Tutorial to Locales and Locale Interpretation*

Clemens Ballarin

Abstract

Locales are Isabelle's approach for dealing with parametric theories. They have been designed as a module system for a theorem prover that can adequately represent the complex inter-dependencies between structures found in abstract algebra, but have proven fruitful also in other applications — for example, software verification.

Both design and implementation of locales have evolved considerably since Kammüller did his initial experiments. Today, locales are a simple yet powerful extension of the Isar proof language. The present tutorial covers all major facilities of locales. It 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 the 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 can provide context specific configuration information for proof procedures and concrete syntax. From a logical perspective, locales are just contexts that have been made persistent. To the user, though, they provide powerful means for declaring and combining contexts, and for the reuse of theorems proved in these contexts.

*Published in L. Lambán, A. Romero, J. Rubio, editors, *Contribuciones Científicas en honor de Mirian Andrés.* Servicio de Publicaciones de la Universidad de La Rioja, Logroño, Spain, 2010. Reproduced by permission.

2 Simple Locales

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"
```

The parameter of this locale is `le`, which is a binary predicate with infix syntax ⊑. The parameter syntax is available in the subsequent assumptions, which are the familiar partial order axioms.

Isabelle recognises unbound names as free variables. In locale assumptions, these are implicitly universally quantified. That is, ⟦x ⊑ y; y ⊑ z⟧ ⟹ x ⊑ z in fact means ⋀x y z. ⟦x ⊑ y; y ⊑ z⟧ ⟹ x ⊑ z.

Two commands are provided to inspect locales: **print_locales** lists the names of all locales of the current theory; **print_locale** n prints the parameters and assumptions of locale n; the variation **print_locale!** n additionally outputs the conclusions that are stored in the locale. We may inspect the new locale by issuing **print_locale!** `partial_order`. The output is the following list of context elements.

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

This differs from the declaration. The assumptions have turned into conclusions, denoted by the keyword **notes**. Also, there is only one assumption, namely `partial_order (⊑)`. The locale declaration has introduced the predicate `partial_order` to the theory. This predicate is the *locale predicate*. Its definition may be inspected by issuing **thm** `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)
```

In our example, this is a unary predicate over the parameter of the locale. It is equivalent to the original assumptions, which have been turned into

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.

conclusions and are available as theorems in the context of the locale. The names and attributes from the locale declaration are associated to these theorems and are effective in the context of the locale.

Each conclusion has a *foundational theorem* as counterpart in the theory. Technically, this is simply the theorem composed of context and conclusion. For the transitivity theorem, this is `partial_order.trans`:

```
⟦partial_order ?le; ?le ?x ?y; ?le ?y ?z⟧ ⟹ ?le ?x ?z
```

2.1 Targets: Extending Locales

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. Isar provides 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)"
```

The strict order `less` with infix syntax ⊏ is defined in terms of the locale parameter `le` and the general equality of the object logic we work in. The definition generates a *foundational constant* `partial_order.less` with definition `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. Here, the abbreviation `less` is available for `partial_order.less le`, and it is printed and parsed as infix ⊏. Finally, the conclusion `less_def` is added to the locale:

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

The treatment of theorem statements is more straightforward. As an example, here is the derivation of a transitivity law for the strict order relation.

```
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, conclusions of the locale may be used like theorems. 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.

2.2 Context Blocks

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 about their uniqueness.

```
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
```

The syntax of the locale commands discussed in this tutorial is shown in Table 3. The grammar is complete with the exception of the context elements

constrains and **defines**, which are provided for backward compatibility. See the Isabelle/Isar Reference Manual [8] 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 kind of inheritance is achieved through *import* of locales. The import part of a locale declaration, if present, precedes the context elements. Here is an example, where partial orders are extended to lattices.

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

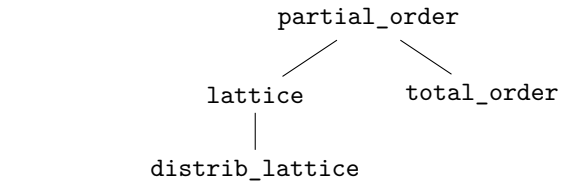
These assumptions refer to the predicates for infimum and supremum defined for `partial_order` in the previous section. We now introduce the notions of meet and join, together with an example theorem.

```
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)"

lemma meet_left: "x ⊓ y ⊑ x"
  ⟨proof⟩

end
```

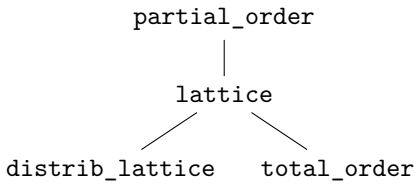
Locales for total orders and distributive lattices follow to establish a sufficiently rich landscape of locales for further examples in this tutorial.

```
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⟩
```

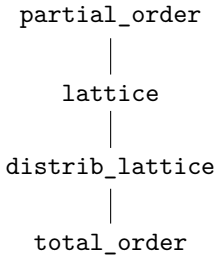
The locale hierarchy obtained through these declarations is shown in Figure 1(a).



(a) Declared hierarchy



(b) Total orders are lattices



(c) Total orders are distributive lattices

Figure 1: Hierarchy of Lattice Locales.

4 Changing the Locale Hierarchy

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*. Locales generalise interpretation from theorems to conclusions, enabling the reuse of definitions and other constructs that are not part of the specifications of the locales.

The first form of interpretation we will consider in this tutorial is provided by the **sublocale** command. It enables to modify the import hierarchy to reflect the *logical* relation between locales.

Consider the locale hierarchy from Figure 1(a). Total orders are lattices, although this is not reflected here, and definitions, theorems and other conclusions from `lattice` are not available in `total_order`. To obtain the situation in Figure 1(b), it is sufficient to add the conclusions of the latter locale to the former. The **sublocale** command does exactly this. The declaration **sublocale** $l_1 \subseteq l_2$ causes locale l_2 to be *interpreted* in the context of l_1. This means that all conclusions of l_2 are made available in l_1.

Of course, the change of hierarchy must be supported by a theorem that reflects, in our example, that total orders are indeed lattices. Therefore the **sublocale** command generates a goal, which must be discharged by the user. This is illustrated in the following paragraphs. First the sublocale relation

is stated.

```
sublocale total_order ⊆ lattice
```

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

```
1. lattice (⊑)
```

must be shown. Now the locale predicate needs to be unfolded — for example, using its definition or by introduction rules provided by the locale package. For automation, the locale package provides the methods `intro_locales` and `unfold_locales`. 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`, which need to be shown, hence `unfold_locales` is appropriate.

```
proof unfold_locales
```

Since the fact that both lattices and total orders are partial orders is already reflected in the locale hierarchy, the assumptions of `partial_order` are discharged automatically, and only the assumptions introduced in `lattice` remain as subgoals

```
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 obtained by constructing an infimum, whose existence is implied by totality.

```
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.

Similarly, we may establish that total orders are distributive lattices with a second **sublocale** statement.

```
sublocale total_order ⊆ distrib_lattice
  ⟨proof⟩
```

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

Locale interpretation is *dynamic*. The statement **sublocale** $l_1 \subseteq l_2$ will not just add the current conclusions of l_2 to l_1. Rather the dependency is

stored, and conclusions that will be added to l_2 in future are automatically propagated to l_1. The sublocale relation is transitive — that is, propagation takes effect along chains of sublocales. Even cycles in the sublocale relation are supported, as long as these cycles do not lead to infinite chains. Details are discussed in the technical report [2]. See also Section 7.1 of this tutorial.

5 Use of Locales in Theories and Proofs

Locales can be interpreted in the contexts of theories and structured proofs. These interpretations are dynamic, too. Conclusions of locales will be propagated to the current theory or the current proof context.[1] The focus of this section is on interpretation in theories, but we will also encounter interpretations in proofs, in Section 5.3.

As an example, consider the type of integers `int`. The relation `(≤)` is a total order over `int`. We start with the interpretation that `(≤)` is a partial order. The facilities of the interpretation command are explored gradually in three versions.

5.1 First Version: Replacement of Parameters Only

The command **interpretation** is for the interpretation of locale in theories. In the following example, the parameter of locale `partial_order` is replaced by `(≤)` and the locale instance is interpreted in the current theory.

```
interpretation int: partial_order "(≤) :: int ⇒ int ⇒ bool"
```

The argument of the command is a simple *locale expression* consisting of the name of the interpreted locale, which is preceded by the qualifier `int:` and succeeded by a white-space-separated list of terms, which provide a full instantiation of the locale parameters. The parameters are referred to by order of declaration, which is also the order in which **print_locale** outputs them. The locale has only a single parameter, hence the list of instantiation terms is a singleton.

The command creates the goal

```
1. partial_order (≤)
```

which can be shown easily:

```
  by unfold_locales auto
```

The effect of the command is that instances of all conclusions of the locale are available in the theory, where names are prefixed by the qualifier.

[1] Strictly speaking, only interpretation in theories is dynamic since it is not possible to change locales or the locale hierarchy from within a proof.

For example, transitivity for `int` is named `int.trans` and is the following theorem:

```
⟦?x ≤ ?y; ?y ≤ ?z⟧ ⟹ ?x ≤ ?z
```

It is not possible to reference this theorem simply as `trans`. This prevents unwanted hiding of existing theorems of the theory by an interpretation.

5.2 Second Version: Replacement of Definitions

Not only does the above interpretation qualify theorem names. The prefix `int` is applied to all names introduced in locale conclusions including names introduced in definitions. The qualified name `int.less` is short for the interpretation of the definition, which is `partial_order.less (≤)`. Qualified name and expanded form may be used almost interchangeably.[2] The former is preferred on output, as for example in the theorem `int.less_le_trans`:

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

Both notations for the strict order are not satisfactory. The constant `(<)` is the strict order for `int`. In order to allow for the desired replacement, interpretation accepts *equations* in addition to the parameter instantiation. These follow the locale expression and are indicated with the keyword **rewrites**. This is the revised interpretation:

```
interpretation int: partial_order "(≤) :: [int, int] ⇒ bool"
  rewrites "int.less x y = (x < y)"
proof -
```

The goals are now:

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

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

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

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

```
  show "partial_order.less (≤) x y = (x < y)"
    unfolding partial_order.less_def [OF ‹partial_order (≤)›]
    by auto
qed
```

Note that the above proof is not in the context of the interpreted locale. Hence, the premise of `partial_order.less_def` is discharged manually with `OF`.

[2] Since `(≤)` is polymorphic, for `partial_order.less (≤)` a more general type will be inferred than for `int.less` which is over type `int`.

5.3 Third Version: Local Interpretation

In the above example, the fact that `(≤)` is a partial order for the integers was used in the second goal to discharge the premise in the definition of `(⊏)`. In general, proofs of the equations not only may involve definitions from the interpreted locale but arbitrarily complex arguments in the context of the locale. Therefore it would be convenient to have the interpreted locale conclusions temporarily available in the proof. This can be achieved by a locale interpretation in the proof body. The command for local interpretations is **interpret**. We repeat the example from the previous section to illustrate this.

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

The inner interpretation is immediate from the preceding fact and proved by assumption (Isar short hand "."). It enriches the local proof context by the theorems also obtained in the interpretation from Section 5.1, and `int.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 succeeding **next** or **qed** statement.

5.4 Further Interpretations

Further interpretations are necessary for the other locales. In `lattice` the operations ⊓ and ⊔ are substituted by `min` and `max`. The entire proof for the interpretation is reproduced to give an example of a more elaborate interpretation proof. Note that the equations are named so they can be used in a later example.

```
interpretation int: lattice "(≤) :: int ⇒ int ⇒ bool"
  rewrites int_min_eq: "int.meet x y = min x y"
    and int_max_eq: "int.join x y = max x y"
proof -
  show "lattice ((≤) :: int ⇒ int ⇒ bool)"
```

We have already shown that this is a partial order,

```
    apply unfold_locales
```

hence only the lattice axioms remain to be shown.

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

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

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

By `is_inf` and `is_sup`,

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

the goals are transformed to these statements:

```
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)
```

This is Presburger arithmetic, which can be solved by the method `arith`.

```
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 int: lattice "(≤) :: int ⇒ int ⇒ bool" .
  show "int.meet x y = min x y"
    by (bestsimp simp: int.meet_def int.is_inf_def)
  show "int.join x y = max x y"
    by (bestsimp simp: int.join_def int.is_sup_def)
qed
```

Next follows that `(≤)` is a total order, again for the integers.

```
interpretation int: total_order "(≤) :: int ⇒ int ⇒ bool"
  by unfold_locales arith
```

Theorems that are available in the theory at this point are shown in Table 2. Two points are worth noting:

- Locale `distrib_lattice` was also interpreted. Since the locale hierarchy reflects that total orders are distributive lattices, the interpretation of the latter was inserted automatically with the interpretation

of the former. In general, interpretation traverses the locale hierarchy upwards and interprets all encountered locales, regardless whether imported or proved via the **sublocale** command. Existing interpretations are skipped avoiding duplicate work.

- The predicate `(<)` appears in theorem `int.less_total` although an equation for the replacement of `(⊏)` was only given in the interpretation of `partial_order`. The interpretation equations are pushed downwards the hierarchy for related interpretations — that is, for interpretations that share the instances of parameters they have in common.

The interpretations for a locale n within the current theory may be inspected with **print_interps** n. This prints the list of instances of n, for which interpretations exist. For example, **print_interps** `partial_order` outputs the following:

```
int : partial_order "(≤)"
```

Of course, there is only one interpretation. The interpretation qualifier on the left is mandatory. Qualifiers can either be *mandatory* or *optional*. Optional qualifiers are designated by "?". Mandatory qualifiers must occur in name references while optional ones need not. Mandatory qualifiers prevent accidental hiding of names, while optional qualifiers can be more convenient to use. The default is mandatory.

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.

A locale for order preserving maps requires three parameters: `le` (**infixl** $\sqsubseteq$) and `le'` (**infixl** $\preceq$) for the orders and φ for the map.

In order to reuse the existing locale for partial orders, which has the single parameter `le`, it must be imported twice, once mapping its parameter to `le` from the new locale and once to `le'`. This can be achieved with a compound locale expression.

In general, a locale expression is a sequence of *locale instances* separated by "+" and followed by a **for** clause. An instance has the following format:

qualifier **:** *locale-name parameter-instantiation*

We have already seen locale instances as arguments to **interpretation** in Section 5. As before, the qualifier serves to disambiguate names from different instances of the same locale, and unless designated with a "?" it must occur in name references.

Since the parameters `le` and `le'` are to be partial orders, our locale for order preserving maps will import the these instances:

```
le: partial_order le
le': partial_order le'
```

For matter of convenience we choose to name parameter names and qualifiers alike. This is an arbitrary decision. Technically, qualifiers and parameters are unrelated.

Having determined the instances, let us turn to the **for** clause. It serves to declare locale parameters in the same way as the context element **fixes** does. Context elements can only occur after the import section, and therefore the parameters referred to in the instances must be declared in the **for** clause. The **for** clause is also where the syntax of these parameters is declared.

Two context elements for the map parameter φ and the assumptions that it is order preserving complete the locale declaration.

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

Here are examples of theorems that are available in the locale:

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

  ⟦le'.less ?x ?y; ?y ≼ ?z⟧ ⟹ le'.less ?x ?z
```

While there is infix syntax for the strict operation associated with `(⊑)`, there is none for the strict version of `(≼)`. The syntax $\sqsubset$ for `less` is only available for the original instance it was declared for. We may introduce infix syntax for `le'.less` with the following declaration:

```
notation (in order_preserving) le'.less (infixl ‹≺› 50)
```

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

```
⟦?x ≺ ?y; ?y ≼ ?z⟧ ⟹ ?x ≺ ?z
```

There are short notations for locale expressions. These are discussed in the following.

6.1 Default Instantiations

It is possible to omit parameter instantiations. The instantiation then defaults to the name of the parameter itself. For example, the locale expression `partial_order` is short for `partial_order le`, since the locale's single parameter is `le`. We took advantage of this in the **sublocale** declarations of Section 4.

6.2 Implicit Parameters

In a locale expression that occurs within a locale declaration, omitted parameters additionally extend the (possibly empty) **for** clause.

The **for** clause is a general construct of Isabelle/Isar to mark names occurring in the preceding declaration as "arbitrary but fixed". This is necessary for example, if the name is already bound in a surrounding context. In a locale expression, names occurring in parameter instantiations should be bound by a **for** clause whenever these names are not introduced elsewhere in the context — for example, on the left hand side of a **sublocale** declaration. There is an exception to this rule in locale declarations, where the **for** clause serves to declare locale parameters. Here, locale parameters for which no parameter instantiation is given are implicitly added, with their mixfix syntax, at the beginning of the **for** clause. For example, in a locale declaration, the expression `partial_order` is short for

```
partial_order le for le (infixl "⊑" 50).
```

This short hand was used in the locale declarations throughout Section 3.

The following locale declarations provide more examples. 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)"
```

The parameter instantiation in the first instance of `lattice` is omitted. This causes the parameter `le` to be added to the **for** clause, and the locale has parameters `le`, `le'` and, of course, φ.

Before turning to the second example, we complete the locale by providing infix syntax for the meet and join operations of the second lattice.

```
context lattice_hom
begin
notation le'.meet (infixl ‹⊓''› 50)
notation le'.join (infixl ‹⊔''› 50)
end
```

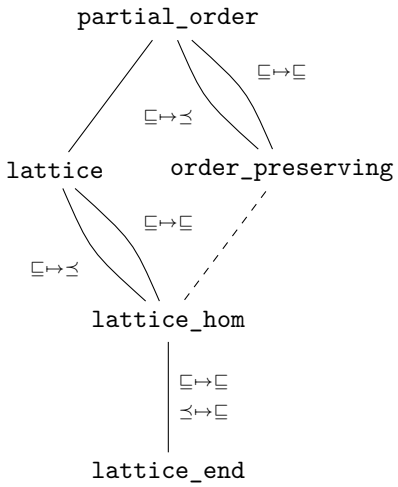


Figure 2: Hierarchy of Homomorphism Locales.

The next example makes radical use of the short hand facilities. A homomorphism is an endomorphism if both orders coincide.

locale `lattice_end = lattice_hom _ le`

The notation `_` enables to omit a parameter in a positional instantiation. The omitted parameter, `le` becomes the parameter of the declared locale and is, in the following position, used to instantiate the second parameter of `lattice_hom`. The effect is that of identifying the first in second parameter of the homomorphism locale.

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. Parameter instantiations are indicated by $\sqsubseteq \mapsto \preceq$ etc. By looking at the inheritance diagram it would seem that two identical copies of each of the locales `partial_order` and `lattice` are imported by `lattice_end`. This is not the case! Inheritance paths with identical morphisms are automatically 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`

This theorem will be useful in the following section.

7 Conditional Interpretation

There are situations where an interpretation is not possible in the general case since the desired property is only valid if certain conditions are fulfilled. Take, for example, the function `λi. n * i` that scales its argument by a constant factor. This function is order preserving (and even a lattice endomorphism) with respect to `(≤)` provided `n ≥ 0`.

It is not possible to express this using a global interpretation, because it is in general unspecified whether `n` is non-negative, but one may make an interpretation in an inner context of a proof where full information is available. This is not fully satisfactory either, since potentially interpretations may be required to make interpretations in many contexts. What is required is an interpretation that depends on the condition — and this can be done with the **sublocale** command. For this purpose, we introduce a locale for the condition.

```
locale non_negative =
  fixes n :: int
  assumes non_neg: "0 ≤ n"
```

It is again convenient to make the interpretation in an incremental fashion, first for order preserving maps, then for lattice endomorphisms.

```
sublocale non_negative ⊆
    order_preserving "(≤)" "(≤)" "λi. n * i"
  ⟨proof⟩
```

While the proof of the previous interpretation is straightforward from monotonicity lemmas for `(*)`, the second proof follows a useful pattern.

```
sublocale non_negative ⊆ lattice_end "(≤)" "λi. n * i"
proof (unfold_locales, unfold int_min_eq int_max_eq)
```

Unfolding the locale predicates *and* the interpretation equations immediately yields two subgoals that reflect the core conjecture.

```
1. ⋀x y. n * min x y = min (n * x) (n * y)
2. ⋀x y. n * max x y = max (n * x) (n * y)
```

It is now necessary to show, in the context of `non_negative`, that multiplication by `n` commutes with `min` and `max`.

```
qed (auto simp: hom_le)
```

The lemma `hom_le` simplifies a proof that would have otherwise been lengthy and we may consider making it a default rule for the simplifier:

```
lemmas (in order_preserving) hom_le [simp]
```

7.1 Avoiding Infinite Chains of Interpretations

Similar situations arise frequently in formalisations of abstract algebra where it is desirable to express that certain constructions preserve certain properties. For example, polynomials over rings are rings, or — an example from the domain where the illustrations of this tutorial are taken from — a partial order may be obtained for a function space by point-wise lifting of the partial order of the co-domain. This corresponds to the following interpretation:

sublocale `partial_order ⊆ f: partial_order "λf g. ∀x. f x ⊑ g x"`
oops

Unfortunately this is a cyclic interpretation that leads to an infinite chain, namely

```
partial_order ⊆ partial_order (λf g. ∀x. f x ⊑ g x) ⊆
  partial_order (λf g. ∀x y. f x y ⊑ g x y) ⊆  ...
```

and the interpretation is rejected.

Instead it is necessary to declare a locale that is logically equivalent to `partial_order` but serves to collect facts about functions spaces where the co-domain is a partial order, and to make the interpretation in its context:

locale `fun_partial_order = partial_order`

sublocale `fun_partial_order ⊆`
`f: partial_order "λf g. ∀x. f x ⊑ g x"`
⟨*proof*⟩

It is quite common in abstract algebra that such a construction maps a hierarchy of algebraic structures (or specifications) to a related hierarchy. By means of the same lifting, a function space is a lattice if its co-domain is a lattice:

locale `fun_lattice = fun_partial_order + lattice`

sublocale `fun_lattice ⊆ f: lattice "λf g. ∀x. f x ⊑ g x"`
⟨*proof*⟩

8 Further Reading

More information on locales and their interpretation is available. For the locale hierarchy of import and interpretation dependencies see [2]; interpretations in theories and proofs are covered in [3]. In the latter, I show how interpretation in proofs enables to reason about families of algebraic structures, which cannot be expressed with locales directly.

Haftmann and Wenzel [4] 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 locales reimplementation for Isabelle 2009 provides, among other improvements, a clean integration with Isabelle/Isar's local theory mechanisms, which are described in another paper by Haftmann and Wenzel [5].

The original work of Kammüller on locales [7] may be of interest from a historical perspective. My previous report on locales and locale expressions [1] describes a simpler form of expressions than available now and is outdated. The mathematical background on orders and lattices is taken from Jacobson's textbook on algebra [6, Chapter 8].

The sources of this tutorial, which include all proofs, are available with the Isabelle distribution at https://isabelle.in.tum.de.

Revision History. The present fourth revision of the tutorial accommodates minor updates to the syntax of the locale commands and the handling of notation under morphisms introduced with Isabelle 2016. For the third revision of the tutorial, which corresponds to the published version, much of the explanatory text was rewritten. Inheritance of interpretation equations was made available with Isabelle 2009-1. The second revision accommodates changes introduced by the locales reimplementation for Isabelle 2009. Most notably locale expressions were generalised from renaming to instantiation.

Acknowledgements. Alexander Krauss, Tobias Nipkow, Randy Pollack, Andreas Schropp, Christian Sternagel and Makarius Wenzel have made useful comments on earlier versions of this document. The section on conditional interpretation was inspired by a number of e-mail enquiries the author received from locale users, and which suggested that this use case is important enough to deserve explicit explanation. The term *conditional interpretation* is due to Larry Paulson.

References

[1] C. Ballarin. Locales and locale expressions in Isabelle/Isar. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs, TYPES 2003, Torino, Italy*, LNCS 3085, pages 34–50. Springer, 2004.

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

Diagnostics

toplevel	::=	**print_locales**
	|	**print_locale** [“**!**”] *name*
	|	**print_interps** *name*

Table 3: Syntax of Locale Commands.

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

[3] 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.

[4] 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.

[5] F. Haftmann and M. Wenzel. Local theory specifications in Isabelle/Isar. In S. Berardi, F. Damiani, and U. de Liguoro, editors, *Types for Proofs and Programs, TYPES 2008, Torino, Italy*, LNCS 5497, pages 153–168. Springer, 2009.

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

[7] 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.

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