Tutorial to Locales and Locale Interpretation

2nd revision, for Isabelle 2009

Clemens Ballarin

Abstract

Locales are Isabelle's mechanism for dealing 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. The second revision accommodates changes introduced by the locales reimplementation for Isabelle 2009. Most notably, in complex specifications (*locale expressions*) renaming has been generalised to instantiation.

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` =

definition	definition through an equation
inductive	inductive definition
primrec	primitive recursion
fun, **function**	general recursion
abbreviation	syntactic abbreviation
theorem, etc.	theorem statement with proof
theorems, etc.	redeclaration of theorems
text, etc.	document markup

Table 1: Isar commands that accept a target.

```
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"
```

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 transformations 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.

This style of working is illustrated in the block below, where notions of infimum and supremum for partial orders are introduced, together with theorems.

```
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
```

Two commands are provided to inspect locales: **print_locales** lists the names of all locales of the current theory; **print_locale** n prints the pa-

rameters 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. The grammer is complete with the exception of additional context elements not discussed here. 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. Import is a separate entity in the locale declaration. If present, it precedes the context elements.

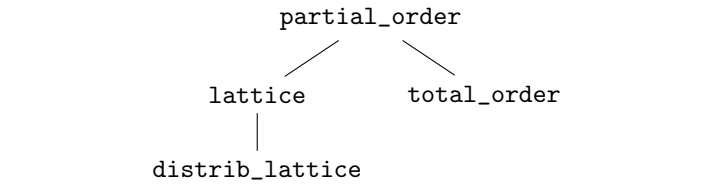
```
locale lattice = partial_order +
  assumes ex_inf: "∃inf. is_inf x y inf"
    and ex_sup: "∃sup. is_sup x y sup"
begin
```

These assumptions refer to the predicates for infimum and supremum defined in `partial_order`. We may now introduce the notions of meet and join.

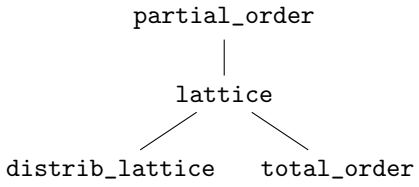
```
definition
  meet (infixl "⊓" 70) where "x ⊓ y = (THE inf. is_inf x y inf)"

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.

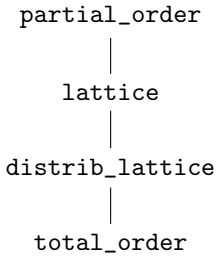
```
locale total_order = partial_order +
  assumes total: "x ⊑ y ∨ y ⊑ x"

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

locale distrib_lattice = lattice +
  assumes meet_distr: "x ⊓ (y ⊔ z) = x ⊓ y ⊔ x ⊓ z"

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



(a) Declared hierachy



(b) Total orders are lattices



(c) Total orders are distributive lattices

Figure 1: Hierarchy of Lattice Locales.

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

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 **sublocale** command.

sublocale `total_order` $\subseteq$ `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.

sublocale `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 **sublocale** $l_1 \subseteq l_2$ 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 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*. 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 qualifier*, here `nat`.
The command creates the goal[1]

```
1. partial_order op ≤
```

which can be shown easily:

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
```

The interpretation qualifier, `nat` in the example, is applied to all names processed by the interpretation. If a qualifer is given in the **interpretation** command, its use is mandatory when referencing the name. For example, the above theorem cannot be referred to simply by `trans`. This prevents unwanted hiding of theorems.

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 the natural strict order for `nat`. Interpretation allows to map concepts introduced by definitions in locales to the corresponding concepts of the theory.

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

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

```
interpretation nat: partial_order "op ≤ :: [nat, nat] ⇒ bool"
  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 as 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 nat_less_eq: "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 preceding fact and proved with ".". It 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 "partial_order.less op ≤ (x::nat) y = (x < y)"
    and nat_meet_eq: "lattice.meet op ≤ (x::nat) y = min x y"
    and nat_join_eq: "lattice.join op ≤ (x::nat) y = max x y"
proof -
  show lattice: "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+
```

For the first of the equations, we refer to the theorem shown in the previous interpretation.

```
  show "partial_order.less op ≤ (x::nat) y = (x < y)"
    by (rule nat_less_eq)
```

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

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 ≤ on the natural numbers.

```
  from lattice 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
```

Next follows that ≤ is a total order.

```
interpretation nat: total_order "op ≤ :: nat ⇒ nat ⇒ bool"
  where "partial_order.less op ≤ (x::nat) y = (x < y)"
    and "lattice.meet op ≤ (x::nat) y = min x y"
    and "lattice.join op ≤ (x::nat) y = max x y"
proof -
  show "total_order (op ≤ :: nat ⇒ nat ⇒ bool)"
    by unfold_locales arith
qed (rule nat_less_eq nat_meet_eq nat_join_eq)+
```

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 only necessary for mapping the definitions to the right operators on `nat`.

```
interpretation nat: distrib_lattice "op ≤ :: nat ⇒ nat ⇒ bool"
  where "partial_order.less op ≤ (x::nat) y = (x < y)"
    and "lattice.meet op ≤ (x::nat) y = min x y"
    and "lattice.join op ≤ (x::nat) y = max x y"
  by unfold_locales [1] (rule nat_less_eq nat_meet_eq nat_join_eq)+
```

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

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 nat_dvd_less_eq:
    "partial_order.less op dvd (x::nat) y = (x dvd y ∧ x ≠ y)"
```
⟨*proof*⟩

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

```
interpretation nat_dvd: lattice "op dvd :: nat ⇒ nat ⇒ bool"
  where "partial_order.less op dvd (x::nat) y = (x dvd y ∧ x ≠ y)"
    and nat_dvd_meet_eq: "lattice.meet op dvd = gcd"
    and nat_dvd_join_eq: "lattice.join op dvd = lcm"
```
⟨*proof*⟩

Equations `nat_dvd_meet_eq` and `nat_dvd_join_eq` are used in the main part the subsequent interpretation.

```
interpretation nat_dvd:
  distrib_lattice "op dvd :: nat ⇒ nat ⇒ bool"
  where "partial_order.less op dvd (x::nat) y = (x dvd y ∧ x ≠ y)"
    and "lattice.meet op dvd = gcd"
    and "lattice.join op dvd = lcm"
proof -
  show "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
qed (rule nat_dvd_less_eq nat_dvd_meet_eq nat_dvd_join_eq)+
```

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

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

```
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.

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*, which enables to combine locales and instantiate parameters. A locale expression is a sequence of locale *instances* followed by an optional **for** clause. Each instance consists of a locale reference, which may be preceded by a qualifer and succeeded by instantiations of the parameters of that locale. Instantiations may be either positional or through explicit mappings of parameters to arguments.

Using a locale expression, a locale for order preserving maps can be declared in the following way.

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

The second and third line contain the expression — two instances of the partial order locale where the parameter is instantiated to `le` and `le'`, respectively. The **for** clause consists of parameter declarations and is similar to the context element **fixes**. The notable difference is that the **for** clause is part of the expression, and only parameters defined in the expression may occur in its instances.

Instances define *morphisms* on locales. Their effect on the parameters is lifted to terms, propositions and theorems in the canonical way, and thus to the assumptions and conclusions of a locale. The assumption of a locale expression is the conjunction of the assumptions of the instances. The conclusions of a sequence of instances are obtained by appending the conclusions

of the instances in the order of the sequence.

The qualifiers in the expression are already a familiar concept from the **interpretation** command (Section 5.1). Here, they serve to distinguish names (in particular theorem names) for the two partial orders within the locale. Qualifiers in the **locale** command (and in **sublocale**) default to optional — that is, they need not occur in references to the qualified names. Here are examples of theorems in locale `order_preserving`:

```
le.less_le_trans: ⟦?x ⊏ ?y; ?y ⊑ ?z⟧ ⟹ ?x ⊏ ?z
hom_le: ?x ⊑ ?y ⟹ φ ?x ≼ φ ?y
```

The 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 operation associated with $\preceq$. This can be declared through an abbreviation.

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

Now the theorem is displayed nicely as `le'.less_le_trans`:

```
⟦?x ≺ ?y; ?y ≼ ?z⟧ ⟹ ?x ≺ ?z
```

Qualifiers not only apply to theorem names, but also to names introduced by definitions and abbreviations. For example, in `partial_order` the name `less` abbreviates `op ⊏`. Therefore, in `order_preserving` the qualified name `le.less` abbreviates `op ⊏` and `le'.less` abbreviates `op ≺`. Hence, the equation in the abbreviation above could have been written more concisely as `less' ≡ le'.less`.

Readers may find the declaration of locale `order_preserving` a little awkward, because the declaration and concrete syntax for `le` from `partial_order` are repeated in the declaration of `order_preserving`. Locale expressions provide a convenient short hand for this. A parameter in an instance is *untouched* if no instantiation term is provided for it. In positional instantiations, a parameter position may be skipped with an underscore, and it is allowed to give fewer instantiation terms than the instantiated locale's number of parameters. In named instantiations, instantiation pairs for certain parameters may simply be omitted. Untouched parameters are implicitly declared by the locale expression and with their concrete syntax. In the sequence of parameters, they appear before the parameters from the **for** clause.

The following locales illustrate this. A map φ is a lattice homomorphism if it preserves meet and join.

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

In this declaration, the first parameter of `lattice_hom`, `le`, is untouched and is then used to instantiate the second parameter. Its concrete syntax is preserved.

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 respective locales appear only once.

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:

```
sublocale 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 `hom_le`:

```
?x ⊑ ?y ⟹ φ ?x ≼ φ ?y
```

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

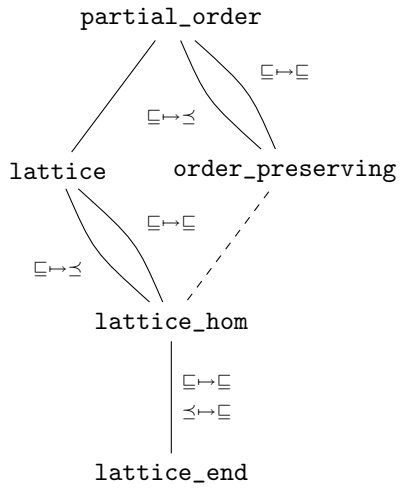


Figure 2: Hierarchy of Homomorphism Locales.

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 ML and 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*
qualifier	::=	*name* [“**?**” | “**!**”]

Context Elements

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

Locale Expressions

pos-insts	::=	(*term* | “_”)*
named-insts	::=	**where** *name* “**=**” *term* (**and** *name* “**=**” *term*)*
instance	::=	[*qualifier* “**:**”] *qualified-name* (*pos-insts* | *named-inst*)
expression	::=	*instance* (“**+**” *instance*)* [**for** *fixes* (**and** *fixes*)*]

Declaration of Locales

locale	::=	$element^+$
	|	*expression* [“**+**” $element^+$]
toplevel	::=	**locale** *name* [“**=**” *locale*]

Interpretation

equation	::=	[*attr-name* “**:**”] *prop*
equations	::=	**where** *equation* (**and** *equation*)*
toplevel	::=	**sublocale** *name* (“$<$” | “$\subseteq$”) *expression proof*
	|	**interpretation** *expression* [*equations*] *proof*
	|	**interpret** *expression 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.