Tutorial to Locales and Locale Interpretation

Clemens Ballarin

Abstract

Locales are Isabelle's mechanism to deal with parametric theories. We present typical examples of locale specifications, along with interpretations between locales to change their hierarchic dependencies and interpretations to reuse locales in theory contexts and proofs.

This tutorial is intended for locale novices; familiarity with Isabelle and Isar is presumed.

1 Introduction

Locales are based on contexts. A *context* can be seen as a formula schema

$$\bigwedge \mathtt{x}_1 \ldots \mathtt{x}_n .\ [\![\ \mathtt{A}_1;\ \ldots\ ;\mathtt{A}_m\ ]\!] \Longrightarrow \ldots$$

where variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$ are called *parameters* and the premises $\mathtt{A}_1, \ldots, \mathtt{A}_m$ *assumptions*. A formula $\mathtt{C}$ is a *theorem* in the context if it is a conclusion

$$\bigwedge \mathtt{x}_1 \ldots \mathtt{x}_n .\ [\![\ \mathtt{A}_1;\ \ldots\ ;\mathtt{A}_m\ ]\!] \Longrightarrow \mathtt{C}.$$

Isabelle/Isar's notion of context goes beyond this logical view. Its contexts record, in a consecutive order, proved conclusions along with attributes, which may control proof procedures. Contexts also contain syntax information for parameters and for terms depending on them.

2 Simple Locales

Locales can be seen as persistent contexts. In its simplest form, a *locale declaration* consists of a sequence of context elements declaring parameters (keyword **fixes**) and assumptions (keyword **assumes**). The following is the specification of partial orders, as locale `partial_order`.

```
locale partial_order =
  fixes le :: "'a ⇒ 'a ⇒ bool" (infixl "⊑" 50)
  assumes refl [intro, simp]: "x ⊑ x"
    and anti_sym [intro]: "⟦ x ⊑ y; y ⊑ x ⟧ ⟹ x = y"
    and trans [trans]: "⟦ x ⊑ y; y ⊑ z ⟧ ⟹ x ⊑ z"
```

definition	definition through an equation
inductive	inductive definition
fun, **function**	recursive function
abbreviation	syntactic abbreviation
theorem, etc.	theorem statement with proof
theorems, etc.	redeclaration of theorems

Table 1: Isar commands that accept a target.

The parameter of this locale is `le`, with infix syntax $\sqsubseteq$. There is an implicit type parameter `'a`. It is not necessary to declare parameter types: most general types will be inferred from the context elements for all parameters.

The above declaration not only introduces the locale, it also defines the *locale predicate* `partial_order` with definition `partial_order_def`:

```
partial_order ?le ≡
(∀x. ?le x x) ∧
(∀x y. ?le x y ⟶ ?le y x ⟶ x = y) ∧
(∀x y z. ?le x y ⟶ ?le y z ⟶ ?le x z)
```

The specification of a locale is fixed, but its list of conclusions may be extended through Isar commands that take a *target* argument. In the following, **definition** and **theorem** are illustrated. Table 1 lists Isar commands that accept a target. There are various ways of specifying the target. A target for a single command may be indicated with keyword **in** in the following way:

```
definition (in partial_order)
  less :: "'a ⇒ 'a ⇒ bool" (infixl "⊏" 50)
  where "(x ⊏ y) = (x ⊑ y ∧ x ≠ y)"
```

A definition in a locale depends on the locale parameters. Here, a global constant `partial_order.less` is declared, which is lifted over the locale parameter `le`. Its definition is the global theorem `partial_order.less_def`:

```
partial_order ?le ⟹
partial_order.less ?le ?x ?y = (?le ?x ?y ∧ ?x ≠ ?y)
```

At the same time, the locale is extended by syntax information hiding this construction in the context of the locale. That is, `partial_order.less le` is printed and parsed as infix $\sqsubset$. Finally, the conclusion of the definition is added to the locale, `less_def`:

```
(?x ⊏ ?y) = (?x ⊑ ?y ∧ ?x ≠ ?y)
```

As an example of a theorem statement in the locale, here is the derivation of a transitivity law.

```
lemma (in partial_order) less_le_trans [trans]:
  "⟦ x ⊏ y; y ⊑ z ⟧ ⟹ x ⊏ z"
  unfolding less_def by (blast intro: trans)
```

In the context of the proof, assumptions and theorems of the locale may be used. Attributes are effective: `anti_sym` was declared as introduction rule, hence it is in the context's set of rules used by the classical reasoner by default.

When working with locales, sequences of commands with the same target are frequent. A block of commands, delimited by **begin** and **end**, makes a theory-like style of working possible. All commands inside the block refer to the same target. A block may immediately follow a locale declaration, which makes that locale the target. Alternatively the target for a block may be given with the **context** command.

In the block below, notions of infimum and supremum together with theorems are introduced for partial orders.

```
context partial_order begin

definition
  is_inf where "is_inf x y i =
    (i ⊑ x ∧ i ⊑ y ∧ (∀z. z ⊑ x ∧ z ⊑ y ⟶ z ⊑ i))"

definition
  is_sup where "is_sup x y s =
    (x ⊑ s ∧ y ⊑ s ∧ (∀z. x ⊑ z ∧ y ⊑ z ⟶ s ⊑ z))"

theorem is_inf_uniq: "⟦is_inf x y i; is_inf x y i'⟧ ⟹ i = i'"
⟨proof⟩

theorem is_sup_uniq: "⟦is_sup x y s; is_sup x y s'⟧ ⟹ s = s'"
⟨proof⟩

end
```

In fact, many more theorems need to be shown for a usable theory of partial orders. The above two serve as illustrative examples.

Two commands are provided to inspect locales: **print_locales** lists the names of all locales of the theory; **print_locale** n prints the parameters and assumptions of locale n; **print_locale!** n additionally outputs the conclusions.

The syntax of the locale commands discussed in this tutorial is shown in Table 4. See the Isabelle/Isar Reference Manual [6] for full documentation.

3 Import

Algebraic structures are commonly defined by adding operations and properties to existing structures. For example, partial orders are extended to lattices and total orders. Lattices are extended to distributive lattices.

With locales, this inheritance is achieved through *import* of a locale. The import comes before the context elements.

```
locale lattice = partial_order +
  assumes ex_inf: "∃inf. partial_order.is_inf le x y inf"
    and ex_sup: "∃sup. partial_order.is_sup le x y sup"
begin
```

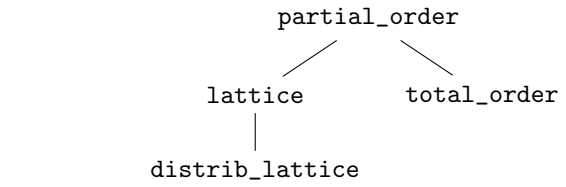
Note that the assumptions above refer to the predicates for infimum and supremum defined in `partial_order`. In the current implementation of locales, syntax from definitions of the imported locale is unavailable in the locale declaration, neither are their names. Hence we refer to the constants of the theory. The names and syntax is available below, in the context of the locale.

```
definition
  meet (infixl "⊓" 70) where "x ⊓ y = (THE inf. is_inf x y inf)"

definition
  join (infixl "⊔" 65) where "x ⊔ y = (THE sup. is_sup x y sup)"

end
```

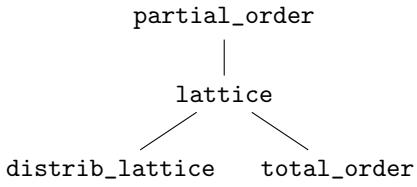
Locales for total orders and distributive lattices follow. Each comes with an example theorem.

```
locale total_order = partial_order +
  assumes total: "x ⊑ y ∨ y ⊑ x"

lemma (in total_order) less_total: "x ⊏ y ∨ x = y ∨ y ⊏ x"
  ⟨proof⟩

locale distrib_lattice = lattice +
  assumes meet_distr:
    "lattice.meet le x (lattice.join le y z) =
    lattice.join le (lattice.meet le x y) (lattice.meet le x z)"

lemma (in distrib_lattice) join_distr:
  "x ⊔ (y ⊓ z) = (x ⊔ y) ⊓ (x ⊔ z)"
  ⟨proof⟩
```

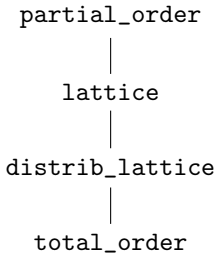
The locale hierachy obtained through these declarations is shown in Figure 1(a).



(a) Declared hierachy



(b) Total orders are lattices



(c) Total orders are distributive lattices

Figure 1: Hierarchy of Lattice Locales.

4 Changing the Locale Hierarchy

Total orders are lattices. Hence, by deriving the lattice axioms for total orders, the hierarchy may be changed and `lattice` be placed between `partial_order` and `total_order`, as shown in Figure 1(b). Changes to the locale hierarchy may be declared with the **interpretation** command.

interpretation `total_order ⊆ lattice`

This enters the context of locale `total_order`, in which the goal

```
1. lattice op ⊑
```

must be shown. First, the locale predicate needs to be unfolded — for example using its definition or by introduction rules provided by the locale package. The methods `intro_locales` and `unfold_locales` automate this. They are aware of the current context and dependencies between locales and automatically discharge goals implied by these. While `unfold_locales` always unfolds locale predicates to assumptions, `intro_locales` only unfolds definitions along the locale hierarchy, leaving a goal consisting of predicates defined by the locale package. Occasionally the latter is of advantage since the goal is smaller.

For the current goal, we would like to get hold of the assumptions of `lattice`, hence `unfold_locales` is appropriate.

proof `unfold_locales`

Since both `lattice` and `total_order` inherit `partial_order`, the assumptions of the latter are discharged, and the only subgoals that remain are the assumptions introduced in `lattice`

```
1. ⋀x y. ∃inf. is_inf x y inf
2. ⋀x y. ∃sup. is_sup x y sup
```

The proof for the first subgoal is

```
fix x y
from total have "is_inf x y (if x ⊑ y then x else y)"
  by (auto simp: is_inf_def)
then show "∃inf. is_inf x y inf" ..
```

The proof for the second subgoal is analogous and not reproduced here.

qed

Similarly, total orders are distributive lattices.

interpretation `total_order ⊆ distrib_lattice`
⟨*proof*⟩

The locale hierarchy is now as shown in Figure 1(c).

5 Use of Locales in Theories and Proofs

Locales enable to prove theorems abstractly, relative to sets of assumptions. These theorems can then be used in other contexts where the assumptions themselves, or instances of the assumptions, are theorems. This form of theorem reuse is called *interpretation*.

The changes of the locale hierarchy from the previous sections are examples of interpretations. The command **interpretation** $l_1 \subseteq l_2$ is said to *interpret* locale l_2 in the context of l_1. It causes all theorems of l_2 to be made available in l_1. The interpretation is *dynamic*: not only theorems already present in l_2 are available in l_1. Theorems that will be added to l_2 in future will automatically be propagated to l_1.

Locales can also be interpreted in the contexts of theories and structured proofs. These interpretations are dynamic, too. Theorems added to locales will be propagated to theories. In this section the interpretation in theories is illustrated; interpretation in proofs is analogous. As an example consider, the type of natural numbers `nat`. The order relation $\leq$ is a total order over `nat`, divisibility `dvd` is a distributive lattice.

We start with the interpretation that $\leq$ is a partial order. The facilities of the interpretation command are explored in three versions.

5.1 First Version: Replacement of Parameters Only

In the most basic form, interpretation just replaces the locale parameters by terms. The following command interprets the locale `partial_order` in the global context of the theory. The parameter `le` is replaced by `op` $\leq$.

interpretation `nat: partial_order ["op ≤ :: nat ⇒ nat ⇒ bool"]`

The locale name is succeeded by a *parameter instantiation*. In general, this is a list of terms, which refer to the parameters in the order of declaration in the locale. The locale name is preceded by an optional *interpretation prefix*, which is used to qualify names from the locale in the global context.
The command creates the goal

```
1. partial_order op ≤
```

which can be shown easily:[1]

by `unfold_locales auto`

Now theorems from the locale are available in the theory, interpreted for natural numbers, for example `nat.trans`:

```
⟦?x ≤ ?y; ?y ≤ ?z⟧ ⟹ ?x ≤ ?z
```

In order to avoid unwanted hiding of theorems, interpretation accepts a qualifier, `nat` in the example, which is applied to all names processed by the interpretation. If present the qualifier is mandatory — that is, the above theorem cannot be referred to simply as `trans`.

5.2 Second Version: Replacement of Definitions

The above interpretation also creates the theorem `nat.less_le_trans`:

```
⟦partial_order.less op ≤ ?x ?y; ?y ≤ ?z⟧
⟹ partial_order.less op ≤ ?x ?z
```

Here, `partial_order.less op` $\leq$ represents the strict order, although `<` is defined on `nat`. Interpretation enables to map concepts introduced in locales through definitions to the corresponding concepts of the theory.[2]

This is achieved by unfolding suitable equations during interpretation. These equations are given after the keyword **where** and require proofs. The revised command, replacing $\sqsubset$ by `<`, is:

interpretation `nat: partial_order ["op ≤ :: [nat, nat] ⇒ bool"]`

[1] Note that `op` binds tighter than functions application: parentheses around `op` $\leq$ are not necessary.

[2] This applies not only to **definition** but also to **inductive**, **fun** and **function**.

```
  where "partial_order.less op ≤ (x::nat) y = (x < y)"
proof -
```

The goals are

```
 1. partial_order op ≤
 2. partial_order.less op ≤ x y = (x < y)
```

The proof that $\leq$ is a partial order is a above.

```
  show "partial_order (op ≤ :: nat ⇒ nat ⇒ bool)"
    by unfold_locales auto
```

The second goal is shown by unfolding the definition of `partial_order.less`.

```
  show "partial_order.less op ≤ (x::nat) y = (x < y)"
    unfolding partial_order.less_def [OF ‘partial_order op ≤‘]
    by auto
qed
```

Note that the above proof is not in the context of a locale. Hence, the correct interpretation of `partial_order.less_def` is obtained manually with `OF`.

5.3 Third Version: Local Interpretation

In the above example, the fact that $\leq$ is a partial order for the natural numbers was used in the proof of the second goal. In general, proofs of the equations may involve theorems implied by the fact the assumptions of the instantiated locale hold for the instantiating structure. If these theorems have been shown abstractly in the locale they can be made available conveniently in the context through an auxiliary local interpretation (keyword **interpret**). This interpretation is inside the proof of the global interpretation. The third revision of the example illustrates this.

```
interpretation nat: partial_order ["op ≤ :: nat ⇒ nat ⇒ bool"]
  where "partial_order.less (op ≤) (x::nat) y = (x < y)"
proof -
  show "partial_order (op ≤ :: nat ⇒ nat ⇒ bool)"
    by unfold_locales auto
  then interpret nat: partial_order ["op ≤ :: [nat, nat] ⇒ bool"] .
  show "partial_order.less (op ≤) (x::nat) y = (x < y)"
    unfolding nat.less_def by auto
qed
```

The inner interpretation does not require an elaborate new proof, it is immediate from the preceeding fact and proved with ".". This interpretation enriches the local proof context by the very theorems also obtained in the interpretation from Section 5.1, and `nat.less_def` may directly be used to

unfold the definition. Theorems from the local interpretation disappear after leaving the proof context — that is, after the closing **qed** — and are then replaced by those with the desired substitutions of the strict order.

5.4 Further Interpretations

Further interpretations are necessary to reuse theorems from the other locales. In `lattice` the operations $\sqcap$ and $\sqcup$ are substituted by `min` and `max`. The entire proof for the interpretation is reproduced in order to give an example of a more elaborate interpretation proof.

```
interpretation nat: lattice ["op ≤ :: nat ⇒ nat ⇒ bool"]
  where "lattice.meet op ≤ (x::nat) y = min x y"
    and "lattice.join op ≤ (x::nat) y = max x y"
proof -
  show "lattice (op ≤ :: nat ⇒ nat ⇒ bool)"
```

We have already shown that this is a partial order,

```
    apply unfold_locales
```

hence only the lattice axioms remain to be shown:

```
1. ⋀x y. ∃inf. partial_order.is_inf op ≤ x y inf
2. ⋀x y. ∃sup. partial_order.is_sup op ≤ x y sup
```

After unfolding `is_inf` and `is_sup`,

```
    apply (unfold nat.is_inf_def nat.is_sup_def)
```

the goals become

```
1. ⋀x y. ∃inf≤x. inf ≤ y ∧ (∀z. z ≤ x ∧ z ≤ y ⟶ z ≤ inf)
2. ⋀x y. ∃sup≥x. y ≤ sup ∧ (∀z. x ≤ z ∧ y ≤ z ⟶ sup ≤ z)
```

which can be solved by Presburger arithmetic.

```
    by arith+
```

In order to show the equations, we put ourselves in a situation where the lattice theorems can be used in a convenient way.

```
  then interpret nat: lattice ["op ≤ :: nat ⇒ nat ⇒ bool"] .
  show "lattice.meet op ≤ (x::nat) y = min x y"
    by (bestsimp simp: nat.meet_def nat.is_inf_def)
  show "lattice.join op ≤ (x::nat) y = max x y"
    by (bestsimp simp: nat.join_def nat.is_sup_def)
qed
```

That the relation $\leq$ is a total order completes this sequence of interpretations.

```
interpretation nat: total_order ["op ≤ :: nat ⇒ nat ⇒ bool"]
```

`nat.less_def` from locale `partial_order`: `(?x < ?y) = (?x ≤ ?y ∧ ?x ≠ ?y)`
`nat.meet_left` from locale `lattice`: `min ?x ?y ≤ ?x`
`nat.join_distr` from locale `distrib_lattice`: `max ?x (min ?y ?z) = min (max ?x ?y) (max ?x ?z)`
`nat.less_total` from locale `total_order`: `?x < ?y ∨ ?x = ?y ∨ ?y < ?x`

Table 2: Interpreted theorems for $\leq$ on the natural numbers.

by `unfold_locales arith`

Theorems that are available in the theory at this point are shown in Table 2.

Note that since the locale hierarchy reflects that total orders are distributive lattices, an explicit interpretation of distributive lattices for the order relation on natural numbers is not neccessary.

Why not push this idea further and just give the last interpretation as a single interpretation instead of the sequence of three? The reasons for this are twofold:

- Often it is easier to work in an incremental fashion, because later interpretations require theorems provided by earlier interpretations.
- Assume that a definition is made in some locale l_1, and that l_2 imports l_1. Let an equation for the definition be proved in an interpretation of l_2. The equation will be unfolded in interpretations of theorems added to l_2 or below in the import hierarchy, but not for theorems added above l_2. Hence, an equation interpreting a definition should always be given in an interpretation of the locale where the definition is made, not in an interpretation of a locale further down the hierarchy.

5.5 Lattice `dvd` on `nat`

Divisibility on the natural numbers is a distributive lattice but not a total order. Interpretation again proceeds incrementally.

interpretation `nat_dvd: partial_order ["op dvd :: nat ⇒ nat ⇒ bool"]`
  where `"partial_order.less op dvd (x::nat) y = (x dvd y ∧ x ≠ y)"`
⟨*proof*⟩

Note that there is no symbol for strict divisibility. Instead, interpretation substitutes `x dvd y ∧ x ≠ y`.

`nat_dvd.less_def` from locale `partial_order`:
`(?x dvd ?y ∧ ?x ≠ ?y) = (?x dvd ?y ∧ ?x ≠ ?y)`
`nat_dvd.meet_left` from locale `lattice`:
`gcd (?x, ?y) dvd ?x`
`nat_dvd.join_distr` from locale `distrib_lattice`:
`lcm (?x, gcd (?y, ?z)) = gcd (lcm (?x, ?y), lcm (?x, ?z))`

Table 3: Interpreted theorems for `dvd` on the natural numbers.

```
interpretation nat_dvd: lattice ["op dvd :: nat ⇒ nat ⇒ bool"]
  where nat_dvd_meet_eq:
      "lattice.meet op dvd (x::nat) y = gcd (x, y)"
    and nat_dvd_join_eq:
      "lattice.join op dvd (x::nat) y = lcm (x, y)"
⟨proof⟩
```

Equations `nat_dvd_meet_eq` and `nat_dvd_join_eq` are named since they are handy in the proof of the subsequent interpretation.

```
interpretation nat_dvd:
  distrib_lattice ["op dvd :: nat ⇒ nat ⇒ bool"]
  apply unfold_locales

 1. ⋀x y z.
       lattice.meet op dvd x (lattice.join op dvd y z) =
       lattice.join op dvd (lattice.meet op dvd x y)
        (lattice.meet op dvd x z)

  apply (unfold nat_dvd_meet_eq nat_dvd_join_eq)

 1. ⋀x y z. gcd (x, lcm (y, z)) = lcm (gcd (x, y), gcd (x, z))

  apply (rule gcd_lcm_distr) done
```

Theorems that are available in the theory after these interpretations are shown in Table 3.

The full syntax of the interpretation commands is shown in Table 4. The grammar refers to *expr*, which stands for a *locale* expression. Locale expressions are discussed in Section 6.

6 Locale Expressions

A map φ between partial orders $\sqsubseteq$ and $\preceq$ is called order preserving if `x` $\sqsubseteq$ `y` implies φ `x` $\preceq$ φ `y`. This situation is more complex than those encountered so far: it involves two partial orders, and it is desirable to use the existing locale for both.

Inspecting the grammar of locale commands in Table 4 reveals that the import of a locale can be more than just a single locale. In general, the import is a *locale expression*. Locale expressions enable to combine locales and rename parameters. A locale name is a locale expression. If e_1 and e_2 are locale expressions then $e_1 + e_2$ is their *merge*. If e is an expression, then $e\, q_1 \ldots q_n$ is a *renamed expression* where the parameters in e are renamed to $q_1 \ldots q_n$. Using a locale expression, a locale for order preserving maps can be declared in the following way.

```
locale order_preserving =
  partial_order + partial_order le' (infixl "⪯" 50) +
  fixes φ :: "'a ⇒ 'b"
  assumes hom_le: "x ⊑ y ⟹ φ x ⪯ φ y"
```

The second line contains the expression, which is the merge of two partial order locales. The parameter of the second one is `le'` with new infix syntax $\preceq$. The parameters of the entire locale are `le`, `le'` and φ. This is their *canonical order*, which is obtained by a left-to-right traversal of the expression, where only the new parameters are appended to the end of the list. The parameters introduced in the locale elements of the declaration follow.

In renamings parameters are referred to by position in the canonical order; an underscore is used to skip a parameter position, which is then not renamed. Renaming deletes the syntax of a parameter unless a new mixfix annotation is given.

Parameter renamings are morphisms between locales. These can be lifted to terms and theorems and thus be applied to assumptions and conclusions. The assumption of a merge is the conjunction of the assumptions of the merged locale. The conclusions of a merge are obtained by appending the conclusions of the left locale and of the right locale.

The locale `order_preserving` contains theorems for both orders $\sqsubseteq$ and $\preceq$. How can one refer to a theorem for a particular order, $\sqsubseteq$ or $\preceq$? Names in locales are qualified by the locale parameters. More precisely, a name is qualified by the parameters of the locale in which its declaration occurs. Here are examples:

`le.less_le_trans:` $[\![$`?x` $\sqsubset$ `?y; ?y` $\sqsubseteq$ `?z`$]\!] \Longrightarrow$ `?x` $\sqsubset$ `?z`

`le_le'_`φ`.hom_le:` `?x` $\sqsubseteq$ `?y` $\Longrightarrow$ φ `?x` $\preceq$ φ `?y`

When renaming a locale, the morphism is also applied to the qualifiers.

Hence theorems for the partial order $\preceq$ are qualified by `le'`. For example, `le'.less_le_trans`:

```
⟦partial_order.less op ⊑ ?x ?y; ?y ⊑ ?z⟧
⟹ partial_order.less op ⊑ ?x ?z
```

This example reveals that there is no infix syntax for the strict version of $\preceq$! This can, of course, not be introduced automatically, but it can be declared manually through an abbreviation.

```
abbreviation (in order_preserving)
  less' (infixl "≺" 50) where "less' ≡ partial_order.less le'"
```

Now the theorem is displayed nicely as $[\![?x \prec ?y; ?y \preceq ?z]\!] \Longrightarrow ?x \prec ?z$.

Not only names of theorems are qualified. In fact, all names are qualified, in particular names introduced by definitions and abbreviations. The name of the strict order of $\sqsubseteq$ is `le.less` and therefore `le'.less` is the name of the strict order of $\preceq$. Hence, the equation in the above abbreviation could have been written as `less' ≡ le'.less`.

Two more locales illustrate working with locale expressions. A map φ is a lattice homomorphism if it preserves meet and join.

```
locale lattice_hom = lattice + lattice le' (infixl "⊑" 50) +
  fixes φ
  assumes hom_meet:
      "φ (lattice.meet le x y) = lattice.meet le' (φ x) (φ y)"
    and hom_join:
      "φ (lattice.join le x y) = lattice.join le' (φ x) (φ y)"

abbreviation (in lattice_hom)
  meet' (infixl "⊓'" 50) where "meet' ≡ le'.meet"
abbreviation (in lattice_hom)
  join' (infixl "⊔'" 50) where "join' ≡ le'.join"
```

A homomorphism is an endomorphism if both orders coincide.

```
locale lattice_end =
  lattice_hom le (infixl "⊑" 50) le (infixl "⊑" 50)
```

The inheritance diagram of the situation we have now is shown in Figure 2, where the dashed line depicts an interpretation which is introduced below. Renamings are indicated by $\sqsubseteq \mapsto \preceq$ etc. The expression imported by `lattice_end` identifies the first and second parameter of `lattice_hom`. By looking at the inheritance diagram it would seem that two identical copies of each of the locales `partial_order` and `lattice` are imported. This is not the case! Inheritance paths with identical morphisms are detected and the conclusions of the respecitve locales appear only once.

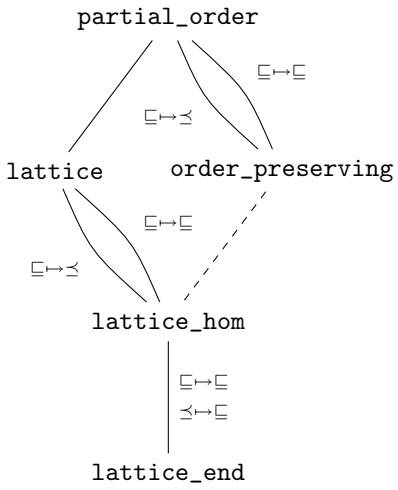


Figure 2: Hierarchy of Homomorphism Locales.

It can be shown easily that a lattice homomorphism is order preserving. As the final example of this section, a locale interpretation is used to assert this.

interpretation `lattice_hom` ⊆ `order_preserving` ⟨*proof*⟩

Theorems and other declarations — syntax, in particular — from the locale `order_preserving` are now active in `lattice_hom`, for example

`le'.less_le_trans`: ⟦`?x` ≺ `?y`; `?y` ≼ `?z`⟧ ⟹ `?x` ≺ `?z`

7 Further Reading

More information on locales and their interpretation is available. For the locale hierarchy of import and interpretation dependencies see [1]; interpretations in theories and proofs are covered in [2]. In the latter, we show how interpretation in proofs enables to reason about families of algebraic structures, which cannot be expressed with locales directly.

Haftmann and Wenzel [3] overcome a restriction of axiomatic type classes through a combination with locale interpretation. The result is a Haskell-style class system with a facility to generate Haskell code. Classes are sufficient for simple specifications with a single type parameter. The locales for orders and lattices presented in this tutorial fall into this category. Order preserving maps, homomorphisms and vector spaces, on the other hand, do not.

The original work of Kammüller on locales [5] may be of interest from a

historical perspective. The mathematical background on orders and lattices is taken from Jacobson's textbook on algebra [4, Chapter 8].

Acknowledgements. Alexander Krauss, Tobias Nipkow, Christian Sternagel and Makarius Wenzel have made useful comments on a draft of this document.

Miscellaneous

attr-name	::=	*name* | *attribute* | *name attribute*

Context Elements

fixes	::=	*name* ["**::**" *type*] ["(" **structure** ")" | *mixfix*]
assumes	::=	[*attr-name* "**:**"] *proposition*
element	::=	**fixes** *fixes* (**and** *fixes*)*
	|	**assumes** *assumes* (**and** *assumes*)*

Locale Expressions

rename	::=	*name* [*mixfix*] | "_"
expr	::=	*renamed-expr* ("+" *renamed-expr*)*
renamed-expr	::=	(*qualified-name* | "(" *expr* ")") *rename**

Declaration of Locales

locale	::=	*element*$^+$
	|	*locale-expr* ["+" *element*$^+$]
toplevel	::=	**locale** *name* ["=" *locale*]

Interpretation

equation	::=	[*attr-name* "**:**"] *prop*
insts	::=	["[" *term*$^+$ "]"]
		[**where** *equation* (**and** *equation*)*]
toplevel	::=	**interpretation** *name* ("<" | "⊆") *expr proof*
	|	**interpretation** [*attr-name* "**:**"] *expr insts proof*
	|	**interpret** [*attr-name* "**:**"] *expr insts proof*

Diagnostics

toplevel	::=	**print_locale** ["**!**"] *locale*
	|	**print_locales**

Table 4: Syntax of Locale Commands.

References

[1] C. Ballarin. Interpretation of locales in Isabelle: Managing dependencies between locales. Technical Report TUM-I0607, Technische Universität München, 2006.

[2] C. Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In J. M. Borwein and W. M. Farmer, editors, *Mathematical knowledge management, MKM 2006, Wokingham, UK*, LNCS 4108, pages 31–43. Springer, 2006.

[3] F. Haftmann and M. Wenzel. Constructive type classes in Isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs, TYPES 2006, Nottingham, UK*, LNCS 4502, pages 160–174. Springer, 2007.

[4] N. Jacobson. *Basic Algebra*, volume I. Freeman, 2nd edition, 1985.

[5] F. Kammüller, M. Wenzel, and L. C. Paulson. Locales: A sectioning concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics: TPHOLs'99, Nice, France*, LNCS 1690, pages 149–165. Springer, 1999.

[6] M. Wenzel. The Isabelle/Isar reference manual. Part of the Isabelle distribution, http://isabelle.in.tum.de/doc/isar-ref.pdf.