The AWE Extension Package

Release 0.5 for Isabelle2005

Maksym Bortin
Universität Bremen

Einar Broch Johnsen
University of Oslo

Christoph Lüth
DFKI Bremen

January 14, 2008

Contents

Chapter 1

Morphisms

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

For a detailed description of the Isabelle/Isar framework we refer to [1] and [2]. A brief and more formal introduction to theory and signature morphisms including advanced examples can be found in [5].

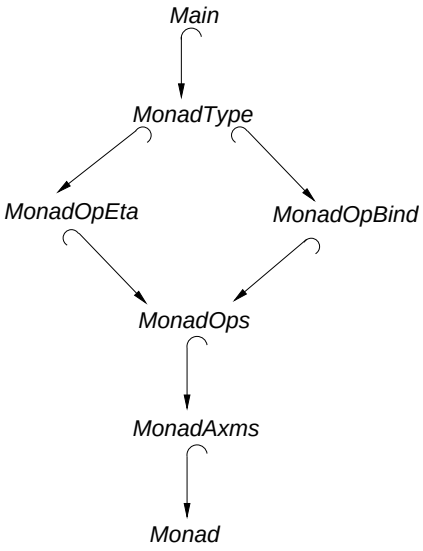


Figure 1.1: Structure of the theories modelling monads.

The monad theory is organised hierarchical as shown in Figure 1.1, and comprises six theories, 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* provides the powerful concrete syntax for monads (for example we can write *({ x<- m; t x })* for *m »= t*), and contains derived propositions.

Note that in order to use monads, we explicitly have to load the monad theory.[1] After loading the theory *Monad*, we start an example theory as follows, and continue to build it up in the following:

theory *Example* **imports** *Main* **begin**

1.2 Constructing a Signature Morphism

1.2.1 Foundations

Theories in Isabelle are structured hierarchically, i.e. there is a basic theory for the meta-logic and any further theory has to *extend* a (finite) set of existing 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 T the finite set of its *ancestor* theories contains T itself together with all theories from which a path to T exists.

The *type signature* of a theory consists of the set of *type constructors* (also called *logical types*) declared in ancestors of the theory. Any type constructor has a fixed rank denoting the number of its arguments. The set of *types* of a theory is then built from the type constructors in its type signature over a fixed infinite set of type variables.

The *operation signature* of a theory consists of the set of *constants* declared in ancestors of this theory. Any constant has a fixed type, taken from the set of types of the theory. The set of *terms* of a theory is then built of the constants in its operation signature over an infinite set of meta-variables. We will consider only well-typed terms, i.e. we assume that every term in the set of terms of a theory has a unique type in the set of types of this theory.

The *signature* of a theory consists of the type and operation signatures of the theory. For detailed description of Isabelle's types and terms, see [2].

A *signature morphism* relates the signatures of two Isabelle theories, the *source* and the *target*. 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.

[1] When using theory morphisms, a theory may require another theory to be loaded which is not an ancestor, so one may have to ***use_thy*** a theory explicitly.

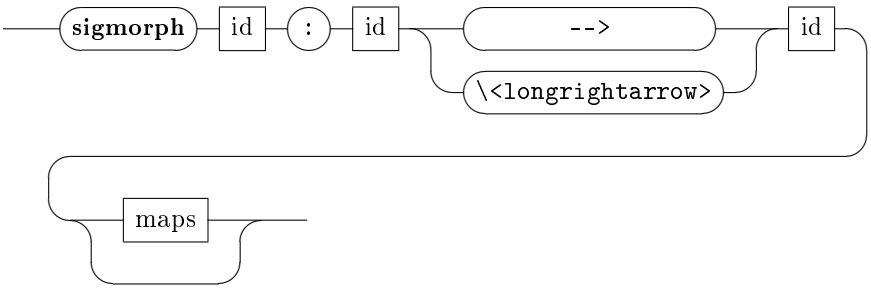
Further, any theory has a finite set of *axioms*, i.e. terms of the meta-logical type `prop`. A *definition* is a special axiom which identifies a constant with a term using the meta-equality `==`. The special form of definitions ensures that they are always *conservative extensions*, i.e. do not affect the consistency of a theory, and allows morphisms to treat definitions in a special way. Axioms, which do not have the form of a definition, we will for simplicity from now on just call axioms.

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 `"`$f\ x_1 \ldots x_n$ `==` t`"` identifying f with a term t. A constant is *logical* if it is not derived, i.e. without a definition in some ancestor.

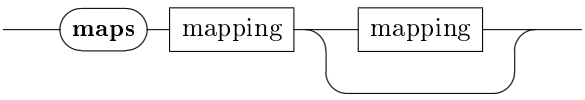
1.2.2 Syntax

The syntax of **sigmorph** is given by the following diagram:

sigmorph



maps



1.2.3 Functionality

The first identifier is the name of the signature morphism to be constructed. The second and third identifier are the source and target theories, respectively. The first mapping defines a *type map*, and the second an *operation map*. The order is important: we will see that any operation map has to respect the type map declared before. The rule *mapping* is given in Appendix A: a mapping consists of a list of assignments ($s \mapsto t$).

Type map

For a type map, both s and t denote arbitrary Isabelle types with the following restrictions:

1. s is a type from the source and t from the target theory of the intended morphism.

2. s has to be a *type pattern*, i.e. has to consist 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.

3. Any type variable which occurs in t has to occur in s.

So, for example, *("('a, 'b) C" ↦ "('b, 'b) D")* is a correct type assignment, if *C* and *D* are type constructors from the source and target respectively with the rank 2.

A type map 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.

Operation map

For an operation map, s and t are strings and will be interpreted as identifiers of constants in source and target signatures of the intended morphism, respectively. The notion of correctness of an operation map is based on a given correct type map: an operation map 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 map and sorts of type variables. Let us explain this by a simple example. Assuming we have the theory

theory *Thy* **imports** *Nat* **begin**
typedecl *'a T*
consts *f :: " 'b ⇒ 'b T ⇒ 'b T "*

then we can start to construct a signature morphism from *Thy* to the HOL-theory *List* starting with a type map as follows:

sigmorph *test : Thy ⟶ List*
maps *[("'a T" ↦ "'a list")]*

The type map is correct, but we further have to map the logical constant *f* to some constant in the target correctly w.r.t to the type map. This can be done by *[("Thy.f" ↦ "List.list.Cons")]*, because the type of the constant *List.list.Cons* is *'a ⇒ 'a list ⇒ 'a list* (where the type variable *'a* is of sort *HOL.type*) and the same type (up to renaming of type variables) we would also obtain replacing *T* by *list* in the type of *Thy.f*, since the type variable *'b* is also of sort *HOL.type* by default. So we can complete the signature morphism:

sigmorph *test : Thy ⟶ List*
maps *[("'a T" ↦ "'a list")]*
[("Thy.f" ↦ "List.list.Cons")]

Note that if we assume that the type variable in the type of the constant *List.list.Cons* would be not of sort *HOL.type* but for example *HOL.plus*, then the mapping above would be incorrect. We could repair this modifiying the declaration

consts *f :: " ('b :: plus) ⇒ 'b T ⇒ 'b T "*

in the theory *Thy*.

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

sigmorph *test : Thy ⟶ List*
maps *[("'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 which satisfy the conditions given by the type map and sorts of type variables. Generally, there are following cases:

1. For some logical constants, more than one correct assignment in the domain exists. In this case the signature morphism construction would fail with a message displaying all possibilities for all this 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, exactly one correct assignment in the domain exists. In this case the signature morphism will be constructed.

3. For some logical constants, no possible assignments in the domain exists. 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 common remove operation on lists having the type *'a ⇒ 'a list ⇒ 'a list*. This means that we have to put *("Thy.f" ↦ "List.list.Cons")* back into the operation map in order to construct the signature morphism *test* as above.

Homomorphic extension of a signature morphism

A signature morphism gives a mapping from the set of terms of its source to the set of terms of its target theory, by replacing all types and constants according to the type and operations maps respectively. The correctness conditions on these maps assure that all type constructors and constants of the source signature are indeed substituted, and that the resulting translated term is well-typed in the target theory. This map is called the *homomorphic extension* of the signature morphism.

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*
maps *[("'a MonadType.M" ↦ "'a Datatype.option")]*

In this case the operation *Datatype.option.Some :: " 'a ⇒ 'a option "* (which is actually a constructor of the datatype *option*) is the unique operation in the target satisfying type map and sort conditions as described above. Thus, we do not need to disambiguate in this case — the following assignment will be derived automatically:

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

1.3 Constructing a Theory Morphism

1.3.1 Foundations

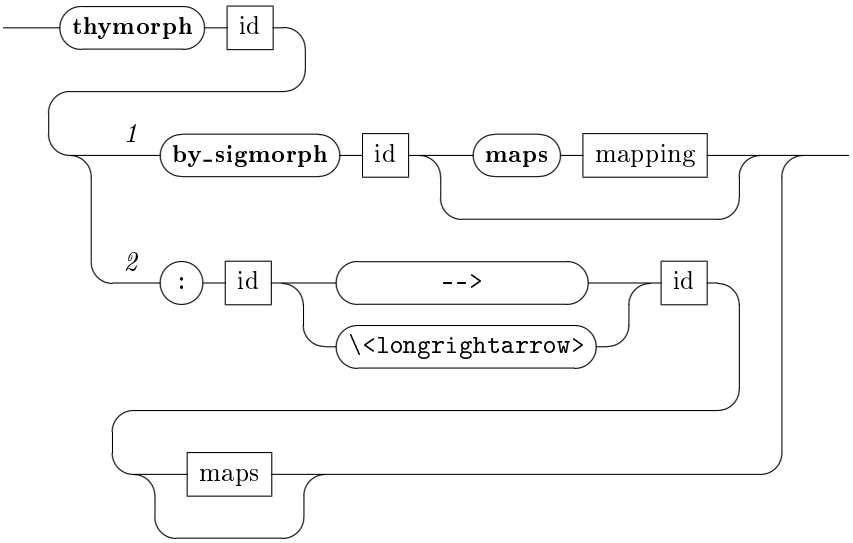
Theory morphisms extend the notion of a signature morphism to axioms (introduced in Section 1.2.1) and *theorems*. Theorems are (as axioms) terms of type `prop`. While axioms are basic propositions without a proof, theorems are derived propositions having a proof built with other propositions. In other words, a *theory morphism* is a signature morphism extended by an *axiom map*.

Domain, codomain, and global theories of a theory morphisms are the same as for its underlying signature morphism.

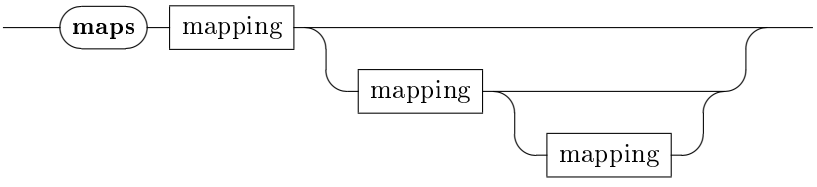
1.3.2 Syntax

The syntax of **thymorph** is given by the following diagram:

thymorph



maps



1.3.3 Functionality

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

1. by giving an identifier of an existing signature morphism and an optional axiom map, or

2. by giving source, target, type, operation and axiom maps.

The second possibility is actually a composed one. Internally it consists of two steps: constructing a signature morphism of the source, target, type and operation maps and then doing the same as before. Notice that the underlying signature morphism is then anonymous, i.e. does not have its own identifier.

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. For an axiom map such an assignment is correct if s denotes a proposition from the domain of the theory morphism to be constructed, while t denotes a proposition which can be either global or from the codomain of the theory morphism, and the term of s translated by the homomorphic extension of the underlying signature morphism yields a term which is equal to the term of t up to renaming of meta-variables.

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 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 1.2 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
 maps [("'c MonadType.M" ↦ "'c Datatype.option")]
```

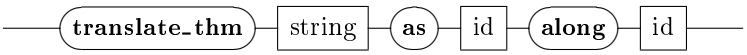
with the response:

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

1.4 translate_thm: Theorem Translation

A theory morphism allows us to move theorems from its domain into its target theory. More exactly, the statement and proof of a theorem can be translated by a theory morphism and then replayed in its target. In order to do so, theorems on which the particular theorem depends (i.e. uses in its proof) have to be translated first. This procedure is implemented by the **translate_thm** command having the following syntax:

translate_thm



`string` is interpreted as the name of the theorem to be translated and has to be fully qualified. The first identifier is the name which will be assigned to the translated proposition, while the third identifier has to refer to an existing theory morphism. One can also use the empty string "" as the name for the translated proposition; in this case a new name will be generated automatically.

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. On the one hand this will make **translate_thm** work much more faster, but on the other hand may cause problems later, since all such translated theorems lose their dependencies.

1.5 Parametrised Theories and Instantiation

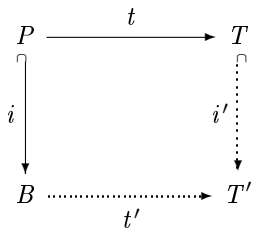
1.5.1 Foundations

If a theory T is an ancestor of a theory T', i.e. T' extends T, then there is a unique theory morphism $i : T \hookrightarrow T'$, the *inclusion*. Its type, operation and axiom maps are empty. In Figure 1.1, all arrows are inclusions. 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 $\langle P, B\rangle$ such that P is an ancestor of B, which gives the inclusion morphism $i : P \hookrightarrow B$. Theory P is called the *parameter* part and B the *body* part of $\langle P, B\rangle$.

The *instantiation* of a parametrised theory is sketched by the following diagram,

where dotted arrows are results of instantiation:



(1.1)

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

First, it is possible to give an assignment for logical types, constants, and axioms of B explicitly during instantiation, which would actually correspond to considering the particular element as a part of the parameter theory P. We will call such elements *explicitly instantiated*.

Then, the theory morphism t' and the theory T' will be constructed as follows:

1. For any type constructor $B.C$ introduced in B we add the type constructor $T.C$ with the same rank to the signature of T, and extend the type map of t by $(\alpha_1, \ldots, \alpha_{i_{B.C}})B.C \mapsto (\alpha_1, \ldots, \alpha_{i_{B.C}})T.C$, unless $B.C$ is explicitly instantiated. If $T.C$ is already in the signature, this requires an explicit renaming. This step gives us the type map for t' and the theory T_1 extending T.

2. For any logical constant $B.f$ introduced in B we add a logical constant $T_1.f$ with a type satisfying the condition given by the type map, constructed in the previous step, to the signature of T_1, and extend the operation map of t by $B.f \mapsto T_1.f$, unless $B.f$ is explicitly instantiated. Here, again, renaming could be required. This step gives us the operation map for t' and the theory T_2 extending T_1, resulting in a signature morphism s with source in B and target in T_2.

3. For any axiom $B.A$ introduced in B we can

   - either *match* $B.A$, i.e. can find a proposition A' searching only in the *codomain* (see Section 1.3.3) of s satisfying the condition given by the operation map of s (see 1.3.1), so that we can extend the axiom map of t by $B.A \mapsto A'$, or
   - *insert* (unless $B.A$ is explicitly instantiated) an axiom $T_2.A$, which again satisfies the condition given by the operation map of s, so that we can extend the axiom map of t by $B.A \mapsto T_2.A$.

This step gives us the theory T' extending T_2 and completes the theory morphism t'.

Note that T has to be the currently developed theory. So, $T \hookrightarrow T_1 \hookrightarrow T_2 \hookrightarrow T'$ denote the development steps of this theory.

Since the third step may introduce new axioms, it is unsafe in the sense that the theory 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 only if all axioms in B have been matched by theorems or axioms in T. The set of unmatched axioms in B we will call *proof obligations*; they can also be displayed with the command **show_thymorph_goals**, described in Section 1.5.3.

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

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

1.5.2 t_instantiate

Syntax and functionality

The syntax of the command **t_instantiate** is shown in Figure 1.2. Referring again to Diagram 1.1, the first identifier corresponds to the theory B and the second to the theory morphism t. The options of **t_instantiate** comprise:

1. A type map (see 1.2.3), which will be interpreted as explicit instantiations for type constructors in the domain of t.
2. An operation map (see 1.2.3), which will be interpreted as explicit instantiations for logical constants in the domain of t.
3. An axiom map (see 1.3.3), which will be interpreted as explicit instantiations for unmatched axioms in the domain of t in any axiomatic instantiation.
4. A renaming map (see the rule for *renaming* in Appendix A), which allows users to change the names of type constructors, constants and axioms as well as the mixfix syntax ot type constructors and constants to be inserted into the theory T.
5. If the keyword **axiomatic** is used then the instantiation will extend the theory T by all unmatched axioms from the domain of t, which are not explicitly instantiated, as described in the previous section.
6. Furthermore, **t_instantiate** will extend T by concrete syntax[2] declared in the domain theories of t, unless the keyword **without_syntax** is used.

[2]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**.

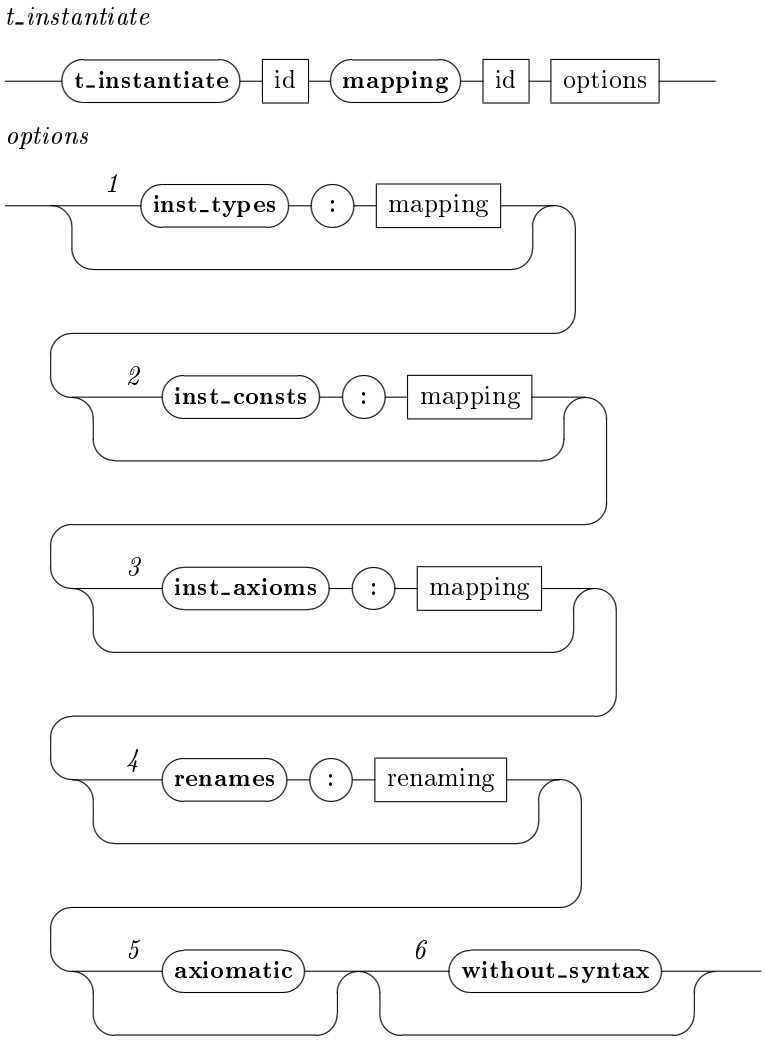


Figure 1.2: Syntax of **t_instantiate**

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

t_instantiate *MonadOps* **mapping** *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`.

Note that in this a case a non-axiomatic instantiation results in a theory morphism, because 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:

t_instantiate *MonadOps* **mapping** *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 if more than one monad instance occurs in a theory, because we can use the concrete monad syntax with only one of them.

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

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

Now, we can instantiate:

t_instantiate *MonadOps* **mapping** *t*
inst_consts: `[("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.

Now let us consider the theory `MonadAxms`. The instantiation

t_instantiate *MonadAxms* **mapping** *t'* **axiomatic**

would insert monad axioms as axioms 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 *bind* operation. So, we choose this possibility and define:

primrec

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

(note that the declaration of the constant *bind* and its infix syntax have been translated during the instantiation of *MonadOps* by *t* above),

and instantiate as follows:

t_instantiate *MonadAxms* **mapping** *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")
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"
lemma mon_eta_inj:
"Some x = Some y ==> x = y"
```

The theorem *Example.bind.simps_2* was automatically derived by Isabelle's primrec package for the definition of *bind* above and proves exactly the monadic left unit property for it. *mon_assoc*, *mon_runit* and *mon_eta_inj* are our proof obligations and are displayed already 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 1.3, so we cannot refer to that morphism any more.

Finally, we instantiate the theory *Monad* by:

t_instantiate *Monad* **mapping** *t*

Now we are able to get any theorem proved for monads as a theorem for *option* by:

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

Instantiation of Isabelle structures

Isabelle provides a number of very useful tools like datatypes and records for HOL. Both concepts are based on the general type definition mechanism in HOL (command **typedef**). Any type definition adds an axiom to the particular theory and requires a non-emptyness proof (see [3]). It is a well-known fact

that this axiom is a conservative extension to the theory. For any instantiation of a parametrised theory having a **typedef** in its body theory this means that it should be axiomatic – a proper instantiation will fail to construct a theory morphism because of the **typedef-**axiom. But since this axiom does not affect the consistency of the body theory, such an axiomatic instantiation will not affect the consistency of any instantiating theory. Hence, a good strategy is to have type definitions in separated theories in order to ensure that no axioms beside **typedef-**axioms will be translated.

For datatypes and records, the AWE Extension Package provides a smoother instantiation possibility: if a datatype (record) is declared in the body of a parametrised theory then the corresponding instantiated datatype (record) 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 underlying **typedef-**axioms 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 some restrictions) of the instantiation of datatypes and records.

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
maps [("T1" ↦ "nat"),
      ("T2" ↦ "nat") ]
t_instantiate B mapping t
```

In this situation **t_instantiate** 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, in contrast, 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 a record works analogous to the instantiation of datatype just described. However, for records extending other records a small patch to the standard Isabelle2005 distribution is required to make some needed selectors visible. This is easily installed, but requires Isabelle to be rebuilt; see Section 3.3 on how to install this patch.

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"*

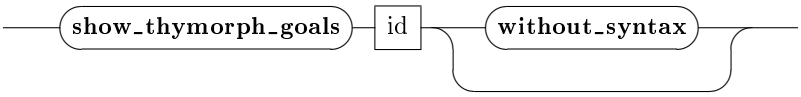
Then, instantiation of a parametrised theory having datatypes or records in its body theory is the same as with type definitions, described at the beginning of this section.

1.5.3 show_thymorph_goals

The command **show_thymorph_goals** supports the extension of a given signature morphism to a theory morphism by showing the proof obligations needed to be proven in order to extend a signatur morphism to a theory morphism. As mentioned in Section 1.5.1, this is useful to construct a proper instantiation.

Syntax and functionality. The syntax of **show_thymorph_goals** is simple:

show_thymorph_goals



where the identifier has to denote an existing signature morphism. For this morphism proof obligations (if any) will be displayed that one can copy and paste into the target theory. Such as with a proper instantiation (Section 1.5.1) 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 ... **by_sigmorph** ...

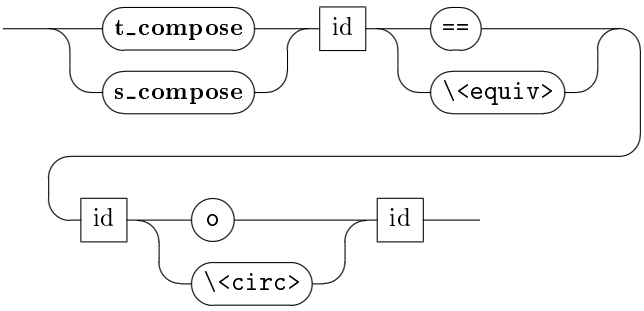
Further, **without_syntax** stops the concrete syntax from the source to be transferred (the same functionality as for **t_instantiate**) into the target, so that goals will be displayed without it. This, of course, does not affect the semantics.

Notice also that since **show_thymorph_goals** may change the signature morphism as well as its target theory, it has to occur in the theory text whenever it was employed.

1.6 Composition

The composition of morphisms provides the possibility to derive new morphisms from already existing ones. The syntax of the composition of theory and signature morphisms (**t_compose** and **s_compose** command, respectively) is given by the following diagram:

composition



The second and third identifiers have to refer to already existing theory (signature) morphisms, while the first identifier will be assigned to the composition morphism. For $t_1 : A \longrightarrow B$ and $t_2 : C \longrightarrow D$ the composition $t \equiv t_2 \circ t_1$, where $t : A \longrightarrow D$, is well-defined if B is an ancestor of C. The type, operation and axiom maps of t are then compositions of the particular maps of t_1 and t_2.

1.7 Construction and Transformation Commands

The AWE Extension Package commands presented so far either construct morphisms between Isabelle theories or work with already constructed morphisms constructing new morphisms and transforming theories. So, we can classify **sig_morph** and **thymorph** as *construction* commands, and in contrast **translate_thm**, **t_instantiate**, **show_thymorph_goals** and **t_compose** as *transformation* commands.

Since transformation commands may change theories, they should be employed according to the following rule:

- Let $t(\phi)$ be some transformation command having morphism ϕ as the argument (for composition this means the morphism t_2). If $t(\phi)$ is used in a theory T then the target of ϕ is an ancestor of T.

We will call this *target-side development*. It was demonstrated by the monad example in the previous sections. The target-side development provides the dynamic target extension of argument morphisms of transformation commands as shown in Figure 1.3: the codomain of ϕ is dynamically adapted to the new theory state at the point of declaration of command involving ϕ.

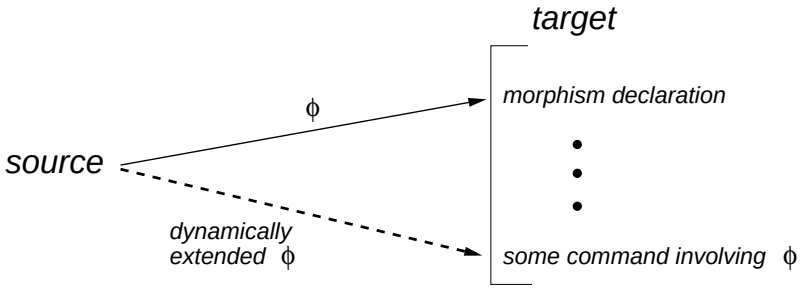


Figure 1.3: Updates of morphisms for target-side development.

In contrast, a construction command can be used in any theory, i.e. it can be for instance the target as well as the source of the intended morphism. The List example in Section 1.2.3 uses the latter possibility, which we will call *source-side construction*. Using the source-side constructions one has to take into account that the source theory of a morphism is fixed at the state where the morphism has been declared, so that all subsequent states will be invisible for the particular morphism. This is pointed up in Figure 1.4: the source of the intended theory morphism **t** would be *not* the theory state at the point of its declaration, but the theory state at the point of the declaration of **s**. This means for instance, that an axiom declared between the declarations of **s** and **t** will not be an element of the source of **t**. In other words, any theory state between the both declarations does not affect neither **s** nor **t**.

1.8 Control

Finally, there is a possibility to get an overview of all already existing theory and signature morphisms, provided by the **t_print** command, which has the

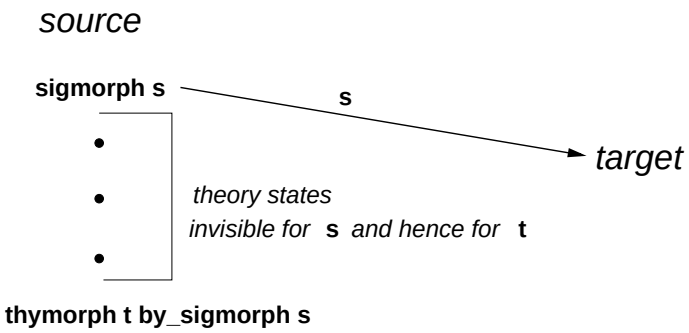
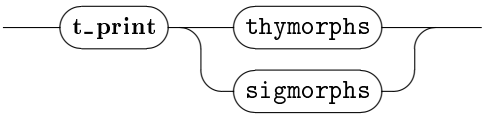


Figure 1.4: Invisible theory states during source-side construction.

following syntax:

t_print



Note that this is an improper command which should not be used inside a theory text.

Chapter 2

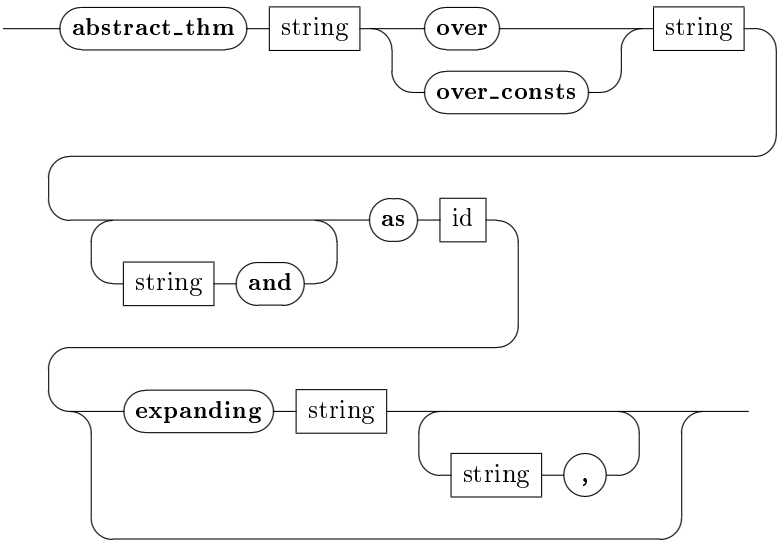
Theorem Abstraction

In this chapter we want to present a technique to make a theorem more generally applicable by making other theorems used in its proof explicit assumptions, replacing eigenvariable operations by meta variables and eigen type variables by type variables. It is implemented by proof term translation, and described in [6] and [7].
The AWE Extension Package provides an Isar command in order to abstract a proposition over given types or constants. To illustrate its use, we will give an example, which first shows the equivalence of a linear-recursive and a tail-recursive functions defined to sum a list of natural numbers and then employs the command to derive a general theorem showing the equivalence of linear- and tail-recursive functions defined over an arbitrary type having the monoid properties.

2.1 Syntax

The syntax of the command is the following:

abstract



The first string should denote the name of theorem to be abstracted. If it is unqualified then it will be interpreted as the name of a theorem from the current

theory. One can refer to a theorem from another theory using a qualified name.

A non-empty sequence of strings after the keyword **over** will be interpreted as Isabelle types to be abstracted, while the sequence of strings after the keyword **over_consts** as constants to be abstracted.

The identifier stands for a name of the resulting abstracted theorem. The optional non-empty sequence of strings after the keyword **expanding** denotes the theorems which have to be expanded in the proof term of the theorem to be abstracted. Next section explains how it can be employed.

2.2 Example

We start with a theory *newNat* which introduces the datatype *nuNat* with a single function *nuPlus* defined on it.

```
theory newNat imports Main begin

datatype nuNat = Zero
               | nuSuc nuNat

consts nuPlus ::  "nuNat ⇒ nuNat ⇒ nuNat"
primrec
"nuPlus Zero x = x"
"nuPlus (nuSuc x) y = nuSuc (nuPlus x y)"
```

Further we prove that *nuPlus* satisfies the associativity law and has *Zero* as its neutral element. This completes the theory *newNat*.

Now we start the theory *TailRec* extending *newNat* and define two versions of summation of a list of *nuNat*, a linear-recursive and a tail-recursive one:

```
consts sum ::  "nuNat list ⇒ nuNat"
recdef sum "measure length"
"sum l = (if l = [] then Zero else nuPlus (hd l) (sum (tl l)))"
consts sum2 ::  "(nuNat list × nuNat) ⇒ nuNat"
recdef sum2 "measure (λ(l, x).  length l)"
"sum2 (l, e) = (if l = [] then e else sum2 (tl l, nuPlus e (hd l)))"
```

We can show that they are equivalent with the following theorems:

```
lemma sum_Cons :  " sum (x#xs) = nuPlus x (sum xs) "
lemma sum2_Cons :  " sum2 (x#xs, a) = sum2 (xs, nuPlus a x) "
lemma nuPlus_1 :  "nuPlus e (sum l) = sum2 (l, e)"
theorem equality :  " sum l = sum2(l, Zero) "
```

where the first two arise just from the definitions, the third is crucial and proved by induction on *l*, while the last is the main result and is proved by instantiation $e \mapsto$ *Zero* in *nuPlus_1*.

Type abstraction over *nuNat* means that we want to make all implicit assumptions referring to *nuNat* in the proof explicit, such that we can replace the type *nuNat* by a type variable. For the theorem *equality*, this is done as follows:

```
abstract_thm "equality" over "newNat.nuNat" as abs_equality
```

This yields the following response:

```
...  abstracting types:  nuNat
...  abstracting ops:  newNat.nuPlus, newNat.nuNat.Zero, TailRec_nat.sum2,
       TailRec_nat.sum
```

```
...  abstracting thms:  TailRec_nat.nuPlus_1, TailRec_nat.sum2.simps,
       newNat.nuPlus.simps_1, TailRec_nat.sum.simps

abstracted thms
abstracted ops
abstracted types.

abs_equality :
"[|
!!l.  ?sum l = (if l = [] then ?Zero else ?nuPlus (hd l) (?sum (tl l)));
!!x.  ?nuPlus ?Zero x = x;
!!l e.  ?sum2.0 (l, e) = (if l = [] then e
                          else ?sum2.0 (tl l, ?nuPlus e (hd l)));
!!e l.  ?nuPlus e (?sum l) = ?sum2.0 (l, e)
|]
⟹ ?sum ?l = ?sum2.0 (?l, ?Zero)"
```

First, we can observe that in the resulting theorem all constants listed in the line *abstracting ops* are abstracted, i.e. they became meta-variables in the derived theorem. Further, the theorems listed in the line *abstracting thms* occur in already abstracted form in the assumptions of *abs_equality*. The reason is that these are exactly the propositions which are used in the proof of *equality*. However, note that the proposition *TailRec_nat.nuPlus_1* occurs among this assumptions, which is derivable from the other ones. This can be avoided by *expanding* the use of *TailRec_nat.nuPlus_1* in the proof of the theorem (with the keyword **expanding**), so it it will not occur in the assumptions (note in general expanding a theorem in a proof may introduce new assumptions):

```
abstract_thm "equality" over "newNat.nuNat" as abs_equality
expanding "TailRec_nat.nuPlus_1"
```

Instead of *TailRec_nat.nuPlus_1*, the following propositions, used in the proof of *TailRec_nat.nuPlus_1*, will occur:

```
newNat.nuPluss_assoc, TailRec_nat.sum2_Cons, TailRec_nat.sum_Cons,
newNat.nuPluss_Zero_Id_right
```

As mentioned above the theorems *TailRec_nat.sum_Cons* and *TailRec_nat.sum2_Cons* follow immediatly from the *.simps* theorems of the recursive definitions of *sum* and *sum2*, respectively, which already occur in the assumptions of the derived theorem. Hence, expanding *TailRec_nat.sum_Cons* and *TailRec_nat.sum2_Cons* will discharge these without introducing new assumptions. So, finally we can write:

```
abstract_thm "equality" over "newNat.nuNat" as abs_equality
expanding "TailRec_nat.nuPlus_1", "TailRec_nat.sum_Cons",
          "TailRec_nat.sum2_Cons"
```

We obtain the following theorem:

```
abs_equality :
"[| !!x.  ?nuPlus ?Zero x = x;
!!u.  ?nuPlus u ?Zero = u;
!!u ua c.  ?nuPlus u (?nuPlus ua c) = ?nuPlus (?nuPlus u ua) c;
!!u.  ?sum u = (if u = [] then ?Zero else ?nuPlus (hd u) (?sum (tl u)));
!!u e.  ?sum2.0 (u, e) = (if u = [] then e
                          else ?sum2.0 (tl u, ?nuPlus e (hd u))) |]
⟹ ?sum ?l = ?sum2.0 (?l, ?Zero)"
```

This we can interpret as follows: for any type `M` equipped with the constants `p :: M ⇒ M ⇒ M` and `e :: M` satisfying the first three assumptions, i.e. the monoid properties, the both functions, satisfying the equations in the fourth and fifth assumptions, also satisfy the equation in the conclusion.

Chapter 3

Installation and Usage

The AWE Extension Package is easy to install and use.

3.1 Installation

First, unpack the sources; they unpack into a called directory `awe-x.y` (where `x.y` is version number), but this can be renamed or moved arbitrarily. Change into that directory. Installation consists of a single

```
./configure
```

To use the instantiation of records extending other records, make the steps described in the paragraph **Records**, Section 1.5.2.

To make use of our extensions, Isabelle needs 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 `"incomplete proof objects"` when running our extensions, this is sign of missing full proof objects.

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

3.2 Usage

To use our extensions, you must install them first (see above), and then use the theory `AWE` in the `Extensions` directory. You can do this by saying

```
use_thy "<path-to-extensions>/Extensions/AWE"
```

in your theory, or you create a soft link from your project to the `AWE.thy` and just use `AWE` as an ancestor.

Bear in mind the remark on page 3: when using theory morphisms, a theory `T` may depend on other theories $\mathtt{S}_1, \ldots, \mathtt{S}_n$ which are not their ancestors. This situation does not normally occur with Isabelle, so users should be careful to handle this extra dependency; in particular, users need to load $\mathtt{S}_1, \ldots, \mathtt{S}_n$ manually.

3.3 Record Patch

The instantiation of records extending another record requires a small patch to the standard Isabelle2005 distribution (to make some needed selectors visible). The patch is performed by the following simple steps:

1. Set the environment variable `ISABELLE_HOME` to the Isabelle's home directory, if not already done so.
2. Call the script `record_patch` in the AWE Extension Package home directory. This will add a selector to the Isabelle/HOL record package and update the AWE Extension Package.
3. (Re)build Isabelle/HOL with full proof objects using, for example, the `build` tool.

3.4 Restrictions

Further, there are the following important notes:

1. The presence of declarations of Isabelle's type classes in the domain of a morphism can lead to exceptions, especially in connection with theorem translation.
2. 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.
3. 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 1.5.2) 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 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

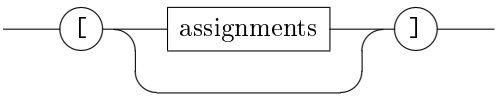
Syntax Primitives

In the following we will use these conventions:

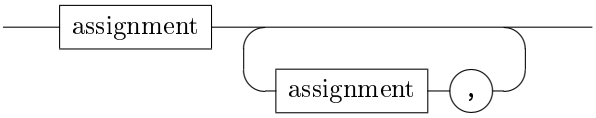
- terminal symbol *id* denote the lexical class of identifiers
- terminal symbol *string* denote the lexical class of quoted strings

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

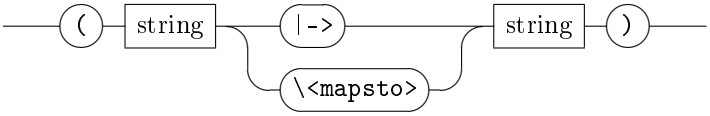
mapping



assignments



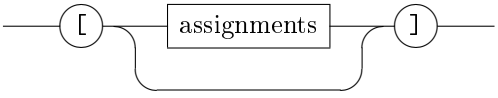
assignment



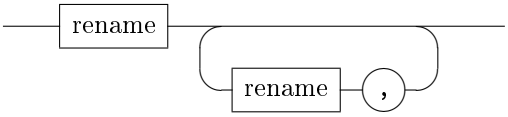
The examples involving *mapping* can be found in Section 1.2 and Section 1.3.

Another primitive is *renaming* which is also a list of assignments, but differ a little bit from *mapping* in the rule for *rename*:

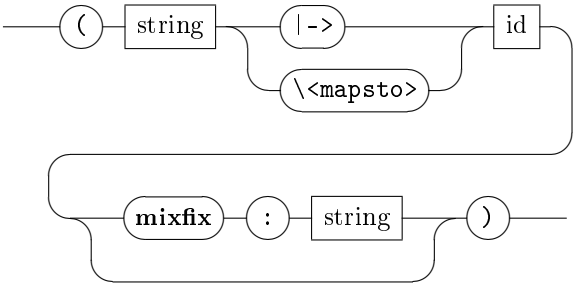
renaming



assignments



rename



The example in Section 1.5.2 shows how *renaming* can be employed.

Bibliography

[1] Wenzel, Markus. 2005. *The Isabelle/Isar Reference Manual*

[2] Paulson, Lawrence C. 2005. *The Isabelle Reference Manual*

[3] Nipkow, Tobias, Paulson, Lawrence C., Wenzel, Markus. 2005. *A Proof Assistant for Higher-Order Logic*

[4] Wenzel, Markus and Berghofer, Stefan. 2005. *The Isabelle System Manual*

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

[6] 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.

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