Locales and Locale Expressions in Isabelle/Isar

Clemens Ballarin
Fakultät für Informatik
Technische Universität München
85748 Garching, Germany
ballarin@in.tum.de

Abstract

Locales provide a module system for the Isabelle proof assistant. Recently, locales have been ported to the new Isar format for structured proofs. At the same time, they have been extended by locale expressions, a language for composing locale specifications, and by structures, which provide syntax for algebraic structures. The present paper presents both and is suitable as a tutorial to locales in Isar, because it covers both basics and recent extensions, and contains many examples.

1 Overview

The present text is based on [3]. It was updated for for Isabelle2005, but does not cover locale interpretation.

Locales are an extension of the Isabelle proof assistant. They provide support for modular reasoning. Locales were initially developed by Kammüller [7] to support reasoning in abstract algebra, but are applied also in other domains — for example, bytecode verification [8].

Kammüller's original design, implemented in Isabelle99, provides, in addition to means for declaring locales, a set of ML functions that were used along with ML tactics in a proof. In the meantime, the input format for proof in Isabelle has changed and users write proof scripts in ML only rarely if at all. Two new proof styles are available, and can be used interchangeably: linear proof scripts that closely resemble ML tactics, and the structured Isar proof language by Wenzel [11]. Subsequently, Wenzel re-implemented locales for the new proof format. The implementation, available with Isabelle2002 or later, constitutes a complete re-design and exploits that both Isar and locales are based on the notion of context, and thus locales are seen as a natural extension of Isar. Nevertheless, locales can also be used with proof scripts: their use does not require a deep understanding of the structured Isar proof style.

At the same time, Wenzel considerably extended locales. The most important addition are locale expressions, which allow to combine locales more freely. Previously only linear inheritance was possible. Now locales support multiple inheritance through a normalisation algorithm. New are also structures, which provide special syntax for locale parameters that represent algebraic structures.

Unfortunately, Wenzel provided only an implementation but hardly any documentation. Besides providing documentation, the present paper is a high-level description of locales, and in particular locale expressions. It is meant as a first step towards the semantics of locales, and also as a base for comparing locales with module concepts in other provers. It also constitutes the base for future extensions of locales in Isabelle. The description was derived mainly by experimenting with locales and partially also by inspecting the code.

The main contribution of the author of the present paper is the abstract description of Wenzel's version of locales, and in particular of the normalisation algorithm for locale expressions (see Section 4.2). Contributions to the implementation are confined to bug fixes and to provisions that enable the use of locales with linear proof scripts.

Concepts are introduced along with examples, so that the text can be used as tutorial. It is assumed that the reader is somewhat familiar with Isabelle proof scripts. Examples have been phrased as structured Isar proofs. However, in order to understand the key concepts, including locales expressions and their normalisation, detailed knowledge of Isabelle is not necessary.

2   Locales: Beyond Proof Contexts

In tactic-based provers the application of a sequence of proof tactics leads to a proof state. This state is usually hard to predict from looking at the tactic script, unless one replays the proof step-by-step. The structured proof language Isar is different. It is additionally based on *proof contexts*, which are directly visible in Isar scripts, and since tactic sequences tend to be short, this commonly leads to clearer proof scripts.

Goals are stated with the **theorem** command. This is followed by a proof. When discharging a goal requires an elaborate argument (rather than the application of a single tactic) a new context may be entered (**proof**). Inside the context, variables may be fixed (**fix**), assumptions made (**assume**) and intermediate goals stated (**have**) and proved. The assumptions must be dischargeable by premises of the surrounding goal, and once this goal has been proved (**show**) the proof context can be closed (**qed**). Contexts inherit from surrounding contexts, but it is not possible to export from them (with exception of the proved goal); they "disappear" after the closing **qed**. Facts may have attributes — for example, identifying them as default to the simplifier or classical reasoner.

Locales extend proof contexts in various ways:

- Locales are usually *named*. This makes them persistent.

- Fixed variables may have *syntax*.

- It is possible to *add* and *export* facts.

- Locales can be combined and modified with *locale expressions*.

The Locales facility extends the Isar language: it provides new ways of stating and managing facts, but it does not modify the language for proofs. Its purpose is to support writing modular proofs.

3 Simple Locales

3.1 Syntax and Terminology

The grammar of Isar is extended by commands for locales as shown in Figure 1. A key concept, introduced by Wenzel, is that locales are (internally) lists of *context elements*. There are five kinds, identified by the keywords **fixes**, **constrains**, **assumes**, **defines** and **notes**.

At the theory level — that is, at the outer syntactic level of an Isabelle input file — **locale** declares a named locale. Other kinds of locales, locale expressions and unnamed locales, will be introduced later. When declaring a named locale, it is possible to *import* another named locale, or indeed several ones by importing a locale expression. The second part of the declaration, also optional, consists of a number of context element declarations.

A number of Isar commands have an additional, optional *target* argument, which always refers to a named locale. These commands are **theorem** (together with **lemma** and **corollary**), **theorems** (and **lemmas**), and **declare**. The effect of specifying a target is that these commands focus on the specified locale, not the surrounding theory. Commands that are used to prove new theorems will add them not to the theory, but to the locale. Similarly, **declare** modifies attributes of theorems that belong to the specified target. Additionally, for **theorem** (and related commands), theorems stored in the target can be used in the associated proof scripts.

The Locales package provides a *long goals format* for propositions stated with **theorem** (and friends). While normally a goal is just a formula, a long goal is a list of context elements, followed by the keyword **shows**, followed by the formula. Roughly speaking, the context elements are (additional) premises. For an example, see Section 4.4. The list of context elements in a long goal is also called *unnamed locale*.

Finally, there are two commands to inspect locales when working in interactive mode: **print_locales** prints the names of all targets visible in the current theory, **print_locale** outputs the elements of a named locale or locale expression.

The following presentation will use notation of Isabelle's meta logic, hence a few sentences to explain this. The logical primitives are universal quantification ($\bigwedge$), entailment ($\Longrightarrow$) and equality ($\equiv$). Variables (not bound variables) are

attr-name	::=	*name* | *attribute* | *name attribute*
locale-expr	::=	*locale-expr1* ("**+**" *locale-expr1*)*
locale-expr1	::=	(*qualified-name* | "**(**" *locale-expr* "**)**")
		(*name* [*mixfix*] | "**_**")*
fixes	::=	*name* ["**::**" *type*] ["**(**" **structure** "**)**" | *mixfix*]
constrains	::=	*name* "**::**" *type*
assumes	::=	[*attr-name* "**:**"] *proposition*
defines	::=	[*attr-name* "**:**"] *proposition*
notes	::=	[*attr-name* "**=**"] (*qualified-name* [*attribute*])$^+$
element	::=	**fixes** *fixes* (**and** *fixes*)*
	|	**constrains** *constrains* (**and** *constrains*)*
	|	**assumes** *assumes* (**and** *assumes*)*
	|	**defines** *defines* (**and** *defines*)*
	|	**notes** *notes* (**and** *notes*)*
element1	::=	*element*
	|	**includes** *locale-expr*
locale	::=	*element*$^+$
	|	*locale-expr* ["**+**" *element*$^+$]
in-target	::=	"**(**" **in** *qualified-name* "**)**"
theorem	::=	(**theorem** | **lemma** | **corollary**) [*in-target*] [*attr-name*]
theory-level	::=	. . .
	|	**locale** *name* ["**=**" *locale*]
	|	(**theorems** | **lemmas**)
		[*in-target*] [*attr-name* "**=**"] (*qualified-name* [*attribute*])$^+$
	|	**declare** [*in-target*] (*qualified-name* [*attribute*])$^+$
	|	*theorem proposition proof*
	|	*theorem element1** **shows** *proposition proof*
	|	**print_locale** *locale*
	|	**print_locales**

Figure 1: Locales extend the grammar of Isar.

sometimes preceded by a question mark. The logic is typed. Type variables are denoted by 'a, 'b etc., and $\Rightarrow$ is the function type. Double brackets $\llbracket$ and $\rrbracket$ are used to abbreviate nested entailment.

3.2 Parameters, Assumptions and Facts

From a logical point of view a *context* is a formula schema of the form

$$\bigwedge \mathtt{x}_1 \ldots \mathtt{x}_n . \ \llbracket \ \mathtt{C}_1; \ \ldots \ ; \mathtt{C}_m \ \rrbracket \Longrightarrow \ldots$$

The variables $\mathtt{x}_1, \ldots, \mathtt{x}_n$ are called *parameters*, the premises $\mathtt{C}_1, \ldots, \mathtt{C}_n$ *assumptions*. A formula $\mathtt{F}$ holds in this context if

(1) $$\bigwedge \mathtt{x}_1 \ldots \mathtt{x}_n . \ \llbracket \ \mathtt{C}_1; \ \ldots \ ; \mathtt{C}_m \ \rrbracket \Longrightarrow \mathtt{F}$$

is valid. The formula is called a *fact* of the context.

A locale allows fixing the parameters $\mathtt{x}_1$, ..., $\mathtt{x}_n$ and making the assumptions $\mathtt{C}_1$, ..., $\mathtt{C}_m$. This implicitly builds the context in which the formula $\mathtt{F}$ can be established. Parameters of a locale correspond to the context element **fixes**, and assumptions may be declared with **assumes**. Using these context elements one can define the specification of semigroups.

```
locale semi =
   fixes prod :: "['a, 'a] ⇒ 'a" (infixl "·" 70)
   assumes assoc: "(x · y) · z = x · (y · z)"
```

The parameter `prod` has a syntax annotation enabling the infix "·" in the assumption of associativity. Parameters may have arbitrary mixfix syntax, like constants. In the example, the type of `prod` is specified explicitly. This is not necessary. If no type is specified, a most general type is inferred simultaneously for all parameters, taking into account all assumptions (and type specifications of parameters, if present).[1]

Free variables in assumptions are implicitly universally quantified, unless they are parameters. Hence the context defined by the locale `semi` is

$$\bigwedge \mathtt{prod}. \ \llbracket \ \bigwedge \mathtt{x} \ \mathtt{y} \ \mathtt{z}. \ \mathtt{prod} \ (\mathtt{prod} \ \mathtt{x} \ \mathtt{y}) \ \mathtt{z} = \mathtt{prod} \ \mathtt{x} \ (\mathtt{prod} \ \mathtt{y} \ \mathtt{z}) \ \rrbracket \Longrightarrow \ldots$$

The locale can be extended to commutative semigroups.

```
locale comm_semi = semi +
   assumes comm: "x · y = y · x"
```

This locale *imports* all elements of `semi`. The latter locale is called the import of `comm_semi`. The definition adds commutativity, hence its context is

$$\bigwedge \mathtt{prod}. \ \llbracket \bigwedge \mathtt{x} \ \mathtt{y} \ \mathtt{z}. \ \mathtt{prod} \ (\mathtt{prod} \ \mathtt{x} \ \mathtt{y}) \ \mathtt{z} = \mathtt{prod} \ \mathtt{x} \ (\mathtt{prod} \ \mathtt{y} \ \mathtt{z});$$

$$\bigwedge \mathtt{x} \ \mathtt{y}. \ \mathtt{prod} \ \mathtt{x} \ \mathtt{y} = \mathtt{prod} \ \mathtt{y} \ \mathtt{x} \ \rrbracket \Longrightarrow \ldots$$

---

[1] Type inference also takes into account type constraints, definitions and import, as introduced later.

One may now derive facts — for example, left-commutativity — in the context of `comm_semi` by specifying this locale as target, and by referring to the names of the assumptions `assoc` and `comm` in the proof.

```
theorem (in comm_semi) lcomm:
  "x · (y · z) = y · (x · z)"
proof -
  have "x · (y · z) = (x · y) · z" by (simp add: assoc)
  also have "... = (y · x) · z" by (simp add: comm)
  also have "... = y · (x · z)" by (simp add: assoc)
  finally show ?thesis .
qed
```

In this equational Isar proof, "..." refers to the right hand side of the preceding equation. After the proof is finished, the fact `lcomm` is added to the locale `comm_semi`. This is done by adding a **notes** element to the internal representation of the locale, as explained the next section.

3.3   Locale Predicates and the Internal Representation of Locales

In mathematical texts, often arbitrary but fixed objects with certain properties are considered — for instance, an arbitrary but fixed group G — with the purpose of establishing facts valid for any group. These facts are subsequently used on other objects that also have these properties.

Locales permit the same style of reasoning. Exporting a fact F generalises the fixed parameters and leads to a (valid) formula of the form of equation (1). If a locale has many assumptions (possibly accumulated through a number of imports) this formula can become large and cumbersome. Therefore, Wenzel introduced predicates that abbreviate the assumptions of locales. These predicates are not confined to the locale but are visible in the surrounding theory.

The definition of the locale `semi` generates the *locale predicate* `semi` over the type of the parameter `prod`, hence the predicate's type is (`['a, 'a] ⇒ 'a) ⇒ bool`. Its definition is

```
semi_def:
     semi ?prod ≡ ∀x y z. ?prod (?prod x y) z = ?prod x (?prod y z).
```

In the case where the locale has no import, the generated predicate abbreviates all assumptions and is over the parameters that occur in these assumptions.

The situation is more complicated when a locale extends another locale, as is the case for `comm_semi`. Two predicates are defined. The predicate `comm_semi_axioms` corresponds to the new assumptions and is called *delta predicate*, the locale predicate `comm_semi` captures the content of all the locale, including the import. If a locale has neither assumptions nor import, no predicate is defined. If a locale has import but no assumptions, only the locale predicate is defined.

The Locales package generates a number of theorems for locale and delta predicates. All predicates have a definition and an introduction rule. Locale predi-

---

Theorems generated for the predicate `semi`.

```
semi_def:
      semi ?prod ≡ ∀x y z. ?prod (?prod x y) z = ?prod x (?prod y z)
semi.intro:
   (⋀x y z. ?prod (?prod x y) z = ?prod x (?prod y z)) ⟹ semi ?prod
```

Figure 2: Theorems for the locale predicate `semi`.

cates that are defined in terms of other predicates (which is the case if and only if the locale has import) also have a number of elimination rules (called *axioms*). All generated theorems for the predicates of the locales `semi` and `comm_semi` are shown in Figures 2 and 3, respectively.

Theorems generated for the predicate `comm_semi_axioms`.

```
comm_semi_axioms_def:
         comm_semi_axioms ?prod ≡ ∀x y. ?prod x y = ?prod y x
comm_semi_axioms.intro:
       (⋀x y. ?prod x y = ?prod y x) ⟹ comm_semi_axioms ?prod
```

Theorems generated for the predicate `comm_semi`.

```
comm_semi_def:   comm_semi ?prod ≡ semi ?prod ∧ comm_semi_axioms ?prod
comm_semi.intro:
         ⟦semi ?prod; comm_semi_axioms ?prod⟧ ⟹ comm_semi ?prod
comm_semi.axioms:
                   comm_semi ?prod ⟹ semi ?prod
             comm_semi ?prod ⟹ comm_semi_axioms ?prod
```

Figure 3: Theorems for the predicates `comm_semi_axioms` and `comm_semi`.

Note that the theorems generated by a locale definition may be inspected immediately after the definition in the Proof General interface [1] of Isabelle through the menu item "Isabelle/Isar>Show me ... >Theorems".

Locale and delta predicates are used also in the internal representation of locales as lists of context elements. While all **fixes** in a declaration generate internal **fixes**, all assumptions of one locale declaration contribute to one internal **assumes** element. The internal representation of `semi` is

fixes	prod :: "['a, 'a] ⇒ 'a"(**infixl**"·"70)
assumes	"semi prod"
notes	assoc : "?x · ?y · ?z = ?x · (?y · ?z)"

7

and the internal representation of `"comm_semi"` is

(2)

fixes	prod :: `"['a, 'a]` $\Rightarrow$ `'a"` (**infixl** `"·"` 70)
assumes	`"semi prod"`
notes	assoc : `"?x · ?y · ?z = ?x · (?y · ?z)"`
assumes	`"comm_semi_axioms prod"`
notes	comm : `"?x · ?y = ?y · ?x"`
notes	lcomm : `"?x · (?y · ?z) = ?y · (?x · ?z)"`

The **notes** elements store facts of the locales. The facts `assoc` and `comm` were added during the declaration of the locales. They stem from assumptions, which are trivially facts. The fact `lcomm` was added later, after finishing the proof in the respective **theorem** command above.

By using **notes** in a declaration, facts can be added to a locale directly. Of course, these must be theorems. Typical use of this feature includes adding theorems that are not usually used as a default rewrite rules by the simplifier to the simpset of the locale by a **notes** element with the attribute `[simp]`. This way it is also possible to add specialised versions of theorems to a locale by instantiating locale parameters for unknowns or locale assumptions for premises.

3.4   Definitions

Definitions were available in Kammüller's version of Locales, and they are in Wenzel's. The context element **defines** adds a definition of the form p x_1 ... $x_n \equiv$ t as an assumption, where p is a parameter of the locale (possibly an imported parameter), and t a term that may contain the x_i. The parameter may neither occur in a previous **assumes** or **defines** element, nor on the right hand side of the definition. Hence recursion is not allowed. The parameter may, however, occur in subsequent **assumes** and on the right hand side of subsequent **defines**. We call p *defined parameter*.

```
locale semi2 = semi +
  fixes rprod (infixl "⊙" 70)
  defines rprod_def: "rprod x y ≡ y · x "
```

This locale extends `semi` by a second binary operation `"⊙"` that is like `"·"` but with reversed arguments. The definition of the locale generates the predicate `semi2`, which is equivalent to `semi`, but no `semi2_axioms`. The difference between **assumes** and **defines** lies in the way parameters are treated on export.

3.5   Export

A fact is exported out of a locale by generalising over the parameters and adding assumptions as premises. For brevity of the exported theorems, locale predicates are used. Exported facts are referenced by writing qualified names consisting of the locale name and the fact name — for example,
`semi.assoc`:

```
    semi ?prod ⟹ ?prod (?prod ?x ?y) ?z = ?prod ?x (?prod ?y ?z).
```

8

Defined parameters receive special treatment. Instead of adding a premise for the definition, the definition is unfolded in the exported theorem. In order to illustrate this we prove that the reverse operation "⊙" defined in the locale `semi2` is also associative.

```
theorem (in semi2) r_assoc: "(x ⊙ y) ⊙ z = x ⊙ (y ⊙ z)"
  by (simp only: rprod_def assoc)
```

The exported fact is
```
semi2.r_assoc:
    semi2 ?prod ⟹ ?prod ?z (?prod ?y ?x) = ?prod (?prod ?z ?y) ?x.
```

The defined parameter is not present but is replaced by its definition. Note that the definition itself is not exported, hence there is no `semi2.rprod_def`.[2]

4 Locale Expressions

Locale expressions provide a simple language for combining locales. They are an effective means of building complex specifications from simple ones. Locale expressions are the main innovation of the version of Locales discussed here. Locale expressions are also reason for introducing locale predicates.

4.1 Rename and Merge

The grammar of locale expressions is part of the grammar in Figure 1. Locale names are locale expressions, and further expressions are obtained by *rename* and *merge*.

Rename. The locale expression $e\ q_1 \ldots q_n$ denotes the locale of e where parameters, in the order in which they are fixed, are renamed to q_1 to q_n. The expression is only well-formed if n does not exceed the number of parameters of e. Underscores denote parameters that are not renamed. Renaming by default removes mixfix syntax, but new syntax may be specified.

Merge. The locale expression $e_1 + e_2$ denotes the locale obtained by merging the locales of e_1 and e_2. This locale contains the context elements of e_1, followed by the context elements of e_2.

In actual fact, the semantics of the merge operation is more complicated. If e_1 and e_2 are expressions containing the same name, followed by identical parameter lists, then the merge of both will contain the elements of those locales only once. Details are explained in Section 4.2 below.

The merge operation is associative but not commutative. The latter is because parameters of e_1 appear before parameters of e_2 in the composite expression.

---

[2]The definition could alternatively be exported using a let-construct if there was one in Isabelle's meta-logic. Let is usually defined in object-logics.

Rename can be used if a different parameter name seems more appropriate —
for example, when moving from groups to rings, a parameter `G` representing
the group could be changed to `R`. Besides of this stylistic use, renaming is im-
portant in combination with merge. Both operations are used in the following
specification of semigroup homomorphisms.

```
locale semi_hom = comm_semi sum (infixl "⊕" 65) + comm_semi +
  fixes hom
  assumes hom: "hom (x ⊕ y) = hom x · hom y"
```

This locale defines a context with three parameters `sum`, `prod` and `hom`. The first
two parameters have mixfix syntax. They are associative operations, the first
of type `['a, 'a] ⇒ 'a`, the second of type `['b, 'b] ⇒ 'b`.

How are facts that are imported via a locale expression identified? Facts are
always introduced in a named locale (either in the locale's declaration, or by
using the locale as target in **theorem**), and their names are qualified by the
parameter names of this locale. Hence the full name of associativity in `semi` is
`prod.assoc`. Renaming parameters of a target also renames the qualifier of facts.
Hence, associativity of `sum` is `sum.assoc`. Several parameters are separated by
underscores in qualifiers. For example, the full name of the fact `hom` in the locale
`semi_hom` is `sum_prod_hom.hom`.

The following example is quite artificial, it illustrates the use of facts, though.

```
theorem (in semi_hom) "hom x · (hom y · hom z) = hom (x ⊕ (y ⊕ z))"
proof -
  have "hom x · (hom y · hom z) = hom y · (hom x · hom z)"
    by (simp add: prod.lcomm)
  also have "... = hom (y ⊕ (x ⊕ z))" by (simp add: hom)
  also have "... = hom (x ⊕ (y ⊕ z))" by (simp add: sum.lcomm)
  finally show ?thesis .
qed
```

Importing via a locale expression imports all facts of the imported locales, hence
both `sum.lcomm` and `prod.lcomm` are available in `hom_semi`. The import is dynamic
— that is, whenever facts are added to a locale, they automatically become
available in subsequent **theorem** commands that use the locale as a target, or
a locale importing the locale.

4.2   Normal Forms

Locale expressions are interpreted in a two-step process. First, an expression is
normalised, then it is converted to a list of context elements.

Normal forms are based on **locale** declarations. These consist of an import sec-
tion followed by a list of context elements. Let $\mathcal{I}(l)$ denote the locale expression
imported by locale l. If l has no import then $\mathcal{I}(l) = \varepsilon$. Likewise, let $\mathcal{F}(l)$ denote
the list of context elements, also called the *context fragment* of l. Note that
$\mathcal{F}(l)$ contains only those context elements that are stated in the declaration of
l, not imported ones.

Example 1. Consider the locales `semi` and `comm_semi`. We have $\mathcal{I}(\texttt{semi}) = \varepsilon$ and $\mathcal{I}(\texttt{comm\_semi}) = \texttt{semi}$, and the context fragments are

$$\mathcal{F}(\texttt{semi}) = \left[\begin{array}{ll} \textbf{fixes} & \texttt{prod} :: \texttt{"['a, 'a]} \Rightarrow \texttt{'a"} \ (\textbf{infixl} \ \texttt{"·"} \ 70) \\ \textbf{assumes} & \texttt{"semi prod"} \\ \textbf{notes} & \texttt{assoc} : \texttt{"?x · ?y · ?z = ?x · (?y · ?z)"} \end{array} \right] ,$$

$$\mathcal{F}(\texttt{comm\_semi}) = \left[\begin{array}{ll} \textbf{assumes} & \texttt{"comm\_semi\_axioms prod"} \\ \textbf{notes} & \texttt{comm} : \texttt{"?x · ?y = ?y · ?x"} \\ \textbf{notes} & \texttt{lcomm} : \texttt{"?x · (?y · ?z) = ?y · (?x · ?z)"} \end{array} \right] .$$

Let $\pi_0(\mathcal{F}(l))$ denote the list of parameters defined in the **fixes** elements of $\mathcal{F}(l)$ in the order of their occurrence. The list of parameters of a locale expression $\pi(e)$ is defined as follows:

$$\pi(l) = \pi(\mathcal{I}(l)) \,\overline{@}\, \pi_0(\mathcal{F}(l)), \text{ for named locale } l.$$
$$\pi(e\, q_1 \ldots q_n) = [q_1, \ldots, q_n, p_{n+1}, \ldots, p_m], \text{ where } \pi(e) = [p_1, \ldots, p_m].$$
$$\pi(e_1 + e_2) = \pi(e_1) \,\overline{@}\, \pi(e_2)$$

The operation $\overline{@}$ concatenates two lists but omits elements from the second list that are also present in the first list. It is not possible to rename more parameters than there are present in an expression — that is, $n \leq m$ — otherwise the renaming is illegal. If $q_i = \_$ then the ith entry of the resulting list is p_i.

In the normalisation phase, imports of named locales are unfolded, and renames and merges are recursively propagated to the imported locale expressions. The result is a list of locale names, each with a full list of parameters, where locale names occurring with the same parameter list twice are removed. Let $\mathcal{N}$ denote normalisation. It is defined by these equations:

$$\mathcal{N}(l) = \mathcal{N}(\mathcal{I}(l)) \,\overline{@}\, [l\, \pi(l)], \text{ for named locale } l.$$
$$\mathcal{N}(e\, q_1 \ldots q_n) = \mathcal{N}(e) \,[q_1 \ldots q_n / \pi(e)]$$
$$\mathcal{N}(e_1 + e_2) = \mathcal{N}(e_1) \,\overline{@}\, \mathcal{N}(e_2)$$

Normalisation yields a list of *identifiers*. An identifier consists of a locale name and a (possibly empty) list of parameters.

In the second phase, the list of identifiers $\mathcal{N}(e)$ is converted to a list of context elements $\mathcal{C}(e)$ by converting each identifier to a list of context elements, and flattening the obtained list. Conversion of the identifier $l\, q_1 \ldots q_n$ yields the list of context elements $\mathcal{F}(l)$, but with the following modifications:

- Rename the parameter in the ith **fixes** element of $\mathcal{F}(l)$ to q_i, $i = 1, \ldots, n$. If the parameter name is actually changed then delete the syntax annotation. Renaming a parameter may also change its type.

- Perform the same renamings on all occurrences of parameters (fixed variables) in **assumes**, **defines** and **notes** elements.

- Qualify names of facts by $q_1\_\ldots\_q_n$.

The locale expression is *well-formed* if it contains no illegal renamings and the following conditions on $\mathcal{C}(e)$ hold, otherwise the expression is rejected:

- Parameters in **fixes** are distinct;

- Free variables in **assumes** and **defines** occur in preceding **fixes**;[3]

- Parameters defined in **defines** must neither occur in preceding **assumes** nor **defines**.

4.3   Examples

Example 2.   We obtain the context fragment $\mathcal{C}(\texttt{comm\_semi})$ of the locale `comm_semi`. First, the parameters are computed.

$$\pi(\texttt{semi}) = [\texttt{prod}]$$

$$\pi(\texttt{comm\_semi}) = \pi(\texttt{semi})\,\overline{@}\,[\,] = [\texttt{prod}]$$

Next, the normal form of the locale expression `comm_semi` is obtained.

$$\mathcal{N}(\texttt{semi}) = [\texttt{semiprod}]$$

$$\mathcal{N}(\texttt{comm\_semi}) = \mathcal{N}(\texttt{semi})\,\overline{@}\,[\texttt{comm\_semi prod}] = [\texttt{semi prod}, \texttt{comm\_semi prod}]$$

Converting this to a list of context elements leads to the list (2) shown in Section 3.3, but with fact names qualified by `prod` — for example, `prod.assoc`. Qualification was omitted to keep the presentation simple. Isabelle's scoping rules identify the most recent fact with qualified name $x.a$ when a fact with name a is requested.

Example 3.   The locale expression `comm_semi sum` involves renaming. Computing parameters yields $\pi(\texttt{comm\_semi sum}) = [\texttt{sum}]$, the normal form is

$$\mathcal{N}(\texttt{comm\_semi sum}) = \mathcal{N}(\texttt{comm\_semi})[\texttt{sum/prod}] = [\texttt{semi sum}, \texttt{comm\_semi sum}]$$

and the list of context elements

fixes	sum :: "['a, 'a] $\Rightarrow$ 'a" (**infixl** "$\oplus$" 65)
assumes	"semi sum"
notes	sum.assoc : "(?x $\oplus$?y) $\oplus$?z = sum ?x (sum ?y ?z)"
assumes	"comm_semi_axioms sum"
notes	sum.comm : "?x $\oplus$?y = ?y $\oplus$?x"
notes	sum.lcomm : "?x $\oplus$ (?y $\oplus$?z) = ?y $\oplus$ (?x $\oplus$?z)"

---

[3]This restriction is relaxed for contexts obtained with **includes**, see Section 4.4.

Example 4. The context defined by the locale `semi_hom` involves merging two copies of `comm_semi`. We obtain the parameter list and normal form:

$$\pi(\texttt{semi\_hom}) = \pi(\texttt{comm\_semi sum} + \texttt{comm\_semi})\,\overline{@}\,[\texttt{hom}]$$

$$= (\pi(\texttt{comm\_semi sum})\,\overline{@}\,\pi(\texttt{comm\_semi}))\,\overline{@}\,[\texttt{hom}]$$

$$= ([\texttt{sum}]\,\overline{@}\,[\texttt{prod}])\,\overline{@}\,[\texttt{hom}] = [\texttt{sum}, \texttt{prod}, \texttt{hom}]$$

$$\mathcal{N}(\texttt{semi\_hom}) = \mathcal{N}(\texttt{comm\_semi sum} + \texttt{comm\_semi})\overline{@}$$

$$[\texttt{semi\_hom sum prod hom}]$$

$$= (\mathcal{N}(\texttt{comm\_semi sum})\,\overline{@}\,\mathcal{N}(\texttt{comm\_semi}))\overline{@}$$

$$[\texttt{semi\_hom sum prod hom}]$$

$$= ([\texttt{semi sum}, \texttt{comm\_semi sum}]\,\overline{@}\,[\texttt{semi prod}, \texttt{comm\_semi prod}])\overline{@}$$

$$[\texttt{semi\_hom sum prod hom}]$$

$$= [\texttt{semi sum}, \texttt{comm\_semi sum}, \texttt{semi prod}, \texttt{comm\_semi prod},$$

$$\texttt{semi\_hom sum prod hom}].$$

Hence $\mathcal{C}(\texttt{semi\_hom})$, shown below, is again well-formed.

fixes	sum :: "['a, 'a] $\Rightarrow$ 'a" (**infixl** "$\oplus$" 65)
assumes	"semi sum"
notes	sum.assoc : "(?x $\oplus$?y) $\oplus$?z = ?x $\oplus$ (?y $\oplus$?z)"
assumes	"comm_semi_axioms sum"
notes	sum.comm : "?x $\oplus$?y = ?y $\oplus$?x"
notes	sum.lcomm : "?x $\oplus$ (?y $\oplus$?z) = ?y $\oplus$ (?x $\oplus$?z)"
fixes	prod :: "['b, 'b] $\Rightarrow$ 'b" (**infixl** "$\cdot$" 70)
assumes	"semi prod"
notes	prod.assoc : "?x $\cdot$?y $\cdot$?z = ?x $\cdot$ (?y $\cdot$?z)"
assumes	"comm_semi_axioms prod"
notes	prod.comm : "?x $\cdot$?y = ?y $\cdot$?x"
notes	prod.lcomm : "?x $\cdot$ (?y $\cdot$?z) = ?y $\cdot$ (?x $\cdot$?z)"
fixes	hom :: "'a $\Rightarrow$ 'b"
assumes	"semi_hom_axioms sum"
notes	sum_prod_hom.hom : hom (x $\oplus$ y) = hom x $\cdot$ hom y

Example 5. In this example, a locale expression leading to a list of context elements that is not well-defined is encountered, and it is illustrated how normalisation deals with multiple inheritance. Consider the specification of monads (in the algebraic sense) and monoids.

locale `monad` =
  fixes prod :: "['a, 'a] $\Rightarrow$ 'a" (**infixl** "$\cdot$" 70) **and** one :: 'a ("1" 100)
  assumes l_one: "1 $\cdot$ x = x" **and** r_one: "x $\cdot$ 1 = x"

Monoids are both semigroups and monads and one would want to specify them as locale expression `semi + monad`. Unfortunately, this expression is not well-

formed. Its normal form

$$\mathcal{N}(\texttt{monad}) = [\texttt{monad prod}]$$
$$\mathcal{N}(\texttt{semi} + \texttt{monad}) = \mathcal{N}(\texttt{semi})\,\overline{@}\,\mathcal{N}(\texttt{monad}) = [\texttt{semi prod}, \texttt{monad prod}]$$

leads to a list containing the context element

$$\textbf{fixes } \texttt{prod} :: \texttt{"['a, 'a]} \Rightarrow \texttt{'a"} \;(\textbf{infixl } \texttt{"·"}\; 70)$$

twice and thus violating the first criterion of well-formedness. To avoid this problem, one can introduce a new locale `magma` with the sole purpose of fixing the parameter and defining its syntax. The specifications of semigroup and monad are changed so that they import `magma`.

locale magma = **fixes** prod (**infixl** "·" 70)

locale semi' = magma + **assumes** assoc: "(x · y) · z = x · (y · z)"
locale monad' = magma + **fixes** one ("1" 100)
  assumes l_one: "1 · x = x" **and** r_one: "x · 1 = x"

Normalisation now yields

$$\mathcal{N}(\texttt{semi'} + \texttt{monad'}) = \mathcal{N}(\texttt{semi'})\,\overline{@}\,\mathcal{N}(\texttt{monad'})$$
$$= (\mathcal{N}(\texttt{magma})\,\overline{@}\,[\texttt{semi' prod}])\,\overline{@}\,(\mathcal{N}(\texttt{magma})\,\overline{@}\,[\texttt{monad' prod}])$$
$$= [\texttt{magma prod}, \texttt{semi' prod}]\,\overline{@}\,[\texttt{magma prod}, \texttt{monad' prod}])$$
$$= [\texttt{magma prod}, \texttt{semi' prod}, \texttt{monad' prod}]$$

where the second occurrence of `magma prod` is eliminated. The reader is encouraged to check, using the **print_locale** command, that the list of context elements generated from this is indeed well-formed.

It follows that importing parameters is more flexible than fixing them using a context element. The Locale package provides the predefined locale `var` that can be used to import parameters if no particular mixfix syntax is required. Its definition is

$$\textbf{locale } \texttt{var} = \textbf{fixes } \texttt{x\_}$$

The use of the internal variable `x_` enforces that the parameter is renamed before being used, because internal variables may not occur in the input syntax. Using `var`, the locale `magma` could have been replaced by the locale expression

$$\texttt{var prod }(\textbf{infixl } \texttt{"·"}\; 70)$$

in the above locale declarations.

4.4 Includes

The context element **includes** takes a locale expression e as argument. It can only occur in long goals, where it adds, like a target, locale context to the proof context. Unlike with targets, the proved theorem is not stored in the locale. Instead, it is exported immediately.

```
theorem lcomm2:
  includes comm_semi shows "x · (y · z) = y · (x · z)"
proof -
  have "x · (y · z) = (x · y) · z" by (simp add: assoc)
  also have "... = (y · x) · z" by (simp add: comm)
  also have "... = y · (x · z)" by (simp add: assoc)
  finally show ?thesis .
qed
```

This proof is identical to the proof of `lcomm`. The use of **includes** provides the same context and facts as when using `comm_semi` as target. On the other hand, `lcomm2` is not added as a fact to the locale `comm_semi`, but is directly visible in the theory. The theorem is

$$comm\_semi \ ?prod \implies ?prod \ ?x \ (?prod \ ?y \ ?z) = ?prod \ ?y \ (?prod \ ?x \ ?z).$$

Note that it is possible to combine a target and (several) **includes** in a goal statement, thus using contexts of several locales but storing the theorem in only one of them.

5 Structures

The specifications of semigroups and monoids that served as examples in previous sections modelled each operation of an algebraic structure as a single parameter. This is rather inconvenient for structures with many operations, and also unnatural. In accordance to mathematical texts, one would rather fix two groups instead of two sets of operations.

The approach taken in Isabelle is to encode algebraic structures with suitable types (in Isabelle/HOL usually records). An issue to be addressed by locales is syntax for algebraic structures. This is the purpose of the **(structure)** annotation in **fixes**, introduced by Wenzel. We illustrate this, independently of record types, with a different formalisation of semigroups.

Let `'a semi_type` be a not further specified type that represents semigroups over the carrier type `'a`. Let `s_op` be an operation that maps an object of `'a semi_type` to a binary operation.

```
typedecl 'a semi_type
consts s_op :: "['a semi_type, 'a, 'a] ⇒ 'a" (infixl "⋆ı" 70)
```

Although `s_op` is a ternary operation, it is declared infix. The syntax annotation contains the token $\imath$ (`\<index>`), which refers to the first argument. This syntax is only effective in the context of a locale, and only if the first argument is a

structural parameter — that is, a parameter with annotation **(structure)**. The token has the effect of subscripting the parameter — by bracketing it between `\<^bsub>` and `\<^esub>`. Additionally, the subscript of the first structural parameter may be omitted, as in this specification of semigroups with structures:

```
locale comm_semi' =
  fixes G :: "'a semi_type" (structure)
  assumes assoc: "(x ⋆ y) ⋆ z = x ⋆ (y ⋆ z)" and comm: "x ⋆ y = y ⋆ x"
```

Here `x ⋆ y` is equivalent to `x ⋆`$_{\text{G}}$` y` and abbreviates `s_op G x y`. A specification of homomorphisms requires a second structural parameter.

```
locale semi'_hom = comm_semi' + comm_semi' H +
  fixes hom
  assumes hom: "hom (x ⋆ y) = hom x ⋆`H hom y"
```

The parameter `H` is defined in the second **fixes** element of $\mathcal{C}($`semi'_comm`$)$. Hence `⋆`$_{\text{H}}$ abbreviates `s_op H x y`. The same construction can be done with records instead of an *ad-hoc* type.

```
record 'a semi = prod :: "['a, 'a] ⇒ 'a" (infixl "·`ı" 70)
```

This declares the types `'a semi` and `('a, 'b) semi_scheme`. The latter is an extensible record, where the second type argument is the type of the extension field. For details on records, see [10] Chapter 8.3.

```
locale semi_w_records = struct G +
  assumes assoc: "(x · y) · z = x · (y · z)"
```

The type `('a, 'b) semi_scheme` is inferred for the parameter `G`. Using subtyping on records, the specification can be extended to groups easily.

```
record 'a group = "'a semi" +
  one :: "'a" ("1`ı" 100)
  inv :: "'a ⇒ 'a" ("inv`ı _" [81] 80)
locale group_w_records = semi_w_records +
  assumes l_one: "1 · x = x" and l_inv: "inv x · x = 1"
```

Finally, the predefined locale

$$\textbf{locale } \textit{struct} = \textbf{fixes } \texttt{S\_} \textbf{ (structure)}.$$

is analogous to `var`. More examples on the use of structures, including groups, rings and polynomials can be found in the Isabelle distribution in the session HOL-Algebra.

6   Conclusions and Outlook

Locales provide a simple means of modular reasoning. They enable to abbreviate frequently occurring context statements and maintain facts valid in these contexts. Importantly, using structures, they allow syntax to be effective only

in certain contexts, and thus to mimic common practice in mathematics, where notation is chosen very flexibly. This is also known as literate formalisation [2]. Locale expressions allow to duplicate and merge specifications. This is a necessity, for example, when reasoning about homomorphisms. Normalisation makes it possible to deal with diamond-shaped inheritance structures, and generally with directed acyclic graphs. The combination of locales with record types in higher-order logic provides an effective means for specifying algebraic structures: locale import and record subtyping provide independent hierarchies for specifications and structure elements. Rich examples for this can be found in the Isabelle distribution in the session HOL-Algebra.

The primary reason for writing this report was to provide a better understanding of locales in Isar. Wenzel provided hardly any documentation, with the exception of [12]. The present report should make it easier for users of Isabelle to take advantage of locales.

The report is also a base for future extensions. These include improved syntax for structures. Identifying them by numbers seems unnatural and can be confusing if more than two structures are involved — for example, when reasoning about universal properties — and numbering them by order of occurrence seems arbitrary. Another desirable feature is *instantiation*. One may, in the course of a theory development, construct objects that fulfil the specification of a locale. These objects are possibly defined in the context of another locale. Instantiation should make it simple to specialise abstract facts for the object under consideration and to use the specified facts.

A detailed comparison of locales with module systems in type theory has not been undertaken yet, but could be beneficial. For example, a module system for Coq has recently been presented by Chrzaszcz [5, 6]. While the latter usually constitute extensions of the calculus, locales are a rather thin layer that does not change Isabelle's meta logic. Locales mainly manage specifications and facts. Functors, like the constructor for polynomial rings, remain objects of the logic.

References

[1] David Aspinall. Proof general: A generic tool for proof development. In Susanne Graf and Michael I. Schwartzbach, editors, *TACAS 2000*, LNCS 1785, pages 38–42. Springer, 2000.

[2] Anthony Bailey. *The machine-checked literate formalisation of algebra in type theory*. PhD thesis, University of Manchester, January 1998.

[3] Clemens Ballarin. Locales and locale expressions in Isabelle/Isar. In Berardi et al. [4], pages 34–50.

[4] Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors. *Types for Proofs and Programs, TYPES 2003, Torino, Italy*, LNCS 3085. Springer, 2004.

[5] Jacek Chrząszcz. Implementing modules in the Coq system. In David Basin and Burkhart Wolff, editors, *TPHOLs 2003*, LNCS 2758, pages 270–286. Springer, 2003.

[6] Jacek Chrząszcz. Modules in Coq are and will be correct. In Berardi et al. [4], pages 130–136.

[7] Florian Kammüller. Modular reasoning in Isabelle. In David McAllester, editor, *CADE 17*, LNCS 1831, pages 99–114. Springer, 2000.

[8] Gerwin Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[9] Tobias Nipkow. Structured proofs in Isar/HOL. In H. Geuvers and F. Wiedijk, editors, *TYPES 2002*, LNCS 2646, pages 259–278. Springer, 2003.

[10] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

[11] Markus Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universität München, 2002. Electronically published as http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html.

[12] Markus Wenzel. Using locales in Isabelle/Isar. Part of the Isabelle2003 distribution, file src/HOL/ex/Locales.thy. Distribution of Isabelle available at http://isabelle.in.tum.de, 2002.

[13] Markus Wenzel. The Isabelle/Isar reference manual. Part of the Isabelle2003 distribution, available at http://isabelle.in.tum.de, 2003.