# Local theory specifications in Isabelle/Isar

Florian Haftmann* and Makarius Wenzel**

Technische Universität München
Institut für Informatik, Boltzmannstraße 3, 85748 Garching, Germany
http://www.in.tum.de/~haftmann/
http://www.in.tum.de/~wenzelm/

**Abstract.** The proof assistant Isabelle has recently acquired a "local theory" concept that integrates a variety of mechanisms for structured specifications into a common framework. We explicitly separate a local theory "target", i.e. a fixed axiomatic specification consisting of parameters and assumptions, from its "body" consisting of arbitrary definitional extensions. Body elements may be added incrementally, and admit local polymorphism according to Hindley-Milner. The foundations of our local theories rest firmly on existing Isabelle/Isar principles, without having to invent new logics or module calculi.

Specific target contexts and body elements may be implemented within the generic infrastructure. This results in a large combinatorial space of specification idioms available to the user. Here we introduce targets for locales, type-classes, and class instantiations. The available selection of body elements covers primitive definitions and theorems, inductive predicates and sets, and recursive functions. Porting such existing definitional packages is reasonably simple, and allows to re-use sophisticated tools in a variety of target contexts. For example, a recursive function may be defined depending on locale parameters and assumptions, or an inductive predicate definition may provide the witness in a type-class instantiation.

## 1 Introduction

Many years ago, Isabelle locales were introduced [12] as a mechanism to organize formal reasoning in a modular fashion: after defining a locale as a context of fixed parameters (**fixes**) and assumptions (**assumes**), theorems could be proved within that scope, while an exported result (with additional premises) would be provided at the same time. Such "**theorem** (*in locale*)" statements have become popular means to organize formal theory developments. A natural extension of results within a local context is "**definition** (*in locale*)". Traditional locales would support **defines** elements within the axiomatic specification, essentially

* Supported by DFG project NI 491/10-1.
** Supported by BMBF project Verisoft XT (grant 01 IS 07008).

simulating definitions by equational assumptions, but this turned out to be unsatisfactory. It is not possible to add further definitions without extending the locale, and there is no support for polymorphism. Moreover, Isabelle/HOL users rightly expect to have the full toolbox of definitional packages available (e.g. inductive predicates and recursive functions), not just primitive equations.

These needs are addressed by our *local theory* concept in Isabelle/Isar. A local theory provides an abstract interface to manage definitions and theorems relative to a context of fixed parameters and assumptions. This generic infrastructure is able to support the requirements of existing module concepts in Isabelle, notably locales and type-classes. Thus we integrate and extend the capabilities of structured specifications significantly, while opening a much broader scope for alternative module mechanisms.

Implementing such local theory targets is a delicate task, but only experts in module systems need to do it. In contrast, it is reasonably easy to produce definitional packages for use in the body of any local theory. Here we have been able to rationalize the traditional theory specification primitives of Higher-Order Logic considerably, such that the local versions are both simpler and more general than their global counterparts.

*Overview.* We examine the flexibility of the local theory concept by an example of type class specification and instantiation (§2). After a careful exposition of the relevant foundations of Isabelle/Pure and Isabelle/Isar (§3), we introduce the main local theory architecture (§4) and describe some concrete target mechanisms (§5).

## 2 Example: Type Classes

The following example in Isabelle/HOL [14] uses type-classes to model general orders and orders that admit well-founded induction. Earlier [11] we integrated traditional axiomatic type-classes with locales, now both theory structuring concepts are also fitted into the bigger picture of local theories.

Basic Isabelle notation approximates usual mathematics, despite some bias towards $\lambda$-calculus and functional languages like Haskell. The general syntax for local theory specifications is "*target* **begin** *body* **end**", where *body* consists of a sequence of specification elements (definitions and theorems with proofs), and *target* determines a particular interpretation of the body elements relative to a local context (with parameters and assumptions).

The most common targets are **locale** and **class**. These targets are special in being explicitly named, and allow further body additions at any time. The syntax for this is "**context** *name* **begin** *body* **end**", with the abbreviation of "*specification* (**in** *name*)" for "**context** *name* **begin** *specification* **end**". The latter also integrates the existing "**theorem** (**in** *locale*)" into our framework.

Other targets, like the **instantiation** shown later, demand that the body is a closed unit that provides required specifications, finishes proof obligations etc.

**General Orders.** We define an abstract algebra over a binary relation *less-eq* that observes the partial ordering laws.

**class** *order* =
  **fixes** *less-eq* :: $\alpha \Rightarrow \alpha \Rightarrow$ *bool* (**infix** $\preceq$ 50)
  **assumes** *refl*: $x \preceq x$
    **and** *trans*: $x \preceq y \Longrightarrow y \preceq z \Longrightarrow x \preceq z$
    **and** *antisym*: $x \preceq y \Longrightarrow y \preceq x \Longrightarrow x = y$
**begin**

This class context provides a hybrid view on our abstract theory specification. The term *less-eq* :: $\alpha \Rightarrow \alpha \Rightarrow$ *bool* refers to a fixed parameter of a fixed type; the parameter *less-eq* also observes assumptions. At the same time, the canonical type-class interpretation [11] provides a polymorphic constant for arbitrary *order* types, i.e. any instance of *less-eq* :: $\beta$::*order* $\Rightarrow \beta \Rightarrow$ *bool*. Likewise, the locale assumptions are turned into theorems that work for arbitrary types $\beta$::*order*.

Our class target augments the usual Isabelle type-inference by a separate *type improvement* stage, which identifies sufficiently general occurrences of *less-eq* with the locale parameter, while leaving more specific instances as constants. By handling the choice of locale parameters vs. class constants within the type-checking phase, we also avoid extra syntactic ambiguities: the above mixfix annotation (**infix** $\preceq$ 50) is associated with the class constant once and for all. See §5.3 for further details.

**end**

Back in the global context, *less-eq* :: $\alpha$::*order* $\Rightarrow \alpha \Rightarrow$ *bool* refers to a global class operation for arbitrary *order* types $\alpha$; the notation $x \preceq y$ also works as expected. Global class axioms are available as theorems *refl*, *trans*, *antisym*.

The old **axclass** [17] would have achieved a similar effect. At this point we could even continue with further definitions and proofs relative to this polymorphic constant only, e.g. *less* :: $\alpha$::*order* $\Rightarrow \alpha \Rightarrow$ *bool* depending on the global *less-eq* :: $\alpha$::*order* $\Rightarrow \alpha \Rightarrow$ *bool*. But then the resulting development would be more special than necessary, with the known limitations of type-classes of at most one instantiation per type constructor. So we now continue within the hybrid class/locale context, which provides type-class results as expected, but also admits general locale interpretations [2].

**context** *order*
**begin**

We now define *less* as the strict part of *less-eq*, and prove some simple lemmas.

**definition** *less* :: $\alpha \Rightarrow \alpha \Rightarrow$ *bool* (**infix** $\prec$ 50)
  **where** $x \prec y \leftrightarrow x \preceq y \wedge \neg\, y \preceq x$

**lemma** *irrefl*: $\neg\, x \prec x$ ⟨*proof*⟩
**lemma** *less-trans*: $x \prec y \Longrightarrow y \prec z \Longrightarrow x \prec z$ ⟨*proof*⟩
**lemma** *asym*: $x \prec y \Longrightarrow y \prec x \Longrightarrow C$ ⟨*proof*⟩

**end**

Again this produces a global constant *less* :: $\alpha$::*order* $\Rightarrow \alpha \Rightarrow$ *bool*, whose definition depends on the original class operation *less-eq* :: $\alpha$::*order* $\Rightarrow \alpha \Rightarrow$ *bool*. The

additional variant *order.less* $(rel :: \alpha \Rightarrow \alpha \Rightarrow bool)$ $(x :: \alpha)$ $(y :: \alpha)$ stems from the associated locale context and makes this dependency explicit. The latter is more flexible, but also slightly more cumbersome to use.

**Well-founded Induction and Recursion.** Next we define well-founded orders by extending the specification of general orders.

**class** *wforder* = *order* +
  **assumes** *less-induct*: $(\bigwedge x{::}\alpha.\ (\bigwedge y.\ y \prec x \Longrightarrow P\ y) \Longrightarrow P\ x) \Longrightarrow P\ x$
**begin**

With this induction rule available, we can define a recursion combinator by means of an inductive relation that corresponds to the function's graph, see also [16].

**inductive** *wfrec-rel* $:: ((\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta \Rightarrow bool$
  **for** $F :: (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$
  **where** *rec*: $(\bigwedge z.\ z \prec x \Longrightarrow wfrec\text{-}rel\ F\ z\ (g\ z)) \Longrightarrow wfrec\text{-}rel\ F\ x\ (F\ g\ x)$

**definition** *cut* $:: \alpha \Rightarrow (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$
  **where** $cut\ x\ f\ y = (if\ y \prec x\ then\ f\ y\ else\ undefined)$
**lemma** *cuts-eq*: $cut\ x\ f = cut\ x\ g \leftrightarrow (\forall y.\ y \prec x \longrightarrow f\ y = g\ y)$ ⟨*proof*⟩

**definition** *adm-wf* $:: ((\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta) \Rightarrow bool$
  **where** $adm\text{-}wf\ F \leftrightarrow (\forall f\ g\ x.\ (\forall z.\ z \prec x \longrightarrow f\ z = g\ z) \longrightarrow F\ f\ x = F\ g\ x)$
**lemma** *adm-lemma*: $adm\text{-}wf\ (\lambda f\ x.\ F\ (cut\ x\ f)\ x)$ ⟨*proof*⟩

**definition** *wfrec* $:: ((\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$
  **where** $wfrec\ F = (\lambda x.\ (THE\ y.\ wfrec\text{-}rel\ (\lambda f\ x.\ F\ (cut\ x\ f)\ x)\ x\ y))$

**lemma** *wfrec-unique*: $adm\text{-}wf\ F \Longrightarrow \exists!y.\ wfrec\text{-}rel\ F\ x\ y$ ⟨*proof*⟩
**theorem** *wfrec*: $wfrec\ F\ x = F\ (cut\ x\ (wfrec\ F))\ x$ ⟨*proof*⟩

This characterizes a polymorphic combinator *wfrec* that works for arbitrary types $\beta$, relative to the locally fixed type parameter $\alpha$. Thus $wfrec\ (\lambda f :: \alpha \Rightarrow nat.\ body)$ or $wfrec\ (\lambda f :: \alpha \Rightarrow bool.\ body)$ may be used in the current context. *THE* is the definite choice operator, sometimes written $\iota$ in literature; *undefined* is an unspecified constant.

**end**

Back in the global context, we may refer either to the exported locale operation $wforder.wfrec\ (rel :: \alpha \Rightarrow \alpha \Rightarrow bool)\ (F :: (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta)$ or the overloaded constant $wfrec\ (F :: (\alpha{::}wforder \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta)$. Here $\alpha$ and $\beta$ are again arbitrary, although the class constraint needs to be observed in the second case.

**Lexicographic Products.** The product $\alpha \times \beta$ of two *order* types is again an instance of the same algebraic structure, provided that the *less-eq* operation is defined in a suitable manner, such that the class assumptions can be proven. We shall establish this in the body of the following **instantiation** target.

**instantiation** $*$ :: (*order*, *order*) *order*
**begin**

We now define the lexicographic product relation by means of a (non-recursive) inductive definition, depending on hypothetical *less-eq* on fixed *order* types $\alpha$ and $\beta$.

**inductive** *less-eq-prod* :: $\alpha \times \beta \Rightarrow \alpha \times \beta \Rightarrow$ *bool*
  **where** *less-eq-fst*: $x \prec v \Longrightarrow (x,\ y) \preceq (v,\ w)$
    | *less-eq-snd*: $x = v \Longrightarrow y \preceq w \Longrightarrow (x,\ y) \preceq (v,\ w)$

This definition effectively involves overloading of the polymorphic constant *less-eq* on product types, but the details are managed by our local theory target context. Here we have even used the derived definitional mechanism **inductive**, which did not support overloading in the past.

The above specification mentions various type instances of *less-eq*, for $\alpha$, $\beta$, and $\alpha \times \beta$. All of these have been written uniformly with the $\preceq$ notation. This works smoothly due to an additional improvement stage in the type-inference process.

The outline of the corresponding instantiation proof follows. The **instance** element below initializes the class membership goal as in the existing global **instance** command [17], but the type arity statement is not repeated here.

**instance**
**proof**
  **fix** $p\ q\ r :: \alpha \times \beta$
  **show** $p \preceq p$ ⟨*proof*⟩
  **{ assume** $p \preceq r$ **and** $r \preceq q$ **then show** $p \preceq q$ ⟨*proof*⟩ **}**
  **{ assume** $p \preceq q$ **and** $q \preceq p$ **then show** $p = q$ ⟨*proof*⟩ **}**
**qed**

**end**

By concluding the instantiation, the new type arity becomes available in the theory, i.e. *less-eq* ($p :: \alpha$::*order* $\times\ \beta$::*order*) ($q :: \alpha \times \beta$) can be used for arbitrary $\alpha$, $\beta$ in *order*.

## 3 Foundations

Isabelle consists of two main layers: the *logical framework* of Isabelle/Pure for higher-order Natural Deduction, and the *architectural framework* of Isabelle/Isar for organizing logical and extra-logical concepts in a structured manner.

Our local theory concepts rely only on existing foundations. We refrain from inventing new logics or module calculi, but merely observe pre-existent properties carefully to employ them according to our needs. This principle of leaving the logical basis unscathed is an already well-established Isabelle tradition. It enables implementation of sophisticated tools without endangering soundness. This is Milner's "LCF-approach" in its last consequence, cf. the discussion in [19, §3].

### 3.1 The Pure Logical Framework

The logic of Isabelle/Pure [15] is a reduced version of Higher-Order Logic according to Church [8] and Gordon [9]. This minimal version of HOL is used as a logical framework to represent object-logics, such as the practically important Isabelle/HOL [14].

**Logical Entities.** The Pure logic provides three main categories of formal entities: types, terms, and theorems (with implicit proofs).

Types $\tau$ are simple first-order structures, consisting of type variables $\alpha$ or type constructor applications $(\tau_1, \ldots, \tau_n)\ \kappa$, usually written postfix. Type *prop* represents framework propositions, and the infix type $\alpha \Rightarrow \beta$ functions.

Terms $t$ are formed as simply-typed $\lambda$-terms, with variables $x :: \tau$, constants $c :: \tau$, abstraction $\lambda x :: \tau.\ t[x]$ and application $t_1\ t_2$. Types are usually left implicit, cf. Hindley-Milner type-inference [13]. Terms of type *prop* are called propositions. The logical structure of propositions is determined by quantification $\bigwedge x :: \alpha.\ B[x]$ or implication $A \Longrightarrow B$; these framework connectives express object-logic rules in Natural Deduction style. Isabelle/Pure also provides built-in equality $t_1 \equiv t_2$ with rules for $\alpha\beta\eta$-conversion.

Theorems *thm* are abstract containers for derivations within the logical environment. Primitive inferences of Pure operate on sequents $\Gamma \vdash \varphi$, where $\varphi$ is the main conclusion and $\Gamma$ its local context of hypotheses. There are standard introduction and elimination rules for $\bigwedge$ and $\Longrightarrow$ operating on sequents.

This low-level inference system is directly implemented in the Isabelle inference kernel; it corresponds to dependently-typed $\lambda$-calculus with propositions as types, although proof terms are usually omitted. It is useful to think of theorems as representing full proof terms, even though the implementation may omit them: the formal system can be categorized as "$\lambda HOL$" within the general setting of Pure Type Systems (PTS) [3]. This provides a unified view of terms and derivations, with terms depending on terms $\lambda x :: \alpha.\ b[x]$, proofs depending on terms $\bigwedge x :: \alpha.\ B[x]$, and proofs depending on proofs $A \Longrightarrow B$.

Object-logic inferences are expressed at the level of Pure propositions, *not* Pure rules. For example, in Isabelle/HOL the modus ponens is represented as $(A \longrightarrow B) \Longrightarrow A \Longrightarrow B$, implication introduction as $(A \Longrightarrow B) \Longrightarrow A \longrightarrow B$. Isabelle provides convenient (derived) principles of *resolution* and *assumption* [15] to back-chain such object rules, or close branches within proofs, respectively.

This second level of Natural Deduction is encountered by Isabelle users most of the time, e.g. when doing "**apply** (*rule r*)" in a tactic script. Thus the first level of primitive inferences remains free for internal uses, to support local scopes of fixed variables and assumptions. Both Isar proof texts [18] and locales [12,1,2] operate on this primitive level of Pure, and the Isabelle/Isar framework ensures that local hypotheses are managed according to the block structure of the text, such that users never have to care about the $\Gamma$ part of primitive sequents.

The notation $\langle\varphi\rangle$ shall refer to some theorem $\Gamma \vdash \varphi$, where $\Gamma$ is clear from the context. The cIsabelle syntax ‘$\varphi$‘ references facts the same way but uses ASCII back-quotes in the source instead of the funny parentheses.

**Theories.** Derivations in Isabelle/Pure depend on a global theory environment $\Theta$, which holds declarations of type-constructors $(\alpha_1, \ldots, \alpha_n)\ \kappa$ (specifying the number of arguments), term constants $c :: \tau$ (specifying the most general type scheme), and axioms $a$: $A$ (specifying the proposition). The following concrete syntax shall be used for these three theory declaration primitives:

| | |
|---|---|
| **type** $\forall \overline{\alpha}.\ (\overline{\alpha})\ \kappa$ | — type constructor $\kappa$ |
| **const** $c :: \forall \overline{\alpha}.\ \tau[\overline{\alpha}]$ | — term constant $c$ |
| **axiom** $a$: $\forall \overline{\alpha}.\ A[\overline{\alpha}]$ | — proof constant $a$ |

These primitives support *global schematic polymorphism*, which means that type variables given in the declaration may be instantiated by arbitrary types. The logic provides admissible inferences for this: moving from $\Gamma \vdash \varphi[\alpha]$ to $\Gamma \vdash \varphi[\tau]$ essentially instantiates whole proof trees.

We take the notational liberty of explicit type quantification $\forall \alpha.\ A[\alpha]$, even though the Pure logic is not really polymorphic. Type quantifiers may only occur in global theory declarations and theorems, but never in hypothetical statements, or the binding position of a $\lambda$-abstraction. This restricted type quantification behaves like schematic type variables, as indicated by question marks in Isabelle: results $\forall \alpha.\ \bigwedge x :: \alpha.\ x \equiv x$ and $\bigwedge x :: ?\alpha.\ x \equiv x$ are interchangeable.

Unrestricted declarations of types, terms, and axioms are rarely used in practice. Instead there are certain disciplined schemes that qualify as *definitional specifications* due to nice meta-theoretical properties. In the Pure framework, we can easily justify the well-known principle of *constant definition*, which relates a polymorphic term constant with an existing term:

$$\textbf{constdef}\ c :: \forall \overline{\alpha}.\ \tau[\overline{\alpha}]\ \textbf{where}\ \forall \overline{\alpha}.\ c[\overline{\alpha}] \equiv \mathit{rhs}[\overline{\alpha}]$$

Here the constant name $c$ needs to be new, and *rhs* needs to be a closed term with all its type variables already included in its type $\tau$. If these conditions hold, **constdef** expands to corresponding **const** $c :: \forall \overline{\alpha}.\ \tau[\overline{\alpha}]$ and **axiom** $\forall \overline{\alpha}.\ c[\overline{\alpha}] \equiv \mathit{rhs}[\overline{\alpha}]$.

This constant definition principle observes *parametric polymorphism*. Isabelle also supports *ad-hoc polymorphism* (overloading) which can be shaped into a disciplined version due to Haskell-style type-classes on top of the logic, see also [17] and [11]. The latter concept of "less ad-hoc polymorphism" allows us to reconstruct overloading-free definitions and proofs, via explicit dictionary terms.

We also introduce an explicit definition scheme for "proof constants", which gives proven facts an explicit formal status within the theory context:

$$\textbf{thmdef}\ \forall \overline{\alpha}.\ b = \langle B[\overline{\alpha}] \rangle$$

Moreover, the weaker variant of **thm** $b = \langle B \rangle$ shall serve the purpose of casual naming of facts, without any impact on the internal structure of derivations.

### 3.2 Isar Proof Contexts

The main purpose of the Isabelle/Isar infrastructure is to elevate the underlying logical framework to a scalable architecture that supports structured reasoning. This works by imposing certain Isar *policies* on the underlying Pure *primitives*. It is important to understand, that Isar is not another calculus, but an architecture to organize existing logical principles, and enrich them by non-logical support

structure. The relation of Pure vs. Isar is a bit like that of a CPU (execution primitives) and an operating system (high-level abstractions via policies).

Isabelle/Isar was originally motivated by the demands for human-readable proofs [18]: the Isar proof language provides a structured walk through the text, maintaining local facts and goals, all relative to a *proof context* at each position. This idea of Isar proof context has turned out a useful abstraction to organize various advanced concepts in Isabelle, with locales [12,1,2] being the classic example. A more recent elaboration on the same theme are LCF-style proof tools that work relative to some local declarations and may be transformed in a concrete application context later; [7] covers a Gröbner Base procedure on abstract rings that may get used on concrete integers etc.

Subsequently we briefly review the main aspects of Isar proof contexts, as required for our local theory infrastructure.

**Logical Context Elements.** The idea is to turn the $\Gamma$ part of the primitive calculus (§3.1) into an explicit environment, consisting of declarations for all three logical categories: type variables, term variables, and assumptions:

| | |
|---|---|
| **type** $\alpha$ | — type variable $\alpha$ |
| **fix** $x :: \tau[\overline{\alpha}]$ | — term variable $x$ |
| **assume** $a$: $A[\overline{\alpha}][\overline{x}]$ | — proof variable $a$ |

Strictly speaking there is no explicit **type** element in Isabelle/Isar, because type variables are handled implicitly according to Hindley-Milner discipline [13]: when entering a new term (proposition) into the context, its type variables are fixed; when exporting results from the context, type variables are generalized as far as possible, unless they occur in the types of term variables that are still fixed.

Exporting proven results from the scope of **fix** and **assume** corresponds to $\bigwedge/\Longrightarrow$ introduction rules. In other words, the logical part of an Isar context may be fully internalized into the Pure logic.

Isar also admits derived context elements, parameterized by a *discharge rule* that is invoked when leaving the corresponding scope. In particular, simple (non-polymorphic) definitions may be provided as follows:

$$\textbf{vardef}\ x :: \tau[\overline{\alpha}]\ \textbf{where}\ x \equiv \mathit{rhs}[\overline{\alpha}]$$

Here the variable name $x$ needs to be new, and *rhs* needs to be a closed term, mentioning only previously fixed type variables. If these conditions hold, **vardef** expands to corresponding **fix** and **assume** elements, with a discharge rule that expands a local result $\langle B[x]\rangle$ to $\langle B[\mathit{rhs}]\rangle$, thanks to reflexivity of $\equiv$.

Although **vardef** resembles the global **constdef**, it only works for fixed types! In particular, **vardef** $\mathit{id} :: \alpha \Rightarrow \alpha$ **where** $\mathit{id} \equiv \lambda x :: \alpha.\ x$ merely results in context elements **type** $\alpha$ **fix** $\mathit{id} :: \alpha \Rightarrow \alpha$ **assume** $\mathit{id} \equiv \lambda x :: \alpha.\ x$, for the fixed (hypothetical) type $\alpha$. There is no *let*-polymorphism at that stage, because the logic lacks type quantification.

**Generic Context Data.** The Isar proof context is able to assimilate arbitrary user-data in a type-safe fashion, using a functor interface in ML (see also [19, §3]). This means almost everything can be turned into context data. Common examples include type-inference information (constraints), concrete syntax (mixfix grammar), or hints for automated reasoning tools.

The global theory is only extended monotonically, but Isar contexts support opening and closing of local scopes. Moving between contexts requires replacing references to hypothetical types, terms, and proofs within user data accordingly. The Isar framework cannot operate on user data due to ML's type-safety, but a slightly different perspective allows us to transform arbitrary content, by passing through an explicit *morphism* in just the right spot, cf. [7, §3–4].

So instead of transforming fully abstract data directly, the framework transforms *data declarations*, i.e. implementation specific functions that maintain the data in the context. The interface for this is the generic context element "**declaration** $d$", where $d$: *morphism* $\to$ *context* $\to$ *context* is a data operation provided by some external module implementing context data. The morphism is provided by the framework at some later stage, it determines the differences of the present abstract context wrt. the concrete application environment, by providing mappings for the three logical categories of types, terms, and theorems. The implementation of $d$ needs to apply this morphism wherever logical entities occur in the data; see [7, §2] for a simple example.

Using the Isabelle/Isar infrastructure on top of the raw logic, we can now introduce the concept of *constant abbreviations* that are type-checked like global polymorphic constants locally, but expanded before the logic ever sees them:

$$\textbf{abbrev}\ c :: \forall \overline{\beta}.\ \tau[\overline{\alpha}, \overline{\beta}]\ \textbf{where}\ \forall \overline{\beta}.\ c[\overline{\beta}] \equiv \mathit{rhs}[\overline{\alpha}, \overline{\beta}]$$

Here the type variables $\overline{\alpha}$ need to be fixed in the context, but $\overline{\beta}$ is arbitrary. In other words, we have conjured up proper *let*-polymorphism in the abstract syntax layer of Isabelle/Isar, without touching the Pure logic.

## 4 Local Theory Infrastructure

We are now ready to introduce the key concept of *local theory*, which models the general idea of interpreting definitional elements relatively to a local context.

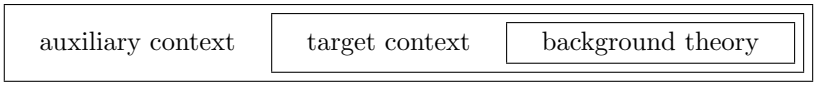
Basic specification elements shall be explicitly separated into two categories: axiomatic **fix**/**assume** vs. definitional **define**/**note**. Together with our implicit treatment of types, this achieves an orthogonal arrangement of $\lambda$- and *let*-bindings for all three logical categories as follows:

| | $\lambda$-binding | *let*-binding |
|---|---|---|
| types | fixed $\alpha$ | arbitrary $\beta$ |
| terms | **fix** $x :: \tau$ | **define** $c \equiv t$ |
| theorems | **assume** $a$: $A$ | **note** $b = \langle B \rangle$ |

A local theory specification is divided into a *target* part, which is derived from the background theory by adding axiomatic elements, and a *body* consisting

of any number of definitional elements. The target also provides a particular interpretation of definitional primitives. Concrete body elements are produced by definitional packages invoked within corresponding **begin**/**end** blocks (cf. §2).

The key duality is that of background theory vs. target context, but there is also an *auxiliary context* that allows to hide the effect of the target interpretation internally. So the general structure of a local theory is a sandwich of three layers:




This allows one to make **define** appear like **vardef** and **note** like **thm** (cf. §3.2), while the main impact on the target context and background theory is exposed to the end-user only later. By fixing the body elements and their effect on the auxiliary context once and for all, we achieve a generic programming interface for definitional packages that work uniformly for arbitrary interpretations.

**Canonical Interpretation via λ-Lifting.** Subsequently we give a formal explanation of local theory interpretation by the blue-print model of $\lambda$-lifting over a fixed context **type** $\alpha$ **fix** $x :: \tau[\alpha]$ **assume** $a$: $A[\alpha][x]$. Restricting ourselves to a single variable of each category avoids cluttered notation; generalization to multiple parameters and assumptions is straightforward.

The idea is that **define** $a \equiv b[\alpha, \beta][x]$ relative to fixed $\alpha$ and $x :: \tau[\alpha]$ becomes a constant definition with explicit abstraction over the term parameter and generalization over the type parameters; the resulting theorem is re-imported into the target context by instantiation with the original parameters. The illusion of working fully locally is completed in the auxiliary context, by using hidden equational assumptions (see below).

The same principle works for **note** $a = \langle B[\alpha, \beta][x]\rangle$, but there is an additional dependency on assumption $a$: $A$, and the lifting over parameters is only partially visible, because proof terms are implicit.

The following $\lambda$-lifting scheme works for independent **define** and **note** elements, in an initial situation where the target and auxiliary context coincide:

| specification | **define** $a \equiv b[\alpha, \beta][x]$ |
|---|---|
| 1. background theory | **constdef** $\forall \alpha\ \beta.\ thy.a \equiv \lambda x.\ b[\alpha, \beta][x]$ |
| 2. target context | **abbrev** $\forall \beta.\ loc.a \equiv thy.a[\alpha, \beta]\ x$ |
| 3. auxiliary context | **vardef** $a \equiv thy.a[\alpha, \beta]\ x$ |
| local result | $\langle a \equiv b[\alpha, \beta][x]\rangle$ |

| specification | **note** $a = \langle B[\alpha, \beta][x]\rangle$ |
|---|---|
| 1. background theory | **thmdef** $\forall \alpha\ \beta.\ thy.a = \langle \bigwedge x.\ A[\alpha][x] \Longrightarrow B[\alpha, \beta][x]\rangle$ |
| 2. target context | **thm** $\forall \beta.\ loc.a = \langle B[\alpha, \beta][x]\rangle$ |
| 3. auxiliary context | **thm** $a = \langle B[\alpha, \beta][x]\rangle$ |
| local result | $\langle B[\alpha, \beta][x]\rangle$ |

This already illustrates the key steps of any local theory interpretation. Step (1) jumps from the auxiliary context right into the background theory to provide

proper foundation, using global primitives of **constdef** and **thmdef**. Step (2) is where the particular target view is produced, here merely by applying fixed entities to revert the abstractions; other targets might perform additional transformations. Note that $\beta$ is arbitrary in the target context. Step (3) bridges the distance of the target and auxiliary context, to make the local result appear literally as specified (with fixed $\beta$).

Extra care is required when mixing several **define** and **note** elements, as subsequent terms and facts may depend on the accumulated auxiliary parameters introduced by **vardef**. Export into the background theory now involves definitional expansion, and import into the auxiliary context folding of hypothetical equations. Here is the interpretation of **note** $a = \langle B[\alpha,\ \beta][c,\ x]\rangle$ depending on a previous **define** $c \equiv b[\alpha,\ \beta][x]$:

| | |
|---|---|
| | **note** $a = \langle B[\alpha,\ \beta][c,\ x]\rangle$ |
| 1. | **thmdef** $\forall\, \alpha\ \beta.\ thy.a = \langle \bigwedge x.\ A[\alpha][x] \Longrightarrow B[\alpha,\ \beta][thy.c[\alpha,\ \beta]\ x,\ x]\rangle$ |
| 2. | **thm** $\forall\, \beta.\ loc.a = \langle B[\alpha,\ \beta][thy.c[\alpha,\ \beta]\ x,\ x]\rangle$ |
| 3. | **thm** $a = \langle B[\alpha,\ \beta][c,\ x]\rangle$ |
| | $\langle B[\alpha,\ \beta][x]\rangle$ |

Each **define** element adds another **vardef** to the auxiliary context, to cater for internal term dependencies of any subsequent **define**/**note** within the same specification package (e.g. **inductive**). Thus package implementors need not care about term dependencies, but work directly with local variables (and with fixed types). Whenever a package concludes, our infrastructure resets the auxiliary context to the current target context, so the user will continue with polymorphic constant abbreviations standing for global terms. Only then, types appear in most general form according to the Hindley-Milner discipline, as expected by the end-user.

# 5 Common Local Theory Targets

## 5.1 Global theories

A global theory is a trivial local theory, where the target context coincides with the background theory. The canonical interpretation (§4) is reduced as follows:

| | |
|---|---|
| | **define** $a \equiv b[\beta]$ |
| 1. | **constdef** $\forall\, \beta.\ thy.a \equiv b[\beta]$ |
| 2. | (omitted) |
| 3. | **vardef** $a \equiv thy.a[\beta]$ |
| | $\langle a \equiv b[\beta]\rangle$ |

| | |
|---|---|
| | **note** $a = \langle B[\beta]\rangle$ |
| 1. | **thmdef** $\forall\, \beta.\ thy.a = \langle B[\beta]\rangle$ |
| 2. | (omitted) |
| 3. | **thm** $a = \langle B[\beta]\rangle$ |
| | $\langle B[\beta]\rangle$ |

Here we trade a fixed term variable $a$ (with fixed type) for a global constant $thy.a$ (with schematic type). The auxiliary context hides the difference in typing and name space details. This abstract view is a considerable advantage for package implementations, even without using the full potential of local theories yet.

The Isabelle/Isar toplevel ensures that local theory body elements occurring on the global level are wrapped into a proper target context as sketched above. Thus a local theory package like **inductive** may be invoked seamlessly in any situation.

### 5.2 Locales

Locales [12] essentially manage named chunks of Isabelle/Isar context elements (§3.2), such as **locale** *loc* = **fixes** $x$ **assumes** $A[x]$, together with **declaration** elements associated with conclusions. Locale expressions [1] allow to combine atomic locales *loc*, either by renaming *loc* $y$ or merge $loc_1$ + $loc_2$ of the underlying axiomatic specifications. Results stemming from a locale expression may be interpreted later, giving particular terms and proofs for the axiomatic part [2].

The key service provided by the locale mechanism is that of breaking up complex expressions into atomic locales. Down at that level, it hands over to the generic local theory infrastructure, by providing a target context that accepts **define** and **note** according to the canonical $\lambda$-lifting again (§4). There is only one modification: instead of working with primitive **fixes** and **assumes** of the original **locale** definition, there is an additional indirection through a global predicate definition **constdef** $thy.loc \equiv \lambda x.\ A[x]$.

The Isabelle/Isar toplevel initializes a locale target for "**context** *loc* **begin** *body* **end**" or the short version "*specification* (**in** *loc*)"; the latter generalizes the traditional "**theorem** (**in** *loc*)" form [12] towards arbitrary definitions within locales, including derived mechanisms like "**inductive** (**in** *loc*)".

### 5.3 Type Classes

**Type-class Specification.** Logically a type class is nothing more than an interpretation of a locale with *exactly one* type variable $\alpha$, see [11]. Given such a locale $c[\alpha]$ with fixed type $\alpha$, *class parameters* $\overline{g[\alpha]}$ and predicate $thy.c$, the corresponding type class $c$ is established by the following interpretation, where the right column resembles the traditional **axclass** scheme [17]:

| locale specification | class interpretation |
|---|---|
| **locale** $c$ = | **classdecl** $c$ = |
| **fixes** $\overline{g :: \tau[\alpha]}$ | **const** $\overline{c.g :: \tau[\gamma::c]}$ |
| **assumes** $thy.c$ $\overline{g :: \tau[\alpha]}$ | **axiom** $thy.c$ $\overline{c.g[\gamma::c]}$ |

The class target augments the locale target (§5.2) by a second interpretation within the background theory where conclusions are relative to global constants $\overline{c.g[\gamma::c]}$ and class axiom $thy.c$ $\overline{c.g[\gamma::c]}$, for arbitrary types $\gamma$ of class $c$.

| specification | **define** $f \equiv t[\alpha, \beta][\overline{g}]$ |
|---|---|
| 1. background theory | **constdef** $\forall \alpha\ \beta.\ thy.f \equiv \lambda \overline{x}.\ t[\alpha, \beta][\overline{x}]$ |
| 2a. locale target | **abbrev** $\forall \beta.\ loc.f \equiv thy.f[\alpha, \beta]\ \overline{g}$ |
| 2b. class target | **constdef** $\forall \gamma{::}c\ \beta.\ c.f \equiv thy.f[\gamma, \beta]\ \overline{c.g}$ |
| 3. auxiliary context | **vardef** $f \equiv thy.f[\alpha, \beta]\ \overline{g}$ |

| specification | **note** $a = \langle B[\alpha, \beta][\overline{g}] \rangle$ |
|---|---|
| 1. background theory | **thmdef** $\forall \alpha\ \beta.\ thy.a = \langle \bigwedge \overline{x}.\ A[\alpha][\overline{x}] \Longrightarrow B[\alpha, \beta][\overline{x}] \rangle$ |
| 2a. locale target | **thm** $\forall \beta.\ loc.a = \langle B[\alpha, \beta][\overline{g}] \rangle$ |
| 2b. class target | **thmdef** $\forall \gamma{::}c\ \beta.\ c.a = \langle B[\gamma, \beta][\overline{c.g}] \rangle$ |
| 3. auxiliary context | **thm** $a = \langle B[\alpha, \beta][\overline{g}] \rangle$ |

The interpretation (2b) of **fixes** and **define** $f\ [\alpha, \beta]$ establishes a one-to-one correspondence with *class constants* $c.f\ [\gamma{::}c, \beta]$, such that $f\ [\alpha, \beta]$ becomes $c.f\ [\gamma{::}c, \beta]$. When interleaving **define** and **note** elements, the same situation occurs as described in §4 — hypothetical definitions need to be folded:

| | **note** $a = \langle B[\alpha, \beta][f, \overline{g}] \rangle$ |
|---|---|
| 1. | **thmdef** $\forall \alpha\ \beta.\ thy.a = \langle \bigwedge \overline{x}.\ A[\alpha][\overline{x}] \Longrightarrow B[\alpha, \beta][thy.f[\alpha, \beta]\ \overline{g}, \overline{g}] \rangle$ |
| 2a. | **thm** $\forall \beta.\ loc.a = \langle B[\alpha, \beta][thy.f[\alpha, \beta]\ \overline{g}, \overline{g}] \rangle$ |
| 2b. | **thmdef** $\forall \gamma{::}c\ \beta.\ thy.a = \langle B[\gamma, \beta][c.f[\gamma, \beta], \overline{c.g}] \rangle$ |
| 3. | **thm** $a = \langle B[\alpha, \beta][f, \overline{g}] \rangle$ |

**Type-class Instantiation.** The **instantiation** target integrates type classes and overloading by providing a Haskell-like policy for class instantiation: each arity $\kappa :: (\overline{s})\ c$ is associated with a set of class parameters $c.g\ [(\overline{\delta{::}s})\ \kappa]$ for which specifications are given which respect the **assumes** of $c$. As auxiliary means, at the **begin** of an **instantiation**, each of these $c.g\ [(\overline{\delta{::}s})\ \kappa]$ is associated with a corresponding shadow variable $\kappa.g\ [\overline{\delta{::}s}]$. These are treated specifically in subsequent **define** elements:

| | **define** $\kappa.g\ [\overline{\delta{::}s}] \equiv t$ |
|---|---|
| 1. | **constdef** $\forall \overline{\delta{::}s}.\ c.g\ [(\overline{\delta{::}s})\ \kappa] \equiv t$ |
| 2. | (omitted) |
| 3. | **vardef** $\kappa.g\ [\overline{\delta{::}s}] \equiv c.g\ [(\overline{\delta{::}s})\ \kappa]$ |

In other words, **define** is interpreted by overloaded **constdef**. Further occurrences of **define** or **note**, unrelated to the class instantiation, are interpreted as in the global theory target (§5.1). As an additional policy the **instantiation** target requires all class parameters to be specified before admitting the obligatory **instance** proof before the **end**.

**Target Syntax.** As seen in §2 both the **class** and **instantiation** context allows us to refer to whole families of corresponding constants uniformly. The idea is to let the user write $c.f$ unambiguously, and substitute this internally according to the actual instantiation of the class type parameter $[\gamma{::}c]$. This works by splitting conventional order-sorted type inference into three phases:

| phase | **class** target | **instantiation** target |
|---|---|---|
| 1. type inference | with class constraint $\gamma::c$ on $c.f$ disregarded | |
| 2a. type improvement | $c.f\ [?\xi] \rightsquigarrow c.f\ [\alpha]$ | $c.f\ [?\xi] \rightsquigarrow c.f\ [(\overline{\delta::s})\ \kappa]$ |
| 2b. substitution | $c.f\ [\alpha] \rightsquigarrow f\ [\alpha]$ | $c.f\ [(\overline{\delta::s})\ \kappa] \rightsquigarrow \kappa.f\ [\overline{\delta}]$ |
| 3. type inference | with all constraints, fixing remaining inference parameters | |

To permit writing terms $c.f\ [\alpha]$ for $\alpha$ even without a class constraint, first the class constraint $\gamma::c$ on $c.f$ is disregarded in phase (1) and is re-considered in phase (3); in between, types left open by type inference are still improvable type inference parameters $?\xi$.

Whenever $c.f$ is used with a characteristic type parameter ($\alpha$ in **class** case, $(\overline{\delta::s})\ \kappa$ in **instantiation** case), it is substituted by the appropriate parameter ($f$ for **class** or $\kappa.f$ for **instantiation**) in phase (2b); more general occurrences $c.f\ [\gamma]$ are left unchanged. This allows to write $c.f$ uniformly for both local and global versions.

To relieve the user from cumbersome type annotations, a *type improvement* step is carried out (2a): if $c.f$ carries a type inference parameter $?\xi$, this is specialized to the characteristic type parameter. This step will hand over $c.f$ with completely determined type information.

As a particularity of the **instantiation** target, the substitution $c.f\ [(\overline{\delta::s})\ \kappa] \rightsquigarrow \kappa.f\ [\overline{\delta}]$ is only carried out while no **define** $\kappa.f\ [\overline{\delta::s}] \equiv t$ has occurred yet; afterwards, occurrences of $c.f\ [(\overline{\delta::s})\ \kappa]$ are taken literally.

When printing terms, substitutions are reverted: $f\ [\alpha] \rightsquigarrow c.f\ [\alpha]$ for **class**, and $\kappa.f\ [\overline{\delta}] \rightsquigarrow c.f\ [(\overline{\delta::s})\ \kappa]$ for **instantiation**. Thus the surface syntax expected by the end-user is recovered.

## 6 Conclusion

*Related Work.* Structured specifications depending on parameters and assumptions are closely related to any variety of "*modular logic*", which may appear in the guise of *algebraic specification*, *little theories* etc. Many module systems for proof assistants have been developed in the past, and this is still a matter of active research. Taking only Coq [4] as example, there are "structures" (a variety of record types), "sections" (groups of definitions and proofs depending on parameters and assumptions), and "modules" that resemble ML functors.

Our approach of building up specification contexts and the canonical interpretation of body elements by $\lambda$-lifting is closely akin to "sections" in Coq. Here we continue the original locale idea [12], which was presented as a "sectioning concept for Isabelle" in the first place. There are two main differences to Coq sections: our axiomatic target needs to be fixed once and for all, but the definitional body may be extended consecutively. In Coq both parts are intermingled, and cannot be changed later. Note that Coq sections vanish when the scope is closed, but a local theory may be recommenced.

Beyond similarities to particular module systems our approach is different in providing a broader scope. Acknowledging the existence of different module concepts, we offer a general architecture for integrating them into a common framework. After implementing a suitable target mechanism, a particular module concept will immediately benefit from body specification elements, as produced by existing definitional packages (**inductive**, **primrec**, **function**, etc.).

Fitting a module system into our framework requires a representation of its logical aspects within Isabelle/Pure, and any auxiliary infrastructure as Isabelle/Isar context. The latter is very flexible thanks to generic context data (covering arbitrary ML values in a type-safe fashion), and generic "declarations" for maintaining such data depending on a logical morphism. The Pure framework [15] supports higher-order logic, but only simple types without type quantification [8]. The particular targets presented here demonstrate that non-trivial modular concepts can indeed be mapped to the Pure logic, including an illusion of local type-quantification (for definitions) according to Hindley-Milner [13].

In fact, there is no need to stay within our canonical interpretation of $\lambda$-lifting at all. This template may be transcended by using explicit proof terms in Pure, to enable more general "admissible" principles in the interpretation. For example, the AWE tool [6] applies theory interpretation techniques directly to global type constructors, constants and axioms. This allows one to operate on polymorphic entities, as required for an abstract theory of monads, for example. In AWE, definitions and theorems depending on such global axiomatizations are transformed extra-logically by mapping the corresponding proof objects, and replaying them in the target context. The present implementation needs to redefine some common specification elements. Alternatively, one could present this mechanism as another local theory target, enabling it to work with any local definitional package, without requiring special patches just for AWE.

*Implementation and Applications.* Since Isabelle2008, local theories are the official interface for implementing derived specification mechanisms within the Isabelle framework. The distribution includes the general framework, with a couple of targets and definitional packages for the new programming interface. We already provide target mechanisms for global theories, locales, type classes and class instantiations as described above. There is another target for raw overloading without the type-class discipline. Moreover, the following body specifications are available:

- **definition** and **theorem** as wrappers for the **define** and **note** primitives
- **primrec** for structural recursion over datatypes
- **inductive** and **coinductive** for recursive predicates and sets (by Stefan Berghofer)
- **function** for general recursive functions (by Alexander Krauss)
- **nominal-inductive** and **nominal-primrec** for specifications in nominal logic (by Christian Urban and Stefan Berghofer)

Experience with such "localization" efforts of existing packages indicates that conversion of old code is reasonably easy; package implementations can usually

be simplified by replacing primitive specifications by our streamlined local theory elements. Some extra care is required since packages may no longer maintain "global handles" on results, e.g. the global constant name of an inductively defined predicate. Such references to logical entities need to be generalized to arbitrary terms. Due to interpretation of the original specification in a variety of targets, one cannot count on particular global results, but needs to work with explicit Isar contexts and morphisms on associated data.

**Acknowledgments.** Tobias Nipkow and Alexander Krauss greatly influenced the initial "local theory" design (more than 2 years ago), by asking critical questions about definitions within locales. Early experiments with inductive definitions by Stefan Berghofer showed that the concept of "auxiliary context" is really required, apart from the "target context". Amine Chaieb convinced the authors that serious integration of locales and type-classes is really needed for advanced algebraic proof tools. Clemens Ballarin helped to separate general local theory principles from genuine features of locales. Norbert Schirmer and other early adopters helped to polish the interfaces.

## References

1. Ballarin, C.: Locales and locale expressions in Isabelle/Isar. In: S. Berardi, et al. (eds.) Types for Proofs and Programs (TYPES 2003), *LNCS*, vol. 3085. Springer-Verlag (2004)
2. Ballarin, C.: Interpretation of locales in Isabelle: Theories and proof contexts. In: J.M. Borwein, W.M. Farmer (eds.) Mathematical Knowledge Management (MKM 2006), *LNAI*, vol. 4108. Springer-Verlag (2006)
3. Barendregt, H., Geuvers, H.: Proof assistants using dependent type systems. In: A. Robinson, A. Voronkov (eds.) Handbook of Automated Reasoning. Elsevier (2001)
4. Barras, B., et al.: The Coq Proof Assistant Reference Manual, v. 8.1. INRIA (2006)
5. Bertot, Y., et al. (eds.): Theorem Proving in Higher Order Logics (TPHOLs 1999), *LNCS*, vol. 1690. Springer-Verlag (1999)
6. Bortin, M., Broch Johnsen, E., Lüth, C.: Structured formal development in Isabelle. Nordic Journal of Computing **13** (2006)
7. Chaieb, A., Wenzel, M.: Context aware calculation and deduction — ring equalities via Gröbner Bases in Isabelle. In: M. Kauers, et al. (eds.) Towards Mechanized Mathematical Assistants (CALCULEMUS and MKM 2007), *LNAI*, vol. 4573. Springer-Verlag (2007)
8. Church, A.: A formulation of the simple theory of types. J. Symbolic Logic (1940)
9. Gordon, M.J.C., Melham, T.F. (eds.): Introduction to HOL: A theorem proving environment for higher order logic. Cambridge University Press (1993)
10. Gunter, E.L., Felty, A. (eds.): Theorem Proving in Higher Order Logics (TPHOLs 1997), *LNCS*, vol. 1275. Springer-Verlag (1997)
11. Haftmann, F., Wenzel, M.: Constructive type classes in Isabelle. In: T. Altenkirch, C. McBride (eds.) Types for Proofs and Programs (TYPES 2006), *LNCS*, vol. 4502. Springer-Verlag (2007)

12. Kammüller, F., Wenzel, M., Paulson, L.C.: Locales: A sectioning concept for Isabelle. In: Bertot et al. [5]
13. Milner, R.: A theory of type polymorphism in programming. J. Computer and System Sciences (17) (1978)
14. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer-Verlag (2002)
15. Paulson, L.C.: Isabelle: the next 700 theorem provers. In: P. Odifreddi (ed.) Logic and Computer Science. Academic Press (1990)
16. Slind, K.: Function definition in higher-order logic. In: Gunter and Felty [10]
17. Wenzel, M.: Type classes and overloading in higher-order logic. In: Gunter and Felty [10]
18. Wenzel, M.: Isar — a generic interpretative approach to readable formal proof documents. In: Bertot et al. [5]
19. Wenzel, M., Wolff, B.: Building formal method tools in the Isabelle/Isar framework. In: K. Schneider, J. Brandt (eds.) Theorem Proving in Higher Order Logics (TPHOLs 2007), *LNCS*, vol. 4732. Springer-Verlag (2007)