J Autom Reasoning (2014) 52:123–153
DOI 10.1007/s10817-013-9284-7

Locales: A Module System for Mathematical Theories

Clemens Ballarin

Received: 30 November 2012 / Accepted: 18 March 2013 / Published online: 10 April 2013

Abstract Locales are a module system for managing theory hierarchies in a theorem prover through theory interpretation. They are available for the theorem prover Isabelle. In this paper, their semantics is defined in terms of local theories and morphisms. Locales aim at providing flexible means of extension and reuse. Theory modules (which are called locales) may be extended by definitions and theorems. Interpretation to Isabelle's global theories and proof contexts is possible via morphisms. Even the locale hierarchy may be changed if declared relations between locales do not adequately reflect logical relations, which are implied by the locales' specifications. By discussing their design and relating it to more commonly known structuring mechanisms of programming languages and provers, locales are made accessible to a wider audience beyond the users of Isabelle. The discussed mechanisms include ML-style functors, type classes and mixins (the latter are found in modern object-oriented languages).

Keywords Theorem prover • Module system • Theory hierarchy • Theory interpretation • Isabelle

1 Introduction

The developers of the computer algebra system Axiom pioneered implementing complex hierarchies of algebraic structures in a computer language. The user manual [13] shows a graph of 45 interconnected algebraic structures at 15 levels in the basic algebra hierarchy all of which are implemented as types in that system. Standard libraries of programming languages usually have many more classes, but hierarchies

C. Ballarin (✉)
Stephanienstr. 61, 76133 Karlsruhe, Germany
e-mail: ballarin@in.tum.de

tend to be less deep. (For example, the Java 6 Standard Edition class library contains almost 3800 classes at only eight levels [20].) It is evident that such libraries are only maintainable if they can be extended easily.

Locales provide flexible means of building and using hierarchic developments of theory modules and were designed so that abstract algebraic theories could be represented in an adequate fashion. Today, locales are used in many domains. Examples include proofs in graph theory [18], set theory [21] and state space management in programming language semantics [22]. Also Isabelle's class package uses locales [10].

Locales provide some of the automation that makes Isabelle's type classes attractive, but they are not restricted to a single carrier type. Theorem reuse is rigourously based on interpretation (often called *theory interpretation* in the context of provers), and locales can deal with important forms of circular theory module dependencies.

A re-implementation of locales was released with Isabelle 2009. Users have mainly benefited from more powerful *locale expressions*, which provide flexible means for composing theory hierarchies. In particular, locale expressions now admit parameter instantiation, while previously only renaming was possible. This is useful, for example, for expressing duality. *Local theories* [11], which became available in Isabelle at that time, helped clarify the design and reduce the code size of the locales implementation to about two thirds.

The purpose of the present paper is to provide an operational semantics of locales relative to local theories, and to outline the design goals. Relations to other structuring mechanisms, both for formal theory developments and programming languages, are established. Users of locales should also consult the tutorial [5].

The following section contains formalisations of algebraic structures that illustrate important features of locales and serve as a base for examples in the subsequent sections. Local theories and other devices necessary to define locales are introduced in Section 3. Section 4 is the core of the paper. Locales and the user-level operations are defined. In Section 5 relations to ML-style modules and other means of reuse in provers and programming languages are discussed.

2 Example—the Lattice of Subgroups

The formalisation presented in this section serves to introduce locales by example. It involves two algebraic structures, lattice and group, who are related by identifying the lattice induced by the subgroup relation.

Isabelle's notation for formulas is close to what is common in mathematics. Both $\bigwedge$ and $\forall$ denote universal quantification, and $\Longrightarrow$ and $\longrightarrow$ denote implication.[1] Double square brackets abbreviate nested implication: $[\![\ \mathtt{A}_1;\ \ldots;\ \mathtt{A}_n\ ]\!] \Longrightarrow \mathtt{B}$ means $\mathtt{A}_1 \Longrightarrow \ldots \Longrightarrow \mathtt{A}_n \Longrightarrow \mathtt{B}$. The double arrow $\longleftrightarrow$ is an alternative notation for equality on Booleans, and with precedence lower than that of the logical connectives $\wedge$, $\vee$ etc.

[1]The differences between Isabelle's meta-logical connectives $\bigwedge$ and $\Longrightarrow$ and the connectives $\forall$ and $\longrightarrow$ of the HOL object-logic are not relevant for understanding the examples.

2.1 Algebraic Structures

An abstract algebraic structure like group or lattice is declared with the **locale** command. Our example is based on lattices and we start with the formalisation of partial orders.

```
locale partial_order =
  fixes S and le (infixl "⊑" 50)
  assumes refl: "x ∈ S ⟹ x ⊑ x"
    and antisym: "⟦ x ⊑ y; y ⊑ x; x ∈ S; y ∈ S ⟧ ⟹ x = y"
    and trans: "⟦ x ⊑ y; y ⊑ z; x ∈ S; y ∈ S; z ∈ S ⟧ ⟹ x ⊑ z"
```

The carrier set `S` and the order relation `le` (with concrete syntax ⊑) are the parameters (**fixes**) of the specification, which consists of the usual axioms (**assumes**).

Infima do not necessarily exist in partial orders, but it is useful to have a notion for the concept already here. The **context** command enables to focus on a locale and to extend it—in this case, by a definition.

```
context partial_order begin
  definition is_inf where "is_inf x y w ⟷ w ⊑ x ∧ w ⊑ y ∧
    (∀z ∈ S. z ⊑ x ∧ z ⊑ y ⟶ z ⊑ w) ∧ x ∈ S ∧ y ∈ S ∧ w ∈ S"
end
```

That is, `is_inf` is a predicate, and `is_inf x y w` means that `w` is the infimum of `x` and `y` in the carrier set. A semilattice is a partial order where infima for any two elements exist.

```
locale semilattice =
  partial_order "S" "le" for S and le (infixl "⊑" 50) +
  assumes existence: "⟦ x ∈ S; y ∈ S ⟧ ⟹ ∃inf. is_inf x y inf"
```

This declaration consists of a *locale expression* (the second line), and an additional axiom. A locale expression contains one or several *locale instances* and an optional **for** clause. Here the expression describes an instance of `partial_order`, which is imported. While in the previous locale the parameters were declared in a **fixes** clause, here they have moved to the **for** clause so that they can be referred to in the instance of the imported locale.

Within `semilattice` we can now define an operation for the infimum, by means of the definite selection operator,[2] and elaborate its properties, for example associativity:

```
context semilattice begin
  definition meet (infixl "⊓" 70)
    where "op ⊓ = (λx ∈ S. λy ∈ S. THE inf. is_inf x y inf)"
  lemma assoc: "(x ⊓ y) ⊓ z = x ⊓ (y ⊓ z)"  ⟨proof⟩

   ⋮

end
```

[2]Since HOL is total, bounded λ-abstraction denotes a function that maps all arguments outside the domain to a fixed but unknown value, about which nothing can be proved. Likewise for the definite selection operator `THE` if the described element does not exist or is not unique.

2.2 Duality

It is immediate from the axioms that the inverse relation of a partial order is again a partial order. With locales, this can be expressed with the **sublocale** command:

```
sublocale partial_order ⊆ dual!: partial_order "S" "λx y. y ⊑ x"
⟨proof⟩
```

The declaration consists of a locale (to the left of ⊆) called the *target* and a locale expression. Based on the provided proof, the target locale is enriched by definitions and theorems of the locale instance given in the expression. The qualifier `dual` identifies these dual versions. For example, `dual.is_inf` is now recognised as the dual of `is_inf`. The exclamation mark asserts that the qualifier is required when referencing names in the dual instance. This prevents accidental hiding of names of the original locale. In contrast to the expression in the locale declaration above, here a **for** clause is not needed: `S` and ⊑ are parameters of the target.

We may now introduce syntax for the supremum predicate.

```
context partial_order begin
  abbreviation is_sup where "is_sup ≡ dual.is_inf"
end
```

Its definition is already available through the sublocale declaration.

A lattice consists of a lower semilattice and a dual upper semilattice. In contrast to the previous situation, where duality only implied new definitions and theorems, we now need to obtain a new axiom, namely the existence of the supremum. This is achieved by declaring a locale that imports two instances of `semilattice`.

```
locale lattice =
  semilattice "S" "le" + dual!: semilattice "S" "λx y. y ⊑ x"
  for S and le (infixl "⊑" 50)
```

Like for `is_sup`, syntax for the supremum operation could now be declared.

2.3 A Concrete Instance

Interpretation facilitates reuse of definitions and theorems from locales in other contexts. Given a proof of an instance of the axioms within a context, the context is enriched by instances of the theorems. To illustrate this, we consider the power set of a set `X`, which is a lattice with respect to the subset relation.

The **interpretation** command interprets a locale in the context of Isabelle's global background theory. We proceed in two steps, first showing that the power set is partially ordered:

```
interpretation power!: partial_order "Pow X" "op ⊆" ⟨proof⟩
```

Since the base set `X` is arbitrary it is represented by a variable. The interpretation yields theorems qualified by `power`—for example, `power.trans`,

```
⟦x ⊆ y; y ⊆ z; x ∈ Pow X; y ∈ Pow X; z ∈ Pow X⟧ ⟹ x ⊆ z
```

and its dual `power.dual.trans`,

```
⟦y ⊆ x; z ⊆ y; x ∈ Pow X; y ∈ Pow X; z ∈ Pow X⟧ ⟹ z ⊆ x
```

The above interpretation merely instantiated the locale parameters. For `lattice` it is desirable to replace definitions in the locale by corresponding concepts from the target context. This is achieved by extending the interpretation.

```
interpretation power!: lattice "Pow X" "op ⊆"
  where "power.meet = (λA ∈ Pow X. λB ∈ Pow X. A ∩ B)"
    and "power.dual.meet = (λA ∈ Pow X. λB ∈ Pow X. A ∪ B)"

⟨proof⟩
```

The infimum is, of course, set intersection and its dual set union. In order to meet the definitions, the operations need to be restricted to the carrier set.

2.4 Interpretation in Generic Contexts

Interpretations occur naturally in the contexts of algebraic structures themselves. A well-known example is the lattice of subgroups of a group.

The carrier set of a group is closed under group operations. Since this notion is required for both the definition of groups and subgroups, we declare a locale for it.

```
locale closed =
  fixes G and mult (infixl "·" 70) and one ("1") and inv
  assumes mult_closed: "⟦ x ∈ G; y ∈ G ⟧ ⟹ x · y ∈ G"
    and one_closed: "1 ∈ G" and inv_closed: "x ∈ G ⟹ inv x ∈ G"
```

The locale declaration for the actual group definition imports this locale:

```
locale group = closed +
  assumes assoc: "⟦ x ∈ G; y ∈ G; z ∈ G ⟧ ⟹ (x · y) · z = x · (y · z)"
    and l_one: "x ∈ G ⟹ 1 · x = x"
    and l_inv: "x ∈ G ⟹ inv x · x = 1"
```

Here, the parameters of `closed` are not instantiated explicitly. A short-hand notation is used that makes the parameters of the instance *implicit parameters* of the declared locale. For details, see the locales tutorial [5].

A subgroup is a subset that is closed under group operations. This naturally leads to the set $\mathscr{G}$ of all subgroups of `G` and the closure ⟨`S`⟩ of a set `S`, which is the smallest subgroup of `G` that contains `S`. The subgroup relation itself is denoted by $\trianglelefteq$.

```
context group begin
  definition subgroup (infixl "⊴" 50)
    where "H ⊴ K ⟷ H ⊆ K ∧ closed H mult one inv"
  definition groups ("𝒢")
    where "𝒢 = {H. H ⊴ G}"
  definition closure ("⟨_⟩")
    where "⟨S⟩ = ⋂{H. S ⊆ H ∧ H ⊴ G}"
end
```

The definition of `subgroup` involves the predicate `closed`, which is generated by the declaration of the locale `closed` and abbreviates its specification.

We are now ready to show that $\mathcal{G}$ is a lattice. By means of the **sublocale** command, we provide an interpretation of `lattice` in the context of `group`, where the supremum operation is set intersection, and the infimum of two subgroups is the group generated by the union of their carrier sets:

```
sublocale group ⊆ sub!: lattice "𝒢" "op ⊴"
   where "sub.meet = (λK ∈ 𝒢. λL ∈ 𝒢. K ∩ L)"
     and "sub.dual.meet = (λK ∈ 𝒢. λL ∈ 𝒢. ⟨K ∪ L⟩)"
  ⟨proof⟩
```

The `group` context is now enriched by instances of lattice theorems qualified by `sub`—for example associativity of the join operation, `sub.dual.assoc`,

```
(λK∈𝒢. λL∈𝒢. ⟨K ∪ L⟩) ((λK∈𝒢. λL∈𝒢. ⟨K ∪ L⟩) x y) z =
(λK∈𝒢. λL∈𝒢. ⟨K ∪ L⟩) x ((λK∈𝒢. λL∈𝒢. ⟨K ∪ L⟩) y z)
```

3 Logic and Architecture Prerequisites

Locales provide means for building and working with large theory developments based on small components or *little theories* [8]. In Isabelle, these components are the local theories implemented by Haftmann and Wenzel [11] on top of the Isabelle/Isar framework. While locales are implemented in the local theories framework, conceptually they are not closely tied to Isabelle and Isar and could be implemented in other provers as well. Properties of the logic and facilities of a theorem prover architecture required by locales are defined in this section.

3.1 Logic Calculus

Locales require certain properties of the calculus implemented by the prover. These, along with notation, are introduced now.

Terms $s, t, \ldots$ and formulas $A, B, \ldots$ are distinguished, and formulas are terms.[3] Theorems are sequents $A_1, \ldots, A_n \vdash B$, where $n \geq 0$ and the hypotheses $A_1, \ldots, A_n$ and the proposition B are formulas. Variables are denoted by $x, y, \ldots$, sequences of variables, terms and formulas by $\overline{x}, \overline{y}, \ldots$ etc. Free variables in theorems are implicitly universally quantified, and theorems are closed under instantiation of variables:

$$\frac{\overline{A}[x] \vdash B[x]}{\overline{A}[t] \vdash B[t]}$$

Instantiation may be restricted—for example, to ensure type correctness if the logic is typed. There is an equivalence $\equiv$ of terms, where $s \equiv t$ is a formula, and implication

[3] Alternatively, terms and formulas may be distinct syntactic categories. Then all requirements for terms are duplicated for formulas.

and conjunction over formulas, denoted by $\Longrightarrow$ and $\wedge$ respectively. Theorems are closed under substitution of equivalent terms:

$$\frac{\overline{A} \vdash s \equiv t \qquad \overline{B}[s] \vdash C[s]}{\overline{A}, \overline{B}[t] \vdash C[t]}$$

3.2 Global Theories

Based on the calculus, the prover provides *global theories*. These are not parametric. Locales require global background theories to store deductive information and so-called *foundational constants*, which are the base for operations provided in local theories. Global theories implement the calculus, and they provide facilities for defining foundational constants and noting theorems. These are the operations on global theories (*thy*):

$$\begin{aligned} \mathsf{base} &: \mathit{thy} \\ \mathsf{def} &: \mathit{name} \to \mathit{term} \to \mathit{thy} \to \mathit{thy} \\ \mathsf{note} &: \mathit{name} \to \mathit{thm} \to \mathit{thy} \to \mathit{thy} \end{aligned}$$

The base theory base is the global theory that implements the logic calculus by providing its connectives and deductive machinery. It may contain additional axioms, operation symbols and definitions that are not part of the calculus. Examples are Isabelle's object logics HOL and ZF.

The prover must implement a mechanism for retrieving axioms and theorems from a theory, and, of course, for deriving new theorems. This is not made explicit here, and axioms and theorems in global theories are not distinguished. Constant and theorem names are qualified—that is, are of the form $q_1. \cdots .q_k.n$ in general.

The operation $\mathsf{def}\, c\ t$ extends a global theory by the foundational constant c along with its definition $\vdash c \equiv t$. For readability, we will write $\mathsf{def}\, c\ \overline{x} \equiv t$ instead of $\mathsf{def}\, c\ (\lambda \overline{x}.\ t)$.

The note operation models binding a theorem: $\mathsf{note}\ b\ (\vdash A)$ extends a theory by binding $\vdash A$ to b. Theorems in global theories may not have hypotheses. Whether derivability of theorems is checked depends on the prover, which—as is the case for Isabelle—may request and check a proof.

3.3 Local Theories

Local theories are parametric. Unlike global theories, whose sets of axioms are extensible (by definitions of foundational constants), the specification of a local theory is fixed. New operation symbols are simulated through abbreviations, and definitions are derived. These operations are available on local theories (*lthy*):

$$\begin{aligned} \mathsf{initialize} &: \mathit{vars} \to \mathit{form} \to \mathit{thy} \to \mathit{lthy} \\ \mathsf{promote} &: (\mathit{thy} \to \mathit{thy}) \to \mathit{lthy} \to \mathit{lthy} \\ \mathsf{abbreviate} &: \mathit{name} \to \mathit{term} \to \mathit{lthy} \to \mathit{lthy} \\ \mathsf{note} &: \mathit{name} \to \mathit{thm} \to \mathit{lthy} \to \mathit{lthy} \end{aligned}$$

A local theory may be obtained from a global theory by initialize $\overline{x}\ \ A[\overline{x}]$. It has the parameters $\overline{x}$ and the *specification* A, whose only free variables are the parameters.[4] The local theory inherits language and theorems of the global theory, which is called its *underlying theory*; promote f changes the underlying theory of a local theory via f. Only extensions of the underlying theory by def and note are allowed.

An operation in a local theory is introduced by adding an abbreviation: abbreviate $c\ t[\overline{x}]$ causes the term $t[\overline{x}]$ to be displayed as c when a term is printed, and c to be stored as $t[\overline{x}]$ in the internal representation when a term is read; $\overline{x}$ refers to the parameters of the local theory that is extended. Operation symbols introduced through abbreviate must be distinct from symbols inherited from the global theory. In contrast to global theories, theorems in local theories may have the local theory specification $A[\overline{x}]$ as a hypothesis: note $b\ \ (A[\overline{x}] \vdash B[\overline{x}])$.

3.4 Morphisms

Morphisms are a key ingredient to the composition of specifications (and their local theories) to hierarchies. They also define how local theories are interpreted in contexts. A morphism

$$\varphi = (\varphi_{\mathrm{n}}, \varphi_{\mathrm{t}}, \varphi_{\mathrm{th}})$$

consists of three mappings: φ_{n} is applied to operation and theorem names, φ_{t} maps terms, and φ_{th} transforms theorems. Application of a morphism φ to a name n, term t or theorem $\overline{A} \vdash B$ is denoted by $\varphi(n)$, $\varphi(t)$ and $\varphi(\overline{A} \vdash B)$, respectively. Composition of morphisms is by component and denoted by "$\circ$".

There are the four primitive morphisms:

$$\begin{aligned}
\mathrm{qual}(q) &= (n \mapsto q.n,\ t[c] \mapsto t[q.c],\ th[c] \mapsto th[q.c]) \\
\mathrm{inst}(t/x) &= (\mathrm{id},\ t'[x] \mapsto t'[t],\ th[x] \mapsto th[t]) \\
\mathrm{intp}(A \vdash B) &= (\mathrm{id}, \mathrm{id},\ B \vdash C \mapsto A \vdash C) \\
\mathrm{rewr}(A \vdash s \equiv t) &= (\mathrm{id},\ t'[s] \mapsto t'[t],\ A \vdash C[s] \mapsto A \vdash C[t])
\end{aligned}$$

All morphisms used in locales are composed from these; id denotes the identity morphism. The *qualification morphism* qual(q) prepends operation and theorem names with the qualifier q. Qualification of operation names is not necessarily a morphism on theorems. It is, though, in the context of a local theory, where operation names are bound names, and thus are renamed in definitions and theorems in a consistent manner.

For the other three to be morphisms, the underlying logic must enjoy the properties outlined in Section 3.1. The *instantiation morphism* inst(t/x) instantiates a variable x by a term t. If some specification A entails some other specification B

[4]Although the parameters are represented by variables, they may not be instantiated within the local theory itself. That would violate the contract of the specification and prohibit interpretation. In the implementation of local theories in Isabelle parameters are represented by free, not schematic variables.

then theorems may be lifted from the weaker to the stronger context. This is known as *theory interpretation*, and we denote the corresponding *interpretation morphism* by $\mathrm{intp}(A \vdash B)$. Finally, the *rewrite morphism* $\mathrm{rewr}(A \vdash s \equiv t)$ replaces all occurrences of s in terms and theorems by t.

4 Locales

Locales are a means of persisting local theories, and they provide flexible means of reuse: a locale declaration may extend one or several locales (import), a locale can be made available in other locales, or in other kinds of contexts the prover provides (interpretation). Locales are defined in this section, and their semantics is given by mapping them to local theories.

The core algorithm will be presented in pseudo code based on Standard ML [16]. Finite sequences (lists) will be denoted by square brackets; ":" is infix notation for the cons operator and "@" concatenation. Juxtaposition denotes function application, and $x \triangleright f$ is an alternative notation for $f\,x$. The function fold folds a binary operation f over a list:

```
fun fold $f$ [] $y = y$
  | fold $f$ $(x : xs)$ $y$ = fold $f$ $xs$ $(f\ x\ y)$
```

Parentheses are used for morphism application: $\varphi(x)$.

4.1 Definition

Locales are named, and there is a *locale environment lenv* that maps locale names to locales. A locale *lenv* n consists of these components:

- The *parameters* parms n, a sequence of variables $\overline{x}$.
- The *specification* spec n, a proposition A.
- The *declarations* decls n, a sequence of declarations of either the form abbreviates $c\ t$ or notes $b\ \ B$. Declarations are templates that will eventually be converted to the corresponding local theory operations—that is, they correspond to definitions and theorems inside the locale.
- The *dependencies* deps n, a sequence of pairs of locale names and morphisms. Such a pair (m, φ) is called *locale interpretation*. Dependencies model the relationship between locales as given by import and sublocale declarations.

Parameters and specification are the *head* of a locale, declarations the *body part*. Parameters, specification and declarations are also called *locale elements*. In the sequel, n is generally used instead of *lenv* n when there is no danger of confusion. Occasionally, locales are denoted as 4-tuples where the components appear in the order (parms n, spec n, decls n, deps n).

4.2 Mapping Locales to Local Theories

A local theory is obtained from a locale through application of local theory operations, which are generated from the locale elements. For a locale without dependencies this is straightforward. For a locale with dependencies, it involves traversing the graph defined by the locale dependencies. In both cases, this takes place in the presence of some global background theory Γ_0 and the locale environment *lenv*.

4.2.1 Locales without Dependencies

The case of a locale without dependencies is considered first. The local theory corresponding to a locale is obtained by initialising a local theory from its parameters and specification and adding the declaration elements. The latter is achieved by means of the activate operator:

```
fun activate $(n, \varphi)$ $ctxt$ =
    fold (fn abbreviates $(c \equiv t) \Rightarrow$ abbreviate $\varphi(c \equiv t)$
           | notes $b\ \ B \Rightarrow$ note $\varphi(b)\ \varphi(B)$) (decls $n$) $ctxt$
```

It folds local theory operations over the sequence of declaration of the locale n. Using this operator, the local theory *corresponding* to locale n is

$$\begin{aligned}\Gamma_0 &\triangleright \text{initialize (parms } n) \text{ (spec } n)\\ &\triangleright \text{activate } (n, \mathrm{id})\end{aligned}$$

The morphism argument φ enables to transform declarations before applying them to the local theory. This is required for resolving locales with dependencies.

4.2.2 Locales with Dependencies

If a locale n has the interpretation (m, φ) as a dependency this means that the declarations of m, transformed by φ, are part of the local theory corresponding to n. For this to be sound, φ must map the parameters of the interpreted locale m to terms in the local theory and the specification of m to a theorem of the local theory.

Since m may have dependencies as well, obtaining the local theory corresponding to n is a recursive process, which traverses the *locale dependency graph* given by the dependencies of all locales in the locale environment, and computes an enumeration of locale interpretations, all of whose declarations become part of the local theory corresponding to n. It is useful to allow cycles in the locale dependency graph—for example, for situations as in Section 2.2, where the locale `partial_order` has an interpretation of itself as a dependency. See also Fig. 1, which shows the locale dependency graph of that example. For obtaining a concrete local theory, the enumeration must be finite.

The enumeration of locale dependencies is based on the principle that an enumeration of locale interpretations contains at most one interpretation for each locale instance. This avoids duplication of declarations and enables to deal with cycles to a certain extent. A *locale instance* is a pair of locale name and terms $(n, (t_1, \ldots, t_k))$ where k is the number of parameters of n. The locale instance of a locale interpretation (n, φ) is $(n, (\varphi(x_1), \ldots, \varphi(x_k)))$, where $x_1, \ldots, x_k$ are the parameters of n. The notion of a locale instance is thus an abstraction of locale interpretation,

Fig. 1 Locale dependency graph for the examples in Section 2

$$\begin{tikzcd}
 & & \texttt{partial\_order} \arrow[loop right, "\begin{array}{l} S \mapsto S \\ \sqsubseteq \mapsto \lambda x y.\, y \sqsubseteq x \end{array}"] \arrow[d] \\
 & & \texttt{semilattice} \arrow[d, bend right, "\begin{array}{l} S \mapsto S \\ \sqsubseteq \mapsto \sqsubseteq \end{array}"'] \arrow[d, bend left, "\begin{array}{l} S \mapsto S \\ \sqsubseteq \mapsto \lambda x y.\, y \sqsubseteq x \end{array}"] \\
\texttt{closed} \arrow[dr] & & \texttt{lattice} \arrow[dl, "\begin{array}{ll} S \mapsto \mathcal{G} & \sqcap \mapsto \cap \\ \sqsubseteq \mapsto \trianglelefteq & \sqcup \mapsto \lambda K L.\, \langle K \cup L \rangle \end{array}"] \\
 & \texttt{group} &
\end{tikzcd}$$

taking only the effect of the interpretation on the locale parameters into account.[5] A locale instance $(n, \bar{s})$ *subsumes* another instance $(n, \bar{t})$ if there is a substitution σ such that $\sigma(s_i) = t_i$ simultaneously for all i. Depending on the logic, subsumption may be modulo an equational theory—for example, modulo α, β and η-conversion in the case of higher-order logic. Lifting subsumption to locale interpretations is straightforward: $(n, \varphi) \lesssim (m, \psi)$ if $n = m$ and the locale instance of (n, φ) subsumes the locale instance of (m, ψ). Subsumption of locale interpretations is a quasi order—that is, it is reflexive and transitive.

We are now ready to introduce the *roundup* algorithm, which is the key to activating locales with dependencies:

```
fun add $\chi$ $(n, \varphi)$ $(interps, marked) =$
  if $\exists (m, \psi) \in marked.\, (m, \psi) \lesssim (n, \chi \circ \varphi)$
  then $(interps, marked)$
  else
    let val $(interps', marked') =$
      fold (add $(\chi \circ \varphi)$) (deps $n$) $([], marked \cup \{(n, \chi \circ \varphi)\})$
    in $(interps \mathbin{@} interps' \mathbin{@} [(n, \chi \circ \varphi)], marked')$ end

fun roundup $activate$ $(n, \varphi)$ $ctxt =$
  let val $(interps, \_) =$ add id $(n, \varphi)$ $([], \emptyset)$
  in fold $activate\ interps\ ctxt$ end
```

[5]It does not matter whether this is achieved through instantiation morphisms, rewrite morphisms or a combination of both.

$\mathsf{roundup}$ *activate* (n, φ) recursively processes the locale interpretation (n, φ) and its dependencies. It computes the enumeration of locale interpretations for (n, φ) and folds the operation *activate* over it. The local theory *corresponding* to locale n with dependencies is defined thus:

$$\begin{aligned}\Gamma_0 &\rhd \mathsf{initialize}\ (\mathsf{parms}\ n)\ (\mathsf{spec}\ n)\\ &\rhd \mathsf{roundup\ activate}\ (n, \mathrm{id})\end{aligned}$$

The $\mathsf{roundup}$ operator traverses the locale dependency graph depth-first. It is important to note that the depth-first search is not on locales but on the graph of locale instances induced by the locale dependency graph reachable from the initial instance $(n, (\varphi(x_1), \ldots, \varphi(x_k)))$.

The function add performs the traversal. $\mathsf{add}\ \chi\ (n, \varphi)\ (\mathit{interps}, \mathit{marked})$ extends the enumeration *interps* by all nodes reachable via the morphism χ pointing to the interpretation (n, φ). Nodes subsumed by nodes that are already marked and their descendants are skipped to avoid duplicate declarations. The enumeration of interpretations is in post-fix order. Post-fix is necessary so that declarations in the dependencies of a locale are available to the declarations in its body.

Roundup terminates if the locale dependency graph is acyclic. It also terminates if every path eventually reaches a locale instance that is subsumed by an instance earlier on the path.

Roundup omits instances that are subsumed by instances occurring earlier in the enumeration. Instances subsumed by later instances are not removed, because there might already be instances in the enumeration whose declarations depend on such an instance. This leads to redundancy in enumerations if a specific interpretation of a locale is declared first and later a more general interpretation of the same locale is added.[6]

4.3 User-Level Operations

Most user-level operations of locales were encountered in Section 2. They are: locale declaration, entering the context of a locale, adding theorems, definitions and syntax abbreviations to a locale, introducing new locale dependencies and interpreting locales in the background theory. In addition to these, locales may also be interpreted in Isar proof contexts. The operations are now explained in terms of locales and local theories. They operate on a global state consisting of the background theory Γ_0 and the locale environment *lenv*. In Isabelle, the locale environment is part of Γ. The background theory is initialised to base, the locale environment is initially empty.

[6]That might be necessary when "bootstrapping" a development, but in practice it appears to happen rarely.

4.3.1 Locale Declaration

A locale declaration consists of an import expression, parameter declarations and assumptions. The general form of a locale declaration is this:

$$\begin{aligned}\textbf{locale}\ n = \ & q_1 : n_1\ \overline{t_1} + \ldots + q_k : n_k\ \overline{t_k} \\ & \qquad \textbf{for}\ \overline{x}\ +\ \textbf{fixes}\ \overline{y}\ +\ \textbf{assumes}\ a_1 : A_1, \ldots, a_j : A_j\end{aligned}$$

It adds a new locale named n, where $\overline{x}$ and $\overline{y}$ are the parameters, $q_1 : n_1\ \overline{t_1} + \ldots + q_k : n_k\ \overline{t_k}$ **for** $\overline{x}$ is the imported expression, and $A_1, \ldots, A_j$ are the assumptions. Each $q_i : n_i\ \overline{t_i}$ denotes a locale instance $(n_i, \overline{t_i})$ with qualifier q_i. The a_i are the names of the assumptions. Of the parameters, the $\overline{x}$ may occur in the imported expression and both $\overline{x}$ and $\overline{y}$ may occur in the assumptions. The latter are versions of the user input where free variables except parameters are universally closed.

The specification of the locale is combined from the import expression and the assumptions. Let $\overline{x_i} = \mathsf{parms}\ n_i$ be the parameters of locale n_i. Instantiation and qualification are described by the instantiation morphism

$$\sigma_i = \mathrm{inst}(\overline{t_i}/\overline{x_i}) \circ \mathrm{qual}(q_i).$$

Let $B_i = \mathsf{spec}\ n_i$ be the specification of locale n_i. The specification of the new locale involves the *locale predicate*

$$P_n\ \overline{x}\ \overline{y} \equiv \sigma_1(B_1) \wedge \ldots \wedge \sigma_k(B_k) \wedge A_1 \wedge \ldots \wedge A_j$$

and is $A \equiv P_n\ \overline{x}\ \overline{y}$.

By definition the specification A of n implies the specification $\sigma_i(B_i)$ for each locale instance $(n_i, \overline{t_i})$. This enables to lift theorems from the instance to the new locale via the interpretation morphism

$$\tau_i = \mathrm{intp}(A \vdash \sigma_i(B_i)).$$

The locale predicate is added to the background theory—that is, Γ_0 becomes

$$\Gamma_0 \triangleright \mathsf{def}\ P_n\ \overline{x}\ \overline{y} \equiv \sigma_1(B_1) \wedge \ldots \wedge \sigma_k(B_k) \wedge A_1 \wedge \ldots \wedge A_j.$$

The locale environment is extended such that

$$\begin{aligned} lenv\ n = \ & ([\overline{x}, \overline{y}], A, \\ & [\mathsf{notes}\ a_1\ (A \vdash A_1), \ldots, \mathsf{notes}\ a_j\ (A \vdash A_j)], \\ & [(n_1, \tau_1 \circ \sigma_1), \ldots, (n_k, \tau_k \circ \sigma_k)]). \end{aligned}$$

Example The declaration of locale partial_order in Section 2.1 defines the locale predicate partial_order by extending the background theory via

$$\mathsf{def}\,\mathrm{partial\_order}\ S\ le \equiv \Big(\bigwedge x.\, x \in S \Longrightarrow le\, x\, x\Big) \wedge \ldots$$

For brevity, only reflexivity is shown; antisymmetry and transitivity are indicated by dots. The locale environment is extended such that

$$\begin{aligned} &\mathit{lenv}\ \text{partial\_order} \\ &\quad = ([S, \mathit{le}], \text{partial\_order}\ S\ \mathit{le}, \\ &\qquad [\mathsf{notes}\ \text{refl}\ \left(\bigwedge x.\, x \in S \Longrightarrow \mathit{le}\, x\, x\right), \mathsf{notes}\ \text{antisym}\ldots, \mathsf{notes}\ \text{trans}\ldots], []) \end{aligned}$$

holds. The locale has no import and consequently no dependencies.

4.3.2 Working in the Context of a Locale

The **context** command enables to access a locale. It is followed by a block of declarations, which form the body:

$$\textbf{context}\ n\ \textbf{begin}\ \ldots\ \textbf{end}$$

In the scope of the body, a current local theory Γ_1 is maintained. Initially it is the local theory corresponding to n:

$$\begin{aligned} \Gamma_0 &\rhd \mathsf{initialize}\ (\mathsf{parms}\ n)\ (\mathsf{spec}\ n) \\ &\rhd \mathsf{roundup}\ \mathsf{activate}\ (n, \text{id}). \end{aligned}$$

Declarations in the body update the current local theory and add declarations to the locale. When leaving the scope of the context command, the current theory is discarded, but it can be recreated from the declarations stored in the locale when entering the locale for the next time.

The commands that are available in the body of the **context** command are syntax abbreviation, theorem declaration and definition:

$$\begin{aligned} \textbf{abbreviation}\ \ c\ \ &\textbf{where}\ c \equiv t \\ \textbf{theorem}\ \ b\ &:\ B \\ \textbf{definition}\ \ c\ \ &\textbf{where}\ c \equiv t \end{aligned}$$

The first two are straightforward, for they correspond directly to local theory operations and locale declarations. For the syntax abbreviation command the current local theory is updated via $\mathsf{abbreviate}\, c\ t$, and the declaration $\mathsf{abbreviates}\ c\ t$ is added to the declarations of the locale n. Likewise, for a theorem declaration the current local theory is extended by $\mathsf{note}\ b\ \ (A \vdash B)$ and the declaration that is added to the locale is $\mathsf{notes}\ b\ \ (A \vdash B)$.

Definition is more complicated, for it involves defining a foundational constant in the background theory. Let $\overline{x}$ be the parameters of the locale n and A its specification. The foundational constant is $n.c$, and its definition is that of c lifted over the parameters of the locale. That is, the background theory is replaced by this:

$$\Gamma_0 \rhd \mathsf{def}\, n.c\ \overline{x} \equiv t$$

The definition is also made in the underlying theory of the current local theory, which is then extended by the foundational constant.

$$\begin{aligned}\Gamma_1 &\rhd \mathsf{promote}\,(\mathsf{def}\, n.c\ \overline{x} \equiv t)\\ &\rhd \mathsf{abbreviate}\ c\ (n.c\ \overline{x})\\ &\rhd \mathsf{note}\ c\_\mathrm{def}\ (A \vdash c \equiv t)\end{aligned}$$

This becomes the new current local theory.

To persist the change, the declarations abbreviates c $(n.c\ \overline{x})$ and notes c_def $(A \vdash c \equiv t)$ are added to the locale.

Examples Further declarations from Section 2.1 can now be explained.

1. The definition of is_inf in the locale partial_order creates the foundational constant partial_order.is_inf in the background theory:

$$\mathsf{def}\,\mathrm{partial\_order.is\_inf}\ S\ le\ x\ y\ w \equiv le\ w\ x \wedge le\ w\ y \wedge \ldots$$

The locale itself is extended by an abbreviation is_inf and the theorem is_inf_def:

$$\begin{aligned}&\mathit{lenv}\ \mathrm{partial\_order}\\ &\quad = ([S, le], \mathrm{partial\_order}\ S\ le,\\ &\qquad [\mathsf{notes}\ \mathrm{refl}\ldots, \mathsf{notes}\ \mathrm{antisym}\ldots, \mathsf{notes}\ \mathrm{trans}\ldots,\\ &\qquad \mathsf{abbreviates}\ \mathrm{is\_inf}\ (\mathrm{partial\_order.is\_inf}\ S\ le),\\ &\qquad \mathsf{notes}\ \mathrm{is\_inf\_def}\ (\mathrm{is\_inf}\, x\ y\ w \longleftrightarrow le\ w\ x \wedge le\ w\ y \wedge \ldots)], [])\end{aligned}$$

2. The locale semilattice extends partial_order. This is reflected in the definition of the locale predicate, which is based on the locale predicate of the extended locale.

$$\begin{aligned}&\mathsf{def}\,\mathrm{semilattice}\ S\ le\\ &\quad \equiv \mathrm{partial\_order}\ S\ le \wedge \left(\bigwedge x\, y.\, x \in S \wedge y \in S \Longrightarrow \exists \mathit{inf}.\ \mathrm{is\_inf}\, x\ y\ \mathit{inf}\right)\end{aligned}$$

The locale environment entry only contains declarations related to semilattices:

$$\begin{aligned}&\mathit{lenv}\ \mathrm{semilattice}\\ &\quad = ([S, le], \mathrm{semilattice}\ S\ le,\\ &\qquad [\mathsf{notes}\ \mathrm{existence}\ \left(\bigwedge x\, y.\, x \in S \wedge y \in S \Longrightarrow \exists \mathit{inf}.\ \mathrm{is\_inf}\, x\ y\ \mathit{inf}\right)],\\ &\qquad [(\mathrm{partial\_order}, \mathrm{intp}(\mathrm{semilattice}\ S\ le \vdash \mathrm{partial\_order}\ S\ le))])\end{aligned}$$

Import of partial_order is reflected in the dependency. It incorporates declarations from partial_order, lifting them to the context of semilattices via the interpretation morphism intp(semilattice S $le \vdash$ partial_order S le).

Enumeration of interpretations for (semilattice, id) via the roundup algorithm yields a sequence with two elements:

$$\begin{array}{l}(\text{partial\_order}, \text{intp}(\text{semilattice } S\ le \vdash \text{partial\_order } S\ le)) \\ (\text{semilattice}, \text{id})\end{array}$$

The local theory corresponding to this sequence of interpretations is obtained by applying the morphisms to the declarations of the locales, which lifts them to the context of semilattice:

$$\begin{array}{l}\Gamma_0 \rhd \textsf{initialize}\,[S, le]\ (\text{semilattice } S\ le) \\ \quad \rhd \textsf{note}\ \text{refl}\ (\text{semilattice } S\ le \vdash \bigwedge x.\, x \in S \Longrightarrow le\, x\, x) \\ \quad \rhd \ldots \\ \quad \rhd \textsf{abbreviate}\ \text{is\_inf}\ (\text{partial\_order.is\_inf } S\ le) \\ \quad \rhd \textsf{note}\ \text{is\_inf\_def}\ (\text{semilattice } S\ le \vdash \text{is\_inf}\, x\ y\ w \longleftrightarrow le\, w\ x \wedge le\, w\ y \wedge \ldots) \\ \quad \rhd \textsf{note}\ \text{existence} \\ \qquad\qquad (\text{semilattice } S\ le \vdash \bigwedge x\, y.\, x \in S \wedge y \in S \Longrightarrow \exists inf.\ \text{is\_inf}\, x\ y\ inf)\end{array}$$

Declarations for antisymmetry and transitivity have again been indicated by dots.

4.3.3 Sublocale Declaration

Theory interpretation relations between locales are established with the sublocale command.

$$\textbf{sublocale}\ n \subseteq q_1 : n_1\ \overline{t_1} + \ldots + q_k : n_k\ \overline{t_k}\ \ \textbf{where}\ \overline{s} \equiv \overline{u}\ \langle proof\rangle$$

This extends the target locale n with interpretations of the locale instances $(n_i, \overline{t_i})$. Equations of the optional rewrite clauses, identified by the keyword **where** after the locale instances, enable to specify more elaborate mappings from the languages of the locale instances to the target locale than what is possible through instantiation. This is intended for (but not restricted to) mapping derived operations to suitable concepts in the target locale as illustrated in Section 2.4.

Let $\overline{x_i}$ again be the parameters of n_i and A_i the specification. Let A be the specification of n. The instantiation morphisms of the locale instances are

$$\sigma_i = \text{inst}(\overline{t_i}/\overline{x_i}) \circ \text{qual}(q_i).$$

Interpretation is based on these theorems:

$$A \vdash \sigma(A_1), \ldots, A \vdash \sigma(A_k)$$
$$A \vdash s_1 \equiv u_1, \ldots, A \vdash s_j \equiv u_j$$

To simplify establishing them, the local theory corresponding to n is provided when presenting the proof obligations. Proofs are provided by the user. The first set of theorems gives rise to the interpretation morphisms

$$\tau_i = \text{intp}(A \vdash \sigma_i(A_i)),$$

the second set to the rewrite morphism υ:

$$\upsilon_i = \mathrm{rewr}(A \vdash s_i \equiv u_i)$$

$$\upsilon = \upsilon_j \circ \ldots \circ \upsilon_1$$

Finally, the locale environment is changed at n by adding the interpretations $(n_i, \upsilon \circ \tau_i \circ \sigma_i)$ for $i = 1, \ldots, k$ after the existing dependencies.

Examples We are now ready to explain the sublocale declarations from Section 2.

1. The sublocale declaration at the beginning of Section 2.2,

   sublocale `partial_order ⊆ dual: partial_order "S" "λx y. y ⊑ x"`

   extends contexts generated from the locale partial_order by facts for the dual partial order induced by $\lambda x\, y.\ le\ y\ x$. This is achieved by adding a dependency on itself to the locale partial_order.
   First, duality needs to be established. This proof obligation is generated, and a proof supplied by the user:

   $$\text{partial\_order}\ S\ le \vdash \text{partial\_order}\ S\ (\lambda x\, y.\ le\ y\ x)$$

   Based on the theorem, the locale is extended by a dependency, which is an interpretation that also takes care of qualification and instantiation of the order relation by its dual:

   $$\begin{aligned} &\mathit{lenv}\ \text{partial\_order} \\ &\quad = ([S, le], \ldots, \\ &\qquad [(\text{partial\_order}, \mathrm{intp}(\text{partial\_order}\ S\ le \vdash \text{partial\_order}\ S\ (\lambda x\, y.\ le\ y\ x)) \circ \\ &\qquad\qquad \mathrm{inst}(\lambda x\, y.\ le\ y\ x / le) \circ \mathrm{qual}(\mathrm{dual})]) \end{aligned}$$

   After this extension, roundup of (semilattice, id) yields a sequence of three locale interpretations:

   $$\begin{aligned} &(\text{partial\_order}, \mathrm{intp}(\text{semilattice}\ S\ le \vdash \text{partial\_order}\ S\ (\lambda x\, y.\ le\ y\ x)) \circ \\ &\qquad \mathrm{inst}(\lambda x\, y.\ le\ y\ x / le) \circ \mathrm{qual}(\mathrm{dual})), \\ &(\text{partial\_order}, \mathrm{intp}(\text{semilattice}\ S\ le \vdash \text{partial\_order}\ S\ le)), \\ &\qquad (\text{semilattice}, \mathrm{id}) \end{aligned}$$

   This corresponds to the sequence

   $$\begin{aligned} &(\text{partial\_order}, [S, (\lambda x\, y.\ le\ y\ x)]) \\ &(\text{partial\_order}, [S, le]) \\ &(\text{semilattice}, [S, le]) \end{aligned}$$

   of three locale instances, and because

   $$\mathrm{inst}(\lambda x\, y.\ le\ y\ x / le) \circ \mathrm{inst}(\lambda x\, y.\ le\ y\ x / le) \equiv \mathrm{id}$$

   in the λ-calculus the sequence of interpretations is complete.

2. The sublocale declaration in Section 2.4 establishes the lattice of subgroups:

   sublocale `group ⊆ sub: lattice "𝒢" "op ⊴"`
     where `"sub.meet = (λK ∈ 𝒢. λL ∈ 𝒢. K ∩ L)"`
       and `"sub.dual.meet = (λK ∈ 𝒢. λL ∈ 𝒢. ⟨K ∪ L⟩)"`

   Supremum and infimum on subgroups are identified in rewrite clauses, which yield rewrite morphisms. Three proof obligations are generated:

   $\text{group}\ G\ \mathit{mult}\ \mathit{one}\ \mathit{inv} \vdash \text{lattice}\ \mathcal{G}\ \text{op}\trianglelefteq$

   $\text{group}\ G\ \mathit{mult}\ \mathit{one}\ \mathit{inv} \vdash \text{semilattice.meet}\ \mathcal{G}\ \text{op}\trianglelefteq\ = (\lambda K \in \mathcal{G}. \lambda L \in \mathcal{G}. K \cap L)$

   $\text{group}\ G\ \mathit{mult}\ \mathit{one}\ \mathit{inv} \vdash \text{semilattice.meet}\ \mathcal{G}\ (\lambda K\ L. L \trianglelefteq K) = (\lambda K \in \mathcal{G}. \lambda L \in \mathcal{G}. \langle K \cup L\rangle)$

   The notations `sub.meet` and `sub.dual.meet` are unfolded to semilattice.meet $\mathcal{G}$ op$\trianglelefteq$ and semilattice.meet $\mathcal{G}$ $(\lambda K\ L.\ L \trianglelefteq K)$ respectively in the obligations.[7] After discharging the proof obligations, the locale group is extended by a dependency to lattice.

4.3.4 Interpretation

These commands interpret locales in global theories and Isar proof contexts, respectively:

$$\textbf{interpretation}\ q_1 : n_1\ \overline{t_1} + \ldots + q_k : n_k\ \overline{t_k}\ \textbf{where}\ \overline{s} \equiv \overline{u}\ \langle \mathit{proof}\rangle$$

$$\textbf{interpret}\ q_1 : n_1\ \overline{t_1} + \ldots + q_k : n_k\ \overline{t_k}\ \textbf{where}\ \overline{s} \equiv \overline{u}\ \langle \mathit{proof}\rangle$$

They are discussed in detail in an earlier publication on locales [4]. Interpretations for all given locale instances, and for all locale instances reachable from these by the roundup algorithm, are added immediately to the global theory or proof context. Equations refine the interpretations as in the sublocale command. The interpreted instances are tracked (they correspond to marked instances in roundup), and interpretations subsumed by earlier interpretations, possibly from previous interpretation commands, are skipped.

Tracking of interpreted instances enables providing two additional services in global theories: whenever a declaration is added to a locale, it is propagated to the global theory for all instances of that locale in the global theory; likewise, whenever a dependency is added to a locale, interpretations of locale instances newly entailed by existing instances are added to the global theory. In this way, global theories “subscribe” to locales via interpretations like locales do to locales via sublocale declarations.

Such facilities are not provided for interpretation in proof contexts: these disappear after closing, and the Isar proof language does not permit extending locales from within the body of a proof.

[7] These abbreviations are declared in lattice and are only introduced to group by the sublocale declaration. To simplify the notation in where clauses, from Isabelle 2011-1, they are already available when the where clauses are processed.

5 Other Theory Module Structuring Mechanisms

Locales employ interpretation as the main means of reuse. This, and the high amount of automation obscure how locales are related to more commonly known structuring mechanisms. In this section, relations to ML-style modules, type classes and also mixin modules, the latter of which are found in modern object-oriented languages, are studied.

5.1 ML-Style Module Systems

The module system of the programming language ML (actually, Standard ML [16]) is a well-understood means for structuring software developments. Locales enable modular development of formal theories. Both languages are different, nonetheless modularity provided by locales can be explained with notions borrowed from ML modules.

In ML a *module* consists of component bindings, which represent data fields and code. A *signature* consists of component declarations, which merely assert the component's types. A module m is said to implement a signature I, written $m : I$, if for every declaration in I there is a binding in m, and each bound value in m is of the type given in the corresponding declaration. This arrangement enables a programmer to code against a module without having it available. The signature is sufficient.

The situation is analogous in formal theory developments, where if the components bindings of modules contain proofs and the component declarations of signatures contain the theorem statements, knowing the signature of an imported theory module is sufficient to use its theorems when providing new proofs.

For explaining locales, this idea is now elaborated. The notation for ML-style modules from Harper and Pierce [12] is modified to accommodate theorems and proofs. A formal development P consists of module and signature bindings:

$$\mathrm{P} ::= \mathrm{B}^{+} \qquad \mathrm{B} ::= \textbf{module}\ \mathrm{m}[:\ \mathrm{I}] = \mathrm{M} \mid \textbf{signature}\ \mathrm{J} = \mathrm{I}$$

A module can be a basic module consisting of component bindings, the reference to a module variable (unqualifed or qualifed), a functor, or be obtained by functor application.[8]

$$\mathrm{F}, \mathrm{M} ::= \textbf{mod}\ \left\{\mathrm{CB}^{+}\right\} \mid \mathrm{m} \mid \mathrm{M.m} \mid \lambda\mathrm{m} : \mathrm{I.M} \mid \mathrm{F(M)}$$

$$\mathrm{CB} ::= \textbf{val}\ \mathrm{x} = \mathrm{t} \mid \textbf{abbrev}\ \mathrm{y} = \mathrm{t} \mid \textbf{thm}\ \mathrm{X} = \mathrm{T} \mid \textbf{module}\ \mathrm{m} = \mathrm{M} \mid \textbf{open}\ \mathrm{M}$$

Conceptually, a component binding either binds a value to its definition or it binds a proof. To model local theories more adequately, value bindings, which instantiate parameters, and syntax abbreviations are distinguished. Qualified and unqualified import of modules is also available.

Terms include values and values bound in nested modules. Likewise for proofs.

$$\mathrm{t} ::= \ldots \mid \mathrm{x} \mid \mathrm{M.x} \qquad \mathrm{T} ::= \langle \vdash \mathrm{t} \rangle \mid \mathrm{X} \mid \mathrm{M.X}$$

Rather than denoting proofs explicitly, we write $\langle \vdash \mathrm{t} \rangle$ for a proof of the theorem $\vdash \mathrm{t}$.

[8]The grammar permits higher-order modules, but they will not be used.

Of signatures only the basic ones are required:

$$\mathrm{I} ::= \mathbf{sig}\ \{\mathrm{CD}^{+}\}$$

$$\mathrm{CD} ::= \mathbf{val}\ \mathrm{x} \mid \mathbf{abbrev}\ \mathrm{y} = \mathrm{t} \mid \mathbf{thm}\ \mathrm{X} : \vdash \mathrm{t} \mid \mathbf{module}\ \mathrm{m} = \mathrm{M}$$

The component declaration syntax of signatures corresponds to the component binding syntax of modules. The notation **thm** X : $\vdash$ t says that X will be bound to a proof of $\vdash$ t.

In terminology of modules and signatures, a locale is a functor that maps a parameter module, consisting of several value bindings and a theorem binding, to a module, which extends the parameter module by abbreviation bindings and (typically many) additional theorem bindings. Developing this connection formally is beyond the scope of this discussion, but we will illustrate key points in a series of examples, which are taken from the previous sections.

The locale created in the initial declaration of the locale `partial_order` in Section 2.1 corresponds to a functor whose parameter has the signature

signature PO =
  sig { **val** S **val** le **thm** partial_order : $\vdash$ partial_order S le }

and the functor itself is this:

module po_fun = λ po : PO.
  mod {
    val S = po.S **val** le = po.le
    thm partial_order = po.partial_order
    thm refl = $\langle \vdash \bigwedge x.\, x \in S \Longrightarrow le\ x\ x \rangle$
    thm antisym = … **thm** trans = …
  }

Reflexivity, antisymmetry and transitivity are derived from the theorem parameter. In favour of concise notation this is not made explicit here. The definition of is_inf extends the functor to this:

module po_fun = λ po : PO.
  mod {
    val S = po.S **val** le = po.le
    thm partial_order = po.partial_order
    thm refl = $\langle \vdash \bigwedge x.\, x \in S \Longrightarrow le\ x\ x \rangle$
    thm antisym = … **thm** trans = …
    abbrev is_inf = partial_order.is_inf S le
    thm is_inf_def = $\langle \vdash \text{is\_inf}\ x\ y\ w \longleftrightarrow le\ w\ x \wedge le\ w\ y \wedge \ldots \rangle$
  }

Composition of locales is achieved through interpretation, either by sublocale declarations, or through interpretations generated from imports in locale declarations. Within locales, interpretations are stored as dependencies, and are resolved by the roundup algorithm. The functor po_fun above models extensibility of the locale `partial_order` by syntax abbreviations and theorems. In order to model extensibility through dependencies, the functor is split into a body functor, modelling the body part of the locale, and a functor for dependencies:

```
module po_body = λ po : PO.
  mod {
    thm partial_order = po.partial_order
    thm refl = ⟨⊢ ⋀ x. x ∈ S ⟹ le x x⟩
    . . .
  }

module po_deps = λ po : PO.
  mod {
    val S = po.S val le = po.le
    open po_body(po)
  }
```

The body functor contains no value bindings, these have moved to the dependency functor, which imports the body functor. When adding a dependency to a locale, this amounts to extending the dependency functor by import declarations or module bindings of applications of body functors of locales as enumerated by roundup. To illustrate this, we consider adding the dependency of its dual to the locale `partial_order`. The dependency functor changes to this:

```
module po_deps = λ po : PO.
  mod {
    val S = po.S val le = po.le
    module dual = po_body(
      mod {
        val S = S val le = (λx y. le y x)
        thm partial_order = ⟨⊢ partial_order S le⟩
      })
    open po_body(po)
  }
```

A second instance of po_body is applied to the partial order obtained by inverting the order relation. The resulting submodule is bound to the module variable dual in order to achieve qualification of identifiers. Within the dependency functor the "wiring" of parameters of the body functors takes place. Notably, while both applications of po_body share the parameter S, one application is to the order relation le, the other to its inverse $\lambda x\, y.\ le\ y\ x$. The theorem partial_order in the functor argument of the module binding dual is derived from the incoming theorem po.partial_order via the theorem provided in the dependency.

5.2 Type Classes

Isabelle's type classes are an adaption of Haskell-style type classes to the type system of Gordon's HOL prover. They replace the plain Hindley–Milner polymorphism of the latter by an order-sorted polymorphism where the sorts are finite sets of classes.

Nipkow [17] discusses the idea and Wenzel [24] shows how the integration with the logic can be done in a sound manner.

A class represents the set of types for which certain operation symbols are available (systematic overloading as in Haskell) and for which certain axioms hold. Classes are ordered and the overloaded operations and axioms of a superclass are available in each of its subclasses. A sort denotes the set of types present in each of the contained classes. The order on classes $\subseteq$ induces an order on sorts $\preceq$. Both are reflexive and transitive.

Instantiation of classes is available in two flavours: arity declarations of type constructors and class inclusion. An arity declaration $tc :: (s_1, \ldots, s_n)\, c$ means that the type constructor tc applied to types of sorts $s_1, \ldots, s_n$ yields a type of class c. Class inclusion $c' \subseteq c$ means that all types in c' also belong to c. Arities and class inclusion must be established formally. That is, proofs that the axioms of class c are fulfilled need to be supplied in both cases.

Both forms of instantiation can be expressed in the framework of locales through interpretation. To illustrate this, here is a formalisation of partial orders and semilattices with type classes:[9]

```
axclass order_syntax ⊆ type

consts le :: "'a::order_syntax ⇒ 'a ⇒ bool" (infixl "⊑" 50)

axclass partial_order ⊆ order_syntax
  refl: "x ⊑ x"
  antisym: "⟦ x ⊑ y; y ⊑ x ⟧ ⟹ x = y"
  trans: "⟦ x ⊑ y; y ⊑ z ⟧ ⟹ x ⊑ z"

definition is_inf where "is_inf x y w ⟷
  w ⊑ x ∧ w ⊑ y ∧ (∀z. z ⊑ x ∧ z ⊑ y ⟶ z ⊑ w)"
```

The class `partial_order` is declared in two steps: `order_syntax` extends the class `type` of all types. At this level the overloaded operation `le` is introduced. The class is then extended with axioms, obtaining `partial_order`. The predicate `is_inf` is, by type inference, also associated to `order_syntax`. A class for (lower) semilattices is obtained by a further extension:

```
axclass semilattice ⊆ partial_order
  existence: "∃inf. is_inf x y inf"

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

[9]Isabelle's type classes are also known as *axiomatic* type classes. The examples here are deliberately based on the old user interface in Isabelle 2009, because it provides more direct access to the discussed mechanisms than the combination of type classes and locales, called constructive type classes, from later versions. The structures' carrier is not made explicit. This is merely a convenience, not a restriction of type classes.

5.2.1 Class Inclusion

A natural example for class inclusion through instantiation are total orders, which are partial orders that fulfill an additional axiom:

```
axclass total_order ⊆ partial_order
  total: "x ⊑ y ∨ y ⊑ x"
```

On the other hand, they are lattices, and the class hierarchy can be changed by adding a class inclusion relation with an instance declaration:

```
instance total_order ⊆ semilattice ⟨proof⟩
```

The formalisation with locales is analogous. The locale for total orders is obtained by extending the locale `partial_order` from Section 2, and the inclusion is established with a sublocale declaration:

```
locale total_order =
   partial_order "S" "le" for S and le (infixl "⊑" 50) +
   assumes total: "⟦x ∈ S; y ∈ S⟧ ⟹ x ⊑ y ∨ y ⊑ x"

sublocale total_order ⊆ lattice "S" "le"  ⟨proof⟩
```

Since the second argument of the sublocale command is an expression, lattices other than the order relation `le` could be interpreted as well. This is not possible with class inclusion, where the second argument is only a class name.

5.2.2 Type Instantiation

There are two ways of translating type instantiation to locales, and which one is applicable depends on the arity of the type constructor. For type constructors without parameters—that is, for primitive types—instantiation is achieved through interpretation in the background theory. For type constructors with parameters, the interpretation is relative to a locale.

The first example involves the primitive type `nat` of natural numbers, which is totally ordered by magnitude. Like declaration, type instantiation of classes proceeds in two steps:

```
instance nat :: order_syntax
```

makes the operation `le` available for `nat`. It can then be defined (using a variant of the definition command with reduced syntactic checks):

```
defs (overloaded) le_nat_def: "(m::nat) ⊑ n ≡ m ≤ n"
```

Finally, the validity of the instance is shown, using facts of the natural numbers.

```
instance nat :: total_order ⟨proof⟩
```

The corresponding construction is achieved in locales via an interpretation in the background theory:

```
interpretation nat: total_order "UNIV::nat set" "op ≤"  ⟨proof⟩
```

An instantiation of a type constructor with parameters requires a locale that represents the sorts of the type parameters. For example, the order on pairs can be defined based on the orders of the left and right components. First, again the formalisation with type classes. Let "$\star$" be the type constructor for pairs. The first instance declaration makes the syntax available for pairs:

```
instance ⋆ :: (order_syntax, order_syntax) order_syntax
```

There are several ways of defining an order relation on pairs. We choose the lexicographic order:

```
defs (overloaded) le_pair_def: "x ⊑ y ≡
  if fst x ≠ fst y then fst x ⊑ fst y else snd x ⊑ snd y"
```

This order is partial if the orders on the left and right components are partial. It is total, if the orders on the components are total. Such a mapping of one class hierarchy to another is common, and it can be expressed through two instance declarations.

```
instance ⋆ :: (partial_order, partial_order) partial_order
  ⟨proof⟩
instance ⋆ :: (total_order, total_order) total_order
  ⟨proof⟩
```

Representing these instantiations in locales requires a target locale per arity. In the first instantiation both parameters are partial orders:

```
locale pair_partial_order =
 left: partial_order "S₁" "le₁" + right: partial_order "S₂"
 "le₂"for S₁ and le₁ (infixl "⊑₁" 50) and S₂ and le₂ (infixl "⊑₂" 50)
begin
  definition le_lex (infixl "⊑lex" 50) where "x ⊑lex y ⟷
   (if fst x ≠ fst y then fst x ⊑₁ fst y else snd x ⊑₂ snd y)"
end
```

The definition of the combined order relation `op ⊑lex` takes place in the target locale, and the dependency is introduced with this sublocale declaration:

```
sublocale pair_partial_order ⊆ lex:
  partial_order "S₁ × S₂" "op ⊑lex" ⟨proof⟩
```

In the target locale for the second instantiation both order relations are total orders:

```
locale pair_total_order =
   left: total_order "S₁" "le₁" + right: total_order "S₂" "le₂"
   for S₁ and le₁ (infixl "⊑₁" 50) and S₂ and le₂ (infixl "⊑₂" 50)
```

This is a special case of the previous target locale, and so the definition and theorems can be carried over from `pair_partial_order` and, by transitivity of the dependency relation, from its dependencies with a first sublocale declaration:

```
sublocale pair_total_order ⊆
  pair_ partial_order "S₁" "le₁" "S₂" "le₂"  ⟨proof⟩
```

$$\begin{tikzcd}
\texttt{partial\_order} \arrow[rr] \arrow[d, bend right=40, swap, "\begin{array}{c} S \mapsto S_1 \\ \sqsubseteq \mapsto \sqsubseteq_1 \end{array}"] \arrow[d, bend left=20, "\begin{array}{c} S \mapsto S_2 \\ \sqsubseteq \mapsto \sqsubseteq_2 \end{array}"] \arrow[d, bend left=60, "\begin{array}{c} S \mapsto S_1 \times S_2 \\ \sqsubseteq \mapsto \sqsubseteq_{\mathtt{lex}} \end{array}"] \arrow[drr] & & \texttt{lattice} \arrow[d] \\
\texttt{pair\_partial\_order} \arrow[drr] & & \texttt{total\_order} \arrow[d, bend right=40, swap, "\begin{array}{c} S \mapsto S_1 \\ \sqsubseteq \mapsto \sqsubseteq_1 \end{array}"] \arrow[d, bend left=20, "\begin{array}{c} S \mapsto S_2 \\ \sqsubseteq \mapsto \sqsubseteq_2 \end{array}"] \arrow[d, bend left=60, "\begin{array}{c} S \mapsto S_1 \times S_2 \\ \sqsubseteq \mapsto \sqsubseteq_{\mathtt{lex}} \end{array}"] \\
& & \texttt{pair\_total\_order}
\end{tikzcd}$$

Fig. 2 Locale dependency graph for the examples in Section 5.2

The interpretation representing the instantiation follows:

sublocale `pair_total_order` $\subseteq$ `lex: total_order "`$S_1 \times S_2$`" "op `$\sqsubseteq_{\mathtt{lex}}$`"`
⟨*proof*⟩

The resulting locale dependencies are shown in Fig. 2.

5.2.3 Comparison

As a mechanism for structuring theory modules, type classes are relatively weak. The type system does not provide dependent types, and in some systems, including Isabelle, a class is restricted to a single type parameter. Locales do not have these shortcomings. On the other hand, classes provide more automation.

A deeper comparison is possible by observing that in terms of a functorial module system type classes are the signatures and instance declarations are the functors. See also Harper and Pierce [12], who discuss this relationship for Haskell's type classes. The order-sorted polymorphism of Isabelle's type classes admits principal types and therefore sort information can be computed by type inference. This means that functor applications are computed "on the fly" when automatic tools such as Isabelle's rewrite engine (commonly known as the *simplifier*) are active.

Locales compute functor applications by resolving locale dependencies with the roundup algorithm. Since this is only executed when entering a context target, locales are required that serve as working contexts. Target locales express specification situations that are the focus of particular mathematical analyses. A type instantiation $tc :: (s_1, \ldots, s_n)\, c$ can be translated to the language of locales by providing a target locale that imports the locales corresponding to $s_1, \ldots, s_n$ and adding c as a dependency by showing that the target locale is a sublocale of c. If there is another type instantiation $tc :: (s'_1, \ldots, s'_n)\, c'$ of the same type constructor and $s_1 \preceq s'_1, \ldots, s_n \preceq s'_n$, then it needs to be shown that the target locale of the former type instantiation is a sublocale of the latter. This enriches the working context by information that would be inferred by type classes.

Type instantiation of primitive types is a special case and dealt with by interpretation in the global background theory. Class inclusion declarations translate directly to sublocale declarations.

5.3 Beyond Parameter Substitution

Many module systems have in common that the desired ways of reuse must be anticipated. For example, in the case of a functor, only the parameters can be instantiated when the functor is applied. It is not possible to identify components defined in the body of one functor with components of some other functor. In general, when combining two modules, components of one may need to be identified with components of the other. This is known as the *coherence problem* [12]. The diamond problem, where one module is inherited through two different paths in an inheritance diagram, is a special case.

5.3.1 Mixin Modules

In object-oriented programming languages the coherence problem occurs with multiple inheritance. A solution adopted by some languages is to restrict multiple inheritance to classes that do not encapsulate state—that is, without member fields. Coherence is achieved by redefining a method inherited from more than one superclass such that the desired version is called. Usually, one superclass with member fields is allowed. The others are said to be *mixed in*. This approach is known as *mixin modules* [1, 6]. Terminology varies. For example, in the programming language Scala, classes that are amenable to mixing in with other classes are called *traits* [19].

The coherence problem also exists when combining mathematical theories. Here, usually some *base operations* are specified via axioms; other, *derived operations* are defined in terms of the base operations. A natural representation of such a theory module as a functor puts the base operations in the parameter signature and the derived operations in the functor body. We have done so in the locale examples in Section 2, where the order relation `le` is a base operation of the `partial_order` locale and the group operations are base operations of groups. Supremum and infimum and the subgroup relation are derived.

When transporting the theorems of a theory module to some other context, replacing the base operations only is in general not sufficient. In Section 2.3 the supremum and infimum operations were mapped to set operations that already existed in the background theory. Likewise, in Section 2.4, they were mapped to group operations of the target locale. Locales enable replacing derived operations by means of rewrite morphisms. There is an analogy to redefining a method in a class: in either case the modified component is not a parameter. In other words, the change is not anticipated. The soundness of rewrite morphisms is rooted in the underlying logical system.

5.3.2 Equivalent Formalisations

An important use case of rewrite morphisms, other than the one described above, are equivalent formalisations. Often there is not only one (the *canonical*) set of base operations for a mathematical theory. For example, while the base operations of groups are usually the binary operation, unit and inverse, the latter two are unique in a semigroup (if they exist) and they can be formalised as derived operations. Gunter [9] proposed this, presumably because fewer parameters are simpler to manage. A more involved example are lattices, which allow for an alternative set of axioms where supremum and infimum are the base operations. This is elaborated in Fig. 3. The top part shows the formalisation based on partial orders (like the

```
locale partial_order =
  fixes le (infixl "⊑" 50)
  assumes refl: "x ⊑ x"
    and antisym: "⟦ x ⊑ y; y ⊑ x ⟧ ⟹ x = y"
    and trans: "⟦ x ⊑ y; y ⊑ z ⟧ ⟹ x ⊑ z"
begin
  definition is_inf where "is_inf x y w ⟷
    w ⊑ x ∧ w ⊑ y ∧ (∀z. z ⊑ x ∧ z ⊑ y ⟶ z ⊑ w)"
end

locale semilattice = partial_order +
  assumes ex_inf: "∃inf. is_inf x y inf"
begin
  definition meet (infixl "⊓" 70) where "x ⊓ y = (THE inf. is_inf x y inf)"
end

locale lattice =
  semilattice "le" + dual!: semilattice "λx y. y ⊑ x" for le (infixl "⊑" 50)
begin
  abbreviation join (infixl "⊔" 65) where "join ≡ dual.meet"
end
```

(a) Lattice based on partial order

```
locale lattice' =
  fixes meet (infixl "⅄" 70) and join (infixl "Y" 65)
  assumes comm: "x ⅄ y = y ⅄ x" "x Y y = y Y x"
    and assoc: "(x ⅄ y) ⅄ z = x ⅄ (y ⅄ z)" "(x Y y) Y z = x Y (y Y z)"
    and absorp: "x ⅄ (x Y y) = x" "x Y (x ⅄ y) = x"
begin
  definition le (infixl "≼" 50) where "x ≼ y ⟷ x = x ⅄ y"
end
```

(b) Lattice as equational theory

```
sublocale lattice ⊆ algebraic: lattice' "op ⊓" "op ⊔"
  where "algebraic.le = op ⊑"
⟨proof⟩
sublocale lattice' ⊆ po: lattice "op ≼"
  where "po.meet = op ⅄" and "po.join = op Y"
⟨proof⟩
```

(c) The formalisations are equivalent.

Fig. 3 Two formalisations of lattice

example in Section 2, but for conciseness omitting the carrier set). Beneath follows the alternative formalisation. At the bottom, equivalence of the two locales is established formally with two circular sublocale declarations. It is worth noting that roundup terminates both when entering the context of `lattice` and when entering the context of `lattice'`. The arrangement achieved with these declarations makes theorems from one formalisation of lattice available in the other and vice versa.

The roundup algorithm operates on locale instances, which are an abstraction of locale interpretations: if there are two interpretations such that the effect of both their morphisms on the locale parameters is the same, then only one interpretation will be generated (the one that appears first in the enumeration). This means that there cannot be two interpretations that agree on the parameters but map a derived operation, via a rewrite morphism, to different (but equivalent) terms. In such a situation, a possible solution is choosing an alternative formalisation where the operation in question is a parameter.[10]

6 Conclusion

Locales are a powerful tool for organising mathematical knowledge. They provide commands for declaring locales, entering the context of a locale, extending locales, identifying logical relations between locales and translating the knowledge of a locale to other contexts—in particular, global theories and proof contexts. And, locales can be integrated with local theories, an abstraction of various forms of theories and contexts found in Isabelle, but which are not fundamentally linked to Isabelle or to its logic.

Locales are organised in a dependency graph that encodes the logical relations between them. A locale is persisted mathematical knowledge that can be "brought to life" by converting it to a local theory, in which reasoning may take place. This is called activation, and relations from the dependency graph are resolved by the roundup algorithm. Activation makes locales *dynamic*: declarations added to a locale are propagated to all instances automatically. This enables users to provide definitions, theorems and interpretations, including locale dependencies, in an order that is natural for the mathematics that is being formalised.

Activation is along morphisms. A locale can be activated to its induced local theory via the identity morphism, or, by interpretation, to other target contexts. For interpretation, the image of the specification under the morphism must be derivable in the target context. Interpretation makes locales first-order functors. By tracking interpreted instances, the dynamic flavour of activation is also provided for interpretation in global theories.

Locale predicates reflect locales, which are by themselves extra-logical, into the logic and enable reasoning about locales. This was used in this paper only in passing, in the definition of the subgroup relation based on the locale `closed`. Locale predicates can, for example, be used to deal with infinite families of locales. This is demonstrated in detail elsewhere [4].

[10] In Isabelle, this may also be resolved by putting the interpretations in different global theories.

Locales are partially correct: if roundup terminates then the generated theorems are derivable from the specification. Roundup terminates if the dependency graph is acyclic. It also terminates for important cyclic cases: logically equivalent specifications and operators that are self-dual.[11]

Activation is a fairly expensive operation. When a locale is activated, morphisms are applied to all its declarations and to the declarations of all dependencies. Nevertheless, the implementation is efficient enough so that locales have become a mainstay of Isabelle's theory libraries. Morphisms can be applied to declarations that are to be activated in parallel, which enables making use of modern, parallel hardware. Users can improve the performance of theory developments by putting several declarations into the block of a single context command, which avoids unnecessary repetitions of activation.

6.1 Management of Theory Module Hierarchies

One can distinguish declared and derived relations between theory modules. Declared relations are given as import in locale declarations, and derived relations are provided with the sublocale command. Both are via morphisms, which enable mapping the language of the source to the language of the target. Internally, both kinds of relations are uniformly implemented through interpretation of locale dependencies.

Module hierarchies in programming languages are usually trees (or directed acyclic graphs if multiple inheritance is supported) and extension is only possible at the fringe. This can lead to the same concept being developed at several places in a library simultaneously. To avoid this redundancy, the *tiny theories* method was proposed [7]. This is a more radical version of the little theories approach, where extensions of theory modules are done by introducing one axiom at a time. This would ensure that in a theory library of, for example, order relations or rings even the more obscure variants of these structures are readily available. While being a great convenience for the user, the tiny theories method can complicate library design, because it requires anticipating all variants.

Locales enable the library designer to insert a theory module into an existing hierarchy via the sublocale command, a feature that is inspired by Isabelle's type classes. This means that theory modules are not required to be built up incrementally in a per-axiom fashion. Neither need more rarely used variants of theories be anticipated from the beginning, just because they are in the middle of the hierarchy. They may be added when needed.

6.2 Extensibility of Theory Modules

While locales can be seen as first-order ML-style functors, this does not capture all operational aspects adequately. In particular, bodies of ML functors are not extensible. But this is an important requirement for a module system for mathematical theories. The Coq module system [23], which implements a higher-order variant of

[11]The latter relies on term equivalence being modulo $\alpha\beta\eta$-conversion in Isabelle. Important other cyclic locale dependencies can be made acyclic by introducing additional logically equivalent locales. For an example, see the tutorial [5].

ML-style functors fairly closely, overcomes this problem by introducing namespaces as an additional layer of abstraction so that bindings from several functors contributing to a theory module can be referred to in a uniform manner. Locales have been designed to be extensible by theorem bindings right away. Extensibility by definitions was introduced with local theories [11].

6.3 Rewrite Morphisms and Coherence

Locales can be combined by means of locale expressions, either in the import section of a locale declaration, or in an interpretation. Locales can be combined with target contexts through interpretation. In the case of the sublocale command, the target context is again a locale.

In order to achieve coherence between the combined locales, relations between parameters may be given through parameter instantiations in the expression. In interpretations, including sublocale declarations, additionally value bindings (i.e., definitions) in locale bodies may be changed through rewrite morphisms, which map bound names to terms in the target context of the interpretation. (In principle, locale declarations could also accept rewrite morphisms, but requesting the needed proofs might seem counter-intuitive to users.)

The need for instantiation as opposed to, for example, renaming, is immediately clear for interpretation. But also in locale declarations instantiation leads to a more expressive system. One may, for example, consider a locale for homomorphisms where one parameter represents the operation of the domain and another parameter the operation of the co-domain. With instantiation, a locale for endomorphisms can be derived easily by setting both parameters to the same operation [5, Section 6.2]. With renaming this is not possible, since distinct names need to remain distinct.

The relation of rewrite morphisms to mixin modules of object-oriented programming languages discussed in Section 5.3 is a striking example of how the need for flexible means of reuse in module systems can lead to related solutions in different domains. This was understood by the author only after conceiving rewrite morphisms as a natural extension to interpretation of definitions as they are handled in local theories.

Acknowledgements The first design of locales was inspired by Coq sections [14, 15]. Wenzel integrated locales with the Isar proof language and provided means for constructing locale hierarchies [2]. Theorem reuse through interpretation and means for changing the locale hierarchy were added by the author [3, 4]. Local theories [11] considerably helped clarify locales. Without them, the redesign for Isabelle 2009 would not have been possible.

References

1. Ancona, D., Zucca, E.: A theory of mixin modules: basic and derived operators. Math. Struct. Comput. Sci. **8**, 401–446 (1998)
2. Ballarin, C.: Locales and locale expressions in Isabelle/Isar. In: Berardi, S., Coppo, M., Damiani, F. (eds.) Types for Proofs and Programs, TYPES 2003, Torino, Italy. LNCS 3085, pp. 34–50. Springer (2004)
3. Ballarin, C.: Interpretation of locales in Isabelle: managing dependencies between locales. Tech. Rep. TUM-I0607, Technische Universität München (2006)

4. Ballarin, C.: Interpretation of locales in Isabelle: theories and proof contexts. In: Borwein, J.M., Farmer, W.M. (eds.) Mathematical Knowledge Management, MKM 2006, Wokingham, UK. LNCS 4108, pp. 31–43. Springer (2006)
5. Ballarin, C.: Tutorial to locales and locale interpretation. In: Lambán, L., Romero, A., Rubio, J. (eds.) Contribuciones Científicas en Honor de Mirian Andrés Gómez. Servicio de Publicaciones de la Universidad de La Rioja, Logroño, Spain (2010). Also part of the Isabelle user documentation
6. Bracha, G.: The programming language Jigsaw: mixins, modulariy and multiple inheritance. Ph.D. thesis, University of Utah (1992). Also Technical Report UUCS-92-007
7. Carette, J., Farmer, W.M., Jeremic, F., Maccio, V., O'Connor, R., Tran, Q.M.: The MathScheme library: some preliminary experiments. Manuscript arXiv:1106.1862v1 (2011)
8. Farmer, W.M., Guttman, J.D., Thayer, F.J.: Little theories. In: Kapur, D. (ed.) Automated deduction, CADE-11: Saratoga Springs, NY, USA. LNCS 607, pp. 567–581. Springer-Verlag (1992)
9. Gunter, E.L.: Doing algebra in simple type theory. Tech. Rep. MS-CIS-89-38, University of Pennsylvania (1989)
10. Haftmann, F., Wenzel, M.: Constructive type classes in Isabelle. In: Altenkirch, T., McBride, C. (eds.) Types for Proofs and Programs, TYPES 2006, Nottingham, UK. LNCS 4502, pp. 160–174. Springer (2007). doi:10.1007/978-3-540-74464-1_11
11. Haftmann, F., Wenzel, M.: Local theory specifications in Isabelle/Isar. In: Berardi, S., Damiani, F., de'Liguoro, U. (eds.) Types for Proofs and Programs, TYPES 2008, Torino, Italy. LNCS 5497, pp. 153–168. Springer (2009). doi:10.1007/978-3-642-02444-3_10
12. Harper, R., Pierce, B.C.: Design considerations for ML-style module systems. In: Pierce, B.C. (ed.) Advanced Topics in Types and Programming Languages. MIT Press (2005)
13. Jenks, R.D., Sutor, R.S.: AXIOM: The Scientific Computation System. Springer-Verlag (1992)
14. Kammüller, F.: Modular reasoning in Isabelle. Ph.D. thesis, University of Cambridge, Computer Laboratory (1999). Also Technical Report No. 470
15. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales: a sectioning concept for Isabelle. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) Theorem Proving in Higher Order Logics: TPHOLs'99, Nice, France. LNCS 1690, pp. 149–165. Springer (1999)
16. Milner, R., Tofte, M.: Commentary on Standard ML. MIT Press, Cambridge (1990)
17. Nipkow, T.: Order-sorted polymorphism in Isabelle. In: Huet, G., Plotkin, G. (eds.) Logical Environments, pp. 164–188. Cambridge University Press, Cambridge (1993)
18. Nipkow, T.: Verified efficient enumeration of plane graphs modulo isomorphism. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) Interactive Theorem Proving (ITP 2011). LNCS 6898, pp. 281–296. Springer (2011)
19. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An overview of the Scala programming language. Tech. Rep. IC/2004/64, École Polytechnique Fédérale de Lausanne (2004)
20. Java platform, standard edition 6 API specification. http://docs.oracle.com/javase/6/docs/api/ (2011)
21. Paulson, L.C.: The reflection theorem: a study in meta-theoretic reasoning. In: Voronkov, A. (ed.) Automated Deduction—CADE-18 International Conference. LNCS 2392, pp. 377–391. Springer (2002)
22. Schirmer, N., Wenzel, M.: State spaces—the locale way. Electr. Notes Theor. Comput. Sci. **254**, 161–179 (2009)
23. Soubiran, E.: Modular development of theories and name-space management for the Coq proof assistant. Ph.D. thesis, École Polytechnique (2012)
24. Wenzel, M.: Type classes and overloading in higher-order logic. In: Theorem Proving in Higher Order Logics. LNCS 1275, pp. 307–322 (1997). doi:10.1007/BFb0028402