Locales

A Sectioning Concept for Isabelle

Florian Kammüller[1], Markus Wenzel[2], and Lawrence C. Paulson[1]

[1] Computer Laboratory, University of Cambridge
[2] Technische Universität München, Institut für Informatik

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.

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.

In mathematical proofs, we often want to define abbreviations for big expressions to enhance readability. These abbreviations might implicitly refer to terms which are arbitrary but fixed values for the entire proof. 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 too strong 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 [KP99], we came across several such local definitions. There, we define a set $\mathcal{M}$ as $\{S \subseteq G.\langle cr\rangle \mid order(S) = p^\alpha\}$ 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 Isabelle98 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. Then we would have to parameterize 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.

1.1 Related Work

There are several theorem provers that support modules, *e.g.* IMPS [FGT93], PVS [OSRSC98], and Larch [GH93]. The authors of these systems suggest to use their modules for the representation of mathematical structures, for example abstract algebraic structures like groups. This representation by modules is often not adequate because the modules have no representation in the logic. The "little theories" of IMPS come closest to an adequate representation of mathematical structures by providing a transformation between types and sets (see also [Kam99a, Chapt. 2]).

However, modules offer locality by providing local contexts in which formulas can make use of local declarations and definitions. Locales provide the locality that is part of a module concept. For adequate representation of mathematical structures we propose the concept of Σ and Π-types as it is common in type theories. The first author adapted this approach for Isabelle/HOL [Kam99b] set theory. In general, type theories are more suited for the adequate representation of mathematical structure than classical logics. But, not everyone wants to use type theory.

Locales implement a sectioning device similar to that in AUTOMATH [dB80] or Coq [Dow90]. In contrast to this kind of sections, locales are defined statically. Also, optional pretty printing syntax and dependent local definitions are part of the concept. Windley [Win93] describes abstract theories for HOL [GM93]. They are more adequate than classical modules, but do not offer the same notational advantages as locales. Deviating from the other approaches, locales do not have an instantiation mechanism, instead they enable exporting of theorems for individual instantiation (cf. Sect. 3.2).

1.2 Overview

Subsequently, we explain a simple approach to sectioning for the theorem prover Isabelle. In Sect. 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 Sect. 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 Sect. 5. Finally, we discuss more general aspects of locales in Sect. 6.

2 Locales — the Concept

Locales delimit a scope of *locally 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 may abbreviate longer terms, just like ordinary Isabelle definitions. On the other hand, 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. Thus the proof result becomes an ordinary theorem of the enclosing Isabelle theory.

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.

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 their 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. They are the antecedents of theorems of Isabelle's meta-logic — a form of the predicate calculus defined by a system of natural deduction rules. Meta-assumptions usually remain fixed throughout a proof and may be used within it in any order. The judgment that ϕ holds under the meta-assumptions $\phi_1, \ldots, \phi_n$ is written as

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

A more conventional notation for this would be $\phi_1, \ldots, \phi_n \vdash \phi$. Note that this implicit $\vdash$ is different from the implication of the meta-logic `==>` (cf. Sect. 5).

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 Isabelle98 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 result is admitted with the additional premises attached as meta-assumptions of the theorem. Hence, if this theorem is used in the same locale, the locale rules will be

matched automatically, rather than producing new subgoals. All locale rules can be used throughout the life time of the locale. The life time is determined by the interactive operations of opening and closing (cf. Sect. 3.2).

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. These arbitrary but fixed terms are the locale constants. The locale constants may be viewed from the outside as parameters, because they become universally quantified variables, when a result theorem is exported.

The idea of the locale constant is to use the locale as a scope such that inside the locale a free variable can be considered as a constant. 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 constructor 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. This export transforms a theorem into a general form whereby the locale is represented in the assumptions of the theorem.

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 Sect. 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. Conceptually, the definition of such logical 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 in this particular proof [KP99].

Another common thing in abstract algebra are formulas which are not so big, but suppress implicit information, e.g. we write Ha for the right 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 `r_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 solve this problem either: neither definitions nor pretty printing syntax can hide arguments, like `G` here, although these are fixed in a local context.

These features are realized by locales. In a locale where `G` is an arbitrary but fixed group for a series of theorems we can use a syntax like `H #> a` instead of `r_coset G H a`. We create a simple *locale definition* mechanism for concrete syntax which implements the concept of a local definition with optional pretty printing syntax. The concept of such local definitions is based on the locale constant: inside a locale, a locale constant can be used to abbreviate longer terms. The terms we define can even be dependent on other locale constants if those are contained in the scope of the locale. Since locale constants are only temporarily fixed the latter feature realizes dependent definitions, i.e. the defined terms may omit implicit information of the context. This concrete syntax may only be used 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 can be transformed into 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 defined generally.

```
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 underlying adequate 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 [NW98,Kam99b] as follows (cf. Sect. 5).

```
locale group =
  fixes
    G         :: "'a grouptype"
    e         :: "'a"
    binop     :: "'a => 'a => 'a"        (infixr "#" 80)
    inv       :: "'a => 'a"              ("i (_)" [90] 91)
  assumes
    Group_G   "G : Group"
  defines
    e_def     "e == (G.<e>)"
    binop_def "x # y == (G.<f>) x y"
    inv_def   "i x == (G.<inv>) x"
```

The above part of an Isabelle theory file 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. As illustrated in the example, the user can define syntactical notations for operators, by defining a pattern for the application, as for the prefix syntax of `inv`. Alternatively, one can use the keywords `infixr` or `infixl`, as in the example of `binop`, to define a right or left associative infix syntax.

The subsequent `assumes` specifies the locale rules. They are defined like Isabelle `rules`, i.e. by an identifier followed by the rule given as a string. Locale rules admit the statement of local assumptions about the locale constants. The `assumes` part is optional. Non-fixed variables in locale rules are automatically bound by the universal quantifier `!!` of the meta-logic. In the above example, 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. Here, locale constants declared in the `fixes` section can be defined using the Isabelle meta-equality `==`. The definition can contain variables on the left hand side, if the defined locale constant has appropriate type. This improves natural style of definition, for example for constants that represent infix operators, e.g. `binop`. The non-fixed variables on the left hand side are considered as schematic variables and are bound automatically by universal quantification of the meta-logic. The right hand side of a definition must not contain variables that are not already on the left hand side. In so far locale definitions behave like theory-level definitions. However, the locale concept realizes *dependent definitions* in that any variable that is fixed as a locale constant can occur on the right hand side of definitions. For example, a definition like

```
e_def "e == (G.<e>)"
```

contains the locale constant `G` on the right hand side. In principle, `G` is a free variable. Hence, this is a dependent definition. In Isabelle `defs` this would cause an error message "extra variable on right hand side". Naturally, definitions can already use the syntax of the locale constants in the `fixes` subsection. The `defines` part is, as the `assumes` part, optional.

Note also, that there are two different ways a locale constant can be used: one is to state its properties abstractly using rules, and one is to declare it as a name for a definition.

3.2 Invocation and Scope

After definition, locales may be opened and closed in a block-structured manner. The list (stack) of currently active locales is called *scope*. The operation for activating locales is *open*, the reverse one is *close*.

Scope The locale scope is part of each theory. It is a dynamic stack containing all active locales at a certain point in an interactive Isabelle session. The scope lives until all locales are explicitly closed. At any time there can be more than one locale open. The contents of these various active locales are all visible in the scope. Locales can be built by extension from other locales (cf. Sect. 3.3), i.e. they are nested. If a locale built by extension is open, the nesting is reflected in the scope, which contains the nested locales as layers. To check the state of the scope during a development the function `Print_scope` may be used. It displays the names of all open locales on the scope. The function `print_locales` applied to a theory displays all locales contained in that theory and in addition also the current scope.

Opening Locales can be *opened* at any point during an Isabelle session where we want to prove theorems concerning the locale. Opening a locale means making its contents visible by pushing it onto the scope of the current theory. Inside a scope of opened locales, theorems can use all definitions and rules contained in the locales on the scope. The rules and definitions may be accessed individually using the function *thm*. This function is applied to the names assigned to locale rules and definitions as strings. The opening command is called `Open_locale` and takes the name of the locale to be opened as its argument. In case of nested locales the opening command has to respect the nested structure (cf. Sect. 3.3).

Closing *Closing* means to cancel the last opened locale, pushing it off the scope. Theorems proved during the life time of this locale will be disabled, unless they have been explicitly exported, as described below. However, when the same locale is opened again these theorems may be used again as well, provided that they were saved as theorems in the first place, using `qed` or ML assignment. The command `Close_locale` takes a locale name as a string and checks if this locale is actually the topmost locale on the scope. If this is the case, it removes this locale, otherwise it prints a warning message and does not change the scope.

Export of Theorems Export of theorems transports theorems out of the scope of locales. Locale rules that have been used in the proof of a theorem inside a

locale are carried by the exported form of the theorem as its individual meta-assumptions. The locale constants are universally quantified variables in the exported theorems, hence such theorems can be instantiated individually. Definitions become unfolded; locale constants that were merely used on the left hand side of a definition vanish. Logically, exporting corresponds to a combined application of introduction rules for implication and universal quantification. Exporting forms a kind of *normalization* of theorems in a locale scope.

According to the possibility of nested locales there are two different forms of export. The first one is realized by the function `export` that exports theorems through all layers of opened locales of the scope. Hence, the application of export to a theorem yields a theorem of the global level, that is, the current theory context without any local assumptions or definitions.

The other export function `Export` transports theorems just one level up in the scope. When locales are nested we might want to export a theorem, but not to the global level of the current theory, i.e. not outside all locales in the nesting, instead just to the previous level, because that is where we need it as a lemma. If we are in a nesting of locales of depth n, an application of `Export` will transform a theorem to one of level $n-1$, i.e. into one that is independent of the definitions and assumptions of the locale that was on level n, but still uses locale constants, definitions and rules of the $n-1$ locales underneath.

3.3 Other Aspects

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.

Polymorphism Isabelle's meta-logic is based on a version of Church's Simple Theory of Types [Chu40] 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 type variables is wider. The quantification of the type variables is placed at the outside of the locale. 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 in the examples. On the other hand, locale definitions may not be polymorphic within the locale's scope.

This feature solves the problem we encountered in case studies from abstract algebra, most prominently in the proof of Sylow's theorem [KP99]. There we had to choose a fixed type `i` in order to model the fixing of a polymorphic type of groups to enable readable formulas. Thereby, we lost the generality of the result.

Augmenting Locales As locales are defined statically in an Isabelle theory, operations on locales may be used to construct locales from other predefined ones statically in an Isabelle theory. A locale can be defined as the extension of a previously defined locale. This operation of extension is optional and is syntactically expressed as

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

The locale `foo` builds on the constants and syntax of the locale `bar`. That is, all contents of the locale `bar` can be used in definitions and rules of the corresponding parts of the locale `foo`. Although locale `foo` assumes the `fixes` part of `bar` it does not automatically subsume its rules and definitions. Normally, one expects to use locale `foo` only if locale `bar` is already active. The opening mechanism is designed such that in the case of a locale built by extension it opens the ancestor automatically. If one opens a locale `foo` that is defined by extension from locale `bar` the function `Open_locale` checks if locale `bar` is open. If so, then it just opens `foo`, if not, then it prints a message and opens `bar` before opening `foo`. Naturally, this carries on, if `bar` is again an extension. The locales `bar` and `foo` become separate layers on the scope; `foo` has to be closed before `bar` can be closed (cf. Sect. 3.2).

In case of name clashes always the innermost definition is visible. That is, a name defined in a locale hides an equal name of a theory during the life time of the locale. When locales are built by extension, names may be hidden similarly. This is not possible if unrelated locales are opened simultaneously.

Another interesting device (which has not yet been implemented) is renaming of locale constants. This can be very useful if we want to have more than one instance of the same locale in the scope, for example when we reason with two different groups. The following illustrates a possible renaming mechanism: `loc_r` is created from `loc_c` by renaming all occurrences of locale constant `c` by `r`.

```
locale loc_r = loc_c [r/c]
```

Merging of locales by naming them could be another operation for locales. It can be explained through extension.

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 enable to implement new theory definition features properly.

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 discussed in the previous section:

(4) modify term read and print functions,
(5) modify proof result operation.

4.1 Theory Data

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 appropriate source-level information as required later for any derived 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 recent Isabelle approach is more powerful, internalizing generic data as first-class components of theory objects.

The ML 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:

[1] This is achieved by invoking most of the black-magic that Standard ML has to offer: exception constructors for introducing new injections into type `exn`, private references as tags for identification and authorization, and functors for hiding. We see that ML is for the Real Programmer, after all!

```
signature THEORY_DATA_ARGS =
sig
  val name: string
  type T
  val empty: 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 `merge` operation is called when theories are joined, as should be the private data. Finally, `print` shall display the theory data in some human readable way; the function obtains the signature of the current theory ("self") as additional argument.

The result structure of `TheoryDataFun` is as follows:

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

The new data slot has to be made known via above `init` operation. This is much like a run-time type declaration within a theory. 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.

4.2 Theory Extension Function

Employing above private theory data slot, we have implemented the actual locale definition mechanism on top of usual Isabelle primitives (e.g. `add_modesyntax`). The ultimate result is the ML function `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, updating the table of existing locales. We also invoke a few other Isabelle primitives to extend the theory's syntax, for example.

4.3 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. Using Isabelle's `ThySyn.add_syntax` operation we can now associate our function `add_locale` with the `locale_decl` parser and plug it into the main theory syntax.

4.4 Interface

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

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

5 Examples from Abstract Algebra

We illustrate the use of the implementation by examples with the abstract algebraic structure of groups. We use a representation of groups that we found to be the most appropriate for abstract algebraic structures [Kam99b]. 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)
```

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. The class of all groups is defined as a typed HOL set over this record type [Kam99b]. This definition entails all the properties of a group and enables to state the group property quite concisely as

```
G : Group
```

[2] We use pretty printing facilities for records that are not yet available. The example remains the same, because one can achieve the same syntax using separate `syntax` declarations manually.

Given that the Isabelle theory for groups contains the locale displayed in Sect. 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 `group`. ML function `print_locales` shows all information about locales in the theory:

```
print_locales Group.thy;
```

This returns all information about the locale `group` and the current scope.[3]

```
locale name space:
  "Group.group" = "group", "Group.group"
locales:
  group =
    consts:
      G :: "'a set * (['a, 'a] => 'a) * ('a => 'a) * 'a"
      e :: "'a"
      binop :: "['a, 'a] => 'a"
      inv :: "'a => 'a"
    rules:
      Group_G: "G : Group"
    defs:
      e_def: "e == (G.<e>)"
      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 is loaded we demonstrate one proof that shows how the inverse can be swapped with the group operation.

```
Goal "[|x : (G.<cr>); y : (G.<cr>)|] ==> i(x # y) = (i y)#(i x)";
```

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

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

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 theorem is now:

[3] The print function is mainly for inspecting and debugging, so the output of terms is in their actual internal form without locale syntax.

```
val inv_prod = "[| ?x : (G.<cr>); ?y : (G.<cr>) |]
   ==> inv (binop ?x ?y) = binop (inv ?y) (inv ?x)
   [!!x. inv x == (G.<inv>) x, G : Group,
    !!x y. binop x y == (G.<f>) x y, e == (G.<e>)]" : 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. The question mark `?` in front of variables labels free schematic variables in Isabelle that may be instantiated later. The assumption `e == (G.<e>)` is included because during the proof it was used to abbreviate the unit element.

To transform the theorem into its global form 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.

The locale facilities for groups are of course even more practical if we carry on to more complex structures like cosets. Assuming an adequate definition for cosets and products of subsets of a group (e.g. [Kam99b])

```
r_coset G H a == (λ x. (G.<f>) x a) `` H
set_prod G H1 H2 == (λ x. (G.<f>) (fst x)(snd x)) `` (H1 × H2)
```

where `` yields the image of a HOL function applied to a set — we use an extension of the locale for groups thereby enhancing the concrete syntax of the above definitions.

```
locale coset = group +
  fixes
    rcos      :: "['a set, 'a] => 'a set"   ("_ #> _" [60,61]60)
    setprod   :: "['a set, 'a set] => 'a set" ("_ <#> _" [60,61]60)
  defines
    rcos_def "H #> x == r_coset G H x"
    setprod_def "H1 <#> H2 == set_prod G H1 H2"
```

This enables us to reason in a natural way that reflects typical objectives of mathematics — in this case abstract algebra. We reason about the behaviour of substructures of a structure, like cosets of a group. Are they a group as well?[4] Therefore, we welcome a notation like

```
  (H #> x) <#> (H #> y) = H #> (x # y)
```

when we have to reason with such substructural properties. While knowing that the underlying definitions are adequate and related properties derivable, we can reason with a convenient mathematical notation. Without locales the formula we had to deal with would be

[4] They are a group if H is normal which is proved conveniently in Isabelle with locales.

```
set_prod G (r_coset G H x)(r_coset G H y) = r_coset G H ((G.<f>) x y)
```

The improvement is considerable and enhances comprehension of proofs and the actual finding of solutions — in particular, if we consider that we are confronted with the formulas not only in the main goal statements but in each step during the interactive proof.

6 Discussion

First of all, term syntax may be greatly improved by locales because they enable dependent local definitions. Locale constants can have pretty printing syntax assigned to them and this syntax can as well be dependent, i.e. use everything that is declared as fixed implicitly. So, locales approximate a natural mathematical style of formalization. Locales are a simpler concept than modules. They do not enable abstraction over type constructors (which rules out modeling monads, for example). Neither do locales support 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 [Kam99a]. In the latter experiment, we implemented the mechanical generation of a first-class representation for a locale. This implementation automatically extends the theory state of an Isabelle formalization by declarations and definitions for a predicate representing the locale logically. But, in many cases we do not think of a locale as a intra-logical object, rather just an theory-level assembly of items. Then, we do not want this overhead of automatically created rules and constants. We prefer to perform the first-class reasoning separately in higher-order logic, using an approach with dependent sets [Kam99b].

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 of the meta-logic. And, because Isabelle has a proper meta-logic this first-class representation is easy to apply.

Generality of proofs is partly revealed in locales: certain premises that are available in a locale are not used at all in the proof of a theorem. In that case the exported version of the theorem will not contain these premises. This may seem a bit exotic, in that theorems proved in the same locale scope might have different premise lists. That is, theorems may generally just contain a subset of the locale

assumptions in their premises. That takes away uniformity of theorems of a locale but grants that theorems may be proved in a locale and will be individually considered for the export. In many cases one discovers that a theorem that one closely linked with, say, groups actually does not at all depend on a specific group property and is more generally valid. That is, locales filter the theorems to be of the most general form according to the locale assumptions.

Locales are, as a concept, of general value for Isabelle independent of abstract algebraic proof. In particular, they are independent of any object logic. That is, they can be applied merely assuming the meta-logic of Isabelle/Pure. They are already applied in other places of the Isabelle theories, e.g. for reasoning about finite sets where the fixing of a function enhances the proof of properties of a "fold" functional and similarly in proofs about multisets and the formal method UNITY [CM88]. Furthermore, the concept can be transferred to all higher-order logic theorem provers. There are only a few things the concept relies on. In particular, the features needed are implication and universal quantification — the two constructors that build the basis for the reflection of locales *via* export and are at the same time the explanation of the meaning of locales. For theorem provers where the theory infrastructure differs greatly from Isabelle's, one may consider dynamic definition of locales instead of the static one.

The simple implementation of the locale idea as presented in this paper works well together with the first-class representation of structures by an embedding using dependent types [Kam99b]. Both concepts can be used simultaneously to provide an adequate support for reasoning in abstract algebra.

References

Chu40. A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, pages 56–68, 1940.

CM88. K. Mani Chandi and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

dB80. N. G. de Bruijn. A Survey of the Project AUTOMATH. In J.P. Seldin and J.R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic*, Academic Press Limited, pages 579–606. 1980.

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

FGT93. W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: an Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–248, 1993.

GH93. John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

GM93. M. J. C. Gordon and T. F. Melham. *Introduction to HOL, a Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

Kam99a. F. Kammüller. *Modular Reasoning in Isabelle*. PhD thesis, University of Cambridge, 1999. submitted.

Kam99b. F. Kammüller. Modular Structures as Dependent Types in Isabelle. In *Types for Proofs and Programs: TYPES '98*, LNCS. Springer-Verlag, 1999. Selected papers. To appear.

KP99. F. Kammüller and L. C. Paulson. A Formal Proof of Sylow's First Theorem – An Experiment in Abstract Algebra with Isabelle HOL. *Journal of Automated Reasoning*, 1999. To appear.

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.

OSRSC98. S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. PVS Language Reference. Part of the PVS Manual. Available on the Web as `http://www.csl.sri.com/pvsweb/manuals.html`, September 1998.

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

Win93. P. J. Windley. Abstract Theories in HOL. In L. Claesen and M. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, IFIP Transactions A-20, pages 197–210. North-Holland, 1993.