The AWE Extension Package

Release 0.9.1 for Isabelle2009-1

Maksym Bortin
DFKI Bremen

Einar Broch Johnsen
University of Oslo

Christoph Lüth
DFKI Bremen

March 18, 2010

Contents

Chapter 1

Isabelle Foundations

In this short chapter we want briefly to introduce the basic concepts of Isabelle, which will play an important role in the entire manual. For a detailed description of the Isabelle/Isar framework we refer to, e.g., [1] and [2].

Isabelle is a logical framework in form of a generic LCF-style theorem prover. Isabelle has a fixed meta-logic which is a weak intuitionistic higher-order logic extended with polymorphism and type classes. Object logics like HOL, FOL, ZF etc. extend the meta-logic and are specified using *theories*. Further, Isar is an extension of the framework for interactive theory development with a powerful high-level human-readable proof language, theory presentation, etc.

Theories. Theories in Isabelle are structured hierarchically, i.e., there is the basic theory `Pure` for the meta-logic and any further theory has to *extend* a (finite) set of existing theories (also called *parent* theories). This is expressed in Isabelle/Isar by the keyword **imports**. In other words, theories in Isabelle form a directed acyclic graph. For a given theory $\mathcal{T}$ the finite set of its *ancestor* theories contains $\mathcal{T}$ together with all theories from which a path to $\mathcal{T}$ exists. *Proper* ancestors of $\mathcal{T}$ are then just the ancestors of $\mathcal{T}$ excluding $\mathcal{T}$.

Class signature. The *class signature* $\Sigma_{class}(\mathcal{T})$ of a theory $\mathcal{T}$ consists of the finite set of *classes* declared in the ancestors of the theory, and a *class relation* $\prec_{class}$. Class relations are constrained to be acyclic, i.e., there is no class c such that $c \prec^{+}_{class} c$, where $\prec^{+}_{class}$ denotes the transitive closure. In following we want also consider the reflexive-transitive closure of $\prec_{class}$ denoted by $\prec^{*}_{class}$. So, if $c \prec^{*}_{class} c'$ we say that c is a *subclass* of c' or equivalently c' is a *super class* of c. In case when $c \prec^{+}_{class} c'$ we will also say that c' is a *proper* super class of c.

Sorts. A *sort* of a theory $\mathcal{T}$ is a subset of the set of classes in the class signature of $\mathcal{T}$. For example, Isabelle/HOL introduces the class `HOL.type` and further declares the sort $\{$`HOL.type`$\}$ as the default sort. Let $Sorts(\mathcal{T})$ denote the set of sorts of $\mathcal{T}$. Any class relation induces the subsort relation $\preceq_{sort} \subseteq Sorts(\mathcal{T}) \times Sorts(\mathcal{T})$ as follows: $S \preceq_{sort} S'$ holds iff for any class $c' \in S'$ there exists a class $c \in S$ with $c \prec^{*}_{class} c'$. In case $S \preceq_{sort} S'$ we also say that S is a subsort of S' or equivalently S' is a super sort of S. By definition we have that the empty sort $\emptyset$ is a super sort of any sort.

Type signature. The *type signature* $\Sigma_{type}(\mathcal{T})$ of a theory $\mathcal{T}$ consists of the finite set of *type constructors* (also called *logical types*) declared in the ancestors

of the theory. Any type constructor has a fixed rank, i.e., the number of its arguments. Further, any type constructor has a finite set of arities. An *arity* for a type constructor C with the rank n is of the form $(S_1, \ldots, S_n, S_{n+1})$ where $S_i \in \mathit{Sorts}(\mathcal{T})$.

For example, the declaration "**typedecl** `'a T`" in a HOL theory introduces the type constructor `T` with the rank 1, and adds the standard HOL-arity for `T`: ({`HOL.type`},{`HOL.type`}).

Further arities can be added to a type constructor, e.g., using the Isar command **arities**.

Types. Let $\mathcal{X}$ be a fixed infinite countable set of *type variables*, and $\vartheta : \mathcal{X} \to \mathit{Sorts}(\mathcal{T})$ a *sort assignment*. The set of *types over* ϑ, denoted by $\mathit{Types}_\vartheta(\mathcal{T})$, is the smallest set containing $\mathcal{X}$ and closed under the application of type constructors of $\mathcal{T}$ w.r.t. rank.

The set $\mathit{Types}(\mathcal{T})$ of *types* is then the union of $\mathit{Types}_\vartheta(\mathcal{T})$ for all possible sort assignments ϑ.

For example, in Isabelle/HOL the declaration `'a ⇒ 'b` is actually a shortcut for `('a ::{type}) ⇒ ('b ::{type})`, since `{HOL.type}` is the default sort in HOL. That is, this type is built over ϑ which assigns to all type variables the default sort. In contrast, specifying, e.g., `('a ::{plus}) ⇒ 'b` we obtain a type built over a ϑ, which maps `'a` to the sort `{HOL.plus}` and all other type variables again to `{HOL.type}`.

Sort assignments together with type constructor arities provide the relation $\mathit{of\text{-}sort}_\mathcal{T} \subseteq \mathit{Types}(\mathcal{T}) \times \mathit{Sorts}(\mathcal{T})$. Firstly, α $\mathit{of\text{-}sort}_\mathcal{T}$ S holds for any type variable $\alpha \in \mathit{Types}_\vartheta(\mathcal{T})$ and any super sort S of $\vartheta(\alpha)$. Secondly, for an application $(\omega_1, \ldots, \omega_n)$ C, we can conclude $(\omega_1, \ldots, \omega_n)$ C $\mathit{of\text{-}sort}_\mathcal{T}$ S if C has an arity $(S_1, \ldots, S_n, S_{n+1})$ where $S_{n+1} \preceq_{sort} S$ and ω_1 $\mathit{of\text{-}sort}_\mathcal{T}$ $S_1, \ldots, \omega_n$ $\mathit{of\text{-}sort}_\mathcal{T}$ S_n hold.

Operation signature. The *operation signature* $\Sigma_{op}(\mathcal{T})$ of a theory $\mathcal{T}$ consists of the finite set of *constants* declared in the ancestors of this theory. Any constant $f \in \Sigma_{op}(\mathcal{T})$ has a fixed type $\mathit{type\text{-}of}_\mathcal{T}(f) \in \mathit{Types}(\mathcal{T})$ taken from the set of types of the theory.

Terms. Let $\mathcal{T}$ be a theory and $S \subseteq \Sigma_{op}(\mathcal{T})$. The set $\mathit{Terms}(\mathcal{T}|_S)$ of *S-restricted terms* of $\mathcal{T}$ is built of the constants in $\Sigma_{op}(\mathcal{T}) \setminus S$ over a fixed infinite set of meta-variables. For $\mathit{Terms}(\mathcal{T}|_\emptyset)$ we write $\mathit{Terms}(\mathcal{T})$ and call it the set of *terms* of $\mathcal{T}$.

All (restricted) terms in Isabelle are well-typed, i.e., for any $t \in \mathit{Terms}(\mathcal{T}|_S)$ there is a type $\omega \in \mathit{Types}(\mathcal{T})$. This relation is denoted by t *::* ω. Further, f *::* $\mathit{type\text{-}of}_\mathcal{T}(f)$ holds for any constant $f \in \Sigma_{op}(\mathcal{T})$.

Signature. The *signature* of an Isabelle theory consists of its class, type, and operation signatures.

Propositions. *Propositions* are terms of the meta-logical type `prop`. These can be either *axioms*, i.e., propositions without a proof, or *theorems*, i.e., propo-

sitions having a proof built with other propositions. Any theory has a finite set of axioms.

Definitions. Let $\mathcal{T}$ be a theory, f a constant from its operation signature and t a term over the signature of $\mathcal{T}$. A *definition* of f is an axiom which identifies f with t using the meta-equality of the form $f\ x_1 \ldots x_n \texttt{ == } t[x_1, \ldots, x_n]$, such that the types of terms on *lhs* and *rhs* are equal, $x_1 \ldots x_n$ are distinct, all type variables which occur on *rhs* also occur on *lhs*, and the constant f does not occur on *rhs*. Since Isabelle's framework ensures that a constant cannot be defined more than once (polymorphic constants can be defined more than once but for different type instances), definitions are always *conservative extensions*, i.e., do not affect the consistency of a theory. This also allows morphisms to treat definitions in a special way.

Logical and derived constants. We will also distinguish between logical and derived constants in a signature. A constant f in the signature of a theory is *derived* if in some ancestor of the theory there exists a definition of f. A constant is *logical* if it is not derived, i.e., without a definition in some ancestor.

Chapter 2

Morphisms

2.1 Introduction

This chapter describes the syntax and the functionality of the AWE Extension Package commands dealing with morphisms. In order to describe the functionality we will also cover the basic foundations of theory and signature morphisms. We will try to explain most situations by example, so that this document can be seen as both a reference manual and a tutorial. A brief and more formal introduction to theory and signature morphisms including advanced examples can be found in [8].

Furthermore, we use a modelling of computational monads in Isabelle/HOL as a running example.

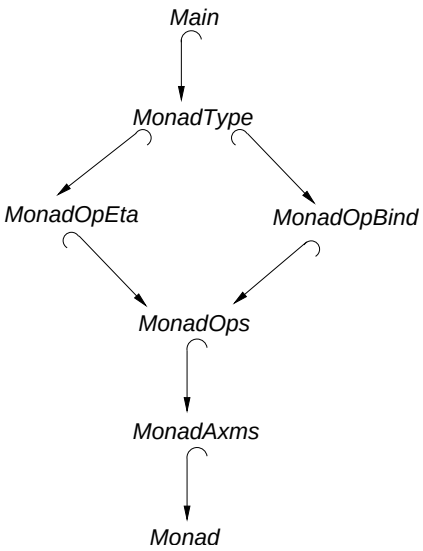


Figure 2.1: Structure of the theories modelling monads.

The monad example is organised hierarchically as shown in Figure 2.1 (where $T_1 \hookrightarrow T_2$ means that the theory T_2 imports T_1), and comprises six theories (*Main* refers to the Isabelle/HOL main theory), starting with *MonadType* which contains only the type declaration

typedecl `'a M`

The theories *MonadOpEta* and *MonadOpBind* contain the declaration of the two monad operations separately, and *MonadOps* is their union:

consts *eta :: " 'a ⇒ 'a M "*
consts *bind :: " 'a M ⇒ ('a ⇒ 'b M) ⇒ 'b M "* **(infixl** *"»="* **5)**

Finally, *MonadAxms* introduces the properties of the monad operations by the following four axioms:

axioms
mon_lunit: "(eta x »= t) = t x"
mon_runit: "(t »= eta) = t"
mon_assoc: "(s »= t »= u) = (s »= (λx. t x »= u))"
mon_eta_inj: "eta x = eta y ⟹ x = y"

The theory *Monad* contains derived constants and propositions. Furthermore, it provides the powerful concrete syntax for monads: for example for *m »= t* we can also write *{ x<- m; t x }* .

Now we start an example theory, and continue to build it up in the following:

theory *Example*
imports *"$AWE_HOME/Extensions/AWE_HOL"*
begin

Notice, that importing the theory *AWE_HOL*, we also import the main HOL-theory *Main*.

2.2 Constructing a Signature Morphism

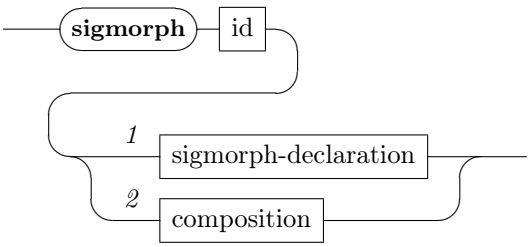
A *signature morphism* relates the signatures of two Isabelle theories, the *source* and the *target*. Any signature morphism will give us a mapping from terms over its source into terms over its target signature. This will be described later in Section 2.3.

Given a signature morphism, the common ancestors of source and target will be called *global*, while the ancestors of the source (target) which are not the ancestors of the target (source) will be called *domain* (*codomain*) of the signature morphism. Classes, type constructors, and constants from global theories we will also call global.

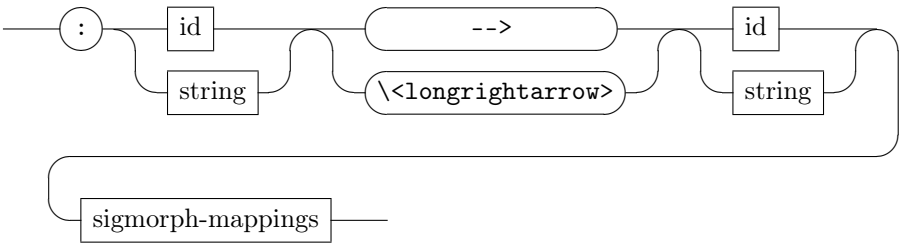
2.2.1 Syntax and functionalty

The command **sigmorph** constructs a new signature morphism, and has the syntax given by the following diagrams:

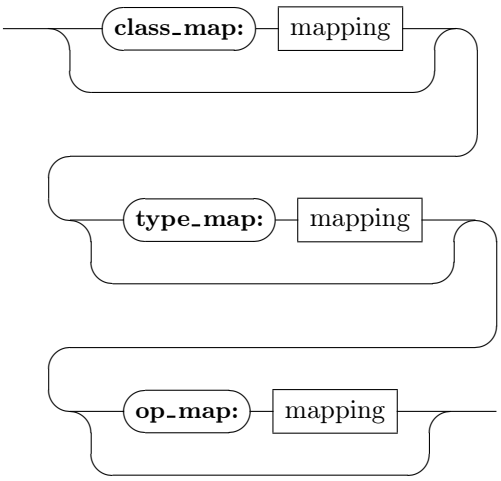
sigmorph



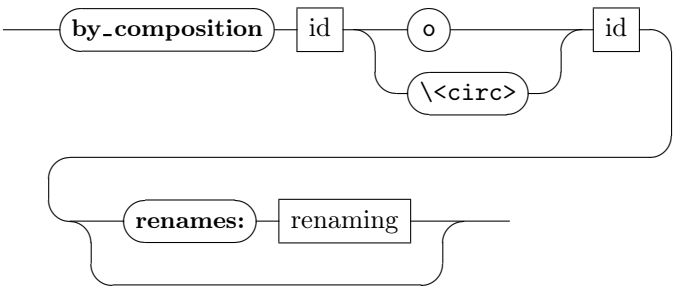
sigmorph-declaration



sigmorph-mappings



composition



The first identifier in the rule *sigmorph* denotes the name σ of the signature morphism to be constructed. Further, there are two possibilities for the construction:

1. By specification of the source theory $\mathcal{T}_1$, target theory $\mathcal{T}_2$, and the particular maps. The syntax is given by the rule *sigmorph-declaration*. The both identifiers (or strings) in the rule are interpreted as the source and target theories, respectively. If the source or target theory has not been loaded by Isabelle yet, the AWE Extensions call `use_thy` with the given

identifier or quoted string to load it. Thus, strings can be used to refer to theories by paths in the same way as with `use_thy` or `import` commands.

The first mapping in the *sigmorph-mappings* rule declares a *class map*, the second a *type map*, and the third an *operation map*. The order is important: we will see that any type map has to respect the class map declared before as well as any operation map has to respect the class and type maps declared before. The rule *mapping* is given in Appendix A: a mapping consists of a list of assignments $(s \mapsto t)$. The particular interpretations of s and t in class, type and operation maps are described in the following sections.

2. By composition of two existing signature morphisms with the syntax given by the rule *composition*. Here, the first identifier refers to a signature morphism $\sigma_2 : \mathcal{C} \longrightarrow \mathcal{D}$, while the second identifier to $\sigma_1 : \mathcal{A} \longrightarrow \mathcal{B}$. The semantics of the composition of signature morphisms is described in Section 2.4. The optional renaming allows to give explicit new names to derived constants from the domain, which will be translated to the target by the normalisation of σ_2 (see Section 2.3). Example of renaming can be found in Section 2.8.2.

Class map

For a class map declaration, both s and t are strings denoting Isabelle classes, and the class assignment $(s \mapsto t)$ is *correct* if

1. s is a type class from the domain and t from some ancestor of the target theory of the intended morphism.
2. s is *not* parametrised over some constants or axioms.

A class map is a list of class assignments and is *correct* if it contains only correct assignments and assigns to any class in the domain exactly one class in the target signature w.r.t. the subclass relation, i.e.,

- if t is assigned to s, s is a subclass of s', and t' is assigned to s' then t is a subclass of t'.

Any correct class map determines the mapping $\sigma_{class} : \Sigma_{class}(\mathcal{T}_1) \to \Sigma_{class}(\mathcal{T}_2)$ where any class from the domain is mapped according to the declaration, while global classes are mapped by identity. By construction, σ_{class} preserves the source and target subclass relations.
Further, σ_{class} induces the mapping $\sigma_{sort} : \mathit{Sorts}(\mathcal{T}_1) \to \mathit{Sorts}(\mathcal{T}_2)$ (called *sort* map), which maps sorts of the source theory to sorts of the target theory replacing source classes according to σ_{class}. In turn, σ_{sort} preserves the source and target subsort relations.

Type map

For a type map, both s and t are strings denoting Isabelle types, and the type assignment $(s \mapsto t)$ is *correct* if

1. s and t are Isabelle types over the source and over the target type signatures of the intended morphism, respectively.

2. s is a *type pattern*, i.e., it consists only of a *single* type constructor applied to a list of *distinct* type variables with the length equal to the rank of the type constructor. That is, an assignment is of the form $(\alpha_1, \ldots, \alpha_n)\ C\ \mapsto\ X$, where C is a type constructor with the rank n.

3. Any type variable which occurs in t has to occur in s. For an assignment $(\alpha_1, \ldots, \alpha_n)\ C\ \mapsto\ X$ this means, that only the type variables $\alpha_1, \ldots, \alpha_n$ may occur in X.

Further, any assignment $(\alpha_1, \ldots, \alpha_n)\ C\ \mapsto\ X[\alpha_1, \ldots, \alpha_n]$ has to respect the arities of C:

4. Let σ_{sort} be the sort map induced by the class map defined before. Let $(S_1, \ldots, S_n, S_{n+1})$ be an arity of C. Then $X[\omega_1, \ldots, \omega_n]$ *of-sort*$_{\mathcal{T}_2}$ $\sigma_{sort}(S_{n+1})$ has to hold for all types $\omega_1, \ldots, \omega_n$ with ω_i *of-sort*$_{\mathcal{T}_2}$ $\sigma_{sort}(S_i)$,.

A type map is a list of type assignments and is *correct* if it contains only correct assignments and for any type constructor from the domain there exists exactly one type pattern to which the type map assigns a type from the target.

Any correct type map determines the mapping $\sigma_{type} : \Sigma_{type}(\mathcal{T}_1) \rightarrow Types(\mathcal{T}_2)$ where any type constructor from the domain is mapped according to the declaration, while global type constructors are mapped by identity. Further, σ_{type} induces the mapping $\overline{\sigma}_{type} : Types(\mathcal{T}_1) \rightarrow Types(\mathcal{T}_2)$ (called the *extension* of type map), which maps types of the source theory to types of the target theory replacing types according to σ_{type}.

Let $\sigma_{sort} : Sorts(\mathcal{T}_1) \rightarrow Sorts(\mathcal{T}_2)$ be the sort map from the previous step. Then $\overline{\sigma}_{type}$ preserves the source and target *of-sort* relations, i.e.,

$$\frac{\omega\ \textit{of-sort}_{\mathcal{T}_1}\ s}{\overline{\sigma}_{type}(\omega)\ \textit{of-sort}_{\mathcal{T}_2}\ \sigma_{sort}(s)}$$

holds.

Operation map

For an operation map, s and t are strings and will be interpreted as constants in source and target signatures of the intended morphism, respectively. An operation map is a list of operation assignments $(s\ \mapsto\ t)$, and is *correct* if it assigns to any *logical* constant from the domain of the intended morphism exactly one constant in the signature of its target w.r.t. the given type and class maps: $\overline{\sigma}_{type}(\textit{type-of}_{\mathcal{T}_1}(s))$ has to be equal to $\textit{type-of}_{\mathcal{T}_2}(t)$ up to a consistent renaming of type variables. This property is called the *operation map* condition.

Let us explain this by a construction of an example signature morphism. Assuming we have the theory

```
theory Thy
imports "$AWE_HOME/Extensions/AWE" Nat
begin
typedecl 'a T
```

classes *cls*
arities *T :: (cls) cls*
consts *f :: " ('b ::{cls}) ⇒ 'b T ⇒ 'b T "*

then we can start to construct a signature morphism *to_list* from *Thy* to the HOL-theory *List*. Notice first, that we import the AWE Extensions here by the theory *AWE*, not *AWE_HOL*. The reason is that if we would take *AWE_HOL*, we would import all main HOL-theories including *List* itself, such that the target of *to_list* would be a proper ancestor of its source. In other words, we would then try to construct a morphism with a non-empty domain and empty codomain, which is possible only for some artificial constructions. In contrast, importing the theory *AWE* instead, we only import the most basic Isabelle theory *Pure* together with the AWE Extensions. Additionaly, we also explicitly import *Nat*, but we could also take any other HOL-theory which is a proper ancestor of *List*.

So, since *Thy* has only two parent theories, namely *Nat* and *AWE*, the domain of *to_list* consists of the theories *Thy* and *AWE*. Notice, that both *AWE* and *AWE_HOL* are empty theories and are only used to load the AWE Extensions, such that they do not play any role for class, type and operation maps.

First, we define class and type maps for *to_list* as follows:

sigmorph *to_list : Thy ⟶ List*
class_map: *[("Thy.cls" ↦ "HOL.type")]*
type_map: *[("'a T" ↦ "'a list")]*

Let us check that the maps above are correct. The only class in the domain of *to_list* is *cls*, which does not have any subclass or superclass. Thus, the class map above is correct, since *HOL.type* is a class in the signature of *List*. If *cls* would have, e.g., a super class *cls'*, then *cls'* could be mapped by the class map only to a super class of *HOL.type*, which can be only *HOL.type* itself, since it has no proper super classes by construction.

So, the derived sort map σ_{sort} replaces *Thy.cls* by *HOL.type* in any sort over the class signature of *Thy*. Next, there are two arities for the type constructor *T*: the HOL-default (*{HOL.type}*, *{HOL.type}*), and the explicitly declared (*{Thy.cls}*, *{Thy.cls}*). Let ω be an arbitrary type of sort σ_{sort}(*{Thy.cls}*), i.e., *{HOL.type}*. Then, by the default arity for the type constructor *list* we can derive (ω *list*) *of-sort*$_{\sigma\ sort}$(*{Thy.cls}*). Thus, the type map respects the arities of the type constructor *T*, and, since there are no other type constructors in *Thy*, is correct.

Now we can consider the actual question: the operation map. This is nessesary, since we still have to map the logical constant *Thy.f* to some constant in the target signature correctly w.r.t. to the previous maps. To be precise, we are looking for a constant in the signature of the theory *List* having the type *('b ::{HOL.type}) ⇒ 'b list ⇒ 'b list*, since this type we would obtain replacing *cls* by *HOL.type* and *T* by *list* in the type of *Thy.f*. The list constructor *List.list.Cons* is such constant, since its declared type is *('a ::{HOL.type}) ⇒ 'a list ⇒ 'a list* – the same type up to the bijective renaming *'a* ↦ *'b* of the occuring type variables.

So, we can complete the declaration of a signature morphism by

sigmorph *to_list : Thy ⟶ List*
class_map: *[("Thy.cls" ↦ "HOL.type")]*
type_map: *[("'a T" ↦ "'a list")]*

op_map: *[("Thy.f" ↦ "List.list.Cons")]*

Altogether, what we have done here manually, the AWE Extensions will be able to derive for us. On the input

sigmorph *to_list : Thy ⟶ List*
class_map: *[("Thy.cls" ↦ "HOL.type")]*
type_map: *[("'a T" ↦ "'a list")]*

it will try to find an assignment for *Thy.f* automatically, searching among all constants in the target signature for those, satisfying the operation map condition. Generally, there are following cases:

1. For some logical constants in the domain, more than one correct assignment exist. In this case the signature morphism construction would fail with a message displaying all possibilities for all such constants. Then the user has to choose one assignment for any of them and put it into the operation map. In other words, the operation map has to contain enough information to disambiguate the assignments derived automatically.
2. For all logical constants in the domain, exactly one correct assignment exists. In this case the signature morphism will be constructed.
3. For some logical constants in the domain no appropriate assignments exist The signature morphism construction will fail with the message showing this constants. To repair this one can modify their types, extend the codomain, or remove them from the domain (if possible).

In particular, on our input we will obtain the first case as the search result – there is at least the remove operation on lists having the same type as *List.list.Cons*. This means that we have to choose one assignment, e.g., *("Thy.f" ↦ "List.list.Cons")*, and put it back into the operation map. This constructs the signature morphism *to_list* as above.

Example

In the monad example we can construct a signature morphism with the source theory *MonadOpEta* and the target theory *Example* as follows:

sigmorph *s : MonadOpEta ⟶ Example*
type_map: *[("'a MonadType.M" ↦ "'a Datatype.option")]*

Since *MonadOpEta.eta* is of type *"'a ⇒ 'a M"*, we are looking in the signature of *Example* for an operation having the type *"'a ⇒ 'a option"*. In our case *Datatype.option.Some* (which is actually a constructor of the datatype *option*) is even the unique operation with this type. Thus, we do not need to disambiguate — the assignment will be derived automatically:

Found: ("MonadOpEta.eta" ↦ "Datatype.option.Some")
Signature morphism s : MonadOpEta ⟶ Example constructed.

2.3 Homomorphic Extension and Normalisation of Signature Morphisms

Let $\sigma : \mathcal{T}_1 \longrightarrow \mathcal{T}_2$ be a signature morphism, and $\Sigma_{der} \subseteq \Sigma_{op}(\mathcal{T}_1)$ the set of *derived* constants from the *domain* of σ. Since the operation map of σ

does not contain assignments for $f \in \Sigma_{der}$, it determines the mapping $\sigma_{op} : \Sigma_{op}(\mathcal{T}_1)\backslash\Sigma_{der} \to \Sigma_{op}(\mathcal{T}_2)$ where any logical constant from the domain is mapped according to the declaration, while global constants are mapped by identity. Then, the operation map condition holds for σ_{op}:

1. on the logical constants in the domain by construction, and
2. on the global constants, since $\overline{\sigma}_{type}$ is an identity mapping for global types.

Thus, σ gives a mapping $\overline{\sigma} : \mathit{Terms}(\mathcal{T}_1|_{\Sigma_{der}}) \to \mathit{Terms}(\mathcal{T}_2)$ where sorts, types and constants are replaced according to the σ_{sort}, σ_{type} and σ_{op} mappings. The correctness conditions on these maps assure that all type classes, type constructors and constants of the source signature are indeed substituted, and that the resulting translated term is well-typed in the target theory, i.e.

$$\frac{t :: \omega}{\overline{\sigma}(t) :: \overline{\sigma}_{type}(\omega)}$$

holds for any $t \in \mathit{Terms}(\mathcal{T}_1|_{\Sigma_{der}})$ and $\omega \in \mathit{Types}(\mathcal{T}_1)$. $\overline{\sigma}$ is called the *homomorphic extension* of the signature morphism σ.

By construction, we have that any global element w.r.t. a signature morphism σ is mapped by the particular map to itself. Thus, $\overline{\sigma}(t) = t$ for any $t \in \mathit{Terms}(\mathcal{T})$, where $\mathcal{T}$ is a global theory.

Furthermore, homomorphic extensions provide the following relation on terms: let σ be a signature morphism, s and t terms over its source and target signatures, respectively; we say that t is a *σ-instance* of s if t is equal to $\overline{\sigma}(s)$ up to a consistent renaming of variables (in other words: t and $\overline{\sigma}(s)$ are α-convertible terms).

It holds, that $\overline{\sigma}(t)$ is a σ-instance of t, for any term t over the source signature. In particular, for any $t \in \mathit{Terms}(\mathcal{T})$, where $\mathcal{T}$ is a global theory, we can conclude that t is a σ-instance of itself.

Provided by $\overline{\sigma}$, the signature morphism σ can be canonicaly extended to the signature morphism $\sigma\downarrow : \mathcal{T}_1 \longrightarrow \mathcal{T}_2'$ with the operation map $\sigma\downarrow_{op} : \Sigma_{op}(\mathcal{T}_1) \to \Sigma_{op}(\mathcal{T}_2')$ where $\mathcal{T}_2'$ is a theory which extends $\mathcal{T}_2$ by the translated derived constants from Σ_{der}, preserving consistency of $\mathcal{T}_2$. The signature morphism $\sigma\downarrow$ is called the *normal form* of σ and the extension procedure is called *normalisation*.

2.4 Composition of Signature Morphisms

Let $\sigma_1 : \mathcal{A} \longrightarrow \mathcal{B}$ and $\sigma_2 : \mathcal{C} \longrightarrow \mathcal{D}$ be signature morphisms. If $\mathcal{B}$ is an ancestor of $\mathcal{C}$, then the signature morphism $(\sigma_2 \circ \sigma_1) : \mathcal{A} \longrightarrow \mathcal{D}'$ is well-defined by the componentwise composition of the particular maps of σ_1 and $\sigma_2\downarrow$. The normalisation of σ_2 is required, since the operation map of σ_1 can assign to logical constants from $\Sigma_{op}(\mathcal{A})$ derived constants from $\Sigma_{op}(\mathcal{B})$. Therefore the theory $\mathcal{D}$ will be (possibly) extended to $\mathcal{D}'$.

2.5 Constructing a Theory Morphism

2.5.1 Foundations

Theory morphisms extend the notion of a signature morphism to axioms and theorems. As mentioned in the previous section, the homomorphic extension of a signature morphism maps terms from its source to terms in its target. Since axioms and theorems are terms of the type `prop`, we can also consider their images under a homomorphic extension, which are propositions as well.

Next, we sketch how signature morphisms with an additional map provide theorem translation. This will motivate introduction of theory morphisms. As mentioned in Chapter 1, theorems are derived propositions. Such derivations (or just proofs) are formalised in Isabelle by *proof terms* (see, e.g., [5]). Roughly, proof terms are trees having axioms as leafs.

Let σ be a signature morphism and η_σ a map assigning to any axiom in the domain some σ-instance of it. We will call such η_σ an *axiom map*, and define it for global axioms as identity map, which is well-defined according to Section 2.3.

Further, let P be a derived proposition in the domain, and π the proof term of P. We obtain the proof term π' in the target replacing all axioms occuring in π according to η_σ. The correctness of the homomorphic extension $\overline{\sigma}$ assures that Isabelle's meta-logical kernel can derive from the proof π' a proposition P', which is in turn a σ-instance of P. Notice the constructivity of the approach: translated theorems are derived in the target, i.e., any theorem translated this way is indeed provable in the target theory (as P' by π' above)

So, this approach justifies our definition of theory morphisms: a *theory morphism* τ is a pair $\langle \sigma, \eta_\sigma \rangle$, i.e., a signature morphism σ equipped with an axiom map.

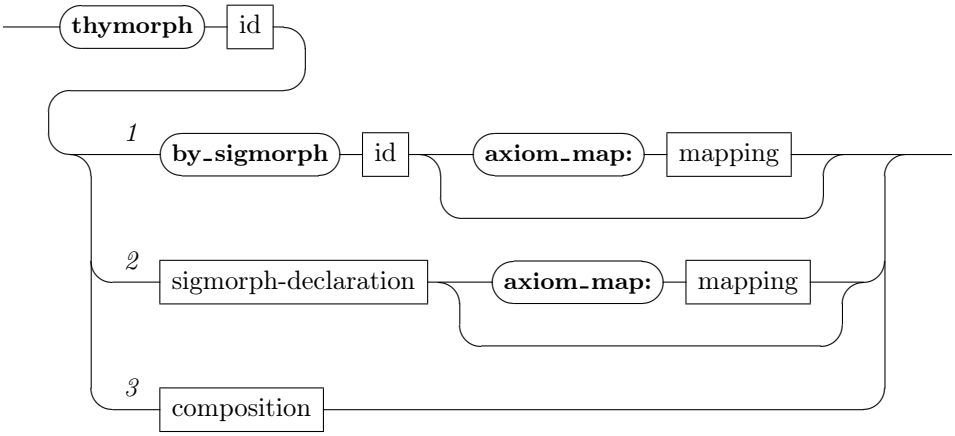
In the sequel, domain, codomain, and global theories of a theory morphism $\tau = \langle \sigma, \eta_\sigma \rangle$ are the same as for its underlying signature morphism σ, and τ-instance means σ-instance.

The normalisation of a theory morphism $\tau = \langle \sigma, \eta_\sigma \rangle$ is defined by: $\tau\downarrow \stackrel{def}{=} \langle \sigma\downarrow, \eta_\sigma \rangle$.

2.5.2 Syntax and functionality

The syntax of the **thymorph** command is given by the following diagram, which refers to the rule *sigmorph-declaration* defined in Section 2.2.1.

thymorph



The first identifier is the name t of the theory morphism to be constructed. As shown in the diagram, there are three ways to construct a theory morphism:

1. By giving an identifier referring to an existing signature morphism and an optional axiom map.

2. By giving source, target, class, type, operation, and an additional optional axiom map. This possibility is actually a composed one. Internally it consists of two steps: constructing an 'intermediate' signature morphism from the given arguments and then doing the same as in *1.* Notice that this intermediate signature morphism will be anonymous, i.e., we cannot refer to it by an identifier.

3. By composition of two existing theory morphisms with the syntax given by the rule *composition* in Section 2.2.1. Here, the first identifier refers to a theory morphism $\tau_2 : \mathcal{C} \longrightarrow \mathcal{D}$, while the second identifier to $\tau_1 : \mathcal{A} \longrightarrow \mathcal{B}$. The semantics of the theory morphism composition is described in Section 2.7. The optional renaming allows to give explicit new names to derived constants from the domain, which will be translated to the target by the normalisation of τ_2 (see Section 2.3) as well as to theorems from the domain which will be translated by composition of axiom maps. Example of renaming can be found in Section 2.8.2.

Axiom map

An axiom map has the same syntax as an operation map: it consists of a list of assignments ($s \mapsto t$), where s and t are strings. According to 2.5.1, for an axiom map such an assignment is correct if the by t denoted proposition is a σ-instance of the proposition denoted by s, where σ is the underlying signature morphism.

An axiom map is *correct* if it contains only correct assignments and maps *all axioms* from the domain of the intended theory morphism.

Just as in the case of the operation map, the AWE Extension Package will automatically try to derive a suitable assignment for any axiom from the domain

searching among all propositions in the codomain. Since ambiguity does not matter in this case, the search may result in:

1. For all axioms from the domain an assignment was found. In this case, a theory morphism will be constructed.

2. For some axioms in the domain no suitable proposition in the codomain could be found. In this case an explicit axiom map may help. Since only propositions from the codomain are searched, suitable global propositions may exist but will not be found automatically. These have to be either

   - assigned explicitly by the axiom map, or
   - made explicit in the codomain using, for instance, the Isabelle command **lemmas**,

   otherwise the theory morphism construction will fail.

Example

We can extend the signature morphism `s` from Section 2.2 to a theory morphism `t` by a declaration as follows:

thymorph `t` **by_sigmorph** `s`

In this case there are no axioms at all in the domain of `t`.

Without referring to the signature morphism `s`, we can also write

thymorph `t : MonadOpEta ⟶ Example`
type_map: `[("'c MonadType.M" ↦ "'c Datatype.option")]`

with the response:

```
Found:  ("MonadOpEta.eta" ↦ "Datatype.option.Some")
Theory morphism t :  MonadOpEta ⟶ Example constructed.
```

The situation is shown in Figure 2.2.

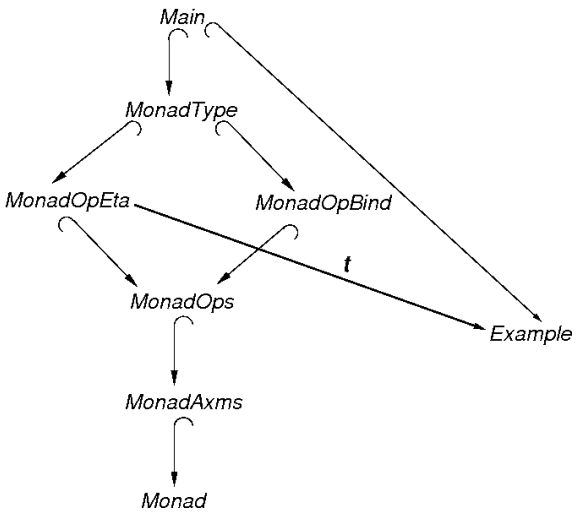


Figure 2.2: Theory morphism from `MonadOpEta` to `Example` constructed

2.6 translate_thm: Theorem Translation

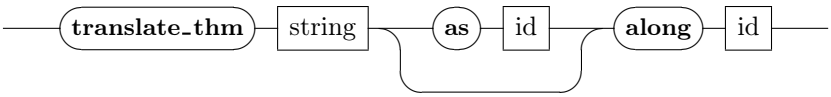
Section 2.5.1 motivated theory morphisms by a basic procedure for translation of derived propositions by proof transformation. The AWE command **translate_thm** implements this procedure, but takes also dependencies between theorems additionaly into account. A theorem P *depends* on a theorem Q if Q is used in the proof of P.

Let us consider the situation when some theorems P_1 and P_2 depend on some theorem Q. Ignoring this dependencies, when moving P_1 and then P_2 along a theory morphism, we would transform the proof of Q twice. The procedure implemented by **translate_thm** avoids this: it moves first all theorems on which P_1 depends (and Q in particular) along the given theory morphism τ, adding their resulting propositions to the target theory of τ. Then, moving P_2 along τ, it can just retrieve the required τ-instance of Q from the target.

Syntax and functionality

The **translate_thm** command has the following syntax:

translate_thm



`string` is interpreted as the name of the theorem to be translated and has to be fully qualified. Using the first `id` one can give an explicit name to the translated proposition. In the case this option is not used, the translated proposition will get the name of the original one as long as it is not already used in the target theory. Otherwise the AWE Extension Package will generate a name automatically. The second `id` has to refer to an existing theory morphism.

There is a possibility to skip the internal translation of proofs by

ML `"set AWE.skip_proofs"`

which will have the same effect as constructing theories in Isar in *quick_and_dirty*-mode. This, however, on the one hand will make **translate_thm** work much more faster, but on the other hand violates the constructivity of theorem translation. Furthermore, it may cause problems later, since all such translated theorems have no proofs, and hence lose their dependencies.

2.7 Composition of Theory Morphisms

Let $\tau_1 : \mathcal{A} \longrightarrow \mathcal{B}$ and $\tau_2 : \mathcal{C} \longrightarrow \mathcal{D}$ be theory morphisms. If $\mathcal{B}$ is an ancestor of $\mathcal{C}$, then the theory morphism $(\tau_2 \circ \tau_1) : \mathcal{A} \longrightarrow \mathcal{D}''$ is well-defined as follows:

1. The signature morphism of $(\tau_2 \circ \tau_1)$ is the composition of the underlying signature morphisms, which normalises τ_2 and (possibly) extends the theory $\mathcal{D}$ to the theory $\mathcal{D}'$.
2. The axiom map of $(\tau_2 \circ \tau_1)$ is the composition of the axiom maps of τ_1 and τ_2, which can require theorem translation by $\tau_2\downarrow$, since the axiom map of

τ_1 can assign derived propositions in $\mathcal{B}$ to axioms in its domain. This step (possibly) extends the theory $\mathcal{D}'$ to the theory $\mathcal{D}''$.

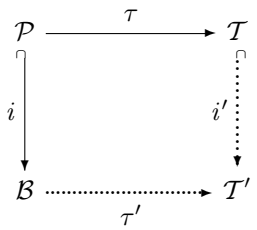
2.8 Parametrised Theories and Instantiation

2.8.1 Foundations

If a theory $\mathcal{T}$ is an ancestor of a theory $\mathcal{T}'$, i.e., $\mathcal{T}'$ extends $\mathcal{T}$, then there is a unique theory morphism $i : \mathcal{T} \hookrightarrow \mathcal{T}'$, the *inclusion*. Its class, type, operation and axiom maps are identities. Since inclusions are determined by the hierarchy of theories there is no need to declare them explicitly, unless one wants to give an explicit name to one of them in order to use it as an argument later.

A *parametrised theory* consists of a tuple of theories $\langle \mathcal{P}, \mathcal{B} \rangle$ such that $\mathcal{P}$ is an ancestor of $\mathcal{B}$, which gives the inclusion morphism $i : \mathcal{P} \hookrightarrow \mathcal{B}$. Theory $\mathcal{P}$ is called the *parameter* part and $\mathcal{B}$ the *body* part of $\langle \mathcal{P}, \mathcal{B} \rangle$.

The *instantiation* of a parametrised theory is sketched by the following diagram, where dotted arrows are results of instantiation:



(2.1)

The theory morphism τ maps all type classes, type constructors, constants and axioms from its domain to type classes, types, constants and propositions in the ancestors of the *instantiating theory* $\mathcal{T}$, respectively. Further, since $\mathcal{B}$ extends $\mathcal{P}$, we can stepwise construct a similar extension $\mathcal{T}'$ of $\mathcal{T}$, and at the same time extend class, type, operation and axiom maps of τ to obtain τ'. In order to do this we need to find a correct assignment for any type class, type constructor, logical constant, and axiom occuring in the domain of $\tau' : \mathcal{B} \longrightarrow \mathcal{T}'$. To keep things simple, we will describe the construction for the case when $\mathcal{B}$ imports only one theory, namely $\mathcal{P}$, such that the domain of τ' is the same as the domain of τ plus theory $\mathcal{B}$.

First, it is possible to give an assignment for classes, logical types, logical constants, and axioms of $\mathcal{B}$ explicitly during instantiation, which would actually correspond to considering the particular element as a part of the parameter theory $\mathcal{P}$, such that the particular map of t can be seen as extended by the given assignment. We will call such elements *explicitly instantiated*. The functionality of explicit instantiations is thus the same as for the particular maps for the construction of morphisms described in 2.2.1 and 2.5.2.

Then, the theory morphism τ' and the theory $\mathcal{T}'$ will be constructed as follows:

1. For any type class c introduced in $\mathcal{B}$, which is not already explicitly instantiated, we add the corresponding type class $\mathcal{T}.c$ to the class signature

of $\mathcal{T}$. Of course, if there exists a super class c' of c in $\mathcal{B}$ then it will be translated (if required) to $\mathcal{T}$ first, such that we can declare $\mathcal{T}.c$ to be a subclass of $\mathcal{T}.c'$ in $\mathcal{T}$ in order to keep our class map consistent.

Notice also, that if c is a type class, which is parametrised by operations or/and axioms (axiomatic type class), then it cannot be explicitly instantiated because of the class map restrictions described in Section 2.2.1. Nevertheless, for such a type class c the corresponding parametrised type class $\mathcal{T}.c$ will be introduced.

This step gives us the class map for τ' and the theory $\mathcal{T}_0$ extending $\mathcal{T}$. The class map extends to the sort map $\sigma_{sort} : \mathit{Sorts}(\mathcal{B}) \to \mathit{Sorts}(\mathcal{T}_0)$.

2. For any type constructor $\mathcal{B}.C$ introduced in $\mathcal{B}$ we add the type constructor $\mathcal{T}.C$ with the same rank and arities, mapped by the sort map from the previous step, to the type signature of $\mathcal{T}$, and extend the type map of τ by $(\alpha_1, \ldots, \alpha_{i_{\mathcal{B}.C}})\mathcal{B}.C \mapsto (\alpha_1, \ldots, \alpha_{i_{\mathcal{B}.C}})\mathcal{T}.C$, unless $\mathcal{B}.C$ is explicitly instantiated.

   This step gives us the type map for τ' and the theory $\mathcal{T}_1$ extending $\mathcal{T}_0$. The type map extends to $\overline{\sigma}_{type} : \mathit{Types}(\mathcal{B}) \to \mathit{Types}(\mathcal{T}_1)$.

3. For any logical constant $\mathcal{B}.f$ introduced in $\mathcal{B}$ we add a logical constant $\mathcal{T}.f$ with the type $\overline{\sigma}_{type}(\mathit{type\text{-}of}_{\mathcal{B}}(f))$, satisfying the operation map condition, to the operation signature of $\mathcal{T}$, and extend the operation map of τ by $\mathcal{B}.f \mapsto \mathcal{T}.f$, unless $\mathcal{B}.f$ is explicitly instantiated. This step gives us the operation map for τ' and the theory $\mathcal{T}_2$ extending $\mathcal{T}_1$. Altogether, we already obtain a signature morphism $\sigma : \mathcal{B} \longrightarrow \mathcal{T}_2$.

4. For any axiom $\mathcal{B}.A$ introduced in $\mathcal{B}$ we can

   - either *match* $\mathcal{B}.A$, i.e., can find a σ-instance A' of $\mathcal{B}.A$ searching only in the *codomain*, such that we can extend the axiom map of τ by $\mathcal{B}.A \mapsto A'$, or
   - *insert* (unless $\mathcal{B}.A$ is explicitly instantiated) an axiom $\mathcal{T}.A$ into $\mathcal{T}_2$, which is the image of $\mathcal{B}.A$ under $\overline{\sigma}$, such that we can correctly extend the axiom map of τ by $\mathcal{B}.A \mapsto \mathcal{T}.A$.

   This step gives us an extension of the axiom map of τ and the theory $\mathcal{T}'$ extending $\mathcal{T}_2$. This axiom map together with σ yield the resulting theory morphism τ'.

There are the following important caveats:

- $\mathcal{T}$ has to be the currently developed theory. So, $\mathcal{T} \hookrightarrow \mathcal{T}_0 \hookrightarrow \mathcal{T}_1 \hookrightarrow \mathcal{T}_2 \hookrightarrow \mathcal{T}'$ denote the development steps of the same theory.
- Since classes, type constructors, constants and axioms can be moved from $\mathcal{B}$ to $\mathcal{T}$ by instantiation, the problem can occur that an element with the same name already exists within the theory $\mathcal{T}$. For example, if a type constructor C is added to a theory where a type constructor C has been already declared then Isabelle throws an exception. The problem can be avoided using explicit renaming: add a renaming $\mathcal{B}.C \mapsto X$, where X is a name which does not occur in the theory $\mathcal{T}$. The same can be done for classes, constants and axioms.

Proper and axiomatic instantiations

Since the fourth step of instantiation may introduce new axioms, it is unsafe in the sense that the theory $\mathcal{T}$ can become inconsistent during such instantiation. This leads to the notion of proper instantiation. A *proper* (or non-axiomatic) instantiation would skip the insertion in the third step above and result in a theory morphism if all axioms in $\mathcal{B}$ have been matched by theorems or axioms in $\mathcal{T}$. Otherwise, it results only in the signature morphism σ, constructed in the third step, together with the set of σ-instances of unmatched axioms from $\mathcal{B}$, also called *proof obligations*. For a given signature morphism these can also be displayed with the command **thymorph_goals**, described in Section 2.8.3.

Proper instantiation is a conservative extension. In contrast, an *axiomatic* instantiation uses the insertion step for axioms, i.e., it may extend the instantiating theory non-conservatively by axioms.

The command **instantiate_theory** implements both proper and axiomatic instantiations.

2.8.2 instantiate_theory

Syntax and functionality

The syntax of the command **instantiate_theory** is shown in Figure 2.3. It uses the rule *sigmorph-mappings* introduced in Section 2.2.1. Referring again to Diagram 2.1, the first identifier (or string) corresponds to the theory $\mathcal{B}$ and the second to the theory morphism τ. Quoted strings are used here in the same way as for construction of morphisms (see 2.2.1), i.e., in order to refer to a theory by a path.

The options of **instantiate_theory** comprise:

1. Optional class, type and operation maps (see 2.2.1), which will be interpreted as explicit instantiations of classes, type constructors and logical constants declared in the domain of τ', respectively.

2. An optional axiom map (see 2.5.2), which will be interpreted as explicit instantiations for unmatched axioms in the domain of τ' in any axiomatic instantiation.

3. A renaming map (see the rule for *renaming* in Appendix A), which allows users to change the names of classes, type constructors, constants and axioms as well as the mixfix syntax ot type constructors and constants to be inserted into the theory $\mathcal{T}$.

4. If the keyword **axiomatic** is used then the instantiation will extend the theory $\mathcal{T}$ by all unmatched axioms from the domain of t, which are not explicitly instantiated, as described in the previous section.

5. Furthermore, **instantiate_theory** will extend $\mathcal{T}$ by concrete syntax[1] declared in the domain theories of τ', unless the keyword **without_syntax** is

[1] These comprise concrete theory syntax built with the following Isar commands: **nonterminals**, **syntax**, **translations**, **parse_translation**, **print_translation**, **parse_ast_translation**, **print_ast_translation**, **token_translation**.

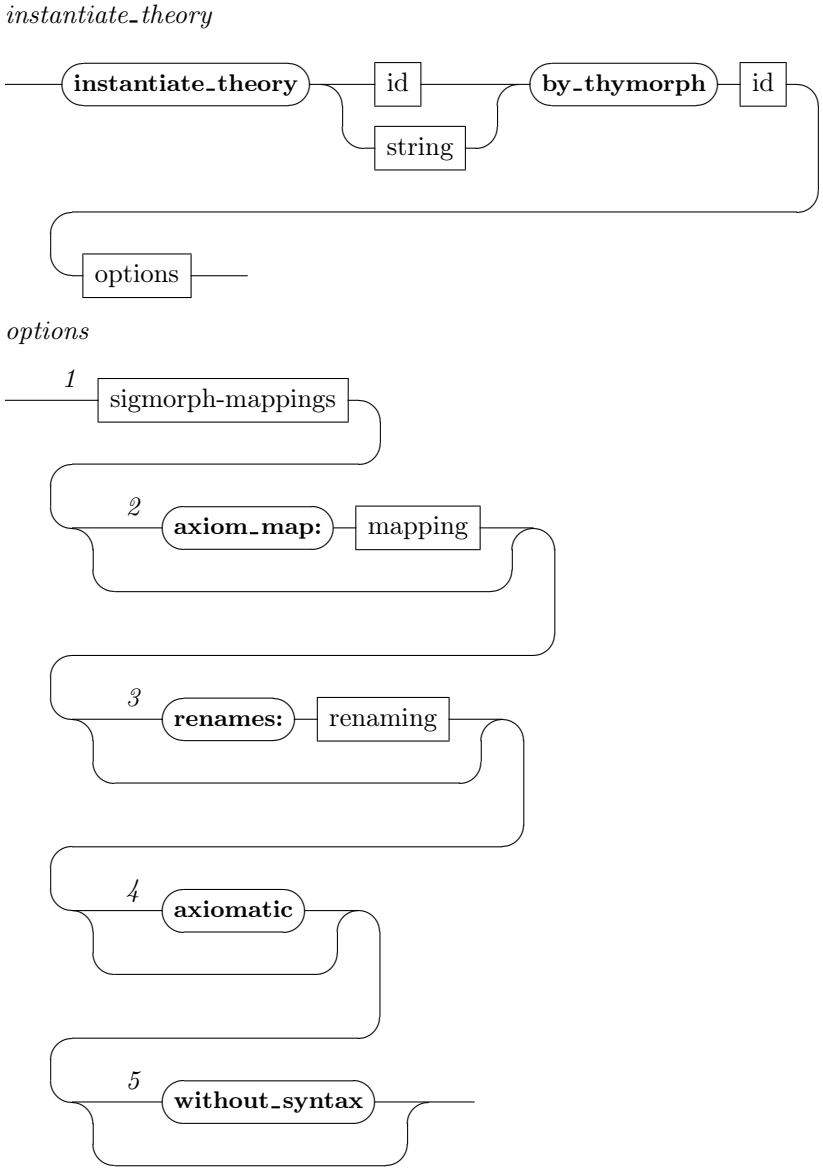


Figure 2.3: Syntax of **instantiate_theory**

used. Apart from that, abbreviations (command **abbreviation**) are also supported if they have the form as presented in [3].

Example

In our monad example we have the theory morphism `t : MonadOpEta ⟶ Example` and further the implicit inclusion `i : MonadOpEta ↪ MonadOps`. This gives us a parametrised theory with parameter `MonadOpEta` and body `MonadOps`, which we will now instantiate by `t`. The input

instantiate_theory `MonadOps` **by_thymorph** `t`

would yield the following response:

```
...  adding logical constant
Example.bind ::  "'a option ⇒ ('a ~⇒ 'b) ~⇒ 'b" (infixl "»=" 5)

Theory morphism t' :  MonadOps ⟶ Example constructed.
```

where `'a ~⇒ 'b` is the Isabelle/HOL's notation for a partial map, equivalent to `'a ⇒ 'b option`. Notice, that in this case our proper instantiation already results in a theory morphism, since there are no axioms at all in `MonadOps`.

We obtain the extended theory `Example` which contains the logical constant `bind` with the same infix annotation as `MonadOpBind.bind` (which is crucial for the concrete monad syntax) and the theory morphism `t'` the operation map of which contains the assignment `MonadOpBind.bind ↦ Example.bind`.

Alternatively, we could also make a similar instantiation involving the renaming together with a re-definition of the mixfix notation:

instantiate_theory `MonadOps` **by_thymorph** `t`
renames`:  [("MonadOpBind.bind" ↦ "option_bind"`
mixfix`:  ("_»=*\ _" [5, 6] 5) )]`

which yields the response:

```
...  adding logical constant
Example.option_bind ::  "'a option ⇒ ('a ~⇒ 'b) ~⇒ 'b"
                                    ("_»=*\ _" [5, 6] 5)
Theory morphism t' :  MonadOps ⟶ Example constructed.
```

Such an instantiation can be helpful when several monad instances occur in a theory, because we can use the concrete monad syntax only with one of them. Only this one will keep the `"»="`-annotation.

As another alternative of the instantiation of `MonadOps`, let us also demonstrate an explicit instantiation of the logical constant `MonadOpBind.bind`. First, we introduce in the theory `Example` the constant:

consts
`option_bind ::  "'a option ⇒ ('a ~⇒ 'b) ~⇒ 'b" ("_»=\ _" [5, 6] 5)`

Now, we can instantiate:

instantiate_theory `MonadOps` **by_thymorph** `t`
op_map: `[("MonadOpBind.bind" ↦ "Example.option_bind")]`

with the response:

```
...  logical constant MonadOpBind.bind explicitly instantiated
Theory morphism t' :  MonadOps ⟶ Example constructed.
```

Of course, for this small example this instantiation is not very useful: we do here something manually what the AWE Extention Package can do for us, as

the first instantiation above shows.
Altogether, we obtain the situation shown in Figure 2.4.

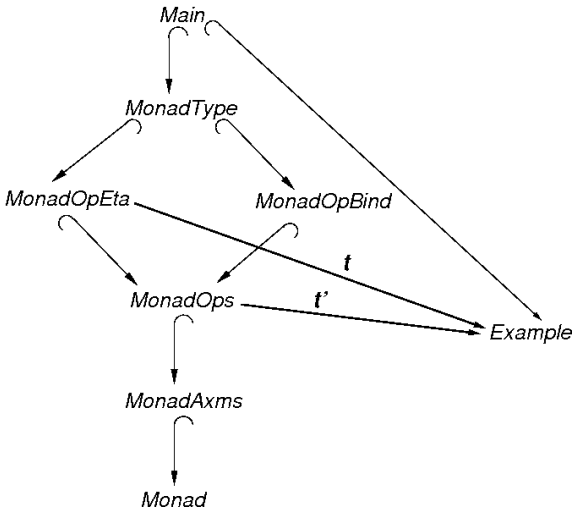


Figure 2.4: Instantiation of *MonadOps* by *t* results in theory morphism *t'*

Now let us consider the theory *MonadAxms*. The instantiation
instantiate_theory *MonadAxms* **by_thymorph** *t'* **axiomatic**
would insert corresponding *t'*-instance for each monad axiom as an axiom into the theory *Example* and construct the theory morphism *t'' : MonadAxms* ⟶ *Example*. This would axiomatically state that the datatype *option* is a monad. Here, we can actually prove the monadic properties for *option* together with a suitable definition of the *Example.bind* operation, obtained in the previous instantiation. So, we choose this possibility and define:

primrec
```
"(None >>= f) = None"
"((Some x) >>= f) = f x"
```

Now we can try a proper instantiation:
instantiate_theory *MonadAxms* **by_thymorph** *t'*
This yields the following response:

```
Instantiating theory MonadAxms by theory morphism
t' : MonadOps ⟶ Example ...

Axiom mapping found: ("MonadAxms.mon_lunit" ↦ "Example.bind.simps_2")
Lemma MonadAxms_mon_eta_inj : "Some x = Some y ⟹ x = y" proven.

Signature morphism t'' : MonadAxms ⟶ Example constructed.

To prove:
lemma mon_assoc: "(s >>= t >>= u) = (s >>= (λx. t x >>= u))"
lemma mon_runit: "(t >>= Some) = t"
```

The lemma *Example.bind.simps_2* was automatically derived for the recursive definition of *bind*, and is actually its second defining equation: *"((Some x) >>= f)*

`= f x"`. But this proposition is a `t'`-instance of the monadic left unit property `MonadAxms.mon_lunit`. This explains the first detected axiom mapping.

Further, the axiom `MonadAxms.mon_eta_inj` would produce the proof obligation `"Some x = Some y ⟹ x = y"`, since no such proposition has been explicitly proven in `Example` yet. But since it claims the injectivity for the constructor `Some`, which is a basic property of any constructor, the automatical reasoner of Isabelle was able to prove it, while the AWE Extension Package has assigned the generated name `MonadAxms_mon_eta_inj` to the resulting lemma. Altogether, this yields the second axiom mapping `("MonadAxms.mon_eta_inj" ↦ "Example.MonadAxms_mon_eta_inj")`.

Generally, any proof obligation will be delegated to Isabelle's automatical reasoner by the AWE Extensions in the following way. First, the tactic `depth_tac`, which provides an exhaustive proof search up to the given depth (see [2]), is applied. The default AWE-value for the depth is 2. In order to change this, users can write:

ML `"AWE.proof_search_depth := N"`

where `N` denotes the new depth.

However, the `depth_tac` does not involve the Isabelle's simplifier sets. So, if the `depth_tac` fails, the proof obligation will be delegated to the Isabelle's `simp_tac`, by default. This tactic is more powerful, but can lead to non-termination in some cases. To get rid of `simp_tac`, users can reset an AWE flag by:

ML `"reset AWE.proof_search_with_simp"`

In particular, the proof obligations generated by the axioms `MonadAxms.mon_assoc` and `MonadAxms.mon_runit` have been also delegated to the Isabelle's reasoner, but the both tactics have failed to prove them.

The current situation in our example is shown in Figure 2.5.

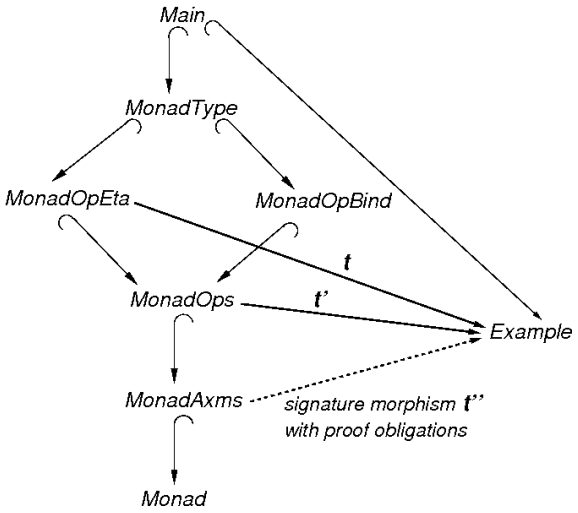


Figure 2.5: Proper instantiation of `MonadAxms` by `t'`

So, our remaining proof obligations are: `mon_assoc` and `mon_runit`, which are already displayed in the translated form, i.e., we can just copy and paste them into `Example` and start to prove them. Done so, we write

thymorph *t* **by_sigmorph** *t''*

which constructs the theory morphism *t* from *MonadAxms* into *Example*. Note that we give to the last theory morphism the same name as for the first theory morphism constructed in Section 2.5, so we cannot refer to that morphism any more.

Finally, we instantiate the theory *Monad* by:

instantiate_theory *Monad* **by_thymorph** *t*

which yields a theory morphism *t' : Monad* ⟶ *Example*.

Now we are able to get a *t'*-instance of any theorem proved for monads as a theorem for *option* by:

translate_thm *"Monad.<some monad theorem>"* **as** *<new name>* **along** *t'*

Let us consider for example the well-known monadic map operation *mapF*, which can be defined in the theory *Monad* as follows:

constdefs

```
 mapF ::  "('a ⇒ 'b) ⇒ 'a M ⇒ 'b M"
"mapF f m == ({a<-m; eta (f a)})"
```

In the same theory the property

lemma *mapF1 : "(mapF f) ∘ eta = eta ∘ f"*

is proven. The theory morphism *t'* allows us to reuse this property by writing in *Example*, e.g.,

translate_thm *"Monad.mapF1"* **along** *t'*

This results in the derivation of the proposition

mapF1 : "(Example.mapF f) ∘ Some = Some ∘ f"

in the theory *Example*. The qualified name *Example.mapF* should just emphasize, that the translated proposition refers to the automatically from *Monad* into *Example* translated derived constant *Monad.mapF*. So, the operation *Example.mapF* is, as expected, of type *"('a ⇒ 'b) ⇒ 'a option ⇒ 'b option"* with the defining equation *"mapF f m == ({a<-m; Some (f a)})"*.

Instantiation of Isabelle/HOL structures

Isabelle provides very useful tools for its different object logics. Since Isabelle/HOL is the most frequently used object logic. there is a number of tools developed for it. Datatypes (**datatype**) and extensible records (**record**) are such tools. Both are based on the general type definition concept for simply typed higher order logics (**typedef**).

These three HOL-tools are supported by the AWE Extension Package in the following sense. If a datatype (record, typedef) is declared in the body of a parametrisation then the corresponding instantiated datatype (record, typedef) will be *generated* by Isabelle in any instantiating theory. This, for example, has the advantage that the instantiation does not need to be axiomatic, since the underlying **typedef**-axiom will be generated as well. Furthermore, such instantiated datatype (record) has all its 'infrastructure' (i.e., induction, cases, simplifiers, etc.) available in the instantiating theory.

The following paragraphs describe more details (including restrictions) of the instantiation of type definitions, datatypes and records.

! Notice, that since type definitions, datatypes and records are Isabelle/HOL tools, the theory *AWE_HOL* has to be loaded in order to use the instantiation mechanism (see also Chapter 3 for technical details).

Type definitions. A type definition has the following form (see [3])

typedef *('a1, 'a2, ...,'aN) T* = S

where S is a predicate depending only on the type variables *'a1, 'a2, ...,'aN*. Such a declaration requires a non-emptyness proof for S, and then introduces the type constructor *T* with the rank *N* together with two functions *Abs_T* and *Rep_T*.

Further, let *t* : *P* $\longrightarrow$ *I* be a theory morphism, and let *B* be a theory containing the type definition above and having *P* as the parent theory. Then the instantiation

instantiate_theory *B* **by_thymorph** *t*
renames: *[(B.T* $\mapsto$ *T')]*

introduces the corresponding type definition *T'* in the instantiating theory *I*, if the following restrictions are satisfied:

1. the term S does not use derived constants which have been introduced in some ancestor theory of *B* which is not an ancestor of *P*; in this special case, when *P* is the single parent theory of *B*, this means that the term S can use any definition from any ancestor of *P*, but no definitions from *B*;
2. the non-emptyness proof can only use global axioms/theorems, i.e. which have been introduced/proven in the common ancestors of *B* and *I*.

Altenatively, loading *AWE* instead of *AWE_HOL*, these restrictions can be avoided, because in this case no type definitions will be *generated* in *I*, but the involved type constructors, constants, etc. will be merely translated separately to *I*. On the other hand, this means that the instantiation now has to be axiomatic, e.g.

instantiate_theory *B* **by_thymorph** *t*
renames: *[(B.T* $\mapsto$ *T')]*
axiomatic

because of the type definition axiom for *T* in the theory *B*. It is a well-known fact that this axiom is a conservative extension to the theory, such that inserting it into the instantiating theory *I* can be seen as a proper instantiation. Hence, a good strategy in this case is to have type definitions in separated theories in order to ensure that no axioms beside **typedef**-axioms will be inserted.

Datatypes. Let us demonstrate the instantiation of datatypes by a small example. Consider the theory *B*, parametrised over two type constructors *T1* and *T2*:

theory *P* **imports** *Main* **begin**
typedecl *T1*
typedecl *T2*
end

theory *B* **imports** *P* **begin**
datatype *T = C1 T1 T*
| C2 T1

end

We could instantiate it as follows:

```
theory I imports Main begin
thymorph t :  P ⟶ I
type_map:  [("T1" ↦ "nat"),
            ("T2" ↦ "nat") ]
instantiate_theory B by_thymorph t
```

In this situation **instantiate_theory** will recognise that there is a datatype structure in the theory `B` and try to instantiate it also as a datatype in the theory `I`, i.e., the instantiation will have the same effect as the declaration:

```
datatype T = C1 nat T
           | C2 nat
```

But such datatype instantiation can also fail in some cases. Consider an alternative body theory:

```
theory B imports P begin
datatype T = C1 T1 T2 T
           | C2 T1
end
```

where we also use the second type parameter in the datatype. Now the same instantiation will fail, although one could think it should just yield the datatype `I.T`:

```
datatype T = C1 nat nat T
           | C2 nat
```

in the theory `I`. The problem is that the datatype package has to distinguish between `T1` and `T2` in `B`, while in `I` this two types coincide. So the datatype package generates in `I` an internal representation for the datatype `I.T`, which structurally differs a little from the representation of `B.T`.

Altogether, the essence is: one has to be very careful with the instantiation of theories parametrised over more than one type constructor, since these can coincide later by a non-injective instantiation.

But in lot of cases there is a way to treat such problematic parametrisations. Consider another alternative body theory:

```
theory B imports P begin
datatype T = C1 "T1 * T2" T
           | C2 T1
end
```

where we now uncurry the constructor `C1` for our two type parameters and obtain essentially the same (isomorphic) datatype. But now the instantiation will succeed, because in this case the datatype package will generate structurally the same representation for `I.T` for any instantiation of `T1` and `T2`.

Finally, consider this quite artifical case where this method will not work:

```
datatype T = C1 T1
           | C2 T2
```

The instantiation above for this datatype will also fail and we cannot represent our type parameters as arguments of some binary type constructor.

Records. The instantiation of a parameterised theory containing records works analogous to the instantiation of datatypes.

Skipping structure instantiations. Users are able to skip the instantiation of datatypes by

ML `"set AWE.skip_datatypes"`

and of records by

ML `"set AWE.skip_records"`

Then, instantiation of a parametrised theory having datatypes or records in its body theory is the same as with type definitions using `AWE` instead of `AWE_HOL`, described above.

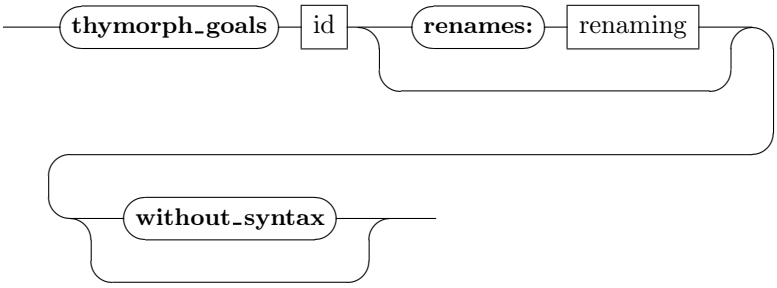
2.8.3 thymorph_goals

The command **thymorph_goals** supports the extension of a given signature morphism σ to a theory morphism. As mentioned already, such an extension can lead to proof obligations: **thymorph_goals** normalises σ, computes the required proof obligations, and displays these. In other words, **thymorph_goals** gives an exact answer to the question: what is to prove in order to extend a signature morphism to a theory morphism.

Syntax and functionality

The syntax of **thymorph_goals** is the following:

thymorph_goals



where the identifier `id` has to denote an existing signature morphism $\sigma : \mathcal{T}_1 \longrightarrow \mathcal{T}_2$. By the normalisation we obtain $\sigma\downarrow: \mathcal{T}_1 \longrightarrow \mathcal{T}_2'$ which (possibly) extends the target theory. The optional renaming map allows to give explicit names in $\mathcal{T}_2'$ for derived constants translated by normalisation.

For $\sigma\downarrow$, the proof obligations (if any) will be displayed that one can copy and paste into the target theory text. Such as with a proper instantiation (Section 2.8.1) one has then to decide which of them to prove and which to assert as axioms. Done so, the signature morphism can be extended to a theory morphism by:

thymorph `<thymorph-id>` **by_sigmorph** `id`

Further, **without_syntax** stops the concrete syntax from the source to be transferred (the same functionality as for **instantiate_theory**) into the target, so that goals will be displayed without it. This, of course, does not affect the semantics.

Notice also, that since **thymorph_goals** may change the signature morphism σ as well as its target theory, it has to stay in the theory text whereever it was employed.

2.9 Control

Finally, we want to present the possibilities to control theory and signature morphisms in Isar.

Overview. To get an overview of all already registered signature or theory morphisms, users can employ the **print_sigmorphs** or **print_thymorphs** commands, respectively.

Printing of morphisms. For any theory or signature morphism from those, which are displayed by the **print_thymorphs** or **print_sigmorphs** commands, users can get its detailed information including domain, codomain, as well as the particular maps by the **print_thymorph** *<id>* or **print_sigmorph** *<id>*, respectively.

Renaming of morphisms. As described in Section 2.8.2, instantiation derives for a given theory morphism `t` its extension, which gets the identifier `t'`. This could be a case, where a more meaningful name for the extended theory morphism is desirable. In order to rename a theory or signature morphism, users can employ the commands **rename_thymorph** *<id>* **to** *<new-id>* or **rename_sigmorph** *<id>* **to** *<new-id>*, respectively.

Chapter 3

Installation and Usage

The AWE Extension Package is easy to install and use.
To make use of the AWE Extensions, Isabelle object logics need to be built with full proof objects. If a logic like HOL has been built without full proofs, one may need to recompile it with the option **-p 2** (for details see [4]); if one sees a message like `"incomplete proof objects"` when running our extensions, this is sign of missing full proof objects.

3.1 Installation

First, unpack the sources; they unpack into a called directory `awe-x.y`, but this can be renamed or moved arbitrarily. We will refer to this directory as *AWE home*. It is recommended to create the environment variable `AWE_HOME` by

```
export AWE_HOME = <path-to-AWE-home>
```

Change directory to the AWE home. Installation consists of a single

```
./configure
```

This will create two files:

1. `<path-to-AWE-home>/Extensions/AWE.thy` – this theory can be used to load the extensions, independent of which object logic of Isabelle has been loaded.
2. `<path-to-AWE-home>/Extensions/AWE_HOL.thy` – this can be used only in the context of Isabelle/HOL or another object logic extending it, e.g., HOLCF, because `AWE_HOL` supports specific concepts of HOL like datatypes and records.

You can now load the extensions into Isabelle as described below.

3.2 Usage

To use the extensions, you must install them first (see 3.1 above).
If you are using Isabelle/HOL or some logic extending HOL, it is recommended to import the theory *AWE_HOL* at the latest in the theory which employs morphisms, by saying, e.g.,
theory ...

imports ... *"<path-to-AWE-home>/Extensions/AWE_HOL"* ...
begin

or,

theory ...
imports ... *"$AWE_HOME/Extensions/AWE_HOL"* ...
begin

if the environment variable `AWE_HOME` is exported as described above in 3.1.

If you are using a different object logic or don't want to import the HOL-theory *Main* with *AWE_HOL*, you can do the same as above, but importing *AWE* instead of *AWE_HOL*.

3.3 ProofGeneral

To integrate the extensions (in particular the new keywords) into ProofGeneral, you must make it use the `isar-keywords.el` file in the directory

```
<path-to-AWE-home>/etc
```

for example by copying that to the directory etc in `ISABELLE_HOME` (usually, isabelle/etc in your home directory). Note that `isar-keywords.el` as given is for use with Isabelle/HOL, but you can change that (see etc/README).

3.4 Restrictions

Further, there are the following important notes:

1. The package does not support the undo mechanism of the Isar-VM, in the sense that undoing changes to a theory, made by the AWE Extension Package commands, will not change existing theory and signature morphisms. In lot of cases this will lead to exceptions. If some changes to a theory are required then it is better also to construct affected theory (signature) morphisms once more.

2. The package does not work well with Isabelle's *quick_and_dirty* flag, i.e., if it is set then especially instantiation of datatypes (described in Section 2.8.2) can fail. One of the reasons is that the *quick_and_dirty* flag omits the construction of proofs, and hence proof terms, and many of the functionalities of the package depend on translating proof terms. ProofGeneral users should check whether it starts Isabelle process in *quick_and_dirty*-mode by default. However, the AWE package resets the flag automatically but, of course, only if it is loaded, so that the situation can occur where the theories loaded before are still in *quick_and_dirty*-mode.

Appendix A

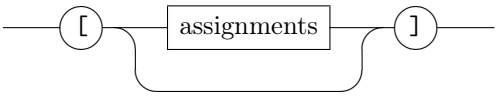
Syntax Primitives

In the following we will use these conventions:

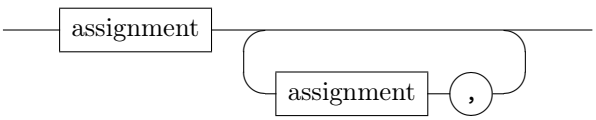
- the non-terminal symbol "string" denotes a quoted string
- the non-terminal symbol "id" denotes a string without quotes
- the non-terminal symbol "declaration" denotes a proper declaration of Isabelle mixfix syntax

The basic syntactic primitive is *mapping*, which is actually a list of assignments. The rules are shown in the following diagram:

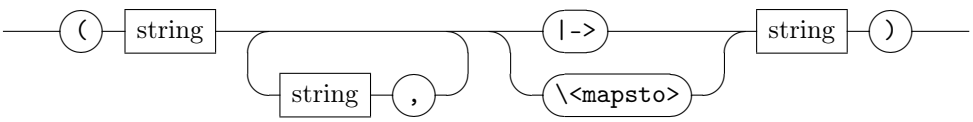
mapping



assignments

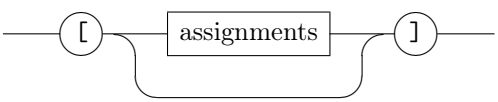


assignment

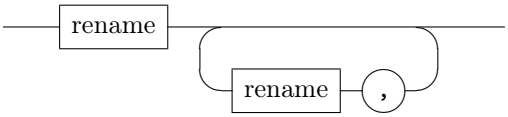


The examples involving *mapping* can be found in Section 2.2 and Section 2.5. Another primitive is *renaming* which is also a list of assignments, but differ a little bit from *mapping* in the rule for *rename*:

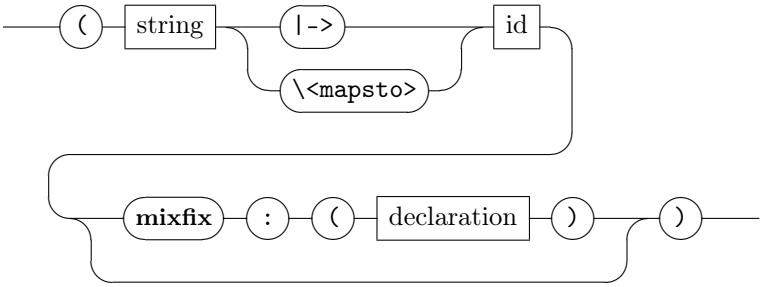
renaming



assignments



rename



The example in Section 2.8.2 shows how *renaming* can be employed.

Bibliography

[1] Wenzel, Markus. 2009. *The Isabelle/Isar Reference Manual*

[2] Paulson, Lawrence C. 2009. *The Isabelle Reference Manual*

[3] Nipkow, Tobias, Paulson, Lawrence C., Wenzel, Markus. 2009. *A Proof Assistant for Higher-Order Logic*

[4] Wenzel, Markus and Berghofer, Stefan. 2009. *The Isabelle System Manual*

[5] Berghofer, Stefan and Nipkow, Tobias. 2000. Proof terms for simply typed higher order logic. In *13th Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'00)*, Volume 1869 of *Lecture Notes in Computer Science*. Springer, 38–52.

[6] Johnsen, Einar Broch, and Lüth, Christoph. Abstracting Refinements for Transformation. *Nordic Journal of Computing* **10**:313–336, 2003.

[7] Johnsen, Einar Broch, and Lüth, Christoph. Theorem Reuse by Proof Term Transformation. In *Proc. 17th Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLs'04)*. Volume 3223 of *Lecture Notes in Computer Science*. Springer, 152-167.

[8] Bortin, Maksym, Johnsen, Einar Broch, and Lüth, Christoph. Structured Formal Development in Isabelle. *Nordic Journal of Computing* **13**:1– 20, 2006.