Technical Report

UCAM-CL-TR-449
ISSN 1476-2986

Number 449

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Locales

A sectioning concept for Isabelle

Florian Kammüller, Markus Wenzel

October 1998

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Locales are a means to define local scopes for the interactive proving process of the theorem prover Isabelle. They delimit a range in which fixed assumption are made, and theorems are proved that depend on these assumptions. A locale may also contain constants defined locally and associated with pretty printing syntax.

Locales can be seen as a simple form of modules. They are similar to sections as in AUTOMATH or Coq. Locales are used to enhance abstract reasoning and similar applications of theorem provers. This paper motivates the concept of locales by examples from abstract algebraic reasoning. It also discusses some implementation issues.

Contents

1 Motivation

In interactive theorem proving it is desirable to get as close as possible to the convenience of paper proof style making developments more comprehensible and self declaring. In mathematical reasoning assumptions and definitions are handled in a casual way. That is, a typical mathematical proof assumes propositions for one proof or a whole section of proofs and local to these assumption definitions are made that depend on those assumptions. The present paper introduces a concept of *locales* for Isabelle [Pau94] that aims to support the described processes of local assumptions and definition. Locales implement a sectioning device similar to that in AUTOMATH [dB80] or Coq [Dow90], but with optional pretty printing syntax and dependent local definitions added.

In mathematical proofs we often want to define abbreviations for big expressions to enhance the readability of the proof. These abbreviations might implicitly refer to terms which are arbitrary but fixed values for the entire proof. Surely, Isabelle's pretty printing and definition possibilities are mostly sufficient for this purpose. But there are still examples where a definition in a theory is a too strong means in the sense that the syntactical constants used for abbreviations are of no global significance. Definitions in an Isabelle theory are visible everywhere.

In the case study of Sylow's theorem [KP97] we came across several such local definitions. There, we define a set $\mathcal{M}$ as $\{S \subseteq G_{cr} \mid order(S) = p^\alpha\}$ where G, p, and α are arbitrary but fixed values with certain properties. This is just for one single big proof, and has no general purpose whatsoever. The formula does not even occur in the main proposition. Still, in Isabelle as it is, we only have the choice of spelling this term out wherever it occurs, or defining it on the global level, which is rather unnatural. On top of that we need in a global definition to parameterize this example over all variables of the right-hand side. In our example we would get something like $\mathcal{M}(G, p, \alpha)$ which is almost as bad as the original formula. This second drawback is because we cannot have something like *local constants*, here G, p, and α.

Subsequently we explain a simple approach towards sectioning for the theorem prover Isabelle. In Section 2 we describe the locale concept and address issues of opening and closing of locales. We present aspects concerning concrete syntax, including a means for local definitions. We continue in Section 3 with the fundamental operations on locales and their features. Section 4 describes the implementation of the ideas. We give a simple example illustrating an application from algebra in Section 5. Finally, we discuss some aspects of locales in Section 6. The Appendix summarizes Isabelle commands and syntax for locales.

2 Locales — the Concept

Locales delimit a scope of *fixed variables*, *local assumptions*, and *local definitions*. Theorems that are proved in the context of locales may make use of these local entities. The result will then depend on the additional hypotheses, while proper local definitions are eliminated.

A locale consists of a set of constants (with optional pretty printing syntax), rules and definitions. Defined as named objects of an Isabelle theory, locales can be invoked later in any proof session. By virtue of such an invocation, any locale rules and definitions are turned into theorems that may be applied in proof procedures like any other theorem. Similarly the definitions, like usual Isabelle definitions, abbreviate longer terms. But, the rules and definitions are only local to the scope that is defined by a locale.

Theorems proved in the scope of a locale can be exported to the surrounding theory context. In that case, rules employed for the proof become *meta-level assumptions* of the exported theorem. For the case of actual definitions, these hypotheses are eliminated via generalization and reflexivity. So, the theorem becomes a usual theorem of the Isabelle theory that contains that locale.

Subsequently we explain several aspects of locales. There are basically two ideas that form the concept of locales: one is the possibility to state local assumptions, and the other one is to make local definitions which can depend on these assumptions, and may use pretty printing. With those two main ideas the notion of a locale constant is strongly connected. We explain other specific features of locales in this section.

2.1 Locale Rules

To explain what locales are it is best to describe the main characteristics of Isabelle that lead to this concept and are the basis of the realization. The feature of Isabelle that builds the basis for the locale rules is Isabelle's concept of meta-assumptions.

In Isabelle each theorem may depend upon meta-assumptions. The theorem that ϕ holds under the meta-assumptions $\phi_1, \ldots, \phi_n$ is written as

$$\phi\,[\phi_1, \ldots, \phi_n]$$

The first main aspect of locales is to build up a local scope, in which a set of rules, the locale rules, are valid. The local rules are realized by using Isabelle's meta-assumptions as an assumption stack. Logically, a locale is a conjunction of meta-assumptions; the conjuncts are the locale rules. Opening the locale corresponds to assuming this conjunction.

In Isabelle as it is, a meta-assumption can be introduced in proofs at any time, but by the end of the proof, Isabelle would complain about extraneous

hypotheses. With the locale concept added to Isabelle, locale rules become meta-assumptions when the locale is invoked. A theorem proved in the scope of some locale, can use these rules. The result extraction process at the end of a proof has been modified accordingly to cope with this: the additional premises stemming from the locale are entailed in the conjunction; the proof is admitted.

2.2 Locale Constants

There is a notion of a *locale constant* that is integral part of the locale concept. A locale implements the idea of "arbitrary but fixed" that is used in mathematical proofs. We can assume certain terms as fixed for a certain section of proofs, and we can state further rules or define other terms depending on them. The locale constants may be viewed from the outside as parameters, because they become universally quantified variables, when a result theorem is exported.

Technically, locale constants behave like logical constants while the locale is open. In particular, they may be subject to the standard Isabelle pretty printing scheme, e.g. equipped with infix syntax.

A locale corresponds to a certain extent to modules in a theorem prover, with some notable restrictions of declaring items, though. In particular, a locale may not contain type declarations and the constants are not persistent. The outside view of locales is realized in a different way. Instead of presenting the entire locale similar to a parameterized module that can be instantiated, one can export theorems from inside the locale.

2.3 Local Definition and Pretty Printing

A major reason for having a sectioning device like locales are user requirements to make temporary abbreviations in the course of a proof development. As pointed out in Section 1, there are large formulas that are used in proofs and do not have a global significance. Moreover, they might not even occur in the final conjecture of the theorem that we want to use. So, conceptually, the definition of such proof terms is not a persistent definition. Nevertheless, we want to use such definitions to make the theorems readable, and the proof process clear. Hence, one aspect is the locality of these definitions. The other aspect, as illustrated by the introductory example as well, is that the local definitions might depend on terms that are constants in a certain scope. For example, we want to write $\mathcal{M}$ only, not a notation like $\mathcal{M}(G, p, \alpha)$ as it would be necessary, if we wanted to refer to the terms that form the other premises of the Sylow theorem.

Another common thing in abstract algebra are formulas which are not so big, but suppress implicit information, e.g. we write Ha for the coset of a with respect to the subset H of a group G. Since the group G containing

this coset is a parameter to this definition we would have to define something like `coset G H a`. This is partly the same problem as with the parameters of the definition $\mathcal{M}$. Note that the normal pretty printing mechanism would not work here because of additional arguments in the definitions.

In a locale where `G` is an arbitrary but fixed group for a whole series of theorems we can have a syntax like `H #> a` instead of `coset G H a`. We can create a simple definition mechanism for concrete syntax which implements the concept of a *locale constant* for which we can define pretty printing syntax. The idea of the locale constant is to give a locale a scope such that inside the locale a free variable can be considered as a constant. Then we can define pretty printing syntax for this constant, but this syntax only exists as long as the locale is open. Viewed from outside the locale, this syntax does not exist. The theorems proved inside the locale using the syntax are global theorems with the syntactical abbreviations unfolded and the locale constants replaced by free variables.

In a locale where we want to reason about a group `G` and its right cosets, we declare `G` as a locale constant. Then we can define another locale constant `#>`, and define this in terms of the underlying theory of groups where the operation `r_coset` is known.

```
rcos_def  "H #> x == r_coset G H x"
```

If the locale containing this definition is open, we can use the convenient syntax `H #> x` for right cosets, and it is defined as the sound operation of right cosets with the parameter `G` fixed for the current scope. If we finish a theorem and want to use it as a general result, we can *export* it. Then, the locale constant `G` will be turned into a universally quantified variable, and the definition will be expanded to the base definition of right cosets.

3 Operations on Locales

Locales are introduced as named syntactic objects within Isabelle theories. They can then be opened in any theory that contains the theory they are defined for.

3.1 Defining Locales

The ideas of locale definitions, rules, and constants can be combined together to realize a sectioning concept. Thereby, we attain a mechanism that constitutes a local theory mechanism. To adjust this rather dynamic idea of definition and declaration to the declarative style of Isabelle's theory mechanism, we integrate the definition of locales into the theories as another language element of Isabelle theory files. The concrete syntax of locale definitions is demonstrated by example below.

Locale group assumes the definition of groups as a set of records [Kam98b] as follows (cf. Section 5):

```
locale group =
  fixes
    G          :: "'a grouptype"
    e          :: "'a"
    binop      :: "'a => 'a => 'a"          (infixr "#" 80)
    inv        :: "'a => 'a"                ("_ ^-1" [90] 91)
  assumes
    Group_G    "G: Group"
  defines
    e_def      "e == unit G"
    binop_def "x # y == bin_op G x y"
    inv_def    "x ^-1 == inverse G x"
```

Above Isabelle theory file section introduces a locale for abstract reasoning about groups.

The subsection introduced by the keyword `fixes` declares the locale constants in a way that closely resembles Isabelle's global `consts` declaration. In particular, there may be an optional pretty printing syntax for the locale constants.

The subsequent `assumes` specifies the locale rules. Their way of definition is the same as for Isabelle `rules`. They admit the statement of the local assumptions we want to state about some of the locale constants. Here, we assume that the locale constant `G` is a member of the set `Group`, i.e. is a group.

Finally, the `defines` part of the locale introduces the definitions that shall be available in this locale. In these definitions we can already use the syntax of the locale constants in the `fixes` subsection and define these locale constants in terms of the underlying theory of groups. Note, that we can define here now in a style that is normally impossible in Isabelle. A definition like

```
e_def  "e == unit G"
```

would appear to contain an additional free variable `G` on the right hand side. In the scope of locale group, though, `G` is bound, and `e_def` becomes a legal *dependent definition.*

Note also, that there are two different kinds of locale constants, one that is used merely in the locale rules, and one that is declared to serve for definitions.

3.2 Invocation and Scope

After definition, locales may be opened and closed in a block-structured manner. The list of currently active locales is called *scope.*

3.2.1 Opening

Locales are *opened* or invoked at points where we want to prove theorems concerning the locale. Inside the scope of an open locale, theorems can use its definitions and rules. An invocation assumes the rules of the locale. The rules get names, so they can be accessed individually. Opening a locale means making its assumptions visible. The primitive Isabelle rule `Thm.assume` applied to a term `P` returns the theorem `P [ P ]`, representing the tautology $P \Rightarrow P$. The term `P` becomes a meta-assumption. The rules are stored in this form in a locale. Opening a locale means activating these meta level assumptions `P [ P ]` for all locale rules `P`.

3.2.2 Scope

Invocation corresponds to assuming an instance of the locale, i.e. we assume the existence of the locale constants fulfilling the locale's rules. The definitions are valid and can be used throughout the lifetime of the locale, i.e. until it is closed again. The opened locales form a scope which lives until all locales are explicitly closed. At any time there can be more than one locale open. The scope manages currently open locales. It contains all assumptions and definitions of opened locales.

3.2.3 Closing

Closing means to cancel the last opened locale, pushing it off the scope. Theorems proved during the life cycle of this locale will also vanish, unless they have been explicitly exported, as described below.

3.2.4 Export of Theorems

Exported theorems become theorems at the global level, that is, the current theory context. Locale rules that have been used in the proof of an exported theorem inside the locale are carried by the global form of the theorem as its individual meta-assumptions. The locale constants are universally quantified variables in these theorems, hence such theorems can be instantiated individually. Definitions become unfolded; locale constants that were merely used for definitions vanish.

3.3 Other Aspects

3.3.1 Instances

We do not provide a general instantiation operation for locales. Nevertheless, theorems proved within some scope of locales can be used outside at any instance, without the need to instantiate the locale first. So, the (conceptual) instantiation of the locale is split up into the instantiation of the individual

theorems. This seems appropriate, because in the kinds of proofs we have in mind, one is only interested in a few of the theorems proved in a locale on the global level.

Still, there is a device to *rename* locale constants (cf. Section 3.3.4). Although this is just a purely syntactical operation it represents in some sense an instantiation.

3.3.2 Proofs

The theorems proved inside a locale can use the locale rules as axioms, accessing them by their names. The used locale rules are held as meta-assumptions. Hence, subgoals created in a proof matching locale assumptions are solved automatically. Theorems proved in a locale can be exported as theorems of the global level under the assumption of the locale rules they use. If a theorem needs only a certain portion of the locale's assumptions, only those will be mentioned in the global form of the theorem.

3.3.3 Polymorphism

Isabelle's meta-logic is based on a version of Church's Simple Theory of Types with schematic polymorphism. Free type variables are implicitly universally quantified at the outer level of declarations and statements. For example, a constant declaration

```
consts  f :: 'a => 'a
```

basically means that `f` has type $\forall\alpha.\alpha \Rightarrow \alpha$. So, if there is a subsequent constant declaration using the same type variable α, those are different type variables. That is, they can be instantiated differently in the same context.

Now, for locales the scope of polymorphic variables is wider. The quantification of the type variables is placed at the outside of the locale. So, variables with equal names are actually the same variables. On the one hand, this difference allows us to define sharing of type domains of operators at an abstract level. This is important for the algebraic reasoning that we are focusing on. On the other hand, locale definitions are not polymorphic within the locale's scope.

3.3.4 Augmenting Locales

We consider some further operations on locales that are not yet implemented in Isabelle, but will be in due course.

One locale can *extend* another locale. This is written as

```
locale foo = bar + ...
```

The dots ... stand for the locale sections `fixes`, `assumes`, and `defines`.

Locales can also be composed by *merging* already existing locales. Here, we compose `foo` by merging `bar1` and `bar2`.

```
locale foo = bar1 + bar2
```

Finally, we can *rename* locale constants. This can be very useful if we want to have more than one instance of the same locale in the scope. Here, `bar` is created from `foo` by renaming all occurrences of locale constant `c` in `bar` by `r`.

```
locale foo = bar [r/c]
```

4 Implementation issues

In this section we briefly highlight some of the implementation issues of locales. In particular, we outline some key features of recent versions of Isabelle that may not yet be folklore in the realm of Isabelle hacking.

Extending the Isabelle theory language by any kind of new mechanism typically consists of the following stages:

(1) providing private theory data,

(2) writing a theory extension function,

(3) installing a new theory section parser.

For our particular mechanism of locales, we also have to adapt parts of the Isabelle goal package to cope with scopes as already hinted in the previous section:

(4) modify term read and print functions,

(5) modify proof result operation.

Subsequently, we briefly describe the idea of locales as theory data, and then carry on to illustrate the implementation. The reader not interested in the internals of Isabelle might well skip the present section and continue with the application example in Section 5.

4.1 Theory Data for Locales

Basically, any new theory extension mechanism boils down to already existing ones, like constant declarations and definitions. For example, the standard Isabelle/HOL `datatype` package could be seen just as a generator of huge amounts of types, constants, and theorems. This pure approach to theory extension has a severe drawback, though. It is like *compiling down* information, losing most of the original source level structure. E.g. it would

be extremely hard to figure out any datatype specification (the set of constructors, say) from the soup of generated primitive extensions left behind in the theory.

The generic theory data concept, introduced in Isabelle98 and improved in later releases, offers a solution to this problem by enabling users to write packages in a *structure preserving* way. Thus one may declare named slots of *any* ML type to be stored within Isabelle theory objects. This way new extensions mechanisms may deposit suitable source-level information as needed later for any advanced operation.

Picking up the `datatype` example again, there may be a generic induction tactic, that figures out the actual rule to apply from the type of some variable. This would be accomplished by doing a lookup in the private `datatype` theory data, containing full information about any HOL type represented as inductive datatype.

Note that traditionally in the LCF system approach, such data would be stored as values or structures within the ML runtime environment, with only very limited means to access this later from other ML programs. Breaking with this tradition, the current Isabelle approach is considered more powerful, internalizing generic data as first class components of theory objects.

The functor `TheoryDataFun` that is part of Isabelle/Pure provides a *fully type-safe* interface to generic data slots[1]. The argument structure is expected to have the following signature:

```
signature THEORY_DATA_ARGS =
sig
  val name: string
  type T
  val empty: T
  val prep_ext: T -> T
  val merge: T * T -> T
  val print: Sign.sg -> T -> unit
end
```

Here `name` and `T` specify the new data slot by name and ML type, while `empty` gives its initial value. The `prep_ext` operation is intended to prepare your data for extension to a new theory node. Usually, this is just the identity function; occasionally one might want to reset some tables, or copy reference values. The `merge` operation is called when theories are joined, as should be your data. Finally, `print` shall display the theory data in some human readable way; the function obtains the current signature as additional argument.

[1]The curious expert may note that this is achieved by invoking most of the black-magic offered by Standard ML: exception constructors for introducing new types, private references as tags for identification and authorization, and functors for hiding the whole mess. We see that ML is for the Real Programmer, after all!

The result structure of `TheoryDataFun` is as follows:

```
signature THEORY_DATA =
sig
  type T
  val init: theory -> theory
  val print: theory -> unit
  val get_sg: Sign.sg -> T
  val get: theory -> T
  val put: T -> theory -> theory
end
```

The new data slot has to be made known to theories (only once) via above `init` operation. Afterwards any derived theory knows about the `print`, `get` and `put` functions as given above.

For locales, we have defined a data slot called "`Pure/locales`" that contains a table of all defined locales, together with their hierarchical name space. There is also a reference variable of the current scope, containing a list of locales identifiers.

Employing this private theory data slot, we have implemented the actual locale definition mechanism on top of usual Isabelle primitives. The ultimate result is function `Locale.add_locale`, which is the actual theory extender that does all the hard work:

```
val add_locale: ...  -> theory -> theory
```

Here the dots refer to the locale specification, including `fixes`, `assumes`, `defines` arguments. After preparing these by parsing, type checking etc., we store the information via above `get` and `put` operations in our theory data slot. We also emit some other Isabelle primitives to extend the theory's syntax, for example.

4.2 Theory Section Parser

Another part of the scheme of adding a theory section to Isabelle is to provide a parsing method. The actual parser `locale_decl` for the locale definitions is just one ML-term constructed from parser combinators as are well-known in the functional programming community. It resides in the file `thy_parse.ML` of Isabelle/Pure.

Via `ThySyn.add_syntax` we can now plug our `Locale.add_locale` function into the main theory parser.

```
val _ = ThySyn.add_syntax
  ["fixes", "assumes", "defines"]
  [(section "locale" "|> Locale.add_locale" locale_decl)];
```

Above we declare new keywords `fixes` etc., and state that `locale` shall introduce a theory file section that is taken care of by the `locale_decl` parser (syntax) and the `Locale.add_locale` theory extension function (semantics).

4.3 Interface

Apart from the actual theory extension function discussed above, there are a few more things to be done for the locale implementation.

The file `locale.ML` (now part of Isabelle/Pure) also contains this creation of the interface. It also contains most of the other changes to Isabelle that implement locales. The function `add_locale` is the main function in this file. It adds a locale to a theory, once it is read in by the appropriate theory parser routine.

The locale constants are realized by developing two layers of terms for them. Basically they are `Frees` for which one can define mixfix syntax. So, we treat a locale constant `c` like `Free c` but produce for it a constant `\<^locale>c`. The pretty printing syntax is assigned to the copy `\<^locale>c`. Terms occurring during proofs really contain the `Free c`, but for printing we use the constant `\<^locale>c`. For reading, we add translation functions permanently to the theory containing the locale.

The read and print functions of terms have to be adjusted to locales: if a locale is open, we want any term that is read in, to respect the bindings of types and terms of that locale. We augment the basic function `Thm.read_cterm` such that it checks if a locale is open, i.e. if the current scope is nonempty, and then bases the type inference on this information. Similarly, we adjust the function `pretty_term`. It is used to print proof states. Here, we replace now all `Free` constants `c` by their double `\<^locale>c`. Then the printing produces the pretty form.

Isabelle's goal package has been adopted to use these modified read and print functions.

5 An Example from Abstract Algebra

We illustrate the use of the implementation by examples with the abstract algebraic structure of groups [Kam98a]. We use a representation of groups that we found in [Kam98b] to be the most appropriate for abstract algebraic structures. The base theory is `Group`. It contains the theory for groups. We define a basic pattern type for the simple structure of groups, by an extensible record definition [NW98][2].

```
record 'a grouptype =
  carrier  :: "'a set"          ("_ .<cr>"  [10] 10)
  bin_op   :: "['a, 'a] => 'a" ("_ .<f>"  [10] 10)
  inverse  :: "'a => 'a"        ("_ .<inv>"  [10] 10)
  unit     :: "'a"              ("_ .<e>"  [10] 10)
```

[2]We use pretty printing facilities for records that are not yet available, but will be in due course. The example doesn't change much, see the proof files.

Now, we have defined a record type with four fields that gives us the projection functions to refer to the constituents of an element of this type. We can see how this works in the following definition of the simple structure — in the sense of [Kam98b] — of groups.

```
constdefs
  Group :: "('a grouptype)set"
  "Group == {G. (G.<f>): (G.<cr>) -> (G.<cr>) -> (G.<cr>)
              & (G.<inv>): (G.<cr>) -> (G.<cr>)
              & (G.<e>): (G.<cr>) &
             (! x: (G.<cr>). ! y: (G.<cr>). !z: (G.<cr>).
                      (G.<f>)(G.<inv> x) x = G.<e>)
                    & (G.<f>)(G.<e>) x = x)
                    & (G.<f>)((G.<f>) x y) z =
                       (G.<f>)(x)((G.<f>) y z)))}"
```

Given that the Isabelle theory for groups contains the locale displayed in Section 3 we can now use it in an interactive Isabelle session. We open the locale group with the ML command

```
Open_locale "group";
```

Now the assumptions and definitions are visible, i.e. we are in the scope of the locale groups. ML function `print_locale` shows all information about locales in the theory.

```
print_locales Group.thy;
```

This returns all information about the locale groups and the current scope.

```
locale name space:
  "Group.group" = "group", "Group.group"
locales:
  group =
    consts:
      G :: "'a set * (['a, 'a] => 'a) * ('a => 'a) * 'a * 'more"
      e :: "'a"
      binop :: "['a, 'a] => 'a"
      inv :: "'a => 'a"
    rules:
      Group_G: "G : Group"
    defs:
      e_def: "e == unit G"
      binop_def: "!!x y. binop x y == (G.<f>) x y"
      inv_def: "!!x. inv x == (G.<inv>) x"
current scope: group
```

Note, how the definitions with free variables have been bound by the meta-level universal quantifier (!!). The locale print function also gives information about the name spaces of the table of locales in the theory Group and displays the contents of the current scope.

As an illustration of the improvement we show how a proof for groups works now. Assuming that the theory of groups [Kam98a] is loaded we demonstrate one proof that shows how the inverse can be swapped with the group operation.

```
Goal "!! x y.[| x:G.<cr>; y:G.<cr> |] ==> (x # y)^-1 = y^-1 # x^-1";
```

Isabelle sets the proof up and keeps the display of the dependent locale syntax.

```
1. !!x y.[| x : G.<cr> ; y : G.<cr> |] ==> (x # y)^-1 = y^-1 # x^-1
```

We can now perform the proof as usual, but with the nice abbreviations and syntax. We can apply all results which we might have proved about groups inside the locale. We can even use the syntax when we use tactics that use explicit instantiation, e.g. `res_inst_tac`. When the proof is finished, we can assign it to a name using `result()`. The proof is now:

```
val inv_prod = "[| ?x : G.<cr>; ?y : G.<cr> |]
   ==> inv (binop ?x ?y) = binop (inv ?y) (inv ?x)
   [!!x. inv x == inverse G x,
     G : Group,
    !!x y. binop x y == bin_op G x y,
     e == unit G]" : thm
```

As meta-assumptions annotated at the theorem we find all the used rules and definitions, the syntax uses the explicit names of the locale constants, not their pretty printing form.

If we want to export the theorem we just type `export inv_prod`.

```
"[| ?G : Group; ?x : ?G.<cr>; ?y : ?G.<cr> |]
   ==> ?G.<inv> (?G.<f> ?x ?y) = ?G.<f> (?G.<inv> ?y) (?G.<inv> ?x)"
```

The locale constant `G` is now a free schematic variable of the theorem. Hence the theorem is universally applicable to all groups. The locale definitions have been eliminated. The other locale constants, e.g. `binop`, are replaced by their explicit versions, and have thus vanished together with the locale definitions.

6 Discussion

Locales seem to be a more primitive concept than modules. They do not enable abstraction over types or type constructors. Neither do they support real schematic polymorphic constants and definitions as the topmost theory level does. On the other hand, these restrictions admit to define a representation of a locale as a *meta-logical predicate* fairly easily. Thereby, locales can be first class citizen of the meta logic. We have developed this aspect of locales elsewhere. Although it works well, we found that it is better to

perform the first class reasoning separately in HOL, using an approach with dependent sets [Kam98b].

We believe that concrete syntax is the most decisive aspect for the actual usability of a locale concept. There is no inherent contradiction that would forbid the integration of the combination presented in this paper with the idea of a first class representation of locales by a locale predicate. In an independent experiment, we even implemented the mechanical generation of a first class representation for a locale. This implementation automatically extends the theory state of an Isabelle formalization. But, in many cases we don't think of a locale as a intra-logical object, rather just an theory-level assembly of items. Then we don't want this overhead of automatically created rules and constants.

In some sense locales do have a first class representation: globally interesting theorems that are proved in a locale may be exported. Then the former context structure of the locale gets dissolved: the definitions become expanded (and thus vanish). The locale constants turn into variables, and the assumptions become individual premises of the exported theorem. Although this individual representation of theorems does not entail the locale itself as a first class citizen of the logic, the context structure of the locale is translated into the meta-logical structure of assumptions and theorems. In so far we mirror the local assumptions — that are really the locale — into a representation in terms of the simple structural language of Isabelle's meta-logic. This translation corresponds logically to an application of the introduction rules for implication and the universal quantifier.

The simple implementation of the locale idea as presented in this paper works well together with the embedding of structures [Kam98b] and both can be used simultaneously.

A Appendix

We sum up the syntax and functions for locales described in this paper to provide a quick reference list. We use usual regular expression constructors $\{\}$, $|$, $[]$ and $*$. Terms enclosed in $\langle\rangle$ are non-terminals; typewriter font denotes terminals; quotation marks enclose single terminal symbols to avoid ambiguity. We use the lexical classes name, id, string, and mixfix of Isabelle [Pau94, Appendix A].

A.1 Syntax

Locale definition:

`locale` name	= ⟨*options*⟩
`fixes`	⟨*consts*⟩
`assumes`	⟨*rules*⟩
`defines`	⟨*defs*⟩

The locale constants have the same format as Isabelle constant declarations.

$$\langle consts \rangle \equiv \{ \text{ name ``::'' string [ ``('' mixfix ``)''] }\}^*$$

Similarly, the locale rules can be defined like Isabelle rules.

$$\langle rules \rangle \equiv \{ \text{ id string } \}^*$$

Definitions have the same outer syntax as general rules, but have to specify a meta-equality ==.

The optional extension of the locale header denotes the operations on locales[3]

$$\begin{aligned} \langle options \rangle \equiv \quad & \text{``+'' name } \{ \text{ ``+'' name } \} \\ & \mid \text{name } \{ \text{ ``+'' name } \} \\ & \mid \text{name ``['' string ``/'' string ``]''} \end{aligned}$$

A.2 Functions for Scope

```
Open_locale  : xstring -> unit
Close_locale : xstring -> unit
export       :     thm -> thm
```

Description:

- `Open_locale` *loc*;

  opens the locale *loc* in the theory of the current context.

- `Close_locale;`

  eliminates the last opened locale from the scope.

- `export` *thm*;

  locale definitions become expanded in *thm* and locale rules that were used in the proof of *thm* become part of its individual assumptions.

References

[dB80] N.G. de Bruijn. *A Survey of the Project AUTOMATH*, pages 579–606. Academic Press Limited, 1980.

[Dow90] G. Dowek. Naming and Scoping in a Mathematical Vernacular. Technical Report 1283, INRIA, Rocquencourt, 1990.

[Kam98a] F. Kammüller. Application Examples for [Kam98b]. Available on the Web as `http://www.cl.cam.ac.uk/users/fk203/algebra`, 1998.

[3] This is not implemented yet.

[Kam98b] F. Kammüller. Modular Structures as Dependent Types in Isabelle. Presented at the TYPES-workshop, Kloster Irsee, Germany, 1998.

[KP97] F. Kammüller and L. C. Paulson. Formal Proof of Sylow's First Theorem – An Experiment of Abstract Algebra in Isabelle HOL. *Journal of Automated Reasoning*, 1997. Submitted for publication.

[NW98] W. Naraschewski and M. Wenzel. Object-oriented Verification based on Record Subtyping in Higher-Order Logic. In *11th International Conference on Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*, ANU, Canberra, Australia, 1998. Springer-Verlag.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.