SML with antiquotations embedded into Isabelle/Isar

Makarius Wenzel* and Amine Chaieb

Technische Universität München
Institut für Informatik, Boltzmannstraße 3, 85748 Garching, Germany

Abstract. We report on some recent experiments with SML embedded into the Isabelle/Isar theory and proof language, such that the program text may again refer to formal logical entities via antiquotations. The meaning of our antiquotations within SML text observes the different logical environments at compile time, link time (of theory interpretations), and runtime (within proof procedures). As a general design principle we neither touch the logical foundations of Isabelle, nor the SML language implementation. Thus we achieve a modular composition of the programming language and the logic within the Isabelle/Isar framework. Our work should be understood as a continuation and elaboration of the original "LCF system approach", which has introduced ML as a programming language for theorem proving in the first place.

1 Introduction

An interesting observation about LCF like theorem provers is the presence of several layers of languages: the implementation language, the meta-language (ML) and the object language (logic). In contrast to the original LCF system [9, 10], the implementation language coincides with ML in most present LCF-style provers. The meta-language of LCF [9] already allowed to "quote" (i.e. refer to) logical entities, which is a very natural requirement on ML. Curiously, all subsequent attempts realized this embedding in the very same manner: they quote the logic inside ML. In this concern, Isabelle made no exception until the advent of Isar [19], which replaced ML as a primary input language and provided a quotation mechanism for SML within this new environment. This reversed embedding of languages turns out to be very important, since it allows to interpret SML code *relative* to a logical context. In general the dependence on a logical environment can be observed at compile time, link time (e.g. of theory interpretations), or runtime (within proof procedures).

This paper presents recent experiments in elaborating SML quotations within Isar, with support for antiquotations that refer to logical entities and system values. So far, antiquotations in Isabelle/Isar have appeared only in preparing LaTeX documents from formal theories, see [14, §4]. An example document is the one you are reading now.

* Supported by BMBF project "Verisoft".

An important design principle we follow is to leave both the logical foundations of Isabelle and the SML language implementation (parser, compiler and runtime system) *intact*. This separation of concerns economizes the implementation of the overall system architecture. It also means that these ideas can in principle be transferred to other programming languages and other logic implementations, by reproducing parts of the integrative Isabelle/Isar infrastructure.

Overview. In §2 we survey some prominent variations on the LCF system architecture; despite its historical flavor, this section is important to highlight some decisive design issues in Isar and the quotation mechanism. In §3 we present the embedding of SML into Isar. In §4 we review some selected antiquotations. §5 shows a further example.

2 Variations on the LCF system architecture

We briefly review the main characteristics of the LCF system family, with special focus on programming support for conducting logical developments.

2.1 Original LCF

The idea of coupling a full functional programming language with a theorem prover was pioneered by the LCF prover [9, 10]. The original system consists of three language layers: LISP, ML (the "meta-language"), and the logic (the "object-language"). LISP serves as implementation platform for basic functionality of symbolic logic (primitive operations on types and terms) and to implement an ML interpreter. When the latter has been bootstrapped, further development of LCF uses the strongly-typed environment of ML, operating on concrete datatypes of types and terms, and an abstract datatype of theorems. By restricting theorem operations to a predefined interface of primitive inferences, further derivations are guaranteed to be "correct-by-construction" thanks to the ML type discipline.

The logic of LCF ("Logic of Computable Functions") describes computational entities, but this does *not* mean that concrete programs may be run within the logical system. There is a difference in modeling computation abstractly and executing concrete programs. This is where ML takes its part as LCF system programming language, providing access to the logic implementation.

In an LCF-style system, a theorem is an abstract value of type *thm*, and a derived rule is a function that transforms proven theorems into proven theorems, such as $thm \to thm$ (unary rules), $thm \to thm \to thm$ (binary rules) or $term \to thm \to thm$ (parameterized rules). For example, the rule *modus-ponens*: $thm \to thm \to thm$ maps $A \longrightarrow B$ and A to B. In LCF, the user is granted full access to the ML programming environment (to implement proof tools, derived specification mechanisms etc.) without affecting soundness. In fact, from the ML level upwards, developers and end-users are treated as equal.

Early on [9], the meta-language of LCF has also supported a *quotation* mechanism for embedding terms and formulas into programs conveniently, by using concrete syntax of the logical environment. For example, the rule for specializing a universal quantifier could be an ML function *forall-spec*: *term* → *thm* → *thm* that maps a term t and a theorem $\forall x.\ P\ x$ to a theorem $P\ t$. Then the particular instance of $a + b$ for x is written in ML with embedded terms as *forall-spec* ⌜$a + b$⌝ *my-thm*. Note that only concrete syntactic entities are quoted directly here, while theorems remain abstract ML entities without external representation.

2.2 Modifications introduced by HOL, Coq, and Isabelle

The LCF approach has spun off a family of successors, with various modifications and further elaboration of the original system architecture. Today, the main representatives of this ongoing process are the HOL familiy [11] (represented by HOL Light, HOL4, and ProofPower), Coq [17, 1], and the Isabelle/Isar framework [21, 20] with Isabelle/HOL [14] as its main application. We shall look into particular aspects of the Isabelle/Isar system architecture in more detail later (§3.1). Here we briefly review some general modifications of the original LCF approach introduced by either of these systems.

ML as implementation language. The original design of the LCF "meta-language" has proven quite successful as general-purpose programming language, thanks to its selection of innovative concepts (algebraic datatypes, higher-order values, let-polymorphism etc.). Independent implementations of ML as a standalone programming language have appeared early on, eventually replacing the original interpreter within LISP. Subsequent generations of LCF-style provers have been implemented directly on top of such ML implementations, notably Coq (OCaml), HOL4 (first SML/NJ then Moscow ML), HOL Light (first Caml-light then OCaml), and Isabelle (mainly Poly/ML and SML/NJ).

Even though the meta-language now coincides with the implementation platform, its role as system extension language is still maintained, although in different degrees. Today, the HOL family and Isabelle still adhere to the strict "correctness-by-construction" paradigm in implementing the logical kernel around an abstract type *thm*, although the version of HOL Light is much smaller than that of Isabelle, for example. In Coq, the kernel is even harder to delimit, and the notion of primitive inferences via an abstract type has been superseded by explicitly stored proof terms that can in principle be checked separately by a small trusted component (cf. the "de Bruijn principle"). Since explicitly stored proof terms can pose resource problems, Coq also provides means to *reduce* proof terms, notably those stemming from internal computation (cf. the "Poincaré principle").

Simplified user command languages. The full programmability of the system by ML has proven both successful and inaccessible to most users. LCF

"system programming" is often perceived as an arcane discipline exercised by a few initiates. Today, regular end-users rarely need to build their own tools before commencing the actual work in producing definitions and proofs. Simplified tactic languages have been implemented on top of the ML kernel of the prover, usually with separate concrete syntax and a separate read-eval-print loop replacing the one of ML.

The systems of the HOL family provide numerous ready-to-use tactics and definitional mechanisms, but only as ML library functions without concrete syntax. In contrast, Coq has provided its own toplevel early on, with specific support for a "mathematical vernacular" (for specifications) and a simple language of tactics (for proof scripts). This restricted language of tactic combinators was later replaced by the more elaborate Ltac [7], which has been carefully designed as an intermediate between full programmability and dumb tactic application. Ltac is especially interesting here, because it has also been embedded into the underlying OCaml implementation language: Ltac expressions may be quoted within OCaml, and OCaml may be quoted within Ltac.

Isabelle has initially followed the same approach as the HOL family, with a collection of library functions for predefined tactics in ML. With the advent of Isar [19], ML was discontinued as the primary input language. Isar provides separate concrete syntax for specifications and proofs (with special focus on human-readable proof texts). The Isar language replaces free programmability by a specific framework with plugins being categorized explicitly as *command*, *method*, *attribute* etc. This arrangement provides some general guidelines, while still leaving sufficient freedom in inventing new concepts within the existing system. In particular, there is no fixed syntax (or datatype representation) for Isar entities — the main concepts are modeled semantically via functions on abstract types.

Internalized natural deduction. This is a specialty of the Isabelle/Pure framework [16]. The idea is to represent (derived) rules not as ML functions, but as first-class theorems within a slightly generalized logical framework. To this end, Isabelle/Pure provides the quantifier "$\bigwedge$" (to express arbitrary, but fixed entities) and the connective "$\Longrightarrow$" (to express entailment from assumptions to conclusions). For example, modus ponens is represented as $\bigwedge A\ B.\ (A \longrightarrow B) \Longrightarrow A \Longrightarrow B$ (it is important to distinguish the framework connective "$\Longrightarrow$" from the one of the object-logic "$\longrightarrow$" here). Isabelle also represents goals as theorems of the framework: $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow C$ means that n pending sub-goals need to be solved in order to establish the main conclusion C; for $n = 0$ the result emerges as a theorem as expected.

The minimal logic of Isabelle/Pure allows to represent rules in the style of Gentzen's natural deduction [8] conveniently, both the traditional ones for the usual logical connectives ($\wedge$, $\vee$, $\forall$, $\exists$ etc.), and any further derived ones emerging in user applications. Instead of a host of primitive ML functions (or tactics), Isabelle provides only a few basic principles to compose rules in a Prolog-like

fashion [15], with *resolution*: $thm \to thm \to thm$ to back-chain a rule from a goal, and *assumption*: $thm \to thm$ to finish a pending branch of a goal by assumption.

Internalized rules impact programmability in the sense that *fewer* operational instructions are required in order to conduct basic inferences. Compared to the HOL family, Isabelle tactic scripts appear more stylized, naming mostly the rules (theorems) to apply. In the Isar proof language [19] this is continued even further towards actual human-readable proof texts. Here programming of proofs has been replaced by textual composition of natural deduction proof schemes. Thus large proofs may be conducted without any programming involved, similar to the Mizar system [18], which lacks programmability altogether. Nevertheless, building add-on tools will require programming again. We shall see later (§3.2), how to re-use some of the infrastructure for Isar proof texts in processing embedded SML sources.

Internalized computation. This works particularly well in the type-theory of Coq. The idea is to use the existing principles of $\delta\iota$-reduction (expansion of primitive and inductive definitions) as a reasonably efficient computational mechanism within the main logical calculus. Thus ML essentially degenerates into a mere implementation platform, and the user works mostly with the logic and its internal programming language.

Sophisticated proof procedures can be modeled, proven correct, and results evaluated — all within the main type-theory of Coq. An important effect of the general correctness proof being provided by the tool implementor is that the proof terms stemming from concrete applications are reduced to equational reflexivity. Since the logic cannot be aware of its own syntactic representation, this approach of internalized computations needs to be combined with an extra-logical mechanism to "reify" formulae as explicit syntactic entities. Note that this idea has been pioneered in [4], where the authors rely on the quoting mechanism of LISP.

There are ongoing efforts to incorporate more and more advanced functional programming technology (taken from OCaml) into the Coq reduction engine. Thus the user is enabled to use internalized proof tools at a realistic scale. On the other hand, this approach demands formal correctness proofs everywhere. Note that the original LCF approach was able to avoid this by checking primitive inferences explicitly at runtime: unproven user tools may fail unexpectedly, but never produce wrong results.

Apart from proof procedures, another important class of system extensions are derived specification mechanisms (called "definitional packages" in Isabelle jargon). Complex packages are presently outside the scope of internalized computations. For example, in Isabelle and the HOL family, higher concepts of inductive sets and types, recursive functions etc. are constructed explicitly from basic logical primitives. This is based on sophisticated ML code outside the main kernel: new definitional principles may be added without affecting the logical foun-

dations. This is in contrast to Coq, which starts with an elaborate type-theory from the very beginning (including inductive constructions as primitives).

3 Isabelle/Isar versus SML

After pointing out the main concepts of Isabelle/Isar, we explain how to embed SML code into the formal text, which in turn may refer to Isabelle entities via antiquotations.

3.1 Isabelle/Isar concepts

Fig. 1 gives an overview of the main concepts of the Isabelle/Isar 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 *method* operates on a *proof-state*).

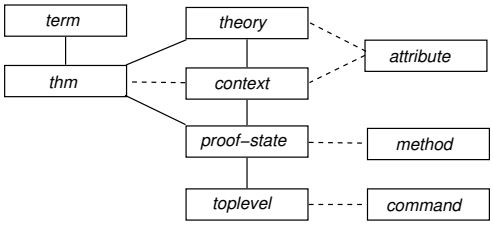

Fig. 1. Isabelle/Isar concepts

These categories are modeled as abstract SML types, except for *term* which provides a concrete syntactic model of λ-terms (with simple types). A *thm* represents a derived proposition as in LCF, but with an explicit *theory* certificate.

A *theory* holds global declarations (e.g. types, constants, axioms), and maintains a unique identification to serve as certificate; there is an efficient sub-theory relation. A *context* models the deductive environment at an arbitrary position in an Isar text (specifications and proofs). The context may hold any kind of local declarations (e.g. type variables, term variables, assumptions). A *proof-state* models an intermediate situation within a block-structured proof; it consists of a stack over $\mathit{context} \times \mathit{goal}^?$, where the optional *goal* is a traditional tactical goal state. The *toplevel* configuration manages a disjoint sum over *theory* or *proof-state*, with additional history information to enable unlimited undo.

The main "active" language elements are *attribute* (subsuming small forward rules and context declarations), *method* (a structured version of traditional tactics, with explicit indication of the proof *context*), and *command* (typed toplevel transactions). There are separate versions for theory specification commands

(*theory* → *theory*), proof commands (*proof-state* → *proof-state*) etc. Users may define their own attributes, methods, commands, while referring to arbitrary user data that is maintained within the *theory* or *context* in a strongly-typed fashion.

3.2 Quoting SML — antiquoting Isabelle

Runtime evaluation. In order to embed SML into Isabelle/Isar, we require some minimal support for "multi-stage" programming from the implementation platform. We shall employ a version of the old-fashioned `EVAL` of LISP, which takes some source code at run-time, and compiles-evaluates it in the toplevel environment, and continues execution of the caller. Thus `EVAL` acts pretty much like a regular function, but may affect the global compiler environment and mutable variables of the runtime environment. In SML this facility may be provided as *use-string*: *string* → *unit*, similar to the better-known *use* function that loads source files. Despite being outside the scope of the official SML standard, this facility is available in all SML implementations (Poly/ML, SML/NJ, Moscow ML etc.) that are in the tradition of *incremental compilation*, as opposed to the *separate compilation* of most other programming languages available these days.

This enables the Isar toplevel loop to invoke SML at runtime, although restricted to global side effects so far. For example, the Isar command **ML** evaluates any SML toplevel declarations, without affecting the Isar toplevel configuration yet. To pass proper values in and out of evaluated code, the usual trick is to manipulate a global reference hidden somewhere in an appropriate module. For example, operating on a *context* works via *Context.poke*: *context* → *unit* and *Context.peek*: *unit* → *context*. Then some source that compiles to a function of type *context* → *context* may be presented as a semantic function as follows:

```
fun eval_context src =
  fn ctxt =>
    (Context.poke ctxt;
      use_string
        ("let val f = " ^ src ^
             "val ctxt = Context.peek () in Context.poke (f ctxt) end");
     Context.peek ());
```

This pasting of source strings is horrible typeless programming! There are interesting research prototypes for strongly typed multi-stage programming in ML (such as MetaOCaml [12]), but this technology is unavailable in mainstream implementations. Luckily the above glue code is only required for setting up the general infrastructure. Users will later encounter properly typed SML elements within Isabelle/Isar, e.g. the method *tactic* that expects source of type *tactic*, or the command **simproc-setup** [6, §2.2] that expects *morphism* → *term* → *thm*.

Expanding antiquotations. Embedded source consists of a list of chunks that alternate between literal text and antiquoted material. Our concrete syntax in

Isabelle/Isar uses $\langle\!\langle \ldots \rangle\!\rangle$ for outermost quoting of source, and @{*name args*} for antiquoted material inside. We maintain a mapping from *name* to parser functions that turn the given *args* into SML text, depending on the Isar *context* available at expansion time (even before the SML compiler is invoked on the generated code).

The most basic antiquotation facility would merely produce literal replacement text for the original source position. The LaTeX antiquotations of the Isabelle document preparation work like that: e.g. @{*term t*} reads term t in the current context and pretty-prints the result into the LaTeX source. For SML antiquotations there are further demands, though, to achieve tight coupling with logical concepts (for proof producing functions etc.). We identify the following main categories tailored to the particular task of *LCF system programming*:

Inline antiquotations represent the most basic concept of replacing text directly in-place. For example, @{*term t*} produces a literal SML source representation using the underlying datatype constructors.

Value antiquotations refer to abstract values taken from the Isabelle/Isar context available at compile time. For example, @{*thm a*} produces code to refer to a theorem from the context, which is bound temporarily within the SML environment; the body code merely refers to a named value, without executing anything at runtime.

Environment markup delimits ranges in the source text that correspond to an implicit λ/*let* binding; the expansion time *context* also follows the indicated block structure. For example, @{*begin* φ} ... @{*end* φ} produces an SML function depending on a *morphism* [6].

Dependent value antiquotations refer to the specified abstractum of a corresponding environment markup. The generated code inserts an additional *projection* to apply values of a particular SML type to the given environment. For example, we use @{*thm* (φ) *a*} to refer to the theorem a abstractly as before, but also to apply the morphism φ provided by the current environment abstraction.

The "compiler technology" required for this model of expanding antiquotations is still fairly simple: we maintain a stack of open environment markups, together with the expansion-time *context* at each stage; we collect named abstract values to be added either to a global compile-time closure, or the local versions relative the functional abstractions stemming from environment markups. User-defined antiquotations are maintained globally as a table of functions that produce source code depending on the expansion-time *context* (which may also be augmented locally during expansion). Note that unlike Camlp4 [13], our expansion mechanism is ignorant of the structure of any literal SML text surrounding the antiquotations.

We illustrate the resulting SML code layout by a schematic example. Consider the following source code within Isabelle/Isar, which indicates antiquotations of all categories introduced above, while any surrounding literal SML code is omitted:

⟪ ...
@{*inline a*}
...
@{*value b*}
...
@{*begin env*}
...
@{*value* (*env*) *c*}
...
@{*end env*}
... ⟫

In the expanded code given below we consider the name prefix `Isabelle` as reserved, it should never occur in user code! The initial structure `Isabelle` serves as global compile time closure; it is discarded later. In contrast, the local environment closures use plain `let` expressions.

```
structure Isabelle =
struct
  val ctxt = Context.peek ()
  val valB = B ctxt
  val valC = C ctxt
end

...
A
...
Isabelle.valB
...
(fn Isabelle_env =>
  let
    val Isabelle_valC' = ApplyEnv.value Isabelle_env Isabelle.valC
  in
    ... Isabelle_valC' ...
  end)
...

structure Isabelle = struct end
```

Here the individual results produced by the parser functions associated with *inline* and *value* have produced expressions `A`, `B` and `C`, respectively. `A` is already a constant SML expression, there are no further dependencies. `B` and `C` are retrieved from the *context* available at compile time (via lookup of *thm* values etc.). `C` is a dependent value, i.e. it is modified further by a specific environment projection inserted into the code. (The collection of projection functions for each kind of environment is maintained together with the antiquotations known to the system.)

3.3 Examples

To illustrate both some basic concepts of Isabelle/Isar (§3.1) and embedded SML with antiquotations (§3.2), we consider proofs of $A \wedge B \longrightarrow B \wedge A$ given as structured text and unstructured script side-by-side:

lemma $A \wedge B \longrightarrow B \wedge A$
proof
  assume $A \wedge B$
  then obtain B **and** A **..**
  then show $B \wedge A$ **..**
qed

lemma $A \wedge B \longrightarrow B \wedge A$
  apply (*rule impI*)
  apply (*erule conjE*)
  apply (*rule conjI*)
  apply *assumption*+
  done

The second version is already close to internal machine operations. To get even further down to traditional ML tactics, we use the method *tactic* that takes SML code representing an expression of type *tactic*. Observe also the references to facts from the current Isar context appearing within the SML source below.

lemma $A \wedge B \longrightarrow B \wedge A$
  apply (*tactic* ⟪ `resolve_tac [@{thm impI}] 1` ⟫)
  apply (*tactic* ⟪ `eresolve_tac [@{thm conjE}] 1` ⟫)
  apply (*tactic* ⟪ `resolve_tac [@{thm conjI}] 1` ⟫)
  apply (*tactic* ⟪ `REPEAT (assume_tac 1)` ⟫)
  done

Next we show how to inspect internal structures of the above Isar text using the elementary **ML** command. Recall that **ML** does not affect the Isar toplevel state, only the one of SML.

lemma $A \wedge B \longrightarrow B \wedge A$
  ML ⟪ `val goal_ctxt = @{context}` ⟫
proof
  assume $A \wedge B$
  then obtain B **and** A **..**
  then show *res*: $B \wedge A$ **..**
  ML ⟪ `val res_ctxt = @{context} and res = @{thm res}` ⟫
qed

Here we have picked the Isar proof context of a goal statement and the extended body context of its proof (which contains additional assumptions to be discharged eventually). The local result retrieved from the body may be exported back into the enclosing context as follows, yielding the rule $A \wedge B \Longrightarrow B \wedge A$:

  ML ⟪ `ProofContext.export res_ctxt goal_ctxt [res]` ⟫

The Isar machinery has used this very rule after finishing the **show** statement, in order to solve the pending subgoal. This example also illustrates how tool developers may learn about the workings of higher-level Isar concepts.

4 An overview of selected SML antiquotations

We now review various concrete antiquotations that are implemented within the framework presented in §3.2. Most of these have already been tried in practical examples, excluding the ideas sketched in §4.6.

4.1 Abstract logical entities

The main logical concepts of Isabelle/Isar are abstract values, which are produced from the static compile-time *context* of the present position in the formal text. The most commonly used value antiquotations are listed in Fig. 2.

Antiquotation	Result
@{*context*}	the compile-time context
@{*theory*}	... its background theory
@{*simpset*}	... its default simplification set
@{*simproc a*}	the simplification procedure named *a*
@{*thm a*}	the theorem named *a*
@{*thms a*}	the theorem list named *a*
@{*cterm t*}	term *t* certified against the theory
@{*ctyp T*}	type *T* certified against the theory

Fig. 2. Abstract logical entities

All these antiquotations refer to abstract values produced in the static compile-time context. For theorems, this means a name lookup. For certified terms and types, this means that a concrete datatype representation is checked against the declarations in the present background theory. This slightly inefficient kernel operation is also performed at compile-time, so there is no runtime penalty for code using @{*cterm*} or @{*ctyp*}.

4.2 Compile-time expressions and patterns

Concrete terms and types may be inlined into the code using @{*term t*} and @{*typ T*}, respectively. Here the given objects are parsed and checked in the compile-time context. In this basic form, the result is a concrete constructor expression in SML.

Being a little more ambitious, we can also produce SML expressions depending on SML values, or even SML `fun/case` patterns that bind variables. We merely need some notational device to indicate term variables as representing SML variables. This works similarly for expressions and patterns alike, although there are some notable differences: patterns are restricted to linear occurrences of variables, but may mention dummies (written with underscore). Since our

Antiquotation	Result
@{*typ* T}	literal type T
@{*term* t}	literal term t
@{*term* t *for* x y z}	term expression over SML variables `x`, `y`, `z`
@{*bterm* t *for* x y z}	term pattern over SML variables `x`, `y`, `z`

Fig. 3. Expressions and patterns

expansion mechanism is ignorant of the surrounding SML syntax, we need to provide separate antiquotations for either situation.

For instance @{*bterm* $x + y$ *for* x y} expands to a pattern that binds `x` and `y` in SML; the remaining structure (an application of constant $+$) is matched literally. By tweaking the Isabelle/Isar term syntax we can easily support dummy-patterns and as-patterns, too, e.g. @{*bterm* $x + \_ + (y$ *as* $(a + b)$) *for* x}. While x is specified as SML variable as before, y is implicitly declared due to its unique role within the as-pattern.

An interesting extension of this scheme would handle naive-polymorphism of the logical framework conveniently. The examples above only work properly for monomorphic $+$, say on natural numbers. With $+ :: \forall\alpha.\ \alpha \Rightarrow \alpha \Rightarrow \alpha$ we need to take care of the actual type instance as well. This works as follows: @{*bterm* $x + y$ *for* $x :: T$ *and* y} matches against the operator's term arguments `x` and `y` as before. The type information is stored within the operator, i.e. a particular instance of the polymorphic $+$. The specification of $x :: T$ indicates that we wish to bind the type of the first argument position of $+$ to an SML variable `T`. This information about typing of constant operators is available from the compile-time context — there is no need to insert code to recompute the type of the actual term `x`.

This idea works similarly for an expression @{*term* $x + y$ *for* $x :: T$ *and* y}. We insert T as pro-forma type variable into the untyped term and invoke the logic's type-inference to propagate this information throughout the whole term skeleton. The result is turned into an SML expression with occurrences of variable `T` in all these inferred positions. If type inference invents additional type variables this indicates an expansion error, and the user needs to specify further type information in the "*for*" part.

4.3 Augmenting the expansion context

The expander state includes a full Isabelle/Isar *context*, which is also subject to the block structure indicated by environment markups. Individual antiquotations may augment that context to affect the processing of subsequent antiquotations (which might involve operations like parsing and type-checking of terms). We provide antiquotations (see Fig. 4) for very basic Isar context extensions of introducing locally fixed variables (with optional type constraints), and binding

term abbreviations via pattern matching. Note that there is no result code being produced here, only the expansion context is affected.

Antiquotation	Effect
@{*vars x y z* :: *T*}	fixing variables x, y, z with type constraint T
@{*let p* = *t*}	binding schematic variables in p by matching with t

Fig. 4. Context elements

How does this impact generated SML code? The general convention is that any value being inlined into the resulting code is understood with respect to the original compilation context. By augmenting the context of the expander we essentially build up a difference between these two contexts, which is discharged on any resulting syntactic entities (terms and types). For example, discharging @{*vars x*} means to turn any occurrence in some term t into a schematic variable *?x* of the Isabelle framework. This impacts runtime operations based on matching or unification.

Discharging abbreviations merely means to replace abbreviated terms by their definitions. This provides a useful macro facility that is guaranteed to be well-typed with respect to the logical context.

4.4 Runtime environments and projections

Many logical operations depend on certain runtime environments in a uniform manner. Instead of writing abstractions and applications explicitly in SML, we may use antiquotations for environment markup and dependent values as described in §3.2. The expander maintains different kinds of runtime environments that are identified by name, like φ for morphism, σ for substitution etc.[1] Defining a new environment requires to specify projection functions for common Isabelle/Isar types (*term*, *thm* etc.). This enables other antiquotations to adapt their result accordingly, achieving strongly-typed code. For a morphism, these projections are *Morphism.term*: *morphism* → *term* → *term*, *Morphism.thm*: *morphism* → *thm* → *thm* etc.

With this setup for morphism environments φ, we may rewrite the generic simproc definition of [6, §2.2] more concisely like this:

```
⟪ @{begin φ}
    … @{cterm (φ) op ⊕} …
    … @{thms (φ) diff-rules} …
  @{end φ} ⟫
```

[1] We may freely use Greek letters φ, ψ, π etc. here, since our SML code is embedded into Isabelle/Isar, which handles a large collection of mathematical symbols.

without the auxiliary `let` expression shown in the published version.

Multiple dependencies can be easily composed, using list notation. For example, @{*term* (φ, σ) $x + y$} refers to the term $x + y$ under morphism φ and substitution σ.

4.5 Runtime matching

Compiled patterns of SML (§4.2) are often insufficient when writing proof-dedicated tools. Typical applications need to take $\beta\eta$-equivalence into account or enforce additional constraints on types (via type-classes).

We can easily support runtime pattern matching by means of a few SML combinators and simple antiquotations to produce match functions from given terms. Technically this yields an environment transformer $env \to env$ to be composed with the right hand side of type $env \to \alpha$, if the match succeeds. Note that this is exactly reverse function composition `#>` that is a prominent combinator in the Isabelle sources. Composing all these several match cases together, with proper handling of failures, merely requires another SML combinator. In any case, the whole expression yields a function of type $env \to \alpha$ which can be applied to an initial environment to form the result.

Matching works uniformly for literal terms, certified terms, and the proposition of theorems, see fig. 5 and the example in §5.

Antiquotation	Result
@{*match* t}	match function for term t
@{*cmatch* t}	match function for certified term t
@{*thmatch* a}	match function for proposition of theorem a

Fig. 5. Runtime match functions

4.6 Further possibilities

Here are some further ideas for potentially useful antiquotations.

- Environment markup for tactical goals. A version of @{*begin goal*} ... @{*end goal*} could support writing tactics conveniently in SML, with separate dependent value antiquotations to match against subgoals etc.
  This would approximate facilities of Ltac [7], but use SML again as programming language, instead of a specialized scripting language.
- Internalized Isabelle/Pure rules as SML functions. Antiquotation @{*rule* a} could turn a named theorem into a rule in the sense of traditional LCF/HOL systems. For example, @{*rule modus-ponens*}: $thm \to thm \to thm$ where *modus-ponens*: $\bigwedge A\ B.\ (A \longrightarrow B) \Longrightarrow A \Longrightarrow B$ within the logical context.

A very simple implementation would merely insert code to invoke the resolution rule of Isabelle/Pure on the specified theorem. A more ambitious solution would attempt to exploit the known theorem structure at compile-time and produce specific code to decompose the run-time arguments, avoiding fully general matching / unification.

– Incorporating SML code generated from Isabelle specifications or proofs. Despite being classical by nature, Isabelle/HOL provides several SML code generation facilities for either propositions or proof terms [3, 2].
We imagine antiquotations @{*code t*} and @{*proof-code p*} to embed the results of code generation / program extraction into SML code that the user writes elsewhere.

These proposed antiquotations are not as easily implemented as the ones considered before. This probably also means that more serious facilities for writing SML generating SML programs will be required, beyond the source string evaluation used so far.

5 Example

The following is the actual source code for the schematic implementation of the generic quantifier elimination given in [5, §3.2], where, for the sake of presentation, the authors *assume* some of the facilities which we have available here in actuality.

```
ML ⟪
  @{vars F G P Q :: bool
    and R :: int ⇒ bool
    and t}

  fun qelim thy qe = divide_and_conquer (fn p =>
    (empty_env, thy) |>
      @{match ¬ P} p #> @{begin σ}
       ([@{term (σ) P}], fwd @{thm cong¬}) @{end σ}
  ||| @{match P ∧ Q} p #> @{begin σ}
      (@{terms (σ) P and Q}, fwd @{thm cong∧}) @{end σ}
  ||| @{match P ∨ Q} p #> @{begin σ}
      (@{terms (σ) P and Q}, fwd @{thm cong∨}) @{end σ}
  ||| @{match P ⟶ Q} p #> @{begin σ}
      (@{terms (σ) P and Q}, fwd @{thm cong⟶}) @{end σ}
  ||| @{match P ⟷ Q} p #> @{begin σ}
      (@{terms (σ) P and Q}, fwd @{thm cong⟷}) @{end σ}
  ||| @{match ∃ x. R x} p #>
     @{begin σ}
      let val (x, px) = dest_abs @{cterm (σ) R}
      in ([px], fn [th] => (empty_env, thy) |>
              @{thmatch F ⟷ G} th #>
                @{begin σ}
```

```
                  let val lift = fwd @{thm cong∃} [gen x th]
                  in fwd @{thm trans} [lift, qe x @{term (σ) G}] end
                @{end σ})
      end
    @{end σ}
| || @{match ∀x. R x} p #>
    @{begin σ} ([@{term (σ) ∃x. ¬R x}], fwd @{thm cong∀})
    @{end σ}
| || @{match t} p #> @{begin σ} ([], fn [] => @{thm (σ) refl})
    @{end σ})
⟫
```

This code typechecks properly in Isabelle/Isar + SML written exactly as above.

6 Conclusion

The idea of alternating quotations / antiquotations (with slight variations of terminology) is fairly old — the LISP community has accumulated such techniques over several decades. So even in the 1978 version of LCF [9], with ML as meta-language and the logic as object-language, quote / unquote mechanisms for the different layers came with little surprise. Later, these preprocessing mechanism have been heavily refined in Camlp4 [13] for Caml-Light and OCaml in particular. The SML community has dropped quotations from the official language definition, although some implementations provide their own version (notably SML/NJ).

Our approach is different in taking the language of Isabelle/Isar as the primary one, and quoting SML code within that. Strictly speaking, this would make SML an *object language* within Isabelle/Isar, but we refrain from this terminology to avoid confusion. Within these quoted pieces of SML, we then allow antiquotations to refer to Isabelle entities from the logical context. We also carefully separate the different environments at compile-time and run-time, where link-time of abstract theory interpretations may count as an additional variant.

In devising concrete SML antiquotations, we have only just started to go beyond the most obvious elements. Further advanced ideas need to be explored and tested with concrete applications, such as the existing code base of tools written for Isabelle/HOL.

References

[1] B. Barras et al. *The Coq Proof Assistant Reference Manual, v. 8.* INRIA, 2006.

[2] S. Berghofer. Program extraction in simply-typed Higher Order Logic. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs (TYPES 2002)*, LNCS 2646. Springer, 2003.

[3] S. Berghofer and T. Nipkow. Executing Higher Order Logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Types for Proofs and Programs (TYPES 2000)*, LNCS 2277. Springer, 2002.
[4] R. S. Boyer and J. S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*. Academic Press, 1981.
[5] A. Chaieb and T. Nipkow. Proof synthesis and reflection for linear arithmetic. Submitted, Nov. 2006.
[6] A. Chaieb and M. Wenzel. Context aware calculation and deduction — ring equalities via Gröbner Bases in Isabelle. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *MKM/Calculemus 2007*, LNAI 4573. Springer, 2007.
[7] D. Delahaye. A proof dedicated meta-language. In *Logical Frameworks and Meta-Languages (LFM 2002)*, ENTCS 70(2), 2002.
[8] G. Gentzen. Untersuchungen über das logische Schließen. *Math. Zeitschrift*, 1935.
[9] M. Gordon, R. Milner, et al. A metalanguage for interactive proof in LCF. In *Principles of programming languages (POPL'78)*, 1978.
[10] M. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. LNCS 78. Springer, 1979.
[11] J. Harrison, K. Slind, and R. Artan. HOL. In Wiedijk [22].
[12] C. Lengauer and W. Taha, editors. *The First MetaOCaml Workshop 2004*, volume 62 of *Science of Computer Programming*. Elsevier, 2006.
[13] M. Mauny and D. de Rauglaudre. A complete and realistic implementation of quotations for ML. In *ACM SIGPLAN Workshop on Standard ML and its Applications*, 1994.
[14] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. 2002.
[15] L. C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3, 1986.
[16] L. C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*. Academic Press, 1990.
[17] L. Théry, P. Letouzey, and G. Gonthier. Coq. In Wiedijk [22].
[18] A. Trybulec. Some features of the Mizar language. Presented at a workshop in Turin, 1993.
[19] M. Wenzel. Isar — a generic interpretative approach to readable formal proof documents. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics (TPHOLs'99)*, LNCS 1690, 1999.
[20] M. Wenzel. *The Isabelle/Isar Reference Manual (for Isabelle2005)*, 2005.
[21] M. Wenzel and L. C. Paulson. Isabelle/Isar. In Wiedijk [22].
[22] F. Wiedijk, editor. *The Seventeen Provers of the World*, LNAI 3600. Springer, 2006.