typeConstraintList*⟩ ‘`)`’

Patterns

⟨*patWild*⟩ ::= ‘`_`’

⟨*patVar*⟩ ::= ⟨*ident*⟩

⟨*patTuple*⟩ ::= ‘`(`’ ⟨*patternList*⟩ ‘`)`’

⟨*patList*⟩ ::= ‘`[`’ ⟨*patternList*⟩ ‘`]`’

⟨*patTyped*⟩ ::= ⟨*pattern*⟩ ‘`::`’ ⟨*type*⟩

⟨*patAs*⟩ ::= ⟨*ident*⟩ ‘`@`’ ⟨*pattern*⟩

⟨*patApp*⟩ ::= ⟨*pattern*⟩ ⟨*pattern*⟩ | ⟨*pattern*⟩ ⟨*operator*⟩ ⟨*pattern*⟩ | ⟨*operator*⟩ ⟨*pattern*⟩

⟨*patRecord*⟩ ::= ‘`<|`’ ⟨*patRecordFieldList*⟩ ‘`|>`’

⟨*patRecordField*⟩ ::= ⟨*identifier*⟩ ‘`=`’ ⟨*pattern*⟩

⟨*patRecordFieldList*⟩ ::= ⟨*patRecordField*⟩ | ⟨*patRecordField*⟩ ‘`,`’ ⟨*patRecordFieldList*⟩

⟨*patParen*⟩ ::= ‘`(`’ ⟨*pattern*⟩ ‘`)`’

⟨*patternList*⟩ ::= ⟨*pattern*⟩ | ⟨*pattern*⟩ ‘`,`’ ⟨*patternList*⟩

⟨*pattern*⟩ ::= ⟨*patWild*⟩
 | ⟨*patVar*⟩
 | ⟨*literal*⟩
 | ⟨*patTuple*⟩
 | ⟨*patList*⟩
 | ⟨*patAs*⟩
 | ⟨*patRecord*⟩
 | ⟨*patApp*⟩
 | ⟨*patTyped*⟩
 | ⟨*patParen*⟩

Let-Patterns

⟨*letpatWild*⟩ ::= ‘`_`’

⟨*letpatVar*⟩ ::= ⟨*ident*⟩

⟨*letpatTyped*⟩ ::= ‘`(`’ ⟨*letPattern*⟩ ‘`::`’ ⟨*type*⟩ ‘`)`’

⟨*letpatTuple*⟩ ::= ‘`(`’ ⟨*letPatternList*⟩ ‘`)`’

⟨*letPatternList*⟩ ::= ⟨*letPattern*⟩ | ⟨*letPattern*⟩ ‘`,`’ ⟨*letPatternList*⟩

⟨*letPattern*⟩ ::= ⟨*letpatWild*⟩
| ⟨*letpatVar*⟩
| ⟨*letpatTyped*⟩
| ⟨*letpatTuple*⟩

Terms

⟨*termVar*⟩ ::= ⟨*ident*⟩

⟨*termTuple*⟩ ::= ‘`(`’ ⟨*terms*⟩ ‘`)`’

⟨*termList*⟩ ::= ‘`[`’ ⟨*terms*⟩ ‘`]`’

⟨*termTyped*⟩ ::= ⟨*term*⟩ ‘`::`’ ⟨*type*⟩

⟨*termApp*⟩ ::= ⟨*term*⟩ ⟨*term*⟩ | ⟨*term*⟩ ⟨*operator*⟩ ⟨*term*⟩ | ⟨*operator*⟩ ⟨*term*⟩

⟨*termRecord*⟩ ::= ‘`<|`’ ⟨*termRecordFieldList*⟩ ‘`|>`’
| ‘`<|`’ ⟨*identifier*⟩ ‘`with`’ ⟨*termRecordFieldList*⟩ ‘`|>`’

⟨*termRecordField*⟩ ::= ⟨*identifier*⟩ ‘`:=`’ ⟨*term*⟩

⟨*termRecordFieldList*⟩ ::= ⟨*termRecordField*⟩ | ⟨*termRecordField*⟩ ‘`,`’ ⟨*termRecordFieldList*⟩

⟨*termParen*⟩ ::= ‘`(`’ ⟨*term*⟩ ‘`)`’

⟨*termCond*⟩ ::= ‘`if`’ ⟨*term*⟩ ‘`then`’ ⟨*term*⟩ ‘`else`’ ⟨*term*⟩

⟨*termLambda*⟩ ::= ‘`fn`’ ⟨*letPattern*⟩+ ‘`->`’ ⟨*term*⟩

⟨*letBinding*⟩ ::= ‘`var`’ ⟨*letPattern*⟩ ‘`=`’ ⟨*term*⟩
| ‘`fun`’ ⟨*ident*⟩ ⟨*letPattern*⟩+ ‘`=`’ ⟨*term*⟩

⟨*termLet*⟩ ::= ‘`let`’ ⟨*letBinding*⟩* ‘`in`’ ⟨*term*⟩ ‘`end`’

⟨*quant*⟩ ::= ‘`forall`’ | ‘`exists`’ | ‘`uexists`’ | ‘`bforall`’ | ‘`bexists`’ | ‘`buexists`’

⟨*termQuant*⟩ ::= ⟨*quant*⟩ ⟨*letPattern*⟩+ ‘`.`’ ⟨*term*⟩

⟨*termBoolCasesCond*⟩ ::= ⟨*term*⟩ ‘`->`’ ⟨*term*⟩
| ‘`otherwise`’ ‘`->`’ ⟨*term*⟩
| ‘`_`’ ‘`->`’ ⟨*term*⟩

⟨*termBoolCasesConds*⟩ ::= ⟨*termBoolCasesCond*⟩ | ⟨*termBoolCasesCond*⟩ ‘`|`’ ⟨*termBoolCasesConds*⟩

```
⟨termBoolCases⟩      :: 'cases' ⟨termBoolCasesConds⟩ 'end'

⟨termCaseRow⟩       ::= ⟨pattern⟩ [ 'when' ⟨term⟩ ] '->' ⟨term⟩

⟨termCaseRows⟩      ::= ⟨termCaseRow⟩ | ⟨termCaseRow⟩ '|' ⟨termCaseRows⟩

⟨termCase⟩           :: 'case' ⟨term⟩ 'of' ⟨termCaseRows⟩ 'end'

⟨termDo⟩             :: 'do' ⟨termDoRows⟩ 'end'

⟨termDoRows⟩         :: ⟨termDoRowStatement⟩ | ⟨termDoRow⟩ ';' ⟨termDoRows⟩

⟨termDoRow⟩          :: ⟨termDoRowLet⟩ | ⟨termDoRowBind⟩ | ⟨termDoRowStatement⟩

⟨termDoRowLet⟩       :: ⟨letPattern⟩ ':=' ⟨term⟩

⟨termDoRowBind⟩      :: ⟨letPattern⟩ '<-' ⟨term⟩

⟨termDoRowStatement⟩ :: ⟨term⟩

⟨terms⟩             ::= ⟨term⟩ | ⟨term⟩ ',' ⟨terms⟩

⟨term⟩              ::= ⟨termVar⟩
                      | ⟨literal⟩
                      | ⟨termTuple⟩
                      | ⟨termList⟩
                      | ⟨termApp⟩
                      | ⟨termTyped⟩
                      | ⟨termParen⟩
                      | ⟨termCond⟩
                      | ⟨termLambda⟩
                      | ⟨termLet⟩
                      | ⟨termQuant⟩
                      | ⟨termBoolCases⟩
                      | ⟨termCase⟩
                      | ⟨termDo⟩
```

Pragmas

```
⟨pragmaNoPrelude⟩ ::= 'NoImplicitPrelude'

⟨pragmaIssue⟩     ::= 'ERROR' string
                    | 'WARNING' string
                    | 'INFO' string
                    | 'HINT' string
                    | 'TODO' string
```

```
⟨severityNoError⟩ ::= 'warning'
                   |  'warn'
                   |  'info'
                   |  'hint'

⟨severity⟩        ::= 'error'
                   |  'err'
                   |  ⟨severityNoError⟩

⟨severityMaybe⟩   ::= ⟨severity⟩
                   |  '-'
                   |  'ignore'

⟨namingConventionDeclaration⟩ ::= 'reset'
                   |  ⟨severityMaybe⟩
                   |  ⟨severity⟩ '"' description '"' '"' regularExp '"'

⟨pragmaNamingConvention⟩ ::= 'NAMING-CONVENTION' name ⟨namingConventionDeclaration⟩

⟨pragmaAllowSimilarNames⟩ ::= 'ALLOW-SIMILAR-NAMES' ⟨nameList⟩

⟨nameList⟩        ::= '"' name '"'
                   |  '"' name '"' ',' ⟨nameList⟩

⟨pragmaSeverity⟩  ::= 'severity' problem ⟨severityMaybe⟩
                   |  'severity' problem 'reset'

⟨pragmaInline⟩    ::= 'inline' ⟨namespaceIdent⟩

⟨pragmaStaticResolve⟩ ::= 'static resolve' ⟨namespaceIdent⟩

⟨showHide⟩        ::= 'show'
                   |  'on'
                   |  'hide'
                   |  'off'

⟨pragmaReporting⟩ ::= 'reporting' [⟨severityNoError⟩] ⟨showHide⟩

⟨pragmaDebug⟩     ::= 'debug on'
                   |  'debug off'

⟨pragmaNumLiteralBoundaries⟩ ::= 'set-num-literal-boundaries' ⟨numLiteralBound⟩
                      ⟨numLiteralBound⟩

⟨numLiteralBound⟩ ::= decimal number
                   |  '-'
```

⟨*pragmaTemplateSteps*⟩ ::= ‘`template-render-steps`’ ‘`reset`’
| ‘`template-render-steps`’ `decimal number`

⟨*pragmaWarningsGeneratedCode*⟩ ::= ‘`warnings-in-generated-code`’ (‘`on`’ | ‘`off`’ | ‘`reset`’)

⟨*pragmaMinimal*⟩ ::= ‘`minimal`’ ⟨*memberGroups*⟩

⟨*memberGroups*⟩ ::= ⟨*identOrOperList*⟩
| ⟨*identOrOperList*⟩ ‘`;`’ ⟨*memberGroups*⟩

⟨*pragma*⟩ :: ⟨*pragmaNoPrelude*⟩
| ⟨*pragmaSeverity*⟩
| ⟨*pragmaNamingConvention*⟩
| ⟨*pragmaInline*⟩
| ⟨*pragmaStaticResolve*⟩
| ⟨*pragmaIssue*⟩
| ⟨*pragmaReporting*⟩
| ⟨*pragmaDebug*⟩
| ⟨*pragmaWarningsGeneratedCode*⟩
| ⟨*pragmaNumLiteralBoundaries*⟩
| ⟨*pragmaTemplateSteps*⟩
| ⟨*pragmaMinimal*⟩

Statements

⟨*stmtPragma*⟩ ::= ‘`{-#`’ ⟨*pragma*⟩ ‘`#-}`’

⟨*stmtAlias*⟩ ::= ‘`alias`’ ⟨*identOrOper*⟩ ‘`:=`’ ⟨*identifier*⟩

⟨*recDecl*⟩ ::= empty | ‘`rec`’ | ‘`mutual-rec`’ ‘`(`’ ⟨*identOrOperList*⟩ ‘`)`’

⟨*stmtDeclare*⟩ ::= ‘`declare`’ [‘`non-exec`’] ⟨*identOrOper*⟩ [‘`::`’ [⟨*typeConstraints*⟩ ‘`=>`’] ⟨*type*⟩]

⟨*stmtDefineClause*⟩ ::= ⟨*identOrOper*⟩ ⟨*pattern*⟩* [‘`when`’ ⟨*term*⟩] ‘`:=`’ ⟨*term*⟩

⟨*stmtDefineClauses*⟩ ::= ⟨*stmtDefineClause*⟩
| ⟨*stmtDefineClause*⟩ ‘`|`’ ⟨*stmtDefineClauses*⟩

⟨*stmtDefine*⟩ ::= ‘`define`’ [‘`non-exec`’] ⟨*recDecl*⟩ ⟨*stmtDefineClauses*⟩

⟨*stmtTest*⟩ ::= ‘`test`’ [⟨*typeConstraints*⟩ ‘`=>`’] ⟨*ident*⟩ ⟨*letPattern*⟩* ‘`:=`’ ⟨*term*⟩

⟨*stmtProperty*⟩ ::= ‘`property`’ [⟨*typeConstraints*⟩ ‘`=>`’] ⟨*ident*⟩ ⟨*letPattern*⟩* ‘`:=`’ ⟨*term*⟩

```
⟨stmtClass⟩          ::= ‘class’ [⟨typeConstraints⟩ ‘=>’] ⟨ident⟩ ⟨typeVar⟩* [‘|’ ⟨functionalDeps⟩]
                         ‘where’
                            ⟨simpleStatement⟩*
                         ‘end-class’

⟨functionalDeps⟩     ::= ⟨functionalDep⟩
                       | ⟨functionalDep⟩ ‘,’ ⟨functionalDeps⟩

⟨functionalDep⟩      ::= ⟨typeVarList⟩ ‘->’ ⟨typeVarList⟩

⟨uniConstraints⟩     ::= ⟨uniConstraint⟩
                       | ⟨uniConstraint⟩ ‘,’ ⟨uniConstraints⟩

⟨uniConstraint⟩      ::= ⟨typeVarList⟩ ‘->’ ⟨type⟩

⟨stmtInstance⟩       ::= ‘instance’ [⟨typeConstraints⟩ ‘=>’] ⟨ident2⟩ ⟨type⟩* [‘|’ ⟨uniConstraints⟩]‘where’
                            ⟨simpleStatement⟩*
                         ‘end-instance’

⟨stmtType⟩           ::= ‘type’ [‘non-exec’] ⟨recDecl⟩ ⟨ident⟩ ⟨typeVar⟩* [ ‘:=’ ⟨type⟩ ]

⟨stmtDatatypeClause⟩ ::= ⟨ident⟩ ⟨type⟩*

⟨stmtDatatypeClauses⟩ ::= ⟨stmtDatatypeClause⟩
                       | ⟨stmtDatatypeClause⟩ ‘|’ ⟨stmtDatatypeClauses⟩

⟨stmtDatatype⟩       ::= ‘datatype’ [‘non-exec’] ⟨recDecl⟩ ⟨ident⟩ ⟨typeVar⟩* ‘:=’
                            ⟨stmtDatatypeClauses⟩


⟨stmtRecordtypeField⟩ ::= ⟨ident⟩ ‘::’ ⟨type⟩

⟨stmtRecordtypeFields⟩ ::= ⟨stmtrRecordtypeField⟩
                       | ⟨stmtDatatypeField⟩ ‘,’ ⟨stmtDatatypeFields⟩

⟨stmtRecordtype⟩     ::= ‘recordtype’ [‘non-exec’] ⟨recDecl⟩ ⟨ident⟩ ⟨typeVar⟩* ‘:=’ ⟨ident⟩
                         ‘<|’
                            ⟨stmtRecordtypeFields⟩
                         ‘|>’ [ ‘deriving’ ‘(’ ⟨deriveGoals⟩ ‘)’ ]

⟨stmtConstructorFamily⟩ ::= ‘constructor-family’ ⟨identOrOper⟩ ‘(’ ⟨identOrOperList⟩
                         ‘)’ [ ‘-’ ]

⟨stmtTemplate⟩       ::= ‘template’ ⟨ident⟩ ‘(’ ⟨templateArgList⟩ ‘)’ [ ‘::’ TemplateType
                         ] ‘:=’ (‘{{’ template statements ‘}}’) | (‘’’’ template body
                         ‘’’’)
```

⟨*stmtRegisterTemplate*⟩ ::= ‘`register-derive-template`’ ⟨*ident*⟩ ⟨*ident2*⟩ [⟨*ident2*⟩ | ‘`-`’] [‘`as`’ ‘`"`’ `label` ‘`"`’]

⟨*stmtDerive*⟩ ::= ‘`derive`’ ⟨*ident2*⟩ ‘`instances`’ (⟨*ident2*⟩ | ‘`(`’ ⟨*identList*⟩ ‘`)`’) [‘`via`’ ‘`"`’ `label` ‘`"`’]

⟨*stmtDeriving*⟩ ::= ‘`deriving`’ ‘`instance`’ [⟨*typeConstraints*⟩ ‘`=>`’] ⟨*ident2*⟩ ⟨*type*⟩* [‘`via`’ ‘`"`’ `label` ‘`"`’]

⟨*simpleStatement*⟩ ::= ⟨*stmtDeclare*⟩
  | ⟨*stmtDefine*⟩
  | ⟨*stmtTest*⟩
  | ⟨*stmtProperty*⟩
  | ⟨*stmtPragma*⟩

⟨*statement*⟩ ::= ⟨*simpleStatement*⟩
  | ⟨*stmtClass*⟩
  | ⟨*stmtInstance*⟩
  | ⟨*stmtType*⟩
  | ⟨*stmtDatatype*⟩
  | ⟨*stmtRecord*⟩
  | ⟨*stmtAlias*⟩
  | ⟨*stmtConstructorFamily*⟩
  | ⟨*stmtTemplate*⟩
  | ⟨*stmtRegisterTemplate*⟩
  | ⟨*stmtDerive*⟩
  | ⟨*stmtDeriving*⟩

Modules

⟨*moduleName*⟩ ::= ⟨*idents*⟩

⟨*modulePart*⟩ ::= ‘`part`’ ⟨*ident*⟩ ‘`+`’ `number`
  | ‘`part`’ ⟨*ident*⟩ ‘`-`’ `number`

⟨*import*⟩ ::= ‘`import`’ [‘`qualified`’] ⟨*moduleName*⟩ [‘`as`’ ⟨*ident*⟩] [[‘`hiding`’] ⟨*moduleImportExports*⟩]

⟨*moduleImportExport*⟩ ::= ⟨*identifier*⟩ [‘`(..)`’]

⟨*moduleImportExportList*⟩ ::= ⟨*moduleImportExport*⟩ ‘`,`’ ⟨*moduleImportExportList*⟩

⟨*moduleImportExports*⟩ ::= ‘`(`’ ‘`)`’
  | ‘`(`’ ⟨*moduleExportList*⟩ ‘`)`’

$\langle$*module*$\rangle$ ::= ‘`module`’ $\langle$*moduleName*$\rangle$ [$\langle$*modulePart*$\rangle$] [$\langle$*moduleImportExports*$\rangle$
] ‘`where`’
  ($\langle$*import*$\rangle$ | $\langle$*pragma*$\rangle$)*
  ($\langle$*statement*$\rangle$ | $\langle$*pragma*$\rangle$)*
‘`end-module`’