

Context aware Calculation and Deduction

Ring Equalities via Gröbner Bases in Isabelle

Amine Chaieb and Makarius Wenzel

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

Abstract. We address some aspects of a proposed system architecture for mathematical assistants, integrating calculations and deductions by common infrastructure within the Isabelle theorem proving environment. Here calculations may refer to arbitrary extra-logical mechanisms, operating on the syntactic structure of logical statements. Deductions are devoid of any computational content, but driven by procedures external to the logic, following to the traditional “LCF system approach”. The latter is extended towards explicit dependency on abstract theory contexts, with separate mechanisms to interpret both logical and extra-logical content uniformly. Thus we are able to implement proof methods that operate on abstract theories and a range of particular theory interpretations. Our approach is demonstrated in Isabelle/HOL by a proof-procedure for generic ring equalities via Gröbner Bases.

1 Introduction

The requirements for mathematical assistants come in various and sometimes conflicting flavors: efficient calculations (computer algebra systems have excelled here), deduction with a precise notion of logical correctness (the theorem proving approach), and ability to work relative to abstract theory contexts as in traditional mathematics. In the present paper we address the issue of combining calculations and deductions with theory abstraction and interpretation in the Isabelle theorem proving environment [14]. Since calculations are external to the logic, we may employ arbitrary programming techniques, with full access to the syntactic structure of statements. Deductions are driven by such external procedures, to replay proofs via primitive inferences, following the traditional “LCF system approach” of correctness-by-construction introduced by Milner [7].

Proof methods are typically inhomogeneous because of a natural division of *computing* results vs. *checking* witnesses within the logic. From the algorithmic perspective it is usually harder to produce a result than to check it, e.g. consider long division of polynomials. From the logical point of view, results may emerge spontaneously by an external “oracle”, but checking requires tedious inferences.

Implementing proof tools by orchestrating non-trivial calculations and deductions is a difficult task. In providing a link to abstract theory mechanisms we contribute to systematic development of advanced methods, being organized according to the logical context required at each stage. Thus we transfer principles of modular program and theory development towards proof procedures.

Related work. There are mainly two other approaches in implementing proof procedures in theorem proving environments: *interpretation* and *assimilation*.

Interpretation means that the proof system provides a specific language for describing procedures, being interpreted at run-time and reduced to basic inferences eventually. This language is usually limited to common expressions for combining terms, theorems, tactics etc. Even if it is made computationally complete, there is normally no access to arbitrary libraries and components of the underlying implementation platform. For example, see *Ltac* in Coq [3], or the language of *macetes* in IMPS [6]. The latter is also notable due to its integration with the concept of “little theories”, by means of *transportable macetes*.

Assimilation means that powerful computational concepts (such as algebraic datatypes, recursive functions) are integrated into the very logical environment itself. Calculations are then native reductions of the underlying calculus, as in the type-theory of Coq [3], which augments plain syntactic $\alpha\beta$ -reduction by $\delta\iota$ -reduction (expansion of simple and inductive definitions). Advanced implementation techniques for functional programming language are assimilated into the Coq inference kernel, in order to achieve reasonable performance.

This unified approach of calculations and deductions (also called “computational reflection”) looks very attractive at first sight: various proof tools have been implemented like this [8, 5]. On the other hand, there is some extra tedium involved, because all manipulations need to be formalized within the logic. First this requires facilities to *quote* (or “reify”) logical statements in order to access their syntactic structure. Then fully formal correctness proofs need to be provided, to get manipulated statements interpreted back into the logic.

An important point of the “LCF approach” is to allow arbitrary manipulations without requiring quoting or formally proven programs — correctness is achieved by checking the logically relevant bits at run-time. There are numerous procedures implemented in HOL Light [10]; see also [11] for particular applications involving Computer Algebra Systems as search oracles for HOL proofs.

Basic notation. Keywords like **theorem**, **locale**, **interpretation** etc. refer to Isabelle theory elements. Types and terms are embedded here as smaller units, and notation approximates mathematical conventions despite a bias towards λ -calculus. Types τ consist of type variables α , type constants like *nat* or *int*, or function spaces $\tau_1 \Rightarrow \tau_2$. Terms t consist of variables x , constants c , abstractions $\lambda x. t$, or applications $t_1 t_2$. Propositions are terms of type *prop*, with the outermost logical rule structure represented via implication $A \Longrightarrow B$, or quantification $\bigwedge x. B$, where outermost quantifiers are implicit.

Mathematical functions $A \rightarrow B$ are approximated by computable functions implemented in SML. We use “curried” notation uniformly for iterated applications: $f x y$ for $f : A \rightarrow B \rightarrow C$ and $x : A$ and $y : B$. This is particularly important for *partial application*, such as $f x : B \rightarrow C$, which requires another argument $y : B$ to continue evaluation. Despite some similarities in notation, a plain mathematical function $(x \mapsto f x) : A \rightarrow B$ should not be confused with a symbolic term $(\lambda x. t x) : \alpha \Rightarrow \beta$ within the logical framework!

Overview. In §2 we present a simple example of calculating with (semi)ring numerals. In §3 we introduce the general principles of contexts and context declarations in Isabelle. In §4 we review some aspects of Isabelle’s Simplifier that use the context infrastructure to implement facilities shown in the example. In §5 we report on a larger application, a proof method for ring equalities via Gröbner Bases.

2 Example: calculations on generic numerals

Our main application (cf. §5) operates on general semiring and ring structures in non-trivial ways. In order to illustrate the integration of proof methods with abstract theory mechanisms of Isabelle, we merely consider the trivial sub-problem of representing constants (abstract numerals) adequately, such that basic calculations (addition, subtraction etc.) can be performed efficiently.

We shall be using Isabelle locales [12, 1, 2] which provide a general concept for abstracting over operations with certain properties. Locales are similar to the “little theories” of IMPS [6], but based on pure predicate definitions together with some infrastructure to manage Isabelle/Isar proof contexts [17].

Existing calculational tools in Isabelle [15] use the more restricted mechanism of axiomatic type classes [16], where the only parameter is the underlying type, but the signature is *fixed*. According to [9], type-classes may be understood as particular locale interpretations, where the variable parameters are replaced by polymorphic constants — fixing everything except the underlying type. Thus our general locale-based proof methods will be able to cover type-classes as well.

2.1 Semirings

A *semiring* provides operations \oplus , \odot , 0 , 1 over type α , with the usual laws for associativity, commutativity, and distributivity etc.

```

locale semiring =
  fixes add ::  $\alpha \Rightarrow \alpha \Rightarrow \alpha$  (infixl  $\oplus$  65)
    and mul ::  $\alpha \Rightarrow \alpha \Rightarrow \alpha$  (infixl  $\odot$  70)
    and zero ::  $\alpha$  (0) and one ::  $\alpha$  (1)
  assumes add-a:  $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ 
    and add-c:  $x \oplus y = y \oplus x$ 
    and add-0:  $0 \oplus x = x$  and ...

```

We shall single out canonical *semiring* expressions as “constants”, corresponding to non-negative natural numbers. To achieve an efficient binary representation, we postulate an operation *bit* for shifting a least-significant 0 or 1:

```

locale semiring-bin = semiring +
  fixes bit ::  $\alpha \Rightarrow \text{bool} \Rightarrow \alpha$ 
  assumes bit-def:  $\text{bit } x \ b = x \oplus x \oplus (\text{if } b \text{ then } 1 \text{ else } 0)$ 

```

We can now represent numerals compactly using binary representation according to the recursive definition $\text{numeral} = 0 \mid 1 \mid \text{bit numeral False} \mid \text{bit numeral True}$

(excluding leading zeros). E.g. 6 is represented as binary *bit* (*bit 1 True False* instead of unary $1 \oplus 1 \oplus 1 \oplus 1 \oplus 1 \oplus 1$). The syntax machinery of Isabelle can be modified to support human-readable (decimal) notation; we shall use *#digits* for this purpose. Then the term *#6* will stand for the previous binary expression.

Symbolic calculation on numerals means to manipulate canonical expressions such that proven equalities emerge in the end. For example, addition is easily implemented as a strongly-normalizing term rewrite system:

theorem (in *semiring-bin*) *binary-add*:

$$\begin{aligned}
x \oplus 0 &= x \\
0 \oplus x &= x \\
1 \oplus 1 &= \text{bit } 1 \text{ False} \\
\text{bit } x \text{ False} \oplus 1 &= \text{bit } x \text{ True} \\
\text{bit } x \text{ True} \oplus 1 &= \text{bit } (\text{add } x \ 1) \text{ False} \\
1 \oplus \text{bit } x \text{ False} &= \text{bit } x \text{ True} \\
1 \oplus \text{bit } x \text{ True} &= \text{bit } (\text{add } x \ 1) \text{ False} \\
\text{bit } x \text{ False} \oplus \text{bit } y \text{ False} &= \text{bit } (x \oplus y) \text{ False} \\
\text{bit } x \text{ False} \oplus \text{bit } y \text{ True} &= \text{bit } (x \oplus y) \text{ True} \\
\text{bit } x \text{ True} \oplus \text{bit } y \text{ False} &= \text{bit } (x \oplus y) \text{ True} \\
\text{bit } x \text{ True} \oplus \text{bit } y \text{ True} &= \text{bit } (x \oplus y \oplus 1) \text{ False}
\end{aligned}$$

Isabelle's Simplifier depends on rewrite rules declared in the context as *[simp]*. The proof method *simp* derives equalities according to a bottom-up strategy, effectively simulating strict functional evaluation with proven results $t = t'$.

declare (in *semiring-bin*) *binary-add* *[simp]*

We can now prove equations involving addition of abstract numerals, e.g.:

theorem (in *semiring-bin*) $\#48734 \oplus \#3762039758274 = \#3762039807008$
by *simp*

The same works for various interpretations of the abstract context, e.g. for type *nat* with the usual operations $+$, $*$, 0 , 1 , and *bit-nat* being defined separately:

definition *bit-nat* $n \ b = n + n + (\text{if } b \text{ then } (1::\text{nat}) \text{ else } (0::\text{nat}))$
interpretation *semiring-bin* [(*op* $+$) (*op* $*$) $0 \ 1$ *bit-nat*] *<proof>*

Now the background context contains corresponding instances of the *semi-ring* locale content (e.g. for type *nat*), including *binary-add* with the *[simp]* declaration. This allows to calculate on interpreted *nat* numerals as follows:

theorem $\#48734 + \#3762039758274 = \#3762039807008$
by *simp*

Multiplication on numerals is handled by the very same technique.

2.2 Ring numerals

The subsequent locale *ring-bin* extends semirings with subtraction and negation. Our particular axioms are not an abstract algebraic characterization, but express the requirements of the proof method introduced later.

```

locale ring-bin = semiring-bin +
  fixes sub ::  $\alpha \Rightarrow \alpha \Rightarrow \alpha$  (infixl  $\ominus$  65)
  and neg ::  $\alpha \Rightarrow \alpha$  ( $\ominus$  - [81] 80)
  assumes neg-mul:  $(\ominus 1) \odot x = \ominus x$ 
  and sub-add [simp]:  $x \ominus y = x \oplus (\ominus y)$ 
  and add-neg:  $z \oplus y = x \Longrightarrow x \oplus (\ominus y) = z$ 

```

Canonical expressions for signed numerals are defined as *numeral* | \ominus *numeral* (excluding negated zeros). For symbolic computation, subtraction is reduced to negation and addition immediately, and negations are normalized as follows:

theorem (in ring-bin) neg-norm [simp]:

```

 $\ominus 0 = 0$ 
 $\ominus (\ominus x) = x$ 
 $(\ominus x) \oplus (\ominus y) = \ominus (x \oplus y)$ 

```

This reduces signed addition to $x \oplus (\ominus y)$ or $(\ominus y) \oplus x$, where x and y are unsigned *semiring* numerals. The result is either some z or $\ominus z$, depending on the sign of the difference of x and y . These cases are covered by *diff-rules*:

theorem (in ring-bin) diff-rules:

```

 $z \oplus x = y \Longrightarrow x \oplus (\ominus y) = \ominus z$ 
 $z \oplus x = y \Longrightarrow (\ominus y) \oplus x = \ominus z$ 
 $z \oplus y = x \Longrightarrow x \oplus (\ominus y) = z$ 
 $z \oplus y = x \Longrightarrow (\ominus y) \oplus x = z$ 

```

Here z needs to be produced separately in order to apply these rules effectively. Checking the preconditions afterwards works by term rewriting, but computing z the same way would be slightly awkward (requiring auxiliary constructions within the logical context, like twos-complement binary numerals). Instead we shall now use the existing programming environment of SML to work out signed addition (including negation and subtraction). Converting symbolic expressions back and forth to SML integers is straight-forward, by providing functions *dest-binary* and *mk-binary*. The remaining task is to test the result and instruct the logical engine to apply an appropriate *diff-rules* case.

The subsequent **simproc-setup** declaration augments the Simplifier context by a function that proves rewrite rules on the fly (depending on the current redex):

```

simproc-setup (in ring-bin) binary-sub ( $x \oplus (\ominus y) \mid (\ominus y) \oplus x =$  (fn  $\varphi \mapsto$ 
  let
    val [rule1, rule2, rule3, rule4] =  $\varphi$  diff-rules
    val int-of = dest-binary  $\varphi$  val of-int = mk-binary  $\varphi$ 
    val add =  $\varphi$  (op  $\oplus$ ) val add-conv = Simplifier.rewrite ( $\varphi$  binary-add)
  in fn (@{term add t u})  $\mapsto$ 
    let
      val x = int-of t val y = int-of u
      val z = x + y val v = of-int (abs z)
      val (w, rule) =
        if z < 0 then if y < 0 then (t, rule1) else (u, rule2)
        else if y < 0 then (operand-of u, rule3) else (operand-of t, rule4)
      in rule OF [add-conv (@{term add v w})] end
    end)

```

Everything depends on a morphism φ that provides the view of abstract entities from *ring-bin* in the particular application context. In the first stage, the procedure applies φ to all required logical entities of *ring-bin*, notably terms and theorems. In the second stage it operates on the particular problem, accepting a term *add t u* and producing a proven result involving another term *add v w*.

We may now establish equalities for abstract and concrete numerals:

theorem (in *ring-bin*) #48734 \ominus #3762039758274 = \ominus #3762039709540
by *simp*

definition *bit-int* $i\ b = i + i + (\text{if } b \text{ then } (1::\text{int}) \text{ else } (0::\text{int}))$

interpretation *ring-bin* [(*op* +) (*op* *) 0 1 *bit-int* (*op* -) *uminus*] <*proof*>

theorem #48734 - #3762039758274 = - #3762039709540
by *simp*

The same technique also works for more complex calculations, such as *div/mod*.

3 Contexts and declarations

3.1 Generic contexts

Isabelle supports two notions of context: *theory* for large-scale library organization, and *proof context* as medium-scale reasoning infrastructure for structured proofs [17] and structured specifications [12, 1, 2, 9]. Here we present a unified view of *generic contexts* that covers the common characteristics of both kinds.

In primitive inferences $\Gamma \vdash A$ means that A is derivable within a context Γ , which typically contains fixed variables and assumptions. We generalize this idea towards a container for arbitrary logical and extra-logical data, including background theory declarations (types, constants, axioms), local parameters and assumptions, definitions and theorems, syntax and type-inference information, hints for automated proof tools (e.g. the Simplifier, arithmetic procedures) etc.

There are two main logical operations on generic contexts:

1. *Context construction* starts with an empty context Γ_0 and proceeds by adding further context elements consecutively. In particular, $\Gamma + \mathbf{fix}\ x$ declares a local variable, and $\Gamma + \mathbf{assume}\ A$ states a local assumption.
2. *Context export* destructs the logical difference of two contexts, by imposing it on local results. The effect of *export* $\Gamma_1\ \Gamma_2$ is to discharge portions of the context on terms and theorems as follows:

$$\begin{aligned} \text{export } (\Gamma + \mathbf{fix}\ x)\ \Gamma\ (t\ x) &= (\lambda x. t\ x) \\ \text{export } (\Gamma + \mathbf{fix}\ x)\ \Gamma\ (\vdash B\ x) &= (\vdash \bigwedge x. B\ x) \\ \text{export } (\Gamma + \mathbf{assume}\ A)\ \Gamma\ (A \vdash B) &= (\vdash A \implies B) \end{aligned}$$

Discharging assumptions on syntactic terms has no effect.

3.2 Context data

Isabelle belongs to the tradition of “LCF-style” provers, which is centered around formally checked entities being implemented as abstract datatypes. All values of such types being produced at run-time are “correct by construction” — relative to the correctness of a few core modules. For example, certified terms may be composed by syntax primitives to produce again certified terms; likewise theorems are composed by primitive rules to produce new theorems.

We extend this principle towards an abstract type of well-formed proof contexts, with additional support for type-safe additions of context data at compile time. Internally, context data consists of an inhomogeneous record of individual data slots (based on dynamically typed disjoint sums). The external programming interface recovers strong static typing by means of an SML functor, involving dependently-typed modules, functor $Data(ARGS): RESULT$, where:

```
ARGS = sig type T val init: T end
RESULT = sig val put: T → context → context val get: context → T end
```

For example, the inference kernel requires a table of axioms, which is declared by structure $Axioms = Data(\text{type } T = (\text{name} \times \text{term}) \text{ list val init} = [])$. The resulting operations $Axioms.put: (\text{name} \times \text{term}) \text{ list} \rightarrow \text{context} \rightarrow \text{context}$ and $Axioms.get: \text{context} \rightarrow (\text{name} \times \text{term}) \text{ list}$ are kept private to the kernel implementation, exposing only interfaces for adding axioms strictly monotonically, and for turning named axioms into theorems. Thus abstract datatype integrity can be maintained, according to the “LCF system approach”.

Slightly less critical applications of the same mechanism involve automated proof tools that depend on “hints” in the context, such as the Simplifier (cf. §4).

3.3 Morphisms

In general a morphism is just abstract non-sense to organize certain logical operations. The idea is that a morphism determines how results may be transferred into another context, providing a different *view* of previous results.

Formally, a morphism φ is a tuple $(\varphi_{type}, \varphi_{term}, \varphi_{thm})$ of mappings on types, terms, and theorems, respectively. φ_0 refers to the identity morphism. We write uniformly $\varphi \tau$ for some type τ , and φt for some term t , and φth for some theorem th , meaning that the appropriate component is applied. Usually morphisms respect the overall syntactic structure of arguments, by mapping types within terms uniformly etc. Then the full morphism φ is already determined by the φ_{thm} part, since types and terms are sub-structurally included in thm .

A morphism is called *well-formed*, if it preserves logical well-formedness of its arguments. This desirable property may be achieved in practice by composing morphisms from basic operations of the inference kernel. For example, a morphism consisting of primitive inference rules (e.g. for instantiation of variables) will map theorems again to theorems by construction.

The following two kinds of morphism, which resemble abstraction and application in λ -calculus or type-theory, are particularly important in practice.

1. *Export morphism*: the export operation between two contexts (cf. §3.1) determines a well-formed morphism $\varphi = \text{export } \Gamma_1 \Gamma_2$. By this view, local results (with fixed variables and assumptions) appear in generalized form (with arbitrary variables and discharged premises).
2. *Interpretation morphism*: given concrete terms for the fixed variables, and theorems for the assumptions of a context, the substitution operation determines a well-formed morphism $\varphi = \text{interpret } [t/x] [th/A]$. By this view, results of an abstract theory are turned into concrete instances.

3.4 Generic declarations

Ultimately we would like to view arbitrary context data after morphism application, but doing this directly turns out as unsatisfactory — essentially violating abstract datatype integrity. This could be amended by incorporating morphisms into the generic data interface (cf. §3.2), but then implementations would become more complicated due to additional invariants (e.g. consider efficient term index structures in the presence of arbitrary morphisms).

Instead, we handle morphisms at the level of *data declarations*. Recalling that arbitrary datatypes can be incorporated into the generic *context*, any operation on data is subsumed by $\text{context} \rightarrow \text{context}$. This motivates the our definition:

$$\text{declaration} = \text{morphism} \rightarrow \text{context} \rightarrow \text{context}$$

This means a declaration participates in applying the morphism: being passed some φ as additional argument it is supposed to apply it to any logical parameters (types, terms, theorems) involved in its operation. Note that immediate declaration in the current context works by passing the identity morphism φ_0 .

A *fact declaration* is represented as a theorem-attribute pair. This is an important special case; the general declaration is recovered by the *apply* operation:

$$\begin{aligned} \text{fact-declaration} &= \text{thm} \times \text{attribute} \\ \text{attribute} &= \text{thm} \rightarrow \text{context} \rightarrow \text{context} \\ \text{apply } (th, att) &= (\varphi \mapsto att (\varphi th)) \end{aligned}$$

Observe that *th* is transformed separately before invoking the *attribute*. Here the internal operation does not have to consider morphisms at all.

The Isabelle syntax for (th, att) is “*th [att]*”, which may appear wherever facts occur, as in “**theorem** *th [att]: A*” or “**declare** *th [att]*”. General declarations are written as “**declaration** *d*”, for $d : \text{morphism} \rightarrow \text{context} \rightarrow \text{context}$.

3.5 Locales

Locales [12] provide a high-level mechanism to organize proof context elements and declarations (cf. §3.4). *Locale expressions* [1] compose existing locales by means of merge and rename operations. *Locale interpretation* [2] transfers results stemming from a locale into another context. *Type-classes* [16] express

properties of polymorphic entities within the type-system; there is a canonical interpretation of classes as specific locales [9]. These building-blocks provide a medium-scale module concept on top of the existing Isabelle logic — locale operations are reduced to basic inferences (usually expressed via morphisms).

A locale specification consists of the following two distinctive parts.

1. *Assumptions* refer to fixed types, terms, and hypotheses. The latter two are specified explicitly by the notation “**fixes** x ” and “**assumes** $A x$ ”; types are maintained implicitly according to occurrences in the given formulae. The assumption part determines the logical meaning of a locale, which is expressed as predicate definition together with a context construction: the definition “**locale** $c = \mathbf{fixes} x \mathbf{assumes} A x$ ” produces a predicate constant $c = (\lambda x. A x)$, and a context construction $\Gamma + \mathbf{fix} x \mathbf{assume} A x$.
2. *Conclusions* are essentially theorems that depend on the locale context and are adjoined to the locale later on, without changing the logical meaning. The notation “(**in** c)” may be added to various theory elements (notably **theorem**) to indicate that the result is established within the context of c , and stored there for later use. Fact declarations may be included, too.

The conclusion part is what really matters in practical use, including declarations of arbitrary extra-logical data. The locale infrastructure maintains a canonical order of declarations d_1, \dots, d_n . Whenever the locale context is reconstructed later, relative to a morphism φ , the context will be augmented by the collective declaration of $d_n \varphi (\dots (d_1 \varphi \Gamma))$. For our purpose, we have generalized the existing fact declaration mechanism towards arbitrary **declaration** functions (cf. §3.4). Thus we may attach arbitrary SML values to a locale, which will be transformed together with the logical content by morphism application.

4 Context-dependent simplification

Isabelle’s generic Simplifier [14] is already a non-trivial example of integrating automated proof tools into our architecture of context data and declarations (cf. §3). The generic proof method *simp* depends on a *simpset* container being maintained as context data. A *simpset* roughly consists of a set of simplification rules and simplification procedures (of type *simproc*), which are maintained by:

$$\begin{aligned} \text{add-simp: } thm &\rightarrow simpset \rightarrow simpset \\ \text{add-simproc: } simproc &\rightarrow simpset \rightarrow simpset \end{aligned}$$

Declaration of plain simplification *rules* (cf. the example in §2.1) follows the general scheme of *fact-declaration* of §3.4. Here *add-simp* is wrapped into an attribute called *simp*. Morphism application is trivial, since rules are theorems.

Declaration of simplification *procedures* is more involved, cf. the example in §2.2. A *simproc* essentially consists of a function $morphism \rightarrow term \rightarrow thm$ that turns a redex t into a theorem $t = t'$. We introduce a derived declaration “**simproc-setup** $name (patterns) = f$ ” for $patterns : term^*$ and $f : morphism$

→ *simproc*, which roughly abbreviates “**declaration** ($\varphi \mapsto \text{add-simproc } (\varphi f)$)”. What does it mean to apply a morphism to a function? We define:

$$\varphi f = (\psi \mapsto f (\psi \circ \varphi))$$

Our main observation is that a function transforms itself by a given morphism. The body will apply the morphism to all required logical entities, or pass it on to other functions. To apply φ to f , we compose φ with any morphism ψ being passed in the future. Actual evaluation in the present context is commenced by invoking $f \varphi_0$ (the identity morphism). Note that this scheme is reminiscent of programming language semantics presented in “continuation-passing style”.

Tool-compliant morphisms. A well-formed morphism (cf. §3.3) guarantees stability of logical entities after transformation. Unfortunately, this is *not* sufficient to achieve stability of arbitrary functions. For example, even with plain simplification rules alone, the Simplifier is not necessarily stable after arbitrary interpretation of its context: although an interpreted rewrite system still represents equational consequences of the original theory, it may cause the rewrite strategy to produce unwanted results or fail to terminate.

A morphism φ is called *f-compliant* iff φf preserves the intended operational behavior of $f \varphi_0$. E.g. for *binary-sub* defined in §2.2, tool-compliant morphisms include those that replace *bit*, *add*, *neg* by rigid first-order terms, but not arbitrary λ -abstractions that destroy the shape of the term patterns involved.

Morphisms stemming from locale import expressions [1] (type instantiations and renaming) are usually tool-compliant. Substitution of parameters by constants works equally well. Assuming that the tool is able to perform unification of simple types at runtime, types may be even generalized here, e.g. replacing the fixed α of locale *ring-bin* by an arbitrary $\beta::c$ constrained by a suitable type-class c that ensures the required algebraic properties. This means that the canonical interpretation of locales as type-classes [9], where fixed parameters are replaced by polymorphic constants yields morphisms that comply with such tools.

5 Application: Gröbner Bases

We use the techniques of §3 to integrate a proof-tool for (semi)-ring equalities. The procedure follows the traditional refutation scheme: negate the input then transform to DNF and disprove each disjunct. Hence we consider formulae P of the form $\bigwedge_{i=1}^n p_i = 0 \wedge f \neq 0$, where the p_i 's and f are polynomials within *semiring*. Recall that we can turn several inequalities into only one. We use an oracle to find a Nullstellensatz refutation consisting of polynomials h_1, \dots, h_n and a number k s.t. $\sum_{i=1}^n h_i \cdot p_i = f^k$. It is clear that by proving this last equality, $\neg P$ follows. The oracle computes a Gröbner basis [4] for the ideal generated by the p_i 's and reduces f in that basis. The whole process keeps track on where the S-polynomials originate from in order to express the final result in terms of the p_i 's. This implementation has been ported from HOL Light [10], see also [13, §3.4.2]. Note that this procedure also solves many problems in *semiring*,

by dealing with $p = q$ as if it were $p - q = 0$ and separating the “negative” and “positive” parts of the Nullstellensatz certificate. This yields a correct yet incomplete procedure, depending only on two axioms stated in *semiringb*:

locale *semiringb* = *semiring* +
assumes *add-cancel*: $(x \oplus y = x \oplus z) \leftrightarrow y = z$
and *idom*: $(w \odot y \oplus x \odot z = w \odot z \oplus x \odot y) \leftrightarrow w = x \vee y = z$

We emphasize that our procedure *proves* results by inference: the oracle only finds a certificate, used to derive the conflicting equation. The final equation emerges by normalizing both sides, using genuine inferences. This approach also allows to integrate other tools solving the detachability problem [13, §3.4.2] or provide the construction information of the S-polynomials (as e.g. in AXIOM).

Context data. The method sketched above is essentially parameterized by the operations and axioms of (semi)ring (in order to *prove* the normal form of polynomials and recognize their structure) and by four non-logical functions: *is-const* recognizes canonical ring expressions as “constants”, *dest-const* produces SML rational numbers from ring constants, *mk-const* produces ring constants from SML rational numbers, and *ring-conv: morphism → term → thm* produces *proven* normalization results of numeral expressions.

Following §3.2, the method reserves a data slot in the context to manage all its instances. We emphasize that instances no longer depend on a morphism, since these are applied at declaration-time, where the corresponding instance of our procedure is computed. At run-time, when we have to prove P , we extract a polynomial q occurring in P and try to find a corresponding interpretation installed in our context-data. Afterwards it proceeds as explained above.

Using the proof method. The end-user sees an Isabelle proof method *algebra*, that works for any tool-compliant interpretation of *semiringb*. The user only needs to interpret the *semiringb* locale to activate the method in his context. The default configuration includes interpretations of *semiringb* to “prominent” types in Isabelle/HOL: natural numbers and the type-class *idom*, which covers e.g. integers and real numbers. Here is a simple example:

theorem $x * (x^2 - x - 5) - 3 = (0::int) \leftrightarrow (x = 3 \vee x = -1)$ **by** *algebra*

6 Conclusion

We have presented a system architecture that emphasizes the explicit context-dependency of both logical and non-logical entities, with specific infrastructure to manage generic data, declarations, morphisms etc. An interesting observation about context-aware proof tools is that there are three phases: compile-time (when loading the sources), declaration-time (when applying morphisms to transfer abstract entities to particular interpretations), and run time (when the tool is invoked on concrete problems).

Traditional “LCF system programming” directly uses features of SML to abstract over theory dependencies. Our approach goes beyond this by providing

specific means to organize the requirements of a tool. Thus the implementor is enabled to build generic methods that are transferred automatically into the application context later on. The end-user merely needs to perform a logical interpretation of the corresponding tool context, without being exposed to SML.

This integration of abstract theory mechanisms and proof methods is achieved without changing the logical foundations of Isabelle; all additional mechanisms are reduced to existing logical principles. Apart from these theoretical considerations there remains the important practical issue to implement tools that are actually stable under application of typical theory morphisms.

References

- [1] C. Ballarin. Locales and locale expressions in Isabelle/Isar. In S. Berardi et al., editors, *Types for Proofs and Programs (TYPES 2003)*, LNCS 3085, 2004.
- [2] C. Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In J. M. Borwein and W. M. Farmer, editors, *Mathematical Knowledge Management (MKM 2006)*, LNAI 4108, 2006.
- [3] B. Barras et al. *The Coq Proof Assistant Reference Manual, v8*. INRIA, 2006.
- [4] B. Buchberger. Ein algorithmisches Kriterium für die Lösbarkeit eines algebraischen Gleichungssystems. *Aequationes mathematicae*, 3:374–383, 1970.
- [5] A. Chaieb. Verifying mixed real-integer quantifier elimination. In U. Furbach and N. Shankar, editors, *IJCAR 2006*, LNCS 4130. Springer, 2006.
- [6] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An interactive mathematical proof system. *Journal of Automated Reasoning*, 11(2), 1993.
- [7] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. LNCS 78. 1979.
- [8] B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in Coq. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2005)*, LNCS 3603, 2005.
- [9] F. Haftmann and M. Wenzel. Constructive type classes in Isabelle. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs (TYPES 2006)*.
- [10] J. Harrison. *HOL Light Tutorial (for version 2.20)*, September 2006.
- [11] J. Harrison and L. Théry. A skeptic’s approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.
- [12] F. Kammüller, M. Wenzel, and L. C. Paulson. Locales: A sectioning concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics (TPHOLs ’99)*, LNCS 1690, 1999.
- [13] B. Mishra. *Algorithmic algebra*. Springer-Verlag, 1993.
- [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. Organizing numerical theories using axiomatic type classes. *Journal of Automated Reasoning*, 33(1), 2004.
- [16] M. Wenzel. Type classes and overloading in higher-order logic. In E. L. Gunter and A. Felty, editors, *Theorem Proving in Higher Order Logics (TPHOLs ’97)*, LNCS 1275, 1997.
- [17] M. Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, TU München, 2002.