Building Formal Method Tools in the Isabelle/Isar Framework

Makarius Wenzel[1*] and Burkhart Wolff[2]

[1] Technische Universität München, Institut für Informatik,
http://www.in.tum.de/~wenzelm/
[2] ETH Zürich, Information Security,
http://www.infsec.ethz.ch/people/wolffb

Abstract We present the generic system framework of Isabelle/Isar underlying recent versions of Isabelle. Among other things, Isar provides an infrastructure for Isabelle plug-ins, comprising extensible state components and extensible syntax that can be bound to tactical ML programs. Thus the Isabelle/Isar architecture may be understood as an extension and refinement of the traditional "LCF approach", with explicit infrastructure for building derivative systems. To demonstrate the technical potential of the framework, we apply it to a concrete formal methods tool: the HOL-Z 3.0 environment, which is geared towards the analysis of Z specifications and formal proof of forward-refinements.

1 Introduction

Nearly 15 years ago, Paulson concluded his handbook article on theorem prover design [18] with the "final advice":

Don't write a theorem prover. Try to use someone else's.

Even today, reuse of existing prover technology is still not common practice — many recent research projects are still based upon attempts to start from scratch, in particular if formal methods are implemented. This has mostly three reasons: Besides (1) general ignorance over or scepticism about the logical framework approach and (2) concerns about efficiency, it is believed that (3) the overhead to represent a particular formal method in a generic system is in fact too large.

There is a remarkable body of literature addressing (1) and (2); various forms of *logical embeddings* have been studied and applied to a wide range of logics or specification languages. Furthermore, a wide range of techniques to *integrate decision procedures* have been developed. Here the usual alternatives are fully expansive tactics, or procedures generating checkable proof objects or external implementations integrated via an oracle-mechanism into the LCF proof engine.

In this paper, we want to address (3) and explore the much less known fact that underlying the most recent versions of Isabelle (which Paulson had in mind

[*] Supported by BMBF project "Verisoft".

in the above quote) there is a generic system framework called Isabelle/Isar that can be compared in a rough analogy to the Eclipse programming system framework. Some key aspects of Isabelle/Isar are

1. hierarchical organization of theory documents,
2. *incremental* document processing for interactive theory and proof development (with unlimited undo),
3. batch-mode processing for high-quality document preparation (LaTeX),
4. extensible syntax for toplevel commands, embedded methods and attributes, and the inner term language,
5. type-safe programming interfaces for generic user data within the logical environment,
6. LCF-style programming interfaces for derived rules, tactics etc.

This framework has been applied to build a family of *formal method tools.* With this category we refer to a class of tools that provide support for a software specification formalism and a pragmatics or formalized "method" to use it for software analysis. Examples of such FM tools implemented as Isabelle/Isar "plugins" are the test-case generation system HOL-TestGen [11, 2], the environment for object-oriented modeling HOL-OCL [4, 10], or the proof-environment for refinement-oriented system development HOL-Z 3.0 [5]. In the present paper we take the latter as a running example, since the logical embedding [16] and the proof-support [9] have been described earlier, the overall plug-in is still relatively small (12000 lines of code if the Java-based front-end is included), and since there exists a "monolithic" implementation [3] of an environment for Z based on a HOL theorem prover offering the potential of a fair comparison.

The analysis method of HOL-Z 3.0 (built on top of HOL within Isabelle2005) is based on the idea that a designer writes a specification document in the specification language Z. This can be done completely independent from Isabelle in an Emacs/ZETA environment providing type-checking and elementary animation of Z formulae [1]. In a further step, the designer might want to state a number of conjectures on his model. In particular, these may be *analytical statements* such as "A is refined by B" or "spec A is consistent" in order to describe the task to proof engineers. Depending on the underlying method, such statements were compiled to *proof-obligations* (PO), i. e. formulas to be proven at the end by the proof-engineer using interactive or automated proof techniques.

The paper proceeds as follows. In §2 we introduce the HOL-Z 3.0 tool as it appears to end-users. In §3 we outline the main aspects of the Isabelle/Isar architecture that are relevant for building formal method tools. In §4 we report on the implementation of HOL-Z 3.0 within the Isabelle/Isar framework. In §5 we compare our Isabelle/HOL-Z system with the more monolithic ProofPower.

2 A Guided Tour through HOL-Z

2.1 The HOL-Z 3.0 System Architecture

As shown in Figure 1, the model development process with Isabelle/HOL-Z is divided into two phases. In the modeling phase, the designer concentrates his attention on describing a model of the system in Z. The model (a LaTeX document) is type-checked by ZETA [1] and passed to the verification phase. In the latter, the proof engineer can prove proof obligations — e. g. stemming from an interactive refinement — using proof commands. The theorem proving process is recorded in proof documents. Both ZETA as well as HOL-Z support "literate specifications", where formal specifications and proofs are mixed with informal explanations and the system generates a final document after checking all proofs. Technically, Isabelle/HOL-Z consists of a shallow embed-

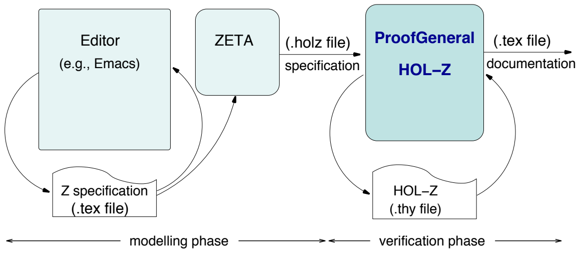

Figure 1. The HOL-Z system architecture perspective

ding of the Z language constructs, a library called the *mathematical toolkit*, a compiler converting ZETA-output into the logical representation, a number of generic proof-procedures providing automatic support for Z-specific constructs like *schemas* (sets of records), or the *schema calculus* with its own quantifier and logical connectives. HOL-Z also provides proof-obligation management instantiated with a concrete method, namely the forward refinement method for Z as described in Spivey's book [19]. A complete HOL-Z spec and proof documentation for Spivey's "BirthdayBook" example, a toy data-base system presented in an abstract and a concrete version and related via refinement proofs, can be downloaded as "standard example" via the HOL-Z home page [5]. There are also references to substantial case-studies that have been done with the system.

2.2 The HOL-Z 3.0 Workflow

The proof document can be processed interactively via Proof General [6]. Subsequently, we highlight the main steps taken when verifying the birthday book example; the reader interested in the details is referred to the complete download of the "standard example". We start by loading the output of ZETA when type-checking the model of the designer:

```
load_holz "BBSpec"
```

This results in an environment holding all definitions and various automatically derived simplification rules. In particular, this includes the state invariants for the abstract state *BirthdayBook* and the concrete *BirthdayBook*1, the abstract insertion operation *AddBirthday* and its concrete counterpart *AddBirthday*1 and the abstraction relation *Abs*. In the BirthdayBook example, all these definitions are presented in the Z idiom as schemas. By typing

```
    set_abs "Abs" [functional]
```

we inform the refinement package that the *Abs* schema represents the abstraction relation and set the system in a mode for functional refinement (i. e. it will generate simplified POs). This setting will also generate a PO requiring that *Abs* actually represents a function, i. e. any concrete state is in fact related to a unique abstract state. After the statement:

```
    refine_op AddBirthday Addbirthday1
```

we will have two new POs (we will show the second below) in the proof obligation database, which can be listed and inspected by the user. The concrete instance for the second PO resulting from the statement above can be referenced via a PO name. For example, if we *discharge* this PO, we can refer to it by:

```
    po "BBSpec_functional.fw_refinementOp_AddBirthday"
```

The system reacts by displaying this well-known condition of forward-simulation. In more detail, this means that whenever the concrete system transition relation *AddBirthday*1 relates a concrete state in *BirthdayBook*1 to a state in *BirthdayBook*1′ and whenever an abstract state *BirthdayBook* can be related via *Abs* to a state in *BirthdayBook*1, then *AddBirthday* must allow a transition to the abstract state *BirthdayBook*′ which is an abstraction of *BirthdayBook*1′:

$$\begin{aligned}&\forall\, BirthdayBook \bullet \forall\, BirthdayBook' \bullet \forall\, BirthdayBook1 \bullet \forall\, BirthdayBook1' \bullet\\ &\quad(\forall\, date? : DATE. \forall\, name? : NAME.\\ &\qquad preAddBirthday \land Abs \land AddBirthday1 \land Abs' \Rightarrow AddBirthday)\end{aligned}$$

Recall that in the above Z formula *BirthdayBook* etc. refer to schemas, which implies that there are implicit parameters (occasionally decorated by ′). The Z specific binding structure is suppressed in pretty-printing. The `po` command initializes the proof state, which may now be refined by a sequence of regular Isabelle proof commands [21] (such as `apply`) that refer to proof methods from generic Isabelle/HOL or specific ones from HOL-Z.

After reaching the final proof state, one can state

```
    discharged
```

whereby this PO will be erased from the proof obligation database.

3 Internals: The Isabelle/Isar System Architecture

The system architecture of Isabelle/Isar [22] extends the original "LCF approach". We briefly review the latter, before explaining our additional concepts.

3.1 The "LCF approach"

The "LCF approach" was pioneered by the original LCF system due to Robin Milner [14]. LCF means "Logic of Computable Functions", but the underlying logic is not really relevant for the architecture. The same principles have been transferred later to the HOL family [13, 12], and some traces are also present in Coq [7]. The main idea of the "LCF approach" is twofold:

1. A full functional programming language acts as "meta language" (ML), which allows arbitrary manipulations of term entities represented as values. ML is *not* part of the logic, but provides access to its implementation.
2. The strong type-discipline of ML ensures "correctness-by-construction", as all critical information is wrapped into abstract datatypes. In particular, an abstract type *thm* guarantees that any value of that type is in fact derivable, relative to a small kernel implementing the primitive inferences of the logic.

Figure 2 illustrates the main ML types of a typical "LCF-style" system. In this diagram a solid line means structural containment (reading downwards, e. g. a *term* is contained in a *thm*), while dashed lines link operand-operation pairs (e. g. a *tactic* operates on a *goal*). The three columns categorize entities according to their typical use: basic values (manipulated directly), state information (handled implicitly), and the main "active" operations invoked by the user.

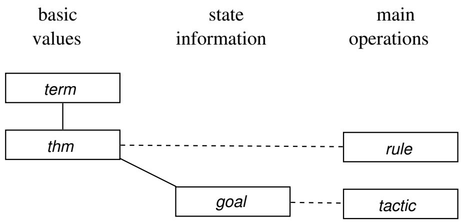

Figure 2. The original "LCF approach"

The concrete datatype *term* provides a syntactic model of λ-terms (consisting of free/bound variables, abstraction, application), with the usual operations (substitution, reduction etc.). Terms are annotated by explicit type-information, which may have to be re-checked explicitly for critical operations (this is usually easy due to the restriction to simple types).

The abstract type *thm* implements primitive inference objects of the underlying logic (e. g. HOL). Since ML enforces type-safety, the *thm* type does not need to store any explicit information about the inferences that lead to a particular theorem. Some systems provide an explicit proof trace as an option, but this typically increases space requirements by several orders of magnitude.

Type *rule* covers functions that produce theorems, e. g. unary $thm \to thm$, binary $thm \to thm \to thm$, or parameterized ones $term \to thm \to thm$. General rules programmed in ML need to replay the constituent inferences for every application at run-time. An important special class of *immediate derived rules* may be implemented more efficiently, using auxiliary theorems that internalize the rule as a statement of the logic; this technique essentially exploits the Deduction Theorem. In Isabelle, derived rules are internalized by default, using the $\bigwedge$ quantifier and $\Longrightarrow$ connective of the Pure logical framework to represent Natural Deduction schemes [17]. Note that in conclusions, the outermost prefix of $\bigwedge x$ quantifiers is turned into schematic variables $?x$ of Isabelle.

Beyond primitive inferences, the LCF approach includes some minimal infrastructure for tactical theorem proving, which supports backwards reasoning from a *goal* by a sequence of *tactic* applications. In Isabelle, a *goal* is again a *thm* stating that the sub-goals entail the main conclusion, and a *tactic* is a lazy function $goal \rightarrow goal^{**}$ that enumerates possible follow-up goals [17]. The tactical layer of the LCF approach imposes a particular discipline of goal-oriented reasoning, which has turned out quite successful very early [14]: the art of representing problems as goals (including auxiliary facts within the statement) and complex manipulations as tactics has thrived over several decades.

Beyond programming tactics, people have also managed to build advanced specification mechanisms that turn high-level user input into primitive definitions and proven theorems. This approach has been employed many times to implement inductive sets and datatypes, recursive functions etc. on top of the core logic implementation. E. g. see [15] or [8] for techniques of bootstrapping higher specification concepts in HOL.

Limitations. Despite this success in building complex logical environments from basic principles, there has traditionally been very little general system infrastructure to support this task systematically. Consequently, "LCF system programming" has often a flavor of working on bare-metal. Moreover, implementors of particular extensions often re-invent auxiliary infrastructure, which duplicates work and tends to result in a diversity of incompatible features.

To illustrate the limitations of raw LCF, consider the problem of combining advanced tactics in classic Isabelle. Given tactic *simp* (the Simplifier, implemented by N.) that depends on a private rule collection, which is represented by type *simpset* and maintained by *addsimp*, *delsimp*: $thm \rightarrow simpset \rightarrow simpset$; given another family of tactics *fast*, *blast* etc. (the Classical Reasoner, implemented by P.) that depend on a *claset* with similar operations; then a combined tactic *force* (implemented by O.) depends on $clasimpset = claset \times simpset$. This is slightly awkward, since declarations on the individual components (*addsimp* on *simpset* etc.) need to be lifted to the tupled container, and users need to know which container type is passed to which tactic exactly. Even worse, this technique requires the participant tactic families and corresponding container types to be known in advance. So this naive composition does not scale well.

Our refined LCF approach addresses such issues by a generic environment that assimilates arbitrary data in a type-safe fashion. So all tactics may depend on a uniform *context*, which is maintained by declarations on the same type.

3.2 The Isabelle/Isar Framework

Although the original motivation for Isabelle/Isar is to support human-readable proof texts [20], the internal system organization has been quite generic from early on, continuing the original idea of Isabelle as a "logical framework" [17] in a broader sense. Consequently, the mechanisms for structured proofs are merely another application of considerably more general infrastructure, which may be re-used in completely different applications as well. Figure 3 gives an overview of the main constituents of the Isabelle/Isar framework. In this diagram, a solid or dashed line means the same as in Figure 2, while an arrow means a backwards reference that may mutate in a monotonic fashion (as explained below).

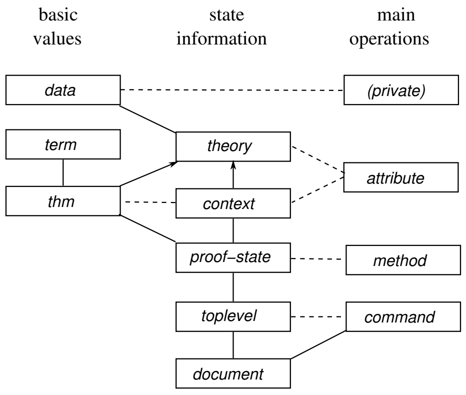

Figure 3. Main concepts of Isabelle/Isar

Here the original entities of the "LCF approach" are still present, although some have been elevated to more sophisticated concepts.

Generic *data*, which can be modeled by the user as arbitrary ML types, is organized explicitly within a *theory* or *context*. Operations on *data* are private to the particular module implementation; access is mediated by higher elements, notably *attribute*, *method*, or *command*.

A *term* is exactly the same as in raw LCF, but a *thm* holds an additional reference to the enclosing *theory* as an explicit certificate. A *context* is certified against its background *theory* in the same way. An *attribute* covers primitive operations on theorems, as well as declarations to the *theory* or *context*.

An Isar *proof-state* wraps the raw LCF *goal* into a rich block-structured configuration, with an explicit *context* at each position. A *method* supersedes *tactic* as the main goal refinement mechanism, depending on additional structure from the *context* and immediate theorems presented in the proof text.

The Isar *toplevel* integrates the main state components, and encapsulates all operations by a transaction concept that supports recovery from errors and unlimited undo. The Isar toplevel loop replaces the bare-bones ML toplevel of LCF: end-users are no longer exposed to the underlying programming environment.

A *document* consists of a *command* sequence, interleaved with the resulting *toplevel* state at each position. This allows to generate output for LaTeX typesetting, involving both the original sources and pretty printed output of logical entities (using "antiquotations" for *term* and *thm* in the text).

We shall now take a closer look at the key concepts required for building formal reasoning tools within the Isabelle/Isar framework.

Logical Environments. Our central concepts of *theory* and *context* are motivated by certain aspects of the underlying calculus. Derivations in Isabelle/Pure [17] (and the HOL family in general [13]) can be described as a judgment $\Gamma \vdash_\Theta \varphi$, meaning that proposition φ is derivable from assumptions Γ, within theory Θ. Both Θ and Γ act as a *logical environment*, but have different characteristics: Θ holds global declarations of polymorphic type constructors, term constants, and axioms; Γ covers locally fixed type variables, term variables, and hypotheses. The following main principles operate on Γ and Θ wrt. the conclusion φ:

- Transfer: due to monotonicity of derivations, results may be transferred into a *larger* environment, i. e. $\Gamma \vdash_\Theta \varphi$ implies $\Gamma' \vdash_{\Theta'} \varphi$ for $\Theta' \supseteq \Theta$ and $\Gamma' \supseteq \Gamma$.
- Export: by discharging assumptions, results may be exported into a *smaller* environment, i. e. $\Gamma' \vdash_\Theta \varphi$ implies $\Gamma \vdash_\Theta \Delta \Longrightarrow \varphi$ where $\Gamma' \supseteq \Gamma$ and $\Delta = \Gamma' - \Gamma$. Note that Θ remains unchanged here, discharge of theory content is not directly supported.

Isabelle/Isar elevates raw Θ to a *theory* and Γ to a *context*, both supporting arbitrarily typed data, being introduced by the user at compile time.

Theories. A *theory* is a named data container with a unique identifier. Theories are related by a nominal sub-theory relation, which corresponds to the dependency graph of the original construction; each theory is derived from a certain sub-graph of ancestor theories. Locally, a theory is updated in a strictly linear discipline, which reduces the total number of theory identifiers. This organization is able to support large-scale logical environments efficiently.

The operation $\mathit{merge}\colon \mathit{theory} \times \mathit{theory} \to \mathit{theory}$ builds the least upper bound of two theories, which is trivial for nominally related theories. The operation $\mathit{begin}\colon \mathit{name} \to \mathit{theory}^* \to \mathit{theory}$ starts a new theory by importing several parent theories and entering a special *draft* mode, which is sustained until the final operation $\mathit{end}\colon \mathit{theory} \to \mathit{theory}$, where the theory is named and identified for permanent storage. An intermediate draft theory acts like a linear type, where updates invalidate earlier versions. The operation $\mathit{checkpoint}\colon \mathit{theory} \to \mathit{theory}$ produces an intermediate stepping stone that will survive the next update: both the original and the changed theory remain valid and are related by the sub-theory relation. Checkpointing essentially recovers pure theory values, at the expense of extra bookkeeping. The operation $\mathit{copy}\colon \mathit{theory} \to \mathit{theory}$ produces an auxiliary version with the same content, but detached from the original.

Theory data may refer to destructive entities, which are maintained in direct correspondence to the globally graph-structured and locally linear evolution of theory values, including explicit copies of impure data. A theory data declaration needs to implement the following ML specification:

type T	representation
val $empty\colon T$	initial value
val $copy\colon T \to T$	refresh impure data
val $extend\colon T \to T$	re-initialize on import
val $merge\colon T \times T \to T$	join on import

A *theory reference* maintains a live link to an evolving theory: updates on drafts are propagated automatically. Derived entities (notably *thm* or *context*) store a theory reference in order to indicate the enclosing logical environment. Soundness substantially depends on monotonicity of the underlying calculus, as the referenced theory may grow spontaneously.

Contexts. A *context* is a pure data container with a back-reference to the enclosing *theory*. The operation $init\colon theory \to context$ creates an empty context from a given theory. Modifications to draft theories are propagated to the context as explained above. The actual context data does not require any special bookkeeping, thanks to the lack of destructive features at this point.

Although contexts may be manipulated arbitrarily (analogous to *thm* values), the common discipline is to follow block structure: a given context is extended consecutively, and results are exported back into the original context. Note that an Isar *proof-state* models block-structure explicitly (using a stack).

Context data is declared by implementing the following ML specification:

type T	representation
val $init\colon theory \to T$	initial value

Generic *theory* and *context* data is used in Isabelle/Pure from the very beginning of bootstrapping the system. Even the inference kernel itself depends on types, constants, and axioms being maintained as theory data. If we liken the original LCF approach to a “micro-kernel” architecture, we may understand the Isabelle/Isar data management facility as a “nano-kernel”.

After having boot-strapped the basic functionality of Isabelle/Pure, with many layers of theory and context data, the same principle is continued in object-logics like Isabelle/HOL. This includes data for reasoning tools (Simplifier, Classical Reasoner etc.) and derived specification mechanisms (inductive sets, recursive functions etc.). There is no reason to stop here, of course (cf. §4).

Attributes. An *attribute* covers any immediate operation on a theorem or the data within the underlying environment (*theory* or *context*):

$$\begin{array}{rll} global\text{-}attribute = & theory \to thm \to thm & \text{global rule} \\ \mid & thm \to theory \to theory & \text{global declaration} \\ local\text{-}attribute = & context \to thm \to thm & \text{local rule} \\ \mid & thm \to context \to context & \text{local declaration} \end{array}$$

Practically speaking, attributes model various kinds of ad-hoc transformations and marginal declarations. Attributes may be adjoined to theorems wherever these occur in other Isar elements, using postfix notation *thm* [*attribute*]. For example, *thm* [*simp*] is a declaration that corresponds to the primitive *addsimp* (cf. §3.1), but operates directly on *theory* or *context*. Another example is *thm* [*symmetric*] which concludes a symmetric version of the given theorem, using a suitable symmetry rule picked from the environment.

Proof States and Methods. A *proof-state* is a stack over $\mathit{context} \times \mathit{goal}^?$, together with some additional information to model the linguistic structure of an Isar proof text [20]. Unstructured proof scripts are incorporated into this model as a trivial case, by ignoring part of this structure.

A *method* operates on the *goal* field of a *proof-state*, depending on the proof *context* and immediate facts stemming from previous reasoning; the result is an enumeration of possible successive *goal* configurations, with optional *case* declarations to augment the context of the subsequent proof body:

$$\mathit{method} = \mathit{context} \to \mathit{thm}^* \to \mathit{goal} \to (\mathit{case}^* \times \mathit{goal})^{**}$$

Non-empty cases only occur in the most advanced methods, notably *cases* and *induct* [21]. Moreover, many methods merely insert the indicated list of theorems into the goal as local premises, and then continue in the plain-old tactical sense. Thus the following simplified version suffices for most situations (recall that $\mathit{tactic} = \mathit{goal} \to \mathit{goal}^{**}$):

$$\mathit{simple\text{-}method} = \mathit{context} \to \mathit{tactic}$$

In other words, a simple method is a plain tactic that refers to the proof context explicitly, retrieving arbitrary data as required (*simp* rules etc.). In order to combine such methods, each constituent part merely needs to pass-on the *context* it has received, such that every component will be able select its own data.

Toplevel Commands. The Isar toplevel maintains an implicit state that is transformed by a sequence of commands, either interactively or in batch-mode. In interactive mode, toplevel state transitions are encapsulated as safe transactions, such that both failure and undo are handled conveniently, without destroying the underlying draft theory. In batch mode, transitions operate in a strictly linear fashion, so an error will abort the present attempt to process the input.

A *toplevel* state consists of a history over $\mathit{empty} \mid \mathit{theory} \mid \mathit{proof\text{-}state}$. Special control commands may revert to previous states (**undo**), or abort the present theory or proof construction (**kill**). Regular commands operate on the topmost history entry, transforming the current *theory* or *proof-state*. The following typed interfaces allow to compose such toplevel commands:

$$\begin{array}{ll}
\mathit{Toplevel.theory} & : (\mathit{theory} \to \mathit{theory}) \to \mathit{transformer} \\
\mathit{Toplevel.begin\text{-}proof} & : (\mathit{theory} \to \mathit{proof\text{-}state}) \to \mathit{transformer} \\
\mathit{Toplevel.proof} & : (\mathit{proof\text{-}state} \to \mathit{proof\text{-}state}^{**}) \to \mathit{transformer} \\
\mathit{Toplevel.end\text{-}proof} & : (\mathit{proof\text{-}state} \to \mathit{theory}) \to \mathit{transformer}
\end{array}$$

Here are some example Isabelle/Isar commands [21] that are based on these.

Toplevel.theory	: **types**, **consts**, **axioms**, **defs**, ...
Toplevel.begin-proof	: **lemma**, **theorem**, **typedef**, ...
Toplevel.proof	: **apply**, **proof**, **fix**, **assume**, **show**, ...
Toplevel.end-proof	: **done**, **qed**, **sorry**, ...

Both *Toplevel.theory* and *Toplevel.begin-proof* may be understood as theory specification elements, the latter with a separate proof obligation; variations on these are regularly introduced by add-on tools. In contrast, *Toplevel.proof* and *Toplevel.end-proof* refer to the Isar proof language [20], which is not so easily extended beyond the predefined elements: deeper insight into the proof engine is required. In practice, it is usually sufficient to plug additional proof tools into the restricted interface for proof methods.

3.3 Syntax

As an "LCF-style" system, Isabelle/Isar works directly with *semantic* entities, modeled as abstract types in ML. Concrete syntax is adjoined superficially between the core system and the end-user, such that theory sources may be presented as plain text. There are essentially three syntax layers in Isabelle/Isar:

1. outer syntax: toplevel commands,
2. embedded syntax: attributes and methods,
3. inner syntax: terms and types.

In particular, a command definition includes a parser function to turn a certain portion of input source into a semantic toplevel transaction. Similar parser functions may be installed for attributes and methods. Term syntax depends on an unrestricted context-free grammar maintained within the theory; new grammar productions may be added by mixfix annotations for constants.

The vertical relationship between abstract system concepts and superficial user syntax may be represented e. g. for type *term* as the pair of functions $read\colon context \to string \to term$ and $print\colon context \to term \to string$. Internally, all components communicate directly in a horizontal manner, bypassing concrete syntax altogether. This works uniformly for pre-defined primitives and add-on tools alike. Taking the detour via generated sources is not recommended: both efficiency and robustness would suffer due to the full round-trip through various (extensible!) syntax layers. This also means that decent tools are obliged to provide proper internal interfaces by default, not just concrete syntax for end-users. Otherwise, the tool development chain would be broken here.

4 Application: Implementing HOL-Z 3.0

We now present concrete instances for the generic slots outlined above.

4.1 Theory Data: Z Environment

HOL-Z requires its own state capturing specific data for parsing and theorem proving, called the Z-environment (of ML type `Zenv`). For example, schema identifiers must have an internal signature ("schema signature") assigning to local names their type and type constraint. This signature is necessary to construct the binding structure of a schema-logical expression. As an example for proof support, there is a set of type-constraint information ("f is a partial function": $f \in A \nrightarrow B$) that is used to eliminate side-conditions of this form which occur in many situations throughout proofs. The data-base of currently unproven proof-obligations is also part of `Zenv`. Our theory data declaration looks like this:

```
structure ZEnv_Data = TheoryDataFun
 (type T = ZEnv.Zenv
  val empty = ZEnv.mt_zenv
  fun copy T = T
  fun extend T = T
  val merge = ZEnv.merge)
```

The core definitions of the `Zenv` type (not shown here) are passed to the functor `TheoryDataFun`, which extends the theory container by an additional data entry in a type-safe way. This results in access primitives `ZEnv_Data.get: theory -> Zenv` and `ZEnv_Data.map: (Zenv -> Zenv) -> theory -> theory`, which we keep private to our module; only some high-level user operations will be exposed later.

Unlike an immediate implementation of such a state component by a global ML reference variable, official theory data is subject to toplevel **undo** operations performed interactively by the user (as part of the Proof General protocol).

4.2 Toplevel Commands

A common scheme of tool interaction is a facility allowing the import of the results of a predecessor in the tool-chain. Instead of ad-hoc attempts to generate Isabelle theory documents indirectly as a text file, we strongly suggest to base such a facility on the internal Isabelle `term`-structure and the type-checker. This way, problems with ambiguous syntax trees can be avoided. Moreover, the parser for the inner term language is designed for flexibility and power, not for efficiency! Such an import mechanism is `load_holz: string -> theory -> theory` (not further discussed here), which will be bound by the subsequent invocation to a new toplevel command of the same name:

```
OuterSyntax.add_parsers
  [OuterSyntax.command "load_holz"
    OuterKeyword.thy_script
    (OuterParse.name >> (Toplevel.theory o load_holz))]
```

The construction re-uses a number of combinators from various libraries such as the parsing combinator library where *parsers* are functions that map a prefix in an input stream into a function representing the *meaning* of this prefix; here, this meaning is a transition function on the theory. There are elementary parsers like `OuterParse.name`, or infix combinators like `>>`, which pipes the meaning of the previously parsed prefix into another function. Summing up, this gives a parser for `load_holz "<holz-file-name>"` which results in a transition from a theory into another one, where the definitions contained in the output file from ZETA were added. The hint of `OuterKeyword.thy_script` tells Proof General [6] about the type of command defined here.

Similarly, the toplevel command `zlemma` (for the toplevel statement with syntax `zlemma <thm-name>: "<goal-text>"`) is defined; in contrast to the standard command `lemma` for starting a proof, `zlemma` uses the HOL-Z parser for Z formulas that can make the implicit binding structure in a schema expression explicit.

4.3 Methods and Attributes

Syntactically, *methods* are embedded into existing toplevel proof commands, such as `apply` or `by`. We bind the specialized tactic `intro_sch_all_tac` for the introduction of the universal schema quantifier is passed to the method `zintro_sch_all`:[3]

```
Method.add_methods [...
  ("zintro_sch_all",
    nat_arg (tac2meth_unary intro_sch_all_tac),
    "intro␣schema␣all␣quantifier"),
  ...]
```

Analogously, *attributes* can be introduced that provide forward reasoning elements into a domain-specific Isar language extension. By a similar piece of setup-code, the attribute syntax *thm* `[zstrip]` can be defined which applies a rule in the sense of §3.1 of type `context -> thm -> thm` to some theorem. In our example, this transformer applies a number of destructive operations on schema-operators (similar to `intro_sch_all_tac`), which lead to a HOL-ified version of a Z-etish definition. Such adoptions are a prerequisite for Isabelle's automated proof methods (*simp*, *blast*, *auto* etc.), such as `apply (simp add: thm [zstrip])`.

4.4 Operator Syntax for Z

The so-called *inner syntax* of Isabelle/Isar is used for denoting terms (occurring in definitions, lemmas, proofs etc.) Inner syntax is usually wrapped inside quotes `"..."`. The grammar rules are derived from mixfix annotations in constant declarations — probably the syntax mechanism that is best-known to end-users.

Z favors the conciseness of mathematical notation to an usual degree; a lot of effort in the language design has been put in inventing mathematical symbols and shortcuts for many operations. The Z standard offers several representations of the "lexems" of the language. HOL-Z takes this into account and supports several

[3] For various reasons, this rule cannot be represented by one single `thm`. Rather, universal schema introduction in Z is a rule scheme that has to be instantiated on the fly for a given goal; see [9] for details.

of these formats. For example, the operator providing the set of partial functions is declared by an (internal) operator symbol having a standard denotation in the *email-format* (allowing keyboard shortcuts at the GUI level in Proof General). Moreover, there is a mapping of the internal operator symbol to the *xsymbols-format*, which is inspired by the LaTeX format of the Z standard.

```
consts
  partial_func ::"['a␣set,'b␣set]␣=>␣('a␣<=>␣'b)␣set"
        ("_␣-|->␣_"   [54,53] 53)

syntax (xsymbols)
  partial_func ::"['a␣set,'b␣set]␣=>␣('a␣<=>␣'b)␣set"
        ("_␣\<pfun>␣_"  [54,53] 53)
```

It is possible to configure Proof General to associate with the xsymbols-format a specific font that allows this concise notation to be directly used on screen during proof work. Furthermore, it is also possible to configure LaTeX style files in a way that LaTeX macros for these symbols are used during the batch-mode document generation; then `\<pfun>` is really shown as $\pfun$.

5 A Comparison to ProofPower

It is instructive to compare HOL-Z to ProofPower [3]. The latter has been built on top of HOL88 in a collaborative project, which ran from 1990 through to the end of 1992. It is free software with the exception of a component for specifying and verifying Ada programs (excluded from our comparison).

ProofPower comprises a developer kit (even providing its own parser generator), an X11/Motif front-end, a HOL prover kernel as well as the key component, a Z specification and proof development system including libraries and customized proof procedures.

ProofPower and HOL-Z have been both used in substantial case studies and possess a similar architecture; it is therefore possible to compare the systems in order to highlight the potential of the generic technologies of Isabelle. The comparison in Figure 4 deliberately excludes generic code from Isabelle/Isar or Isabelle/HOL. Under "document generation" we summarize only the LaTeX style

	HOL-Z	**ProofPower**
LaTeX-Frontend with Type-Checker	10340 lines (Java)	not available
document generation	1200 lines	2400 lines
user interface	100	14000 lines (C)
emb. + lib. + tactics	1521+7150+3543 lines	329965 lines
build process	165 lines	4700 lines

Figure 4. Statistics on the two implementations

files; for HOL-Z, the generation mechanism itself is generic. Under "user interface", we summarize the code for the Motif frontend, which is roughly equivalent to Proof General. The main difference comes in position "logical embedding + library + tactic support": for ProofPower, we essentially count all document files (.doc) of the system comprising the own HOL-system, HOL-libraries already geared to Z, Z libraries, and the overall proof-technical environment. On the HOL-Z side, we just consider the embedding in HOL (not Isabelle/HOL itself), the Z library, and specialized tactics for proof support and methodological support (Proof-Obligation Management for consistency and refinement, which is lacking on the ProofPower side). In our view, the proof-scripts are notably more concise; this is due to the Isar proof language as well as more automated proof procedures available in Isabelle.

One could object that counting lines of literate specification .doc files punishes ProofPower for its excellent documentation. However, even if taking this into account by a very conservative factor 3, the code size of ProofPower is still an order of magnitude larger than HOL-Z. Another objection is that a monolithic development strategy can build sometimes on better components (e.g. Proof General/Emacs is not really a show-case in GUI design). Further, a generic development strategy has also costs in terms of work not reflected in code sizes: namely, from time to time, new versions of Isabelle are released requiring adoptions to modified interfaces; it takes some effort to share the work of others. Still, we believe that the advantages outweigh these costs by far.

6 Conclusion

Presenting Isabelle/Isar as a framework for building formal method tools might appear as a bit of a surprise at first sight: the Pure logical framework of Isabelle [17] was originally conceived as a playground for experimenting with various versions of constructive type theory; the Isar layer [20] was mainly motivated by human readable proof texts. None of this is directly relevant to HOL-Z 3.0.

On the other hand, Isabelle has been conceived as a platform for a broad range of applications from early on. Strictly speaking, this idea is already present in the original LCF/HOL family, but Isabelle has also managed to reduce the dependency of a specific logic. This "generic-everything" attitude has been emphasized even further when revising the key system concepts for Isar.

Thus the Isabelle/Isar framework (as of Isabelle2005) already comprises a solid basis for building advanced applications such as Isabelle/HOL, and HOL-Z as presented here. Nevertheless, this only marks an intermediate stage in further development, both of the framework and its applications. For example, in post-Isabelle2005 versions the notions of *theory* and *context* are refined further into a *local-theory*, which will support very general notions of derived theory specifications, relative to local parameters and assumptions.

A more practical limit for the systematic re-use of existing system infrastructure is that of documentation and education. There is plenty of folklore wisdom and oral tradition, which is not easily available in a systematized form. The present paper — together with an ongoing effort to clean up the actual implementation and provide up-to-date manuals for implementors — is intended as an initiative to communicate concepts of the framework at a higher level.

Acknowledgment. Achim Brucker made valuable suggestions for the design of the proof-obligation manager and provided many valuable comments.

References

[1] ZeTa system, Aug. 1997. `http://uebb.cs.tu-berlin.de/zeta/`.
[2] HOL-TestGen, Jan. 2005. `http://www.brucker.ch/projects/hol-testgen/`.
[3] ProofPower, Jan. 2005. `http://www.lemma-one.com/ProofPower/index/`.
[4] Isabelle/HOL-OCL, Mar. 2006. `http://www.brucker.ch/projects/hol-ocl/`.
[5] Isabelle/HOL-Z, Jan. 2007. `http://www.brucker.ch/projects/hol-z/`.
[6] D. Aspinall. Proof General: A generic tool for proof development. In *European Joint Conferences on Theory and Practice of Software (ETAPS)*, 2000.
[7] B. Barras et al. *The Coq Proof Assistant Reference Manual, v. 8.* INRIA, 2006.
[8] S. Berghofer and M. Wenzel. Inductive datatypes in HOL — lessons learned in Formal-Logic Engineering. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, LNCS 1690, 1999.
[9] A. D. Brucker, F. Rittinger, and B. Wolff. HOL-Z 2.0: A proof environment for Z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, Feb. 2003.
[10] A. D. Brucker and B. Wolff. The HOL-OCL book. Technical Report 525, ETH Zürich, 2006.
[11] A. D. Brucker and B. Wolff. Using HOL-TestGen for test-sequence generation with an application to firewall testing. In B. Meyer and Y. Gurevich, editors, *TAP 2007: Tests And Proofs*, LNCS. Springer, 2007.
[12] M. J. C. Gordon. From LCF to HOL: a short history. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner.* MIT Press, 2000.
[13] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.
[14] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation.* LNCS 78. Springer, 1979.
[15] J. Harrison. Hol light: A tutorial introduction. In *FMCAD'96*, LNCS 1166, 1996.
[16] Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: TPHOLs'96*, LNCS 1125. Springer, 1996.
[17] L. C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science.* Academic Press, 1990.
[18] L. C. Paulson. *Handbook of Logic in Computer Science*, volume 2, chapter Designing a Theorem Prover, pages 415–475. Clarendon Press, 1992.
[19] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.
[20] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs'99*, LNCS 1690, 1999.
[21] M. Wenzel. *The Isabelle/Isar Reference Manual*, 2005. Part of Isabelle2005.
[22] M. Wenzel and L. C. Paulson. Isabelle/Isar. In F. Wiedijk, editor, *The Seventeen Provers of the World*, LNAI 3600. Springer, 2006.