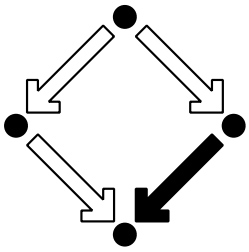
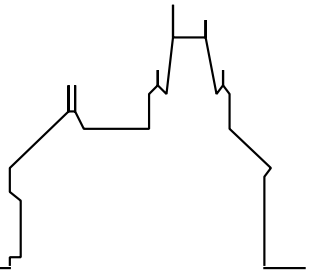


RISC-Linz
Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria, Europe



Programming Languages for Mechanized Mathematics Workshop

Jacques CARETTE and Freek WIEDIJK (Eds.)

Hagenberg, Austria
June 29–30, 2007

RISC-Linz Report Series No. 07-10

Editors: RISC-Linz Faculty
B. Buchberger, R. Hemmecke, T. Jebelean, M. Kauers, T. Kutsia, G. Landsmann, F. Lichtenberger, P. Paule, H. Rolletschek, J. Schicho, C. Schneider, W. Schreiner, W. Windsteiger, F. Winkler.

Supported by:

PML – a new proof assistant

Christophe Raffalli

Abstract

We will present our ongoing work on a new proof assistant and deduction system named PML. The basic idea is to start from an ML-like programming language and add specification and proof facilities.

On the programming language side, the language unifies certain concepts: PML uses only one notion of sum types (polymorphic variants) and one notion of products (extensible records). These can then be used to encode modules and objects. PML's typing algorithm is based on a new constraint consistency check (as opposed to constraint solving).

We transform the programming language into a deduction system by adding specification and proofs into modules. Surprisingly, extending such a powerful programming language into a deduction systems requires very little work. For instance, the syntax of programs can be reused for proofs.

URL: `http://www.lama.univ-savoie.fr/~raffalli/pml/`

Declarative Representation of Proof Terms

Claudio Sacerdoti Coen*

Department of Computer Science, University of Bologna
sacerdot@cs.unibo.it

Abstract. We present a declarative language inspired by the pseudo-natural language used in Matita for the explanation of proof terms. We show how to compile the language to proof terms and how to automatically generate declarative scripts from proof terms. Then we investigate the relationship between the two translations, identifying the amount of proof structure preserved by compilation and re-generation of declarative scripts.

1 Introduction

In modern interactive theorem provers, proofs are likely to have several alternative representation inside the system. For instance, in a system based on Curry-Howard implementation techniques, proofs could be input by the user in either a declarative or a procedural proof language; then the script could be interpreted and executed yielding a proof tree; from the proof tree we can generate a proof term; from the proof term, the proof tree or the initial script we can generate a description of the proof in a pseudo-natural language; finally, from the proof term, the proof tree or a declarative script we can generate a content level description of the proof, for instance in the OMDoc + MathML content language. For instance, the Coq proof assistant [12] has had in the past or still has all these representations but the last one; our Matita interactive theorem prover [2] also has all these representations but proof trees.

It is then natural to investigate the translations between the different representations, wondering how much proof structure can be preserved in the translations. In [10] we started this study by observing that $\bar{\lambda}\mu\tilde{\mu}$-proof-terms are essentially isomorphic to the pseudo-natural language we proposed in the HELM and MoWGLI projects. In [3] we extended the result to OMDoc documents. At the same time we started investigating the possibility of giving an executable semantics to the grammatical constructions of our pseudo-language, obtaining the declarative language described in this paper. The language, which still lacks the justification sub-language, is currently in use in the Matita proof assistant.

In this paper we investigate the mutual translation between declarative scripts in this language and proof terms. We use λ-terms for an extension of System F to keep the presentation simple but close to the actual implementation in Matita, which is not based on $\bar{\lambda}\mu\tilde{\mu}$-proof-terms.

* Partially supported by the Strategic Project DAMA (Dimostrazione Assistita per la Matematica e l'Apprendimento) of the University of Bologna.

Our main result is that the two translations preserve the proof structure and behave as inverse functions on declarative scripts generated by proof terms. Compilation and re-generation of a user-provided declarative script results in a script where the original proof steps and their order are preserved, and additional steps are added to make explicit all the justifications previously proved automatically. Misuses of declarative statements are also corrected by the process.

Translation of procedural scripts to declarative scripts can now be achieved for free by compiling procedural scripts to proof terms before generating the declarative scripts. In this case the proof structure is preserved only if it is preserved (by the semantics of tactic compilation) during the first translation.

In the companion paper [6] Ferruccio Guidi investigates the translation between proof terms of the Calculus of (Co)Inductive Constructions and a subset of the procedural language of Matita. Thus the picture about the different translations is now getting almost complete, up to the fact that the papers presented do not agree on the intermediate language used by all the translations, which is the proof terms language.

An immediate application of this investigation, also explored in [6], is the possibility to take a proof script from a proof assistant (say Coq), compile it to proof terms, transmit them to another proof assistant (say Matita) based on the same logic and rebuild from them either a declarative or a procedural proof script that is easier to manipulate and to be evolved. A preliminary experiment in this sense is also presented in the already cited paper.

The requirement for the translations investigated in this paper are presented in Section 2. Then in Section 3 we present the syntax and the informal semantics of our declarative proof language. Compared with other state of the art declarative languages such as Isar [13] and Mizar [14] we do not address the (sub-)language for justification of proof steps. This is left to future work. Right now justifications are either omitted (and provided by automation) or they are proof terms.

In Section 4 we show the small steps operational semantics of the language which, scripts being sequences of statements, is naturally unstructured in the spirit of [11]. The semantics of a statement is a function from partial proof terms to partial proof terms, i.e. a procedural tactic. Thus the semantics of a declarative script is a compilation to proof terms mediated by tactics in the spirit of [7].

In Section 5 we show the inverse of compilation, i.e. the automatic generation of a declarative script from a proof term. We prove that the two translations form a retraction pair and that their composition is idempotent.

2 Requirements

In this paper we explain how to translate declarative scripts into proof terms and back. By going through proof terms, procedural scripts can also be translated to declarative scripts. Before addressing the details of the translations, we consider

here their informal requirements. We classify the requirements according to two interesting scenarios we would like to address.

Re-generation of declarative scripts from declarative scripts (via proof terms). In this scenario a declarative script is executed obtaining a proof term that is then translated back to a declarative script. The composed translation should preserve the structure of the user provided text, but can make more details explicit. For instance, it can interpolate a proof step between two user provided proof steps or it can add an omitted justification for a proof step. The translation must also reach a fix-point in one iteration. The latter requirement is a consequence of the following stronger requirement: the proof term generated executing the obtained declarative script should be exactly the same proof term used to generate the declarative script. In other terms, the composed translation should not alter the proof term in any way and can only reveal hidden details.

Re-generation of declarative scripts from procedural scripts (via proof terms). In this scenario a procedural script is executed obtaining a proof terms that is then translated back to a declarative script. Ideally the two scripts should be equally easy to modify and maintain. Moreover, the "structure" of the procedural script (if any) should be preserved. Gory details or unnecessary complex sub-proofs that are not explicit in the procedural proof should be hidden in the declarative one. This last requirement is not really a constraint on the declarative language, but on the implementation of the tactics of the proof assistant [9].

Some of the requirements, in particular the preservation of the structure of the user provided text, seem quite difficult to obtain. In [10] we claimed that the latter requirement is likely to be impossible to fulfil when proof terms are Curry-Howard isomorphic to natural deduction proof trees, i.e. when proof terms are simply λ-terms. On the contrary, we expect to be able to fulfil the requirements if proof terms are Curry-Howard isomorphic to sequent calculus. This is the case, for instance, for the $\bar{\lambda}\mu\tilde{\mu}$-terms [5] we investigated as proof terms in [10,3]. In particular, automatic structure preserving generation of Mizar/Isar procedural scripts from $\bar{\lambda}\mu\tilde{\mu}$-terms have been attempted in the Fellowship theorem prover [8] (joint work with Florent Kirchner).

Matita proof terms are λ-terms of the Calculus of (Co)Inductive Constructions (CIC). The calculus is so rich that several of the required constructs of the $\bar{\lambda}\mu\tilde{\mu}$-calculus are somehow available. Thus we expect to be able to fulfil at least partially the requirements just presented. Even in case of failure it is interesting to understand exactly how close we can get.

In the present paper we restrict ourselves to a fragment of CIC, although the implementation in Matita considers the whole calculus. The fragment is an extension of System F obtained by adding explicit type conversions, local definitions and local proofs. The distinction between the last two corresponds to the distinction between λ and Λ in System F. We will present the proof terms for this fragment in Section 4.

assume id **:** type [**that is equivalent to** type]
suppose prop [(id)] [**that is equivalent to** prop]
let id **:=** term
by just **we proved** prop (id) [**that is equivalent to** prop]
by just **we proved** prop [**that is equivalent to** prop] **done**
by just **done**
by just **let** id **:** type **such that** prop (id)
by just **we have** prop (id) **and** prop (id)
we need to prove prop [(id)] [**or equivalently** prop]
we proceed by [**cases**|**induction**] **on** term **to prove** prop
case id [(id**:**type)|(id**:**prop)]*
by induction hypothesis we know prop (id) [**that is equivalent to** prop]
the thesis becomes prop [**or equivalently** prop]
conclude term rel term **by** just [**done**]
obtain id term rel term **by** just [**done**]
rel term **by** just [**done**]

Non terminals:

id	identifiers	term	inhabitants of data types
type	data types	just	justifications, e.g. proof terms or automation "_"
prop	propositions	rel	transitive relations (e.g. $=, \leq, <$)

Table 1. Syntax

3 The Declarative Language

The syntax of the declarative language we propose is an adaptation of the syntax of the pseudo natural language already generated by Matita and studied in [1]. It is also a super-set of the language proposed in [10] and studied also in [3]. The sub-language for justifications has not been developed yet. Thus currently a justification is either provided as a proof term or it is omitted and recovered by automation. Because of the lack of the sub-language for justifications we post-pone a comparison with other declarative languages.

We have explicit statements that deal with conversion, which is a feature of the logical framework of Matita. Two formulae are convertible when they can be reduced by computation to a common value. For instance, $2 * 2$ is convertible with $3 + 1$. Since conversion is a decidable property (in a confluent and strongly normalizable calculus), conversion and reduction steps are not recorded in the proof term (e.g. as rewriting steps). However, since conversion steps are not always obvious to the reader, it is sometimes necessary to make them explicit in the declarative language. Thus the need for the additional statements. In Isar the same steps would be represented by (chains of) rewriting steps.

We now present informally the semantics of the proposed language statements, whose syntax is summarised in Table 1.

assume *id* **:** *type1* [**that is equivalent to** *type2*]
Introduces in the context a new generic but fixed term *id* whose type is

type1. If specified, *type2* must be convertible to *type1*. In this case *id* will be used later on with type *type2*, but in the conclusion of the proof *type1* will be used. Example:

we need to prove $\forall x : T.P(x)$
  assume x: carrier of T that is equivalent to $N \times N$
  let $y :=$ $\pi_1(x)$ *(* first projection *)*
  ...
  by _ **we proved** $P(x)$
done

suppose *prop1* [(*id*)] [**that is equivalent to** *prop2*]
Introduces in the context the hypothesis *prop1* labelled by *id*. If the proposition *prop2* is specified, it must be convertible with *prop1*. In this case *id* will stand later on for the hypothesis *prop2*, but in the conclusion of the proof *prop1* will be used.

let *id* **:=** *term*
Introduced in the context a new local definition.

by *just* **we proved** *prop1* **(***id***)** [**that is equivalent to** *prop2*]
Concludes the proposition *prop1* by means of the justification *just*. The (proof of the) proposition is labelled by *id* for further reference. If *prop2* is specified, it must be convertible with *prop1*. In this case *id* will stand for a proof of the proposition *prop2*.

by *just* **we proved** *prop1* [**that is equivalent to** *prop2*] **done**
Similar to the previous statement. However, the conclusion *prop1* (or *prop2* if specified and convertible with *prop1*) is the current thesis. Thus this statement ends the innermost sub-proof.

by *just* **done**
Similar to the previous statement. However, the conclusion, equal to the current thesis, is not repeated.

by *just* **let** *id1* **:** *type* **such that** *prop* **(***id2***)**
Concludes the proposition $\exists id1 : type$ s.t. *prop* by means of the justification *just*. Exist-elimination is immediately performed yielding the new generic but fixed term *id1* of type *type* and the new hypothesis *prop* labelled by *id2*.

by *just* **we have** *prop1* **(***id1***) and** *prop2* **(***id2***)**
Concludes the proposition $prop1 \wedge prop2$ by means of the justification *just*. And-elimination is immediately performed yielding the new hypotheses *prop1* and *prop2* labelled respectively by *id1* and *id2*.

we need to prove *prop1* [(*id*)] [**or equivalently** *prop2*]
If *id* is omitted, it repeats the current thesis *prop1*. Moreover, if *prop2* is specified and convertible with *prop1*, it replaces the current thesis with *prop2*. Otherwise, if *id* is specified, it starts a nested sub-proof of *prop1* that will be labelled by *id*. If *prop2* is specified and convertible with *prop1*, the thesis of the nested sub-proof is *prop2*, but *id* will label *prop1*.

we proceed by [**cases**|**induction**] **on** *term* **to prove** *prop*
case *id1* [(*id2*:*type2*)|(*id2*:*prop*)]*
by induction hypothesis we know *prop1* (*id*) [**that is equiv. to** *prop2*]
the thesis becomes *prop1* [**or equivalently** *prop2*]

This set of statements are used for proofs by structural induction or by case analysis. The initial statement must be followed by a proof for each case. Each proof must be started by the **case** *id* statement, where *id* is the label of the case (i.e. the name of the inductive constructor the case refers to). The list of arguments that follows *id* binds the local non inductive assumptions for the case. The inductive assumptions are postponed and introduced by the next statement in the set. Only proofs by inductions have inductive assumptions. The last statement in the set, **the thesis becomes**, is used to state explicitly what is the current thesis for each proof. Example:

we proceed by induction on n **to prove** $P(n)$
case 0
  the thesis becomes $P(0)$
  ...
case $S(m : nat)$
  by induction hypothesis we know $P(m)$
  the thesis becomes $P(S(m))$
  ...

conclude *term1 rel term2* **by** *just* [**done**]
obtain *id term rel term* **by** *just* [**done**]
rel term **by** *just* [**done**]

This set of statements are used for chains of (in)equalities. A chain is started by either the first or the second command in the set. All the remaining steps in the chain are made by the third command. In all commands *rel* must be a transitive relation. Chains with mixed relations are possible as soon as the different relations enjoy generalised transitivity (e.g. $x \leq y \wedge y < z \Rightarrow x < z$). Every step in the chain must have a justification *just*. The end of the chains is marked by **done**. In every step but the first one the left hand side of the inequation is the right hand side of the previous step.

If the first step of the chain is a **conclude** statement, then the chain must prove the current thesis, and the last step of the chain ends the innermost sub-proof. Otherwise, if the first step of the chain is an **obtain** statement, the chain only proves a local lemma that is labelled by *id* in the rest of the innermost sub-proof.

obtain H
  $(x+y)^2 = (x+y)(x+y)$ **by** _
  $= x(x+y) + y(x+y)$ **by** *distributivity*
  $= x^2 + xy + yx + y^2$ **by** *distributivity*
  $= x^2 + 2xy + y^2$ **by** _
done

Types		
T	$::= T \to T$	function space
	$\mid$ nat	basic type
Propositions		
P	$::= P \Rightarrow P$	logical implication
	$\mid \forall x : T.P$	universal quantification
	$\mid \exists x : T.P$	existential quantification
	$\mid P \wedge P$	conjunction
	$\mid E = E$	equality
	$\mid F(E_1, \ldots, E_n)$	n-ary predicate
Expressions (inhabitants of types)		
E	$::= x$	bound variable ranging over expressions
	$\mid O(E_1, \ldots, E_n)$	n-ary function
	$\mid ?$	placeholder
Proof terms		
t	$::= \lambda x : T.t$	function
	$\mid \Lambda H : P.t$	type abstraction
	$\mid$ let $x := E$ in t	local definition
	$\mid$ Let $H : P := t$ in t	logical cut
	$\mid (t : P \equiv P)$	explicit type conversion
	$\mid (H\ E_1\ \ldots\ E_n\ H_1 \ldots\ H_m)$	application with 0 or more arguments; H is a bound variable ranging over proof terms or one of the constants below
	$\mid \text{and\_elim}_{P,P,P}$	conjunction elimination
	$\mid$ ex_elim	existential elimination
	$\mid \text{nat\_ind}_P$	induction over Peano natural numbers
	$\mid \text{nat\_cases}_P$	case analysis over Peano natural numbers
	$\mid$ eq_transitive	transitivity of equality

Table 2. Proof term syntax

4 Formal Semantics

We now show the formal semantics of our language in terms of compilation of a declarative script to a proof term. In Tables 2 and 3 we show the syntax and typing rules for the proof terms we will use to encode first order logic natural deduction trees. We only show the inference rules for proof terms, omitting all the conditions about the well-formedness of contexts, types and propositions occurring in the inference rules, since they are quite standard and not relevant to the present work. Moreover we restrict induction and case analysis to natural numbers and we only consider chains of equalities over natural numbers.

The semantics of each statement of Table 1 is a function from a partial proof term to a partial proof term. Intuitively, a partial proof term is a proof

Proof term typing rules.

$$\frac{\Gamma \vdash H : \forall x_1 : T_1 . \ldots . \forall x_n : T_n . P_1 \Rightarrow \ldots \Rightarrow P_m \Rightarrow P \quad \Gamma \vdash E_i : T_i \;\; \forall i \in \{1, \ldots, n\} \quad \Gamma \vdash H_i : P_i\{E_1/x_1 \; ; \; \ldots \; ; \; E_n/x_n\} \;\; \forall i \in \{1, \ldots, m\}}{\Gamma \vdash (H \; E_1 \; \ldots \; E_n \; H_1 \; \ldots \; H_m) : P}$$

$$\frac{\Gamma, x : T \vdash t : P}{\Gamma \vdash \lambda x : T . t : \forall x : T . P} \qquad \frac{\Gamma, H : P_1 \vdash t : P_2}{\Gamma \vdash \Lambda H : P_1 . t : P_1 \Rightarrow P_2} \qquad \frac{\Gamma \vdash t : P_1 \qquad \Gamma \vdash P_1 \equiv P_2}{\Gamma \vdash (t : P_1 \equiv P_2) : P_2}$$

$$\frac{\Gamma, x := E \vdash t : P}{\Gamma \vdash \text{let } x := E \text{ in } t : P\{E/x\}} \qquad \frac{\Gamma \vdash t_1 : P_1 \qquad \Gamma, H : P_1 \vdash t_2 : P_2}{\Gamma \vdash \text{Let } H : P_1 := t_1 \text{ in } t_2 : P_2}$$

We also assume the following constant schemes (that are always supposed to be applied to arguments in β-long normal form):

$\text{and\_elim}_{P_1,P_2,P_3}$:	$P_1 \wedge P_2 \Rightarrow (P_1 \Rightarrow P_2 \Rightarrow P_3) \Rightarrow P_3$
$\text{ex\_elim}_{T,P_1,P_2}$:	$(\exists x : T . P_1) \Rightarrow (\forall x : T . P_1 \Rightarrow P_2) \Rightarrow P_2$
nat_ind_P :	$\forall n : \text{nat} . P(0) \Rightarrow (\forall m : \text{nat} . P(m) \rightarrow P(S(m))) \Rightarrow P(n)$
nat_cases_P :	$\forall n : \text{nat} . P(0) \Rightarrow (\forall m : \text{nat} . P(m)) \Rightarrow P(n)$
eq_transitive :	$\forall x, y, z : \text{nat} . x = y \Rightarrow y = z \Rightarrow x = z$

Table 3. Proof term typing rules (standard well-formed conditions on expressions, contexts, and types omitted)

term with linear placeholders for missing sub-proofs and non-linear placeholders for missing sub-expressions. Each placeholder must be replaced with a proof term or an expression, of the appropriate type, closed in the logical context of the placeholder. The logical context of the placeholder is the ordered set of hypothesis, definitions and declarations collected navigating the proof term from the root to the placeholder. A partial proof term is complete (i.e. it represents a completed proof) when it is placeholder-free. When a proof is started, it is represented by the partial proof term made of just one placeholder.

Formally, we represent a partial proof term as a triple (Σ, Σ', Π). Σ is an ordered list of sequents $\Gamma \vdash P$ providing context and type for the proof term placeholders occurring in the partial proof. Σ' does the same for expression placeholders. Π, the actual partial proof term, is a function from "fillings" for both kinds of placeholders to placeholder-free proof terms.

```
partial_proof :=
  (context * proposition) list *
  (context * type) list *
  (proof_term list  * expression list → proof_term)
```

We denote the empty list with $[]$, the concatenation of two lists with $l_1 @ l_2$ and the insertion of an element at the beginning of a list with $x :: l$. With $(l, l') \mapsto t$ we denote an anonymous function from pairs of lists to terms. With $C[l, l']$ we

represent a proof term having all the proof-terms in l and all the expressions in l' as sub-terms. Finally, π_3 is the third projection of a tuple.

The semantic function $\mathcal{C}[\![\cdot]\!]$ shown in Table 4 maps statements to functions from partial proof terms to partial proof terms. $\mathcal{C}[\![\cdot]\!]^\star$ extends the semantics to a list of statements (a declarative script). Given a declarative script $S_1\ \cdots\ S_n$, the proof term generated executing the script $\mathcal{S}$ from the initial proof state for a proposition P is given by $\mathcal{C}[\![\cdot]\!]_s$ applied to $(\mathcal{S}, P)$.

$$\mathcal{C}[\![\cdot]\!] : \text{statement} \to \text{partial\_proof} \to \text{partial\_proof}$$

$$\mathcal{C}[\![\cdot]\!]^\star : \text{statement list} \to \text{partial\_proof} \to \text{partial\_proof}$$
$$\mathcal{C}[\![S_1 \cdots S_n]\!]^\star = \mathcal{C}[\![S_n]\!] \circ \cdots \circ \mathcal{C}[\![S_1]\!]$$

$$\mathcal{C}[\![\cdot]\!]_s : \text{statement list} * \text{proposition} \to \text{proof\_term}$$
$$\mathcal{C}[\![S_1 \cdots S_n]\!]_s = \pi_3(\mathcal{C}[\![S_1\ \cdots\ S_n, P]\!]^\star\ ([\vdash P], [], ([H], []) \mapsto H))\ ([], [])$$

For instance, consider the statement $\forall x : \text{nat}.P(x)$ and a script "**assume** x : nat $\mathcal{S}$" where we suppose that $\mathcal{S}$ produces for the sequent $x : \text{nat} \vdash P(x)$ a proof term π (i.e. that $\mathcal{C}[\![\mathcal{S}]\!]^\star([x : \text{nat} \vdash P(x)], [], \Pi) = ([], [], ([], []) \mapsto \Pi([\pi], []))$)

We have:

$$\begin{aligned}
&\mathcal{C}[\![\textbf{assume } x : \text{nat } \mathcal{S}, \forall x : \text{nat}.P(x)]\!]_s \\
&= \pi_3((\mathcal{C}[\![\mathcal{S}]\!]^\star \circ \mathcal{C}[\![\textbf{assume } x : T]\!])\ ([\vdash \forall x : \text{nat}.P(x)], [], (H, []) \mapsto H))\ ([], []) \\
&= \pi_3(\mathcal{C}[\![\mathcal{S}]\!]^\star([x : \text{nat} \vdash P(x)], [], ([hd], []) \mapsto \lambda x : \text{nat}.hd))\ ([], []) \\
&= \pi_3((([], [], ([], []) \mapsto \lambda x : \text{nat}.\pi))\ ([], []) \\
&= \lambda x : \text{nat}.\pi
\end{aligned}$$

"**obtain** $H\ E_1 = E_2$" is the only statement whose semantics introduces in Σ' a new expression placeholder ?. The placeholder ? stands for the right hand side of the last expression of the chain. Its instantiation will be known only in the last step of the chain, i.e. in the next "$= E_3$ **done**" statement. Since equation chains cannot be nested, in a partial proof term there can be at most one placeholder, i.e. Σ' can have at most one element and one placeholder symbol ? is sufficient.

Table 4: Formal semantics

$$\begin{aligned}
&\mathcal{C}[\![\textbf{assume } x : T]\!](\Gamma \vdash \forall x : T.P :: \Sigma, \Sigma', \Pi) = \\
&\quad ((\Gamma\ ;\ x : T \vdash P) :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi((\lambda x : T.hd) :: tl, l)) \\
&\mathcal{C}[\![\textbf{assume } x : T_1 \textbf{ that is equivalent to } T_2]\!](\Gamma \vdash \forall x : T_1.P :: \Sigma, \Sigma', \Pi) = \\
&\quad ((\Gamma\ ;\ x : T_2 \vdash P) :: \Sigma, \Sigma', \\
&\qquad \begin{cases} (hd : P' \equiv P) :: tl, l) \mapsto \Pi((\lambda x : T_2.hd : \forall x : T_2.P' \equiv \forall x : T_1.P) :: tl, l) \\ (hd :: tl, l) \qquad\qquad\quad \mapsto \Pi((\lambda x : T_2.hd : \forall x : T_2.P \equiv \forall x : T_1.P) :: tl, l) \end{cases} \\
&\mathcal{C}[\![\textbf{suppose } P_1\ (H)]\!](\Gamma \vdash \forall P : P_1.P_2 :: \Sigma, \Sigma', \Pi) = \\
&\quad ((\Gamma\ ;\ H : P_1 \vdash P_2) :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi((\Lambda H : P_1.hd) :: tl, l)) \\
&\mathcal{C}[\![\textbf{suppose } P_1\ (H) \textbf{ that is equivalent to } P_2]\!](\Gamma \vdash \forall H : P_1.P :: \Sigma, \Sigma', \Pi) = \\
&\quad ((\Gamma\ ;\ H : P_2 \vdash P) :: \Sigma, \Sigma',
\end{aligned}$$

$$\begin{cases} ((hd : P_3 \equiv P) :: tl, l) \mapsto \Pi((\Lambda H : P_2.hd : P_2 \to P_3 \equiv P_1 \to P) :: tl, l) \\ (hd :: tl, l) \qquad\qquad\quad \mapsto \Pi((\Lambda H : P_2.hd : P_2 \to P \equiv P_1 \to P) :: tl, l) \end{cases}$$

$\mathcal{C}[\![\textbf{let } x := E]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$((\Gamma\ ;\ x := E \vdash P) :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi((\text{let } x := E \text{ in } hd) :: tl, l))$

$\mathcal{C}[\![\textbf{by } j \textbf{ we proved } P_1\ (H)]\!](\Gamma \vdash P_2 :: \Sigma, \Sigma', \Pi) =$
$((\Gamma\ ;\ H : P_1 \vdash P_2) :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi((\text{Let } H : P_1 := j \text{ in } hd) :: tl, l))$

$\mathcal{C}[\![\textbf{by } j \textbf{ we proved } P_1\ (H) \textbf{ that is equivalent to } P_2]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$((\Gamma\ ;\ H : P_2 \vdash P) :: \Sigma, \Sigma',$
$(hd :: tl, l) \mapsto \Pi((\text{Let } H : P_1 := (j : P_1 \equiv P_2) \text{ in } hd) :: tl, l))$

$\mathcal{C}[\![\textbf{by } j \textbf{ we proved } P \textbf{ done}]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$(\Sigma, \Sigma', (tl, l) \mapsto \Pi(j :: tl, l))$

$\mathcal{C}[\![\textbf{by } j \textbf{ we proved } P_1 \textbf{ that is equivalent to } P_2 \textbf{ done}]\!](\Gamma \vdash P_2 :: \Sigma, \Sigma', \Pi) =$
$(\Sigma, \Sigma', (tl, l) \mapsto \Pi((j : P_1 \equiv P_2)) :: tl, l))$

$\mathcal{C}[\![\textbf{by } j \textbf{ done}]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) = (\Sigma, \Sigma', (tl, l) \mapsto \Pi(j :: tl, l))$

$\mathcal{C}[\![\textbf{by } j \textbf{ let } x : T \textbf{ such that } P_1\ (H)]\!](\Gamma \vdash P_2 :: \Sigma, \Sigma', \Pi) =$
$((\Gamma\ ;\ x : T\ ;\ H : P_1 \vdash P_2) :: \Sigma, \Sigma',$
$(hd :: tl, l) \mapsto \Pi((\text{ex\_elim}_{T, P_1, P_2} j\ (\lambda x : T.\Lambda H : P_1.hd))) :: tl, l))$

$\mathcal{C}[\![\textbf{by } j \textbf{ we have } P_1\ (H_1) \textbf{ and } P_2\ (H_2)]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$((\Gamma\ ;\ H_1 : P_1\ ;\ H_2 : P_2 \vdash P) :: \Sigma, \Sigma',$
$(hd :: tl, l) \mapsto \Pi((\text{and\_elim}_{P_1, P_2, P}\ j\ (\Lambda H_1 : P_1.\Lambda H_2 : P_2.hd))) :: tl, l))$

$\mathcal{C}[\![\textbf{we need to prove } P]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$((\Gamma \vdash P) :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi((hd) :: tl, l))$

$\mathcal{C}[\![\textbf{we need to prove } P_1 \textbf{ or equivalently } P_2]\!](\Gamma \vdash P_1 :: \Sigma, \Sigma', \Pi) =$
$((\Gamma \vdash P_2) :: \Sigma, \Sigma', (hd :: tl, l) \mapsto \Pi((hd : P_2 \equiv P_1)) :: tl, l))$

$\mathcal{C}[\![\textbf{we need to prove } P_1\ (H)]\!](\Gamma \vdash P_2 :: \Sigma, \Sigma', \Pi) =$
$((\Gamma \vdash P_1) :: (\Gamma\ ;\ H : P_1 \vdash P_2) :: \Sigma, \Sigma',$
$(hd_1 :: hd_2 :: tl, l) \mapsto \Pi((\text{Let } H : P_1 := hd_1 \text{ in } hd_2) :: tl, l))$

$\mathcal{C}[\![\textbf{we need to prove } P_1\ (H)\ \textbf{ or equivalently } P_2]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$((\Gamma \vdash P_2) :: (\Gamma\ ;\ H : P_1 \vdash P) :: \Sigma, \Sigma',$
$(hd_1 :: hd_2 :: tl, l) \mapsto \Pi((\text{Let } H : P_1 := (hd_1 : P_2 \equiv P_1) \text{ in } hd_2) :: tl, l))$

$\mathcal{C}[\![\textbf{conclude } E_1 = E_2 \textbf{ by } j]\!](\Gamma \vdash E_1 = E_3 :: \Sigma, \Sigma', \Pi) =$
$((\Gamma \vdash E_2 = E_3) :: \Sigma, \Sigma',$
$(hd :: tl, l) \mapsto \Pi(((\text{eq\_transitive } E_1\ E_2\ E_3\ j\ hd) :: tl, l)))$

$\mathcal{C}[\![\textbf{conclude } E_1 = E_2 \textbf{ by } j \textbf{ done}]\!](\Gamma \vdash E_1 = E_2 :: \Sigma, \Sigma', \Pi) =$
$(\Sigma, \Sigma', (tl, l) \mapsto \Pi(j :: tl, l))$

$\mathcal{C}[\![\textbf{obtain } H\ E_1 = E_2 \textbf{ by } j]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$((\Gamma \vdash E_2 = ?) :: (\Gamma\ ;\ H : E_1 = ? \vdash P) :: \Sigma, (\Gamma \vdash \text{nat}) :: \Sigma',$
$(hd_1 :: hd_2 :: tl, hd' :: tl') \mapsto$
$\Pi((\text{Let } H : E_1 = hd' := (\text{eq\_transitive } E_1\ E_2\ hd'\ j\ hd_1) \text{ in } \ hd_2) :: tl, tl'))$

$\mathcal{C}[\![\textbf{obtain } H\ E_1 = E_2 \textbf{ by } j \textbf{ done}]\!](\Gamma \vdash P :: \Sigma, \Sigma', \Pi) =$
$((\Gamma\ ;\ H : E_1 = E_2 \vdash P) :: \Sigma, \Sigma',$
$(hd :: tl, l) \mapsto \Pi((\text{Let } H : E_1 = E_2 := j \text{ in } hd) :: tl, l))$

$\mathcal{C}[\![= E_2 \textbf{ by } j \textbf{ done}]\!](\Gamma \vdash E_1 = ? :: \Sigma, (\Gamma \vdash \text{nat}) :: \Sigma', \Pi) =$
$(\Sigma, \Sigma', (tl, l) \mapsto \Pi(j :: tl, E_2 :: l))$

$\mathcal{C}[\![= E_2 \textbf{ by } j \textbf{ done}]\!](\Gamma \vdash E_1 = E_2 :: \Sigma, \Sigma', \Pi) = (\Sigma, \Sigma', (tl, l) \mapsto \Pi(j :: tl, l))$

$\mathcal{C}[\![\textbf{we proceed by induction on } n \textbf{ to prove } P(n)]\!](\Gamma \vdash P(n) :: \Sigma, \Sigma', \Pi) =$

$((\Gamma \vdash P(O)) :: (\Gamma \vdash \forall m : \mathrm{nat}.P(m) \Rightarrow P(S(m))) :: \Sigma, \Sigma',$
$\quad (hd_1 :: hd_2 :: l, l') \mapsto \Pi((\mathrm{nat\_ind}_P\ n\ hd_1\ hd_2) :: l, l'))$
$\mathcal{C}[\![\textbf{we proceed by cases on } n \textbf{ to prove } P(n)]\!](\Gamma \vdash P(n) :: \Sigma, \Sigma', \Pi) =$
$\quad ((\Gamma \vdash P(O)) :: (\Gamma \vdash \forall m : \mathrm{nat}.P(S(m))) :: \Sigma, \Sigma',$
$\quad (hd_1 :: hd_2 :: l, l') \mapsto \Pi((\mathrm{nat\_cases}_P\ n\ hd_1\ hd_2) :: l, l'))$
$\mathcal{C}[\![\textbf{case } H\ arg_1 \cdots arg_n]\!] = \quad \mathcal{C}[\![arg_1]\!]_a\ \cdots \mathcal{C}[\![arg_n]\!]_a$
$\mathcal{C}[\![(x : T)]\!]_a = \mathcal{C}[\![\textbf{assume } x : T]\!]$
$\mathcal{C}[\![(H : P)]\!]_a = \mathcal{C}[\![\textbf{suppose } P\ (H)]\!]$
$\mathcal{C}[\![\textbf{by induction hypothesis we know } P\ (H)]\!] = \mathcal{C}[\![\textbf{suppose } P\ (H)]\!]$
$\mathcal{C}[\![\textbf{by induction hypothesis we know } P_1\ (H) \textbf{ that is equivalent to } P_2]\!] =$
$\quad \mathcal{C}[\![\textbf{suppose } P_1\ (H) \textbf{ that is equivalent to } P_2]\!]$
$\mathcal{C}[\![\textbf{the thesis becomes } P]\!] = \mathcal{C}[\![\textbf{we need to prove } P]\!]$
$\mathcal{C}[\![\textbf{the thesis becomes } P_1 \textbf{ or equivalently } P_2]\!] =$
$\quad \mathcal{C}[\![\textbf{we need to prove } P_1 \textbf{ or equivalently } P_2]\!]$

5 Natural Language Generation

We present in Table 5 the inverse translation $\mathcal{G}[\![-]\!]$ from proof terms to declarative proof scripts. The translation is recursive and proceeds by pattern matching over the proof term. Rules coming first take precedence.

Recursion on equality chains is performed by the auxiliary function $\mathcal{G}[\![-]\!]^{=}_{-}$ where the argument in subscript position is used to remember the right hand side of the last step in the chain.

Table 5: Natural language generation

$\mathcal{G}[\![\lambda x : T.t]\!] = \textbf{assume } x : T\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![\Lambda H : P.t]\!] = \textbf{suppose } P\ (H)\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![let\ x := E\ in\ t]\!] = \textbf{let } x := E\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![(\lambda x : T_2.t : \forall x : T_2.P \equiv \forall x : T_1.P)]\!] =$
$\quad \textbf{assume } x : T_1 \textbf{ that is equivalent to } T_2\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![(\Lambda H : P_2.t : P_2 \Rightarrow P \equiv P_1 \Rightarrow P)]\!] =$
$\quad \textbf{suppose } P_1\ (H) \textbf{ that is equivalent to } P_2\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![(\lambda x : T_2.t : \forall x : T_2.P_2 \equiv \forall x : T_1.P_1)]\!] =$
$\quad \textbf{assume } x : T_1 \textbf{ that is equivalent to } T_2$
$\quad \textbf{we need to prove } P_1 \textbf{ or equivalently } P_2\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![(\Lambda H : P_2.t : P_2 \Rightarrow P_4 \equiv P_1 \Rightarrow P_3)]\!] =$
$\quad \textbf{suppose } P_1\ (H) \textbf{ that is equivalent to } P_2$
$\quad \textbf{we need to prove } P_3 \textbf{ or equivalently } P_4\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![Let\ K : P := (H\ E_1 \ldots E_n\ H_1 \ldots H_m)\ in\ t]\!] =$
$\quad \textbf{by } (H\ E_1 \ldots E_n\ H_1 \ldots H_m) \textbf{ we proved } P\ (K)\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![Let\ K : P_2 := ((H\ E_1 \ldots E_n\ H_1 \ldots H_m) : P_1 \equiv P_2)\ in\ t]\!] =$
$\quad \textbf{by } (H\ E_1 \ldots E_n\ H_1 \ldots H_m) \textbf{ we proved } P_1\ (K)$
$\quad \textbf{that is equivalent to } P_2\ \mathcal{G}[\![t]\!]$
$\mathcal{G}[\![Let\ H : P_2 := (t_1 : P_1 \equiv P_2)\ in\ t_2]\!] =$

we need to prove $P_2\ (H)$ **or equivalently** $P_1\ \mathcal{G}[\![t_1]\!]\ \mathcal{G}[\![t_2]\!]$

$\mathcal{G}[\![(\text{eq\_transitive}\ E'_1\ E'_2\ E'_3\ (H\ E_1 \dots E_n\ H_1 \dots H_m)\ t_{2,3})]\!] =$

conclude $E'_1 = E'_2$ by $(H\ E_1 \dots E_n\ H_1 \dots H_m)\ \mathcal{G}[\![t_{2,3}]\!]^{=}_{E'_3}$

$\mathcal{G}[\![Let\ H : E'_1 = E'_3 :=$

$(\text{eq\_transitive}\ E'_1\ E'_2\ E'_3\ (H\ E_1 \dots E_n\ H_1 \dots H_m)\ t_{2,3})\ in\ t]\!] =$

obtain $H\ E'_1 = E'_2$ **by** $(H\ E_1 \dots E_n\ H_1 \dots H_m)\mathcal{G}[\![t_{2,3}]\!]^{=}_{E'_3}$

$\mathcal{G}[\![Let\ H : P := t_1\ in\ t_2]\!] =$ **we need to prove** $P\ (H)\ \mathcal{G}[\![t_1]\!]\ \mathcal{G}[\![t_2]\!]$

$\mathcal{G}[\![(H\ E_1 \dots E_n\ H_1 \dots H_n)]\!]^{=}_{E'} =$

$= E'$ **by** $(H\ E_1 \dots E_n\ H_1 \dots H_n)$ **done**

$\mathcal{G}[\![(\text{eq\_transitive}\ E'_1\ E'_2\ E'_3\ (H\ E_1 \dots E_n\ H_1 \dots H_m)\ t_{2,3})]\!]^{=}_{E'_3} =$

$= E_2$ **by** $(H\ E_1 \dots E_n\ H_1 \dots H_m)\mathcal{G}[\![\ t_{2,3}]\!]^{=}_{E'_3}$

$\mathcal{G}[\![(\text{ex\_elim}\ t\ P_1\ P_2\ H\ (\lambda x : T.\Lambda H_2 : P_2.t))]\!] =$

by H **let** $x : T$ **such that** $P_2\ (H_2)\ \mathcal{G}[\![t]\!]$

$\mathcal{G}[\![(\text{and\_elim}\ P_1\ P_2\ P_3\ H\ (\Lambda H_1 : P_1.\Lambda H_2 : P_2.t))]\!] =$

by H **we have** $P_1\ (H_1)$ **and** $P_2\ (H_2)\ \mathcal{G}[\![t]\!]$

$\mathcal{G}[\![(\text{nat\_ind}_P\ n\ t_1\ (\lambda m : \text{nat}.\Lambda H : P(m).t_2))]\!] =$

we proceed by induction on n **to prove** $P(n)$

case O

the thesis becomes $P(O)$

$\mathcal{G}[\![t_1]\!]$

case $S\ (m : nat)$

by induction hypothesis we know $P(m)\ (H)$

the thesis becomes $P(S(m))$

$\mathcal{G}[\![t_2]\!]$

$\mathcal{G}[\![(\text{nat\_ind}_P\ n\ t_1\ (\lambda m : \text{nat}.$

$(\Lambda H : P(m).t_2 : P_2 \Rightarrow P(S(m)) \equiv P(m) \Rightarrow P(S(m)))))]\!] =$

we proceed by induction on n **to prove** $P(n)$

case O

the thesis becomes $P(O)$

$\mathcal{G}[\![t_1]\!]$

case $S\ (m : nat)$

by induction hypothesis we know $P(m)\ (H)$

that is equivalent to P_2

the thesis becomes $P(S(m))$

$\mathcal{G}[\![t_2]\!]$

$\mathcal{G}[\![(\text{nat\_ind}_P\ n\ t_1\ (\lambda m : \text{nat}.$

$(\Lambda H : P_2.t_2 : P(m) \Rightarrow P_3 \equiv P(m) \Rightarrow P(S(m)))))]\!] =$

we proceed by induction on n **to prove** $P(n)$

case O

the thesis becomes $P(O)$

$\mathcal{G}[\![t_1]\!]$

case $S\ (m : nat)$

by induction hypothesis we know $P(m)\ (H)$

that is equivalent to P_2

the thesis becomes $P(S(m))$ **or equivalently** P_3

$\mathcal{G}[\![t_2]\!]$

$\mathcal{G}[\![(\text{nat\_ind}_P\ n\ t_1\ (\lambda m : \text{nat}.\Lambda H : P(m).(t_2 : P_2 \equiv P(S(m)))))]\!] =$

we proceed by induction on n **to prove** $P(n)$

case O

the thesis becomes $P(O)$

$\mathcal{G}[\![t_1]\!]$

case $S\ (m : nat)$

by induction hypothesis we know $P(m)\ (H)$

the thesis becomes $P(S(m))$ **or equivalently** P_2

$\mathcal{G}[\![t_2]\!]$

$\mathcal{G}[\![((H\ E_1 \ldots E_n\ H_1 \ldots H_m) : P_1 \equiv P_2)]\!] =$

by $(H\ E_1 \ldots E_n\ H_1 \ldots H_m)$ **we proved** P_1

that is equivalent to P_2 **done**

$\mathcal{G}[\![(H\ E_1 \ldots E_n\ H_1 \ldots H_m)]\!] =$ **by** $(H\ E_1 \ldots E_n\ H_1 \ldots H_m)$ **done**

The following important theorem shows that the proof term generated processing a declarative script generated from a given proof term is identical to the starting proof term. Thus, we fully satisfy the strongest requirement of Section 2 about re-generation of declarative scripts.

Theorem 1 (Round-tripping from proof terms).
$\forall \Gamma, \forall P, \forall t$ *such that* $\Gamma \vdash t : P$, $\forall \Sigma, \Sigma', \Pi$ *we have*

$\mathcal{C}[\![\mathcal{G}[\![t]\!], P]\!]_s((\Gamma \vdash P) :: \Sigma, \Sigma', \Pi) = (\Sigma, \Sigma'', \Pi')$

and

either $\Sigma' = \Sigma''$ *and* $\forall l, l', \Pi(t :: l, l') = \Pi'(l, l')$

or $\Sigma' = (\Gamma' \vdash \text{nat}) :: \Sigma''$ *and* $\exists E, \forall l, l', \Pi(t :: l, E :: l') = \Pi'(l, l')$.

The two branches of the statement deserve an explanation. They differ in whether a placeholder for expressions can be closed. This is the case only if $\mathcal{G}[\![t]\!]$ syntactically contains the last step of a rewriting chain and if one placeholder occurs in Π (and, consequently, Σ' is not empty). The existentially quantified placeholder instantiation E is proved to be the right hand side of the last step in the equality chain.

Proof. The proof is by structural induction on t. We only show one significant case.

Let t be $\lambda x : T_2.t' : \forall x : T_2.P_2 \equiv \forall x : T_1.P_1$. We have $\mathcal{G}[\![t]\!] = S_1\ S_2\ \mathcal{G}[\![t']\!]$ where $S_1 =$ "**assume** $x : T_1$ **that is equivalent to** T_2" and $S_2 =$ "**we need to prove** P_1 **or equivalently** P_2". Assume generic, but fixed Σ, Σ', Π. We have

$$
\begin{aligned}
&\mathcal{C}[\![S_1\ S_2\ \mathcal{G}[\![t']\!]]\!]^\star((\Gamma \vdash \forall x : T_1.P_1) :: \Sigma, \Sigma', \Pi) \\
&= (\mathcal{C}[\![\mathcal{G}[\![t']\!]]\!] \circ \mathcal{C}[\![S_2]\!] \circ \mathcal{C}[\![\text{“ }\textbf{assume } x : T_1 \textbf{ that is equivalentto } T_2''\text{]}\!]) \\
&\quad ((\Gamma \vdash \forall x : T_1.P_1) :: \Sigma, \Sigma', \Pi) \\
&= \mathcal{C}[\![\mathcal{G}[\![t']\!]]\!](\mathcal{C}[\![S_2]\!]((\Gamma\ ;\ x : T_2 \vdash P_1) :: \Sigma, \Sigma', \\
&\quad \begin{cases} ((hd : P_2 \equiv P_1) :: tl, l) \mapsto \Pi((\lambda x : T_2.hd : \forall x : T_2.P_2 \equiv \forall x : T_1.P_1) :: tl, l) \\ (hd :: tl, l) \qquad\qquad\quad \mapsto \Pi((\lambda x : T_2.hd : \forall x : T_2.P_1 \equiv \forall x : T_1.P_1) :: tl, l) \end{cases}))
\end{aligned}
$$

$= \mathcal{C}[\![\mathcal{G}[\![t']\!]]\!](((\Gamma\ ;\ x : T_2 \vdash P_2) :: \Sigma, \Sigma',$
$\quad (hd :: tl, l) \mapsto \Pi((\lambda x : T_2.hd : \forall x : T_2.P_2 \equiv \forall x : T_1.P_1) :: tl, l)))$
$= (\Sigma, \Sigma', (l, l') \mapsto \Pi((\lambda x : T_2.t' : \forall x : T_2.P_2 \equiv \forall x : T_1.P_1) :: l, l')$

The last identity is justified by the inductive hypothesis on t'.

Thus $\Sigma'' = \Sigma'$ and we have to prove the "either" part of the thesis i.e. $\forall l, l', \Pi((\lambda x : T_2.t' : \forall x : T_2.P_2 \equiv \forall x : T_1.P_1) :: l, l') = \Pi((\lambda x : T_2.t' : \forall x : T_2.P_2 \equiv \forall x : T_1.P_1) :: l, l')$, which is trivially true. □

The next theorem shows that all the requirements about re-generation of declarative scripts of Section 2 are fulfilled: the declarative script re-generated from a proof term is an improved version of the starting declarative script. Moreover re-generation is idempotent. Improvement is captured by the relation $\preceq$ (not formally described here due to space constraints). It consists in:

1) interpolating new statements corresponding to the explicitation of justifications previously found automatically or given by means of a proof term more complex than an application. For instance

$$\text{“}\textbf{by}\ \Lambda H : A.H\ \textbf{done}\text{”} \preceq \text{“}\textbf{suppose}\ A\ (H)\text{”}\ \ \text{“}\textbf{by}\ H\ \textbf{done}\text{”}$$

2) replacing statements with other statements that are more appropriate with respect to the context and have the same semantics. For instance the formal semantics of Table 4 shows that **the thesis becomes** P is equivalent to **we need to prove** P. However the former is supposed to be used only to state the thesis of a branch in a proof by induction or case analysis. The relation $\preceq$ also captures the notion of "being less appropriate than". For instance

$$\begin{array}{r} \text{“}\textbf{conclude}\ E_1 = E_2\ \textbf{by}\ j_1\text{”}\ \ \text{“}\textbf{by}\ j_2\ \textbf{done}\text{”} \\ \preceq \text{“}\textbf{conclude}\ E_1 = E_2\ \textbf{by}\ j_1\text{”}\ \ \text{“} = E_3\ \textbf{by}\ j_2\ \textbf{done}\text{”} \end{array} \quad (1)$$

since, once a chain of inequation is started, the same style must be used until the end of the chain.

Note that the relation $\preceq$ is not an order relation since it is reflexive only on scripts that cannot be improved (i.e. only on scripts that have reached the fixpoint).

Lemma 1 (Idempotence of improvement).
$\forall S_1, \ldots, S_n, S'_1, \ldots S'_m, S''_1, \ldots S''_{m'}$, *if* $S_1\ \cdots\ S_n \preceq S'_1\ \cdots\ S'_m \preceq S''_1\ \cdots\ S''_{m'}$ *then* $m = m'$ *and* $\forall i \leq m, S'_i = S''_i$

Theorem 2 (Round-tripping from declarative scripts).
$\forall S_1, \ldots, S_n, \forall \Sigma, \Sigma', \Pi$, *if* $\mathcal{C}[\![S_1\ \cdots\ S_n]\!]^\star(\Sigma, \Sigma', \Pi) = (\Sigma_1, \Sigma'_1, \Pi')$ *where* $\Sigma_1 = \Sigma_2@\Sigma$ *(*Σ_2 *of length* k*) then* $\exists! C$ *such that*
either $\Sigma' = \Sigma'_1$ *and* $\forall l, l', l_1, l'_1, \Pi(C[l_1, l'_1] :: l, l') = \Pi'(l_1@l, l'_1@l')$
or $\Sigma' = E :: \Sigma'_1$ *and* $\forall l, l', l_1, l'_1, \Pi(C[l_1, l'_1] :: l, E :: l') = \Pi'(l_1@l, l'_1@l')$
and $\forall t_1, \ldots, t_k, \forall E_1, \ldots, E_h$
if $\mathcal{G}[\![C[t_1, \ldots, t_k, E_1, \ldots, E_h]]\!] = S'_1 \cdots S'_m$ *then*
$S_1\ \cdots\ S_n\ \mathcal{G}[\![t_1]\!]\ \cdots\ \mathcal{G}[\![t_k]\!] \preceq S'_1\ \cdots\ S'_m$.

The second branch in the statement correspond to the case where one S_i is "**obtain** H $E_1 = E_2$" and the equality chain is not terminated in $S_1, \ldots, S_n$. In this case Σ' will contain a placeholder whereas Σ was empty.

Proof. The proof is by induction on n and then by structural induction on S_1. We only show one simple, but significant case (since it shows an improvement of the script).

Let $n = 2, S_1 =$ "**conclude** $E_1 = E_2$ **by** j_1", $S_2 =$ "**by** j_2 **done**".

$$\begin{aligned}
&\mathcal{C}[\![S_1\ S_2]\!]^\star((\Gamma \vdash E_1 = E_3) :: \Sigma, \Sigma', \Pi) \\
&= (\mathcal{C}[\![S_2]\!] \circ \mathcal{C}[\![S_1]\!])((\Gamma \vdash E_1 = E_3) :: \Sigma, \Sigma', \Pi) \\
&= \mathcal{C}[\![S_2]\!](\mathcal{C}[\![S_1]\!]((\Gamma \vdash E_1 = E_3) :: \Sigma, \Sigma', \Pi)) \\
&= \mathcal{C}[\![S_2]\!]((\Gamma \vdash E_2 = E_3) :: \Sigma, \Sigma', \\
&\qquad (hd :: tl, l) \mapsto \Pi(((\text{eq_transitive } E_1\ E_2\ E_3\ j_1\ hd) :: tl, l))) \\
&= (\Sigma, \Sigma', (l, l') \mapsto \Pi(((\text{eq_transitive } E_1\ E_2\ E_3\ j_1\ j_2) :: l, l'))
\end{aligned}$$

Since $\Sigma_1 = \Sigma$ and $\Sigma'_1 = \Sigma'$ we have $k = 0$ and $C[] =$ (eq_transitive E_1 E_2 E_3 j_1 j_2). Hence we need to prove: $\mathcal{G}[\![C[]]\!] = S'_1\ \ldots\ S'_m$ for some m and $S_1\ S_2 \preceq S'_1\ \ldots\ S'_m$. Now

$$\begin{aligned}
&\mathcal{G}[\![C[]]\!] \\
&= \mathcal{G}[\![(\text{eq_transitive } E_1\ E_2\ E_3\ j_1\ j_2)]\!] \\
&= \text{"}\mathbf{conclude}\ E_1 = E_2\ \mathbf{by} j_1\text{"}\ \mathcal{G}[\![j_2]\!]^{=}_{E_3} \\
&= \text{"}\mathbf{conclude}\ E_1 = E_2\ \mathbf{by} j_1\text{"}\ \text{" } = E_3\ \mathbf{by}\ j_2\ \mathbf{done}\text{"}
\end{aligned}$$

Hence the thesis by (1). □

6 Conclusions

In this paper we study the compilation of declarative scripts into proof terms, and the opposite translation of proof terms into declarative scripts. The study is done on the declarative language of the Matita interactive theorem prover (which lacks an elaborated sub-language for justifications) and on proof terms for a sub-calculus of the Calculus of (Co)Inductive Constructions used in Matita. The actual implementation in Matita already considers a larger calculus that comprises, for instance, fully general inductive types.

We observe that the translation from declarative scripts to declarative scripts via proof terms respects the initial script structure and can even improve it by fixing misuses of statements. Moreover this (double) translation is idempotent. It is an open question whether the same results can be achieved for more complex declarative languages whose statements could alter partial proof terms in a non structural way. Our understanding is that this is the case at least for the proof language presented in [4].

Exportation of formalised results between proof assistants having the same proof terms but different high level proof languages is an immediate application

of our technique. Another obvious application is the translation of procedural scripts into executable declarative scripts for their use in education. This way it is possible to present to students or mathematicians, which only understand the declarative language, proofs found in the procedural style.

The work must be completed by designing the (mostly orthogonal) language for proof step justifications and by extending the translations proposed to cover the full language. Whether we will be able to be as close to the mathematical vernacular as Isar is, retaining automatic generation of declarative scripts with the current properties, is another open question.

References

1. Andrea Asperti, Iris Loeb, and Claudio Sacerdoti Coen. Stylesheets to intermediate representation and presentational stylesheets. MoWGLI Report D2d,D2f, 2003.
2. Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 2007. Special Issue on User Interfaces for Theorem Proving. To appear.
3. Serge Autexier and Claudio Sacerdoti Coen. A formal correspondence between omdoc with alternative proofs and the lambda-bar-mu-mu-tilde-calculus. In *Proceedings of Mathematical Knowledge Management 2006*, volume 4108 of *Lectures Notes in Artificial Intelligence*, pages 67–81. Springer-Verlag, 2006.
4. Pierre Corbineau. A declarative proof language for the Coq proof assistant. Poster at Bricks midterm Simposium.
5. Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 233–243, New York, NY, USA, 2000. ACM Press.
6. Ferruccio Guidi. Procedural representation of CIC proof terms. Submitted to the Programming Languages for Mechanized Mathematics Workshop.
7. John Harrison. A Mizar Mode for HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *LNCS*, pages 203–220. Springer-Verlag, 1996.
8. Florent Kirchner and Claudio Sacerdoti Coen. The Fellowship super-prover. `http://www.lix.polytechnique.fr/Labo/Florent.Kirchner/fellowship/`.
9. Claudio Sacerdoti Coen. Tactics in modern proof-assistants: the bad habit of overkilling. In *Supplementary Proceedings of the 14th International Conference TPHOLS 2001*, pages 352–367, 2001.
10. Claudio Sacerdoti Coen. Explanation in natural language of lambda-bar-mu-mu-tilde-terms. In Andrea Asperti, Bruno Buchberger, and James H. Davenport, editors, *Post-Proceedings of the Fourth International Conference on Mathematical Knowledge Management, MKM 2005*, volume 3863 of *Lecture Notes in Computer Science*, pages 234–249. Springer-Verlag, 2006.
11. Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Tinycals: step by step tacticals. In *Proceedings of User Interface for Theorem Provers 2006*, Electronic Notes in Theoretical Computer Science. Elsevier Science, 2006. To appear.
12. The Coq Development Team. The Coq proof assistant reference manual, 2005.
13. Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *Theorem Proving in Higher Order Logics*, pages 167–184, 1999.
14. Markus Wenzel and Freek Wiedijk. A comparison of mizar and isar. *J. Autom. Reasoning*, 29(3-4):389–411, 2002.

Mei – A Module System for Mechanized Mathematics Systems

Jian Xu

Department of Computing and Software, McMaster University
1280 Main Street West, Hamilton, Ontario, Canada L8S 4L7
xuj5@mcmaster.ca
June 15, 2007

Abstract. This paper presents a module system named Mei designed for mechanized mathematics systems. Mei is a λ-calculus style module system that supports higher-order functors in a natural way. The semantics of functor application is based on substitution. A novel coercion mechanism integrates a parameter passing mechanism based on theory interpretations with simple λ-calculus style higher-order functors.

1 Introduction

A mechanized mathematics system (MMS) is a computer system that supports and improves mathematical processing, by which we mean building mathematical models of the real world, and formulating and reasoning about properties of the real world within the context of the mathematical models. Examples of MMSs are theorem provers and computer algebra systems. An MMS needs to have a powerful library in order to be practical. A powerful library for an MMS should (1) contain sufficient mathematical knowledge to support mathematical activities and (2) be well organized so that new knowledge can be easily developed from existing knowledge. Currently, more and more mathematical knowledge is formalized in different systems, which largely achieves the first goal. Thus, a good module system is necessary for MMSs in order to organize this mathematical knowledge. A good modularity mechanism aids the expressivity of the MMSs. It helps to build up contexts and allows the user to reuse the theorems developed within one context in other contexts with similar structures. However, the development of the module systems for MMSs significantly lags behind the development of underlying logics, proof strategies etc. Very few MMSs have a sophisticated module system which supports the development of large pieces of mathematical knowledge.

Mei[1] is a module system designed for MMSs. There are two *chief* goals we want Mei to achieve: (1) Mei should be applicable to arbitrary logics. Although most module systems are associated with some particular logic, we believe, module systems should be orthogonal to their underlying logics. As a result, Mei is a

[1] Mei is the Chinese name of *Prunus mume*, a species of Asian plum in the family *Rosaceae*. The tree flowers in late winter, typically late January or February.

layer above the underlying logic. Otherwise the module system may put too much constraint on the underlying logic. (2) Mei should integrate higher-order functors with theory interpretation [Far94] based parameter passing mechanisms, e.g. fitting morphisms[2] [Ore99] in specification languages. We can find higher-order functors with type matching as the parameter passing mechanism in many functional languages. On the other hand, fitting morphisms with first-order parametric mechanisms are widely used in specification languages. However, no system supports both of them.

Mei has a core system called Mei Core. As a module system designed for MMSs, Mei Core supports the following features:

(1) Mathematical knowledge is organized as a collection of theories. Informally, a theory consists of a *language*, a set of *axioms*, and a set of *theorems with their proofs*.
(2) Operations that build new theories from existing theories: (a) extending a theory by adding new language and axioms, (b) combining a set of existing theories to produce a new theory, and (c) making a copy of an existing theory by renaming its symbols.
(3) Parameterized theories called functors. A functor is a generic theory with respect to its parameter. It can be instantiated by providing a concrete parameter theory to generate a new theory. In particular, functors can be higher-order in the sense that they are functions over parameterized theories.
(4) Theory interfaces. An interface specifies the information of a theory that is exposed to the outside world, such as the language, axioms, and theorems.
(5) A subtyping relation over (parameterized) theories, which enhances the flexibility of parameter passing.

The syntax of Mei Core resembles that of typed λ-calculus with additional syntax for the theory operations.

Mei is Mei Core plus a coercion mechanism. Mei supports a theory interpretation based parameter passing mechanism called a *view*. Given a view, Mei constructs a corresponding coercion functor of Mei Core. The semantics of Mei is defined in terms of that of Mei Core by substituting the generated coercion functors for the views.

The remainder of this paper is organized as follows. §2 and §3 present the module systems Mei Core and Mei respectively. In §4, we show the power of Mei by comparing it with the ML-style module systems, the module systems of some specification languages (Maude, CASL, Specware) and MMSs (IMPS, PVS, Isabelle, Coq). The paper ends with a short summary in §5.

2 A Module System with Subtyping – Mei Core

We first informally present the features of Mei Core, followed by a formal presentation of its syntax (typing rules) and semantics (evaluation rules).

[2] A fitting morphism is a special theory interpretation that is one-one and maps a symbol in the source theory to a symbol (not an arbitrary expression) in the target theory.

2.1 Informal Presentation

As indicated in the introduction, Mei Core is a module system enjoying the beauty of typed λ-calculus, in which theories are the base objects and parameterized theories are the functions. As a result, the semantics of Mei Core is defined via the simple notion of substitution.

Theories A *theory*[3] in Mei Core consists of a language L, a set Φ of axioms, and a set Δ of theorems provable from the axioms in a sound proof system. In other words, each theory in Mei Core is a concrete representation of some mathematical theory.

A *language* L is a set of symbols of various categories that we can use to build syntactic objects. The set of all sentences over a language L, denoted by $\boldsymbol{Sen}(L)$, is a set of syntactic objects constructed inductively via a set of sentence constructors. The set of axioms is a set of sentences, i.e. $\Phi \subseteq \boldsymbol{Sen}(L)$. Each MMS supports a set of proof strategies to derive sentences from sentences using certain inference rules. A sentence φ is *provable* from Φ if it is derivable from Φ. A proof of φ is then any derivation of φ from Φ. A *theorem* is a pair of a sentence and its proof. A set of theorems is denoted by Δ. The set of sentences of Δ is denoted by $\mathbf{sen}(\Delta)$.

For instance, a language L for a many-sorted first-order logic can be a tuple $(\mathcal{S}, \mathcal{C}, \mathcal{F}, \mathcal{P})$ of sets of *sort symbols*, *constant symbols*, *function symbols*, and *predicate symbols* respectively. Terms and formulas can be defined in the usual way. A sentence is then a closed formula. An example of a proof system for it would be a Hilbert-style proof system. A proof of a sentence is a derivation of the sentence starting from the logical and non-logical axioms and then applying the inference rules such as the modus ponens rule.

A *theory* T is a *closed* tuple (L, Φ, Δ), in the sense that the symbols used in Φ and Δ must be in L and the axioms used in the proofs of Δ must be in Φ.

Remark 1. Our notion of a theory is based on the *little theories* approach [FGT92], in contrast to the modules in most programming language paradigms which are based on the *big theory* approach [FGT92]. In the big theory approach, one powerful set of axioms is used to model all objects of interest. Consequently, modules are a name scope mechanism, where an object developed in one module can be referred to from within another module by qualifying the object name. In the little theories approach, reasoning is carried out under different contexts, modeled by theories. To use an object developed in another theory under the current context, we need to build a theory interpretation between them. Effectively, it imports a translated version of that theory implicitly or explicitly.

Theory Extension, Renaming, and Union Mei Core supports several operations to construct new theories from those existing theories.

[3] A theory in Mei contains only those theorems that are already proved, in contrast to approaches where a theory contains all the theorems that are provable.

Extension. To develop a new theory, instead of starting from scratch, we can start from an existing theory and extend it by adding new language symbols and axioms. For instance, to build a theory of groups, we might rather start by inheriting the language and axioms of a theory of monoids.
Renaming. To make a new copy of an existing theory by *renaming* the symbols of the given theory. We need it for two reasons: (1) to avoid naming conflicts for use in the union operation below, and (2) to name the symbols in a meaningful way according to the intended semantics.
Union. To combine two or more theories, we need the *union* operation. For instance, a natural way to build a theory of rings is to extend the union of a theory of groups and a theory of monoids.

Remark 2. There are two approaches to solve the name conflict issue for large system developments: (1) Names from different modules are distinct. Sharing of names is expressed by equation constraints. (2) Name from different modules are not distinct. Sharing is expressed by names. Renaming is used to solve unintended name conflict. In our little theories approach a theory provides a simple name scope, though it cannot be nested. Since we have the renaming mechanism, we prefer (2) within a theory, i.e. the same name refers to the same object. As shown in the following example, the same name is used to represent the carrier sets of the group theory and the monoid theory to force them to be identical in the ring theory. Renaming is used to distinguish their respective binary operators as the addition and the multiplication operators in the ring theory.

Example 1 illustrates the use of theories and theory operations of Mei Core. Although we do not define the syntax for the underlying logic, the meaning of the example should be clear. For the syntax of Mei Core, please refer to §2.2.

Example 1.

$$
\begin{aligned}
\mathit{Monoid} \equiv\ & \texttt{language sort } mm \\
& \qquad \texttt{const } e : mm \\
& \qquad \texttt{opr } \ \circ : mm^2 \to mm \\
& \texttt{axioms } \ \forall x_1, x_2, x_3 : mm.\ x_1 \circ (x_2 \circ x_3) = (x_1 \circ x_2) \circ x_3 \\
& \qquad \forall x : mm.\ x \circ e = x \wedge e \circ x = x \\
& \texttt{theorems } \ \ldots \\
\mathit{Group} \equiv\ & \mathit{Monoid} \texttt{ with } \{mm \mapsto gg\} \texttt{ extended by} \\
& \texttt{language opr } \ -1 : gg \to gg \\
& \texttt{axioms } \ \forall x : gg.\ x \circ x^{-1} = e = x^{-1} \circ x \\
& \texttt{theorems } \ \ldots \\
\mathit{Ring} \equiv\ & (\mathit{Group} \texttt{ with } \{gg \mapsto rr,\ e \mapsto 0, \circ \mapsto +\} \ \oplus \\
& \mathit{Monoid} \texttt{ with } \{mm \mapsto rr,\ e \mapsto 1, \circ \mapsto *\}) \texttt{ extended by} \\
& \texttt{language} \\
& \texttt{axioms } \ \forall x, y, z : rr.\ x * (y + z) = (x * y) + (x * z) \ \ldots \\
& \texttt{theorems } \ \ldots
\end{aligned}
$$

Parameterized Theories A parameterized theory is a generic theory with respect to its formal parameter theory (the argument type). It can be instantiated provided that the actual parameter theory matches the argument type. The new theory is built from both the body specification and the actual parameter theory. We use the name *functor* for a parameterized theory. The instantiation of functors is based on the simple notion of substitution.

Types of Theories. We use the notion of a *type* of a theory to specify the class of theories that can be used to instantiate a functor. A *type* T of a theory $Thy \equiv (L, \Phi, \Delta)$ is a pair $(L_{\mathsf{T}}, \Phi_{\mathsf{T}})$ such that (a) $L_{\mathsf{T}} \subseteq L$, and (b) $\Phi_{\mathsf{T}} \subseteq (\Phi \cup \mathbf{sen}(\Delta))$. A type of a theory is also used to specify an interface of the theory, i.e. the exported language and sentences. Casting the type of a theory effectively changes its interface. Note that not the entire language and not all facts (axioms and theorems) are necessarily exported. For example, the auxiliary theorems (lemmas) of a theory should be hidden and removed from its type.

Higher-Order Functors. Our module system also supports *higher-order functors*, in the sense that functors can be used as parameters as well as return values of other functors. This provides more flexible reusability of theories. The semantics of the instantiation of higher-order functors is based on substitution as for first-order functors, except that functors can be used in substitutions. As a result, we have a notion of a *type of a functor* to specify the class of functor parameters or the class of resulting functors.

Example 2. We show two theory types, Group and Set, and a functor, *GroupAct*, representing a generic group action theory in this example.

$$
\begin{aligned}
\mathsf{Group} \equiv\ & \texttt{language sort } gg \\
& \texttt{const } e : gg \\
& \texttt{opr } \circ : gg^2 \to gg \\
& \quad -1 : gg \to gg \\
& \texttt{axioms } \forall x_1, x_2, x_3 : gg.\ x_1 \circ (x_2 \circ x_3) = (x_1 \circ x_2) \circ x_3 \\
& \quad \forall x : gg.\ x \circ e = x \wedge e \circ x = x \\
& \quad \forall x : gg.\ x \circ x^{-1} = e = x^{-1} \circ x \\
\mathsf{Set} \equiv\ & \texttt{language sort } ss \\
& \texttt{axioms } \ldots \\
\mathit{GroupAct} \equiv\ & \texttt{functor } \mathsf{X} : \mathsf{Group}.\ \texttt{functor } \mathsf{Y} : \mathsf{Set}.\ (\mathsf{X} \oplus \mathsf{Y})\ \texttt{extended by} \\
& \texttt{language opr } act : gg \times ss \to ss. \\
& \texttt{axioms } \forall x : ss.\ act(e, x) = x \\
& \quad \forall g, h : gg, x : ss.\ act(g \circ h, x) = act(g, act(h, x))
\end{aligned}
$$

GroupAct is a higher-order functor since it can be partially instantiated to return a functor as output. The type of *GroupAct* is trivial from the syntax defined in §2.2. The instantiation of functors will be illustrated in Section §2.2 Example 3.

Subtyping Subtyping is a mechanism that allows the user to treat an object of one type (subtype) as an object of another type (supertype). In general, the subtype is more informative than the supertype. Thus, it is safe to use an object of the subtype in the places where an object of the supertype is required. Let *Group* be a theory of groups of type Group. Let Monoid be the type of theories of monoids. Let F be a functor whose formal argument is of type Monoid. It is thus reasonable to allow the application of F to *Group*, since Group is a subtype of Monoid. This idea is generalized to functor types in Mei Core.

It is easy to put our system into the λ-calculus frame. The theory types and the functor types are the base types and function types respectively. Module expressions are terms, where functor abstractions are λ-abstractions, and functor applications are function applications, whose semantics is given by β-reduction. The theory operations are analogous to extra term constructors, like the arithmetic operators in many functional languages.

2.2 Syntax of Mei Core

The following is the concrete syntax of Mei Core. A module expression represents a theory (or functor), denoted by a sans-serif E with subscripts when necessary. It describes the way a theory (or functor) is constructed. A module type represents a class of theories (or functors), denoted by a sans-serif T. We will use "expression" and "type" for "module expression" and "module type" when there is no possibility of confusion.

Rules for Types. $\mathbf{type}(\mathsf{T})$ is read as "T is a module type". $\mathbf{closed}(L, \Phi)$ asserts that all symbols used in Φ are in L.

$$\frac{\mathsf{T} \equiv (L, \Phi) \quad \mathbf{closed}(L, \Phi)}{\mathbf{type}(\mathsf{T})} \qquad \frac{\mathbf{type}(\mathsf{T}_1) \quad \mathbf{type}(\mathsf{T}_2)}{\mathbf{type}(\mathsf{T}_1 \to \mathsf{T}_2)}$$

The functor type is associated to the right, i.e. $\mathsf{T}_1 \to \mathsf{T}_2 \to \mathsf{T}_3$ means $\mathsf{T}_1 \to (\mathsf{T}_2 \to \mathsf{T}_3)$.

Subtyping Rules. $\mathsf{T}_1 <: \mathsf{T}_2$ is read as "T_1 is a subtype of T_2".

$$\frac{L_2 \subseteq L_1 \quad \Phi_2 \subseteq \Phi_1}{(L_1, \Phi_1) <: (L_2, \Phi_2)} \qquad \frac{\mathsf{T}_{\mathsf{s}_2} <: \mathsf{T}_{\mathsf{s}_1} \quad \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{t}_2}}{\mathsf{T}_{\mathsf{s}_1} \to \mathsf{T}_{\mathsf{t}_1} <: \mathsf{T}_{\mathsf{s}_2} \to \mathsf{T}_{\mathsf{t}_2}}$$

Rules for Typing Module Expressions. $\Gamma \vdash \mathsf{E} : \mathsf{T}$ is read as "E is an expression of type T under context Γ". Γ is a *context* that binds variables to types. We assume that previously defined module expressions can be named for later reference. σ is an *environment* that maps names, called constants, to expressions. We use a sans-serif X and Y for the variables and sans-serif C for the constants. $\mathbf{map}(\rho)$ asserts that ρ is a mapping, i.e. a set of symbol pairs, where $\mathbf{source}(\rho)/\mathbf{target}(\rho)$ is the domain/range of ρ. $\rho(L)$ is the image of L via ρ, and $\rho(\Phi)$ is the translation of Φ in which symbols in L are replaced by their images in $\rho(L)$.

$$\frac{\mathsf{X} : \mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{X} : \mathsf{T}} \quad \textbf{(variable)} \qquad \frac{\sigma(\mathsf{C}) = \mathsf{E} \quad \Gamma \vdash \mathsf{E} : \mathsf{T}}{\Gamma \vdash \mathsf{C} : \mathsf{T}} \quad \textbf{(constant)}$$

$$\frac{L_{\mathsf{T}} \subseteq L \quad \Phi_{\mathsf{T}} \subseteq (\Phi \cup \mathbf{sen}(\Delta)) \quad \mathbf{closed}(L, \Phi, \Delta)}{\vdash (L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta) : (L_{\mathsf{T}}, \Phi_{\mathsf{T}})} \quad \textbf{(theory definition)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L_{\mathsf{E}}, \Phi_{\mathsf{E}}) \quad (L_{\mathsf{E}}, \Phi_{\mathsf{E}}) <: (L, \Phi)}{\Gamma \vdash (L, \Phi)\ \mathsf{E} : (L, \Phi)} \quad \textbf{(casting)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L_{\mathsf{E}}, \Phi_{\mathsf{E}}) \quad \mathbf{closed}(L_{\mathsf{E}} \cup L, \Phi_{\mathsf{E}} \cup \Phi \cup \mathbf{sen}(\Delta))}{\Gamma \vdash \mathsf{E}\ \texttt{extended by}\ (L,\ \Phi,\ \Delta) : (L_{\mathsf{E}} \cup L, \Phi_{\mathsf{E}} \cup \Phi \cup \mathbf{sen}(\Delta))} \quad \textbf{(extension)}$$

$$\frac{\Gamma \vdash \mathsf{E}_1 : (L_1, \Phi_1) \quad \Gamma \vdash \mathsf{E}_2 : (L_2, \Phi_2)}{\Gamma \vdash \mathsf{E}_1\ \oplus\ \mathsf{E}_2 : (L_1 \cup L_2, \Phi_1 \cup \Phi_2)} \quad \textbf{(union)}$$

$$\frac{\Gamma \vdash \mathsf{E} : (L, \Phi) \quad \mathbf{map}(\rho) \quad \mathbf{source}(\rho) = L}{\Gamma \vdash \mathsf{E}\ \texttt{with}\ \rho : (\rho(L), \rho(\Phi))} \quad \textbf{(renaming)}$$

$$\frac{\Gamma,\ \mathsf{X} : \mathsf{T}_1 \vdash \mathsf{E}_2 : \mathsf{T}_2}{\Gamma \vdash (\texttt{functor}\ \mathsf{X} : \mathsf{T}_1.\ \mathsf{E}_2) : \mathsf{T}_1 \to \mathsf{T}_2} \quad \textbf{(functor abstraction)}$$

$$\frac{\Gamma \vdash \mathsf{E}_{\mathsf{f}} : \mathsf{T}_1 \to \mathsf{T}_2 \quad \Gamma \vdash \mathsf{E}_{\mathsf{p}} : \mathsf{T}_{\mathsf{p}} \quad \mathsf{T}_{\mathsf{p}} <: \mathsf{T}_1}{\Gamma \vdash (\mathsf{E}_{\mathsf{f}}\ \mathsf{E}_{\mathsf{p}}) : \mathsf{T}_2} \quad \textbf{(functor application)}$$

We will drop parentheses whenever there is no loss of meaning.

2.3 Semantics

Substitution of Module Expression The semantics of Mei Core is based on the *substitution function*, $\mathsf{E}[\mathsf{X} := \mathsf{E}_{\mathsf{p}}]$, defined as follows.

$$\begin{aligned}
\mathsf{Y}[\mathsf{X} := \mathsf{E}_{\mathsf{p}}] &= \begin{cases} \mathsf{E}_{\mathsf{p}} \text{ if } \mathsf{X} \equiv \mathsf{Y} \\ \mathsf{Y} \text{ otherwise.} \end{cases} \\
\mathsf{C}[\mathsf{X} := \mathsf{E}_{\mathsf{p}}] &= \mathsf{C} \\
(L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta)[\mathsf{X} := \mathsf{E}_{\mathsf{p}}] &= (L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta) \\
((L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ \mathsf{E})[\mathsf{X} := \mathsf{E}_{\mathsf{p}}] &= (L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ \mathsf{E}[\mathsf{X} := \mathsf{E}_{\mathsf{p}}] \\
(\mathsf{E}\ \texttt{extended by}\ \mathsf{S})[\mathsf{X} := \mathsf{E}_{\mathsf{p}}] &= \mathsf{E}[\mathsf{X} := \mathsf{E}_{\mathsf{p}}]\ \texttt{extended by}\ \mathsf{S} \\
(\mathsf{E}_1\ \oplus\ \mathsf{E}_2)[\mathsf{X} := \mathsf{E}_{\mathsf{p}}] &= \mathsf{E}_1[\mathsf{X} := \mathsf{E}_{\mathsf{p}}]\ \oplus\ \mathsf{E}_2[\mathsf{X} := \mathsf{E}_{\mathsf{p}}] \\
(\mathsf{E}\ \texttt{with}\ \rho)[\mathsf{X} := \mathsf{E}_{\mathsf{p}}] &= \mathsf{E}[\mathsf{X} := \mathsf{E}_{\mathsf{p}}]\ \texttt{with}\ \rho \\
(\texttt{functor}\ \mathsf{Y} : \mathsf{T}_1.\ \mathsf{E}_2)[\mathsf{X} := \mathsf{E}_{\mathsf{p}}] &= \begin{cases} \texttt{functor}\ \mathsf{Y} : \mathsf{T}_1.\ \mathsf{E}_2[\mathsf{X} := \mathsf{E}_{\mathsf{p}}] & \text{if } \mathsf{Y} \not\equiv \mathsf{X} \\ \texttt{functor}\ \mathsf{Y} : \mathsf{T}_1.\ \mathsf{E}_2 & \text{otherwise.} \end{cases} \\
(\mathsf{E}_1\ \mathsf{E}_2)[\mathsf{X} := \mathsf{E}_{\mathsf{p}}] &= (\mathsf{E}_1[\mathsf{X} := \mathsf{E}_{\mathsf{p}}])\ (\mathsf{E}_2[\mathsf{X} := \mathsf{E}_{\mathsf{p}}])
\end{aligned}$$

The most important cases are the variable case and the second subcase of the functor abstraction case. They are the base cases for the inductive definition.

Operational Semantics Following the normal operational semantics of λ-calculus, we define the operational semantics of Mei Core via the evaluation rules, which transfer an expression to another preserving its type. The evaluation eventually *converges* in the sense that it terminates in finitely many steps at a *normal form*: expression that is a theory definition or a functor abstraction.

Definition 1. *$\uplus$ is an amalgamation union operator with respect to the types. Let $(L_{T_1}, \Phi_{T_1})\ (L_1,\ \Phi_1,\ \Delta_1)$, $(L_{T_2}, \Phi_{T_2})\ (L_2,\ \Phi_2,\ \Delta_2)$ be two module expressions, $L_1 \uplus L_2 = L_{T_1} \cup L_{T_2} \cup (L_1 \backslash L_{T_1} \sqcup L_2 \backslash L_{T_2})$, where $\sqcup$ is the disjoint union.*

The full definition of $\uplus$ can be found in [Xu07]. The idea is that two symbols are considered identical in the union if they are in the type specifications, and are considered distinct otherwise. For example, let Mult be a type declaring a sort *ele* and a binary operator $\circ$, *Monoid* is a theory of monoids with additional constant e. According to the evaluation rule (*) below, the language of (Mult) *Monoid* $\oplus$ (Mult) *Monoid* contains one *ele*, one $\circ$, but two copies of e, since e is not in Mult. The two e's have to be systematically or heuristically renamed to be distinct.

$$\frac{}{\mathsf{C} \longmapsto \sigma(\mathsf{C})} \qquad \frac{}{\mathsf{T}\ (\mathsf{T}'\ (L,\ \Phi,\ \Delta)) \longmapsto \mathsf{T}\ (L,\ \Phi,\ \Delta)} \qquad \frac{\mathsf{E} \longmapsto \mathsf{E}'}{\mathsf{T}\ \mathsf{E} \longmapsto \mathsf{T}\ \mathsf{E}'}$$

$$((L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ (L, \Phi, \Delta))\ \texttt{extended by}\ (L_{\mathsf{S}}, \Phi_{\mathsf{S}}, \Delta_{\mathsf{S}}) \longmapsto (L_{\mathsf{T}} \cup L_{\mathsf{S}}, \Phi_{\mathsf{T}} \cup \Phi_{\mathsf{S}} \cup \mathbf{sen}(\Delta_{\mathsf{S}}))(L \uplus L_{\mathsf{S}}, \Phi \uplus \Phi_{\mathsf{S}}, \Delta \uplus \Delta_{\mathsf{S}})$$

$$(L_{\mathsf{T}_1}, \Phi_{\mathsf{T}_1})\ (L_1, \Phi_1, \Delta_1) \oplus (L_{\mathsf{T}_2}, \Phi_{\mathsf{T}_2})\ (L_2, \Phi_2, \Delta_2) \longmapsto (L_{\mathsf{T}_1} \cup L_{\mathsf{T}_2}, \Phi_{\mathsf{T}_1} \cup \Phi_{\mathsf{T}_2})\ (L_1 \uplus L_2, \Phi_1 \uplus \Phi_2, \Delta_1 \uplus \Delta_2) \qquad (*)$$

$$((L_{\mathsf{T}}, \Phi_{\mathsf{T}})\ (L,\ \Phi,\ \Delta))\ \texttt{with}\ \rho \longmapsto (\rho(L_{\mathsf{T}}), \rho(\Phi_{\mathsf{T}}))\ (\rho(L), \rho(\Phi), \rho(\Delta))$$

$$\frac{\mathsf{E} \longmapsto \mathsf{E}'}{\mathsf{E}\ \texttt{extended by}\ \mathsf{S} \longmapsto \mathsf{E}'\ \texttt{extended by}\ \mathsf{S}}$$

$$\frac{\mathsf{E}_2 \longmapsto \mathsf{E}_2'}{(L_{\mathsf{T}_1}, \Phi_{\mathsf{T}_1})\ (L_1,\ \Phi_1,\ \Delta_1)\ \oplus\ \mathsf{E}_2 \longmapsto (L_{\mathsf{T}_1}, \Phi_{\mathsf{T}_1})\ (L_1,\ \Phi_1,\ \Delta_1)\ \oplus\ \mathsf{E}_2'}$$

$$\frac{\mathsf{E}_1 \longmapsto \mathsf{E}_1'}{\mathsf{E}_1\ \oplus\ \mathsf{E}_2 \longmapsto \mathsf{E}_1'\ \oplus\ \mathsf{E}_2} \qquad \frac{\mathsf{E} \longmapsto \mathsf{E}'}{\mathsf{E}\ \texttt{with}\ \rho \longmapsto \mathsf{E}'\ \texttt{with}\ \rho}$$

$$\frac{}{(\texttt{functor}\ \mathsf{X} : \mathsf{T}_1.\ \mathsf{E}_2)\ \mathsf{E}_{\mathsf{p}} \longmapsto \mathsf{E}_2[\mathsf{X} := \mathsf{E}_{\mathsf{p}}]}\ (**) \qquad \frac{\mathsf{E}_{\mathsf{f}} \longmapsto \mathsf{E}_{\mathsf{f}}'}{\mathsf{E}_{\mathsf{f}}\ \mathsf{E}_{\mathsf{p}} \longmapsto \mathsf{E}_{\mathsf{f}}'\ \mathsf{E}_{\mathsf{p}}}$$

Note that $(**)$ is β-reduction. To avoid variable capture, it is only applicable if E_{p} is free for X in E_2. The order of evaluation is similar to the normal order reduction in λ-calculus.

Theorem 1. *If E is a well-typed module expression, then either E is a theory definition or a functor abstraction or else there is an E' such that $\mathsf{E} \longmapsto \mathsf{E}'$.*

Theorem 2. *If $\mathsf{E} :\ \mathsf{T}$ and $\mathsf{E} \longmapsto \mathsf{E}'$, then $\mathsf{E}' :\ \mathsf{T}$.*

Theorem 3. *If $\vdash \mathsf{E} :\ \mathsf{T}$, the evaluation of E converges.*

Theorem 1 and 2 shows the soundness of the typing system. The interested readers can refer to [Xu07] for the proofs.

3 A Module System with Subtyping and Coercion – Mei

The subtyping mechanism allows us to regard an object of one type as an object of its supertype. However, it is limited: (1) Theories specified in different languages are not related to each other. For instance, a theory of monoids formalized in a language other than that of type Monoid is not of type Monoid. (2) Two different axiomatizations of the same mathematical theory may not be related. For instance, let Nat be a theory of natural numbers with the zero element and the successor function, and let NatNat be another theory of natural numbers with the constants zero, one, ..., and the functions plus, minus, times. Though they should be equivalent, neither is a subtype of the other in Mei Core. (3) One type is regarded as a subtype of another only in one way since the subtyping relation is based purely on syntax. However, a theory of rings can be treated as a theory of monoids in two ways because there are two copies of monoids with respect to the addition and multiplication residing in a ring structure. We introduce a new mechanism, *coercion*, to solve these issues.

3.1 Syntax of Mei

The syntax and rules of Mei extend those of Mei Core by adding a new syntactic class, *views*, and a new kind of expression, functor applications with views. This subsection presents only the new syntactic rules of Mei.

View rules. The notion of a *view* is a generalization of both a theory interpretation (a view can be defined over functor types) and a subtyping relation (a symbol mapping is involved in a view). A view from $\mathsf{T_s}$ to $\mathsf{T_t}$ shows how an object in $\mathsf{T_t}$ can be treated as an object in $\mathsf{T_s}$. The simplest views are the views between theory types, which are *theory translations* that map the expressions of one theory (the source) to the expressions of the other theory (the target). Syntactically, it is a triple $(\mathsf{T_s}, \mathsf{T_t}, \rho)$, in which $\mathsf{T_s}$ and $\mathsf{T_t}$ are the source and target theory types respectively, and ρ is a *mapping* that maps symbols in $\mathsf{T_s}$ to those in $\mathsf{T_t}$. Views between functor types are defined in terms of the views between their source and target types inductively:

$$\frac{\mathbf{source}(\rho) = L_{\mathsf{s}} \quad \mathbf{target}(\rho) \subseteq L_{\mathsf{t}}}{\mathbf{view}((L_{\mathsf{s}}, \Phi_{\mathsf{s}}), (L_{\mathsf{t}}, \Phi_{\mathsf{t}}), \rho)} \qquad \frac{\mathbf{view}(\mathsf{T}_{\mathsf{t}_1}, \mathsf{T}_{\mathsf{s}_1}, \rho_1) \quad \mathbf{view}(\mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_{\mathsf{t}_2}, \rho_2)}{\mathbf{view}(\mathsf{T}_{\mathsf{s}_1} \to \mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_{\mathsf{t}_1} \to \mathsf{T}_{\mathsf{t}_2}, (\rho_1, \rho_2))}$$

Remark 3. Syntactically, a theory view is not necessarily a theory interpretation, i.e. meaning preserving. However, a *good* theory view should be a theory interpretation, but this is not enforced by Mei. The underlying MMS system is responsible for guaranteeing that a view is indeed an interpretation, i.e. for providing the proofs for the obligations. In other words, the view rules are syntactically decidable.

Coercion Semantics of Views. Intuitively, if $\mathsf{V} \equiv (\mathsf{T_s}, \mathsf{T_t}, \rho)$ is a view from $\mathsf{T_s}$ to $\mathsf{T_t}$, its semantics, denoted by $[\![\mathsf{V}]\!]_{\mathsf{v}} : \mathsf{T_t} \to \mathsf{T_{tt}}$, is a functor in Mei Core, called

the coercion functor, that converts every object in $\mathsf{T_t}$ to an object in $\mathsf{T_{tt}}$, where $\mathsf{T_{tt}} <: \mathsf{T_s}$. Therefore, when we want to use an object $\mathsf{E} : \mathsf{T_t}$ in the place where an object of $\mathsf{T_s}$ is required, we can instead use the coerced object, $[\![\mathsf{V}]\!]_\mathsf{v}\ (\mathsf{E})$.

(1) **Theory view.** $\mathsf{V} \equiv ((L_\mathsf{s}, \Phi_\mathsf{s}), (L_\mathsf{t}, \Phi_\mathsf{t}), \rho)$.

$$[\![\mathsf{V}]\!]_\mathsf{v} = \texttt{functor}\ \mathsf{X} : (L_\mathsf{t}, \Phi_\mathsf{t}).\ (\mathsf{X}\ \texttt{extended by}\ (L_\phi,\ \Phi_\phi,\ \Delta_\mathsf{obl}))\ \texttt{with}\ \mathsf{lift}(\rho^{-1}),$$

where L_ϕ is the empty language, Φ_ϕ is the empty axiom set, Δ_obl is the set of proof obligations generated by V, and lift is a function that adds identity maps for symbols not in L_s to the given mapping. Intuitively, the coercion functor adds to the input theory the information expressed by $\mathsf{T_s}$ which is not expressed by $\mathsf{T_t}$, and translates the theory into the language of $\mathsf{T_s}$.

(2) **Functor view.** $\mathsf{V} \equiv (\mathsf{T}_{\mathsf{s}_1} \to \mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_{\mathsf{t}_1} \to \mathsf{T}_{\mathsf{t}_2}, (\rho_1, \rho_2))$. Let $c_1 = [\![\mathsf{T}_{\mathsf{t}_1}, \mathsf{T}_{\mathsf{s}_1}, \rho_1]\!]_\mathsf{v}$ and $c_2 = [\![\mathsf{T}_{\mathsf{s}_2}, \mathsf{T}_{\mathsf{t}_2}, \rho_2]\!]_\mathsf{v}$.

$$[\![\mathsf{V}]\!]_\mathsf{v} = \texttt{functor}\ \mathsf{F} : \mathsf{T}_{\mathsf{t}_1} \to \mathsf{T}_{\mathsf{t}_2}.\ \texttt{functor}\ \mathsf{X} : \mathsf{T}_{\mathsf{s}_1}.\ c_2\ (\mathsf{F}\ (c_1\ \mathsf{X}))$$

Remark 4. For the sake of simplicity, this paper assumes that a theory view is a fitting morphism (see Footnote 2). The views without this restriction are properly handled in [Xu07].

Rules for Typing Module Expressions.

$$\frac{\Gamma \vdash \mathsf{E_f} : \mathsf{T}_1 \to \mathsf{T}_2 \quad \Gamma \vdash \mathsf{E_p} : \mathsf{T_p} \quad \mathbf{view}(\mathsf{T}_1, \mathsf{T_p}, \rho)}{\Gamma \vdash \mathsf{E_f}\ \mathsf{E_p}\ \text{with view}\ (\mathsf{T}_1, \mathsf{T_p}, \rho) : \mathsf{T}_2}$$

Example 3. Let *Ring* be the ring theory, Group be the group theory, and *GroupAct* be the group action functor defined in Example 1 and 2 respectively. We can partially instantiate *GroupAct* by *Ring* with a view as follows.

$$\mathit{GroupAct}\ \mathit{Ring}\ \texttt{with view}\ (\mathsf{Group}, \mathsf{Ring}, \{gg \mapsto rr,\ e \mapsto 0, \circ \mapsto +\})$$

where Ring is the type of *Ring*. The resulting functor is trivial and will not be shown here to save space. The corresponding coercion functor is

$$\texttt{functor}\ \mathsf{X} :\ \mathsf{Ring}.\ \mathsf{X}\ \texttt{with}\ \{rr \mapsto gg,\ 0 \mapsto e, + \mapsto \circ, 1 \mapsto 1, * \mapsto *\}$$

3.2 Semantics of Mei

The semantics of Mei, $[\![]\!]_e$, is defined in terms of the expressions of Mei Core in the sense that we translate every expression of Mei to an expression of Mei Core.

$$\begin{aligned}
[\![\mathsf{X}]\!]_e &= \mathsf{X} \\
[\![\mathsf{C}]\!]_e &= \mathsf{C} \\
[\![(L_\mathsf{T},\ \Phi_\mathsf{T})\ (L,\ \Phi,\ \Delta)]\!]_e &= (L_\mathsf{T},\ \Phi_\mathsf{T})\ (L,\ \Phi,\ \Delta) \\
&\cdots \\
[\![\mathsf{E_f}\ \mathsf{E_p}]\!]_e &= [\![\mathsf{E_f}]\!]_e\ [\![\mathsf{E_p}]\!]_e \qquad (***) \\
[\![\mathsf{E_f}\ \mathsf{E_p}\ \text{with view}\ \mathsf{V}]\!]_e &= [\![\mathsf{E_f}]\!]_e\ ([\![\mathsf{V}]\!]_\mathsf{v}\ [\![\mathsf{E_p}]\!]_e)
\end{aligned}$$

Only the last case is interesting, where the parameter is coerced before it is fed to the functor. We omit some induction cases which are similar to (***).

Theorem 4. *If* $\Gamma \vdash \mathsf{E} : \mathsf{T}$ *in Mei, then* $\Gamma \vdash [\![\mathsf{E}]\!]_e : \mathsf{T}$ *in* Mei Core.

Theorem 5. *If* $\vdash \mathsf{E} : \mathsf{T}$, *the evaluation of* E *converges.*

The interested readers can refer to [Xu07] for the proofs.

4 Comparison with Other Module Systems

We show, in this section, that a very simple module system like Mei can be powerful. We show that many modular mechanisms supported by various systems can be implemented in Mei. In particular, we investigate the modular mechanisms for ML family languages, algebraic specification languages, and MMSs.

4.1 ML-style Module systems

Every ML family language has a module language to specify the structures of large software systems. We compare Mei with one ML-style module system, Leroy's *modular module system* [Ler00]. As in other ML-style module systems, the components defined in the core language are grouped into *structures*; *signatures* are used to specify the interface of structures. They correspond to theories and theory types in Mei. Leroy's system also supports higher-order functors and subtyping. However, renaming and the fitting morphism style parameter passing mechanism are not provided in Leroy's system. Most importantly, Leroy's system is also designed to be language independent as Mei. In fact, the implementation of Mei in [Xu07] is inspired largely by [Ler00]. In addition, modules of Leroy's system also provide a namespace management while in Mei renaming is used to solve the name conflict as stated in Remark 2.

4.2 Algebraic Specification Languages

While most specification languages support the extension, union, and renaming operations as in Mei, they usually have a different parameterized specification mechanism. The body of the parameterized specification is, in general, defined as an extension of the parameter specification(s). Therefore, only one copy of the parameter occurs in the body definition. Parameter passing is via fitting morphisms, another name for theory views. The semantics is defined by pushouts and only first-order parameterized specifications are supported, which can be simulated easily in Mei. All the languages discussed in this section follow this approach. In the following discussion, we will present only those mechanisms that are not supported directly in Mei.

It is worth indicating that R. Jimémez and F. Orejas present a framework in which the body of the parameterized theory is defined as an arbitrary expression over the parameter, including λ-abstraction and application [JO99]. As in Mei, it supports the higher-order parameterized specification with a fitting morphism style parameter passing mechanism. However, its β-reduction is not properly defined, making its semantics not operational, as fully discussed in [Xu07].

Maude. Maude [CDE+04] is a specification language based on a rewriting logic. Modules are the basic building blocks of Maude. A module consists of sorts, operators on these sorts, and equations specifying how they interact. A module in Maude defines the initial algebra specified by its axioms. "Theory" is a notion similar to "module", which has loose semantics [CDE+04]. Maude has two novel parametric mechanisms: parameterized views and parameterized theories.

Parameterized Views A parameterized view is used in the instantiation of a parameterized module to produce a new parameterized module. Let LIST(X :: TRIV) and SET(X::TRIV) be parameterized modules. Then

(view Set(X::TRIV) from TRIV to SET(X) is
  sort Elt to Set(X).
endv)

is a parameterized view named Set. LIST(Set) is a parameterized module of lists of sets of elements and can be instantiated by a view Nat that maps TRIV to NAT.

This can be simulated in Mei as follows. Let $LIST$: TRIV $\rightarrow$ LIST and SET : TRIV $\rightarrow$ SET be functors.

$$LIST\text{-}SET \equiv \texttt{functor}\ \mathsf{X} : \mathsf{TRIV}.\ LIST\ (SET\ \mathsf{X}\ \texttt{with view}\ \mathsf{V}),$$

where V is a view similar to the view Set in Maude. By a little abuse of notation, $LIST\text{-}SET = LIST \circ SET$, the composition of *List* and *Set*. Clearly *LIST-SET* functions in the same way as LIST(Set).

A parameterized view itself can be instantiated by other view(s) in Maude. For instance, Set(Nat) is a view from TRIV to SET(Nat), a set of natural numbers. Effectively, it is a composition of the view Set with a view from SET(X::TRIV) to SET(Nat), which is a view lifted from Nat. It can be simulated by the view composition and view lifting mechanism presented in [Xu07].

Parameterized Theories Parameterized theories are theories that are parameterized by other (possibly parameterized) theories. For instance, we can define LIST(X::SET(Y::TRIV)), where SET(Y::TRIV) is a parameterized theory. Let FINSET(Y::TRIV) be a parameterized module of finite sets. Then

(view FinSet(X::TRIV) from SET(X) to FINSET(X) is
  sort Set(X) to FinSet(X) .
endv)

is a parameterized view named FinSet. LIST(FinSet) is then a parameterized module of lists of finite sets of elements. Note that LIST is not a higher-order parameterized module. The instantiation of LIST is enforced to be a parameterized module composition, since FinSet must be a parameterized view.

Parameterized theories correspond to a type $(\mathsf{T}_1 \rightarrow \mathsf{T}_2)\ \mathsf{T}_1$ that does not exist in Mei. It specifies a class of expressions that have to be derived from functor applications. For instance, let F : $\mathsf{T}_1 \rightarrow \mathsf{T}_2$ and $X : \mathsf{T}_1$, then $(F\ X)$ is of type $(\mathsf{T}_1 \rightarrow \mathsf{T}_2)\ \mathsf{T}_1$. $(F\ X)$ can then be used to instantiate a functor $G : ((\mathsf{T}_1 \rightarrow \mathsf{T}_2)\ \mathsf{T}_1) \rightarrow T_3$. In Maude G is only used to compose with F as in the following context:

$$H = \texttt{functor}\ \mathsf{X} : \mathsf{T}_1.\ G\ (F\ \mathsf{X}) = G \circ F$$

Parameterized theories is also a simple sharing mechanism, e.g. there is one copy of SET in any instantiation of MOD(X::LIST(Z::SET),Y::STACK(Z::SET)), but two copies in one of MOD(X::LIST(Z1::SET),Y::STACK(Z2::SET)).

The new type is not necessary in Mei: (1) Declaring $G : \mathsf{T}_2 \to T_3$ does not prevent us from composing G with F. (2) Sharing is expressed by naming in Mei, i.e. same name refers to the same object within a theory. (3) It complicates the type system. For example, H defined above will be typed $\mathsf{T}_1 \to ((((\mathsf{T}_1 \to \mathsf{T}_2)\ \mathsf{T}_1) \to \mathsf{T}_3)\ ((\mathsf{T}_1 \to \mathsf{T}_2)\ \mathsf{T}_1))$, which is hard to comprehend.

Specware. Specware [SJ95] has a novel module system. It follows closely a category-theoretic approach. The *diagram* and *colimit* operations are the most important modular mechanisms of Specware. Although most algebraic specification languages use the notion of a colimit to define the semantics, very few of them directly support the colimit operation.

A diagram is a directed multigraph where specifications are nodes and morphisms are edges. A colimit of a diagram is a specification constructed by first taking the disjoint union of the node specifications, and then building the quotient from the equivalent relations induced by the morphisms. The composition of specifications is then constructed by building a diagram and calculating its colimit. For instance, the union operation, parameterized specifications and their instantiations can be seen as particular diagrams. The semantics of these operations can be defined as the colimits of the appropriate diagrams.

The current version of Specware supports a notion of substitution, which is similar to parameterized specification. Any sub-specification in the importation hierarchy of a specification can be used as the formal parameter. The formal parameter actually used for instantiation is determined by the fitting morphism, i.e. its source specification. To make it more practical, it is necessary to build a more sophisticated mechanism of parameterized specifications on top of the colimit mechanism. In fact, the parameterized theories and parameterized views in Maude can be translated directly to diagram constructions as shown in [DM00].

In one sense, the colimit operation is more general than the parametric mechanism in Mei, since colimit is defined over arbitrary diagrams that may not be representable in Mei. In another sense, the parametric mechanism in Mei is more general, since it supports higher-order functors, which do not have a counterpart in Specware. In addition, functors in Mei may contain multiple copies of the parameter theory along different morphisms. The semantics of functor instantiations is then akin to the multiple pushouts in [Ore99], which may not be expressed by the colimit operator. One particular drawback of Specware is that a solid category theory background is required in order for users to use it.

Casl. CASL [Gro01] is an algebraic specification language based on partial first-order logic. A basic specification consists of a signature Σ and a set of sentences (axioms or constraints) over Σ. A signature declares a set of S of sorts, sets TF and PF of total function symbols and partial function symbols respectively, and a set P of predicate symbols. Signatures are related by signature morphism:

$(S, TF, PF, P) \rightarrow (S', TF', PF', P')$, consists of a mapping from S to S', and mappings between the corresponding sets of function and predicate symbols respectively [ABK+02]. CASL provides a number of mechanisms for structuring specifications, among which specification reductions, local specifications, and parameterized views deserve mention.

Reductions SP `hide` SL and SP `reveal` SM.
The hiding reduction hides all symbols listed by SL. The revealing reduction is the complement of the hiding reduction, which reveals all symbols specified by SM. Specification reduction hides auxiliary information, which has the same functionality of up-casts in Mei, i.e. casting a theory to its supertype. The supertype effectively show the revealing part. Both reductions can be implemented in terms of up-casts as shown in [Xu07].
Local specification `local` SP_1 `within` SP_2.
It is equivalent to: $\{SP_1$ `then` $SP_2\}$ `hide` SY_1, ..., SY_n. Thus it can be implemented in terms of specification reduction, and in turn up-casts in Mei, as shown in [Xu07].
View `view` VN $[SP_1]$ ... $[SP_n]$ `given` SP''_1, ..., SP''_m : SP `to` $SP' = SM$ `end`
This defines, according to a symbol mapping SM, a morphism from SP to a parameterized specification which has a body SP', a list of parameters SP_1, ..., SP_n, and a set of imports SP''_1, ..., SP''_m. A view can be instantiated with different fitting arguments, giving compositions of the morphism defined by the view with other fitting morphisms. The views in CASL are a restricted version of the parameterized views in Maude, in that (a) the parameter specification occurs only in the target specification, (b) the parameter specification is not parameterized, and (c) the parameterized view has to be instantiated when it is referred. Hence they can be implemented by view compositions and view lifting in Mei, as shown in **Maude**.

4.3 Mechanized Mathematics Systems

We will see in this section that the modular mechanisms of MMSs are weak, making them the right application area for Mei. We will focus on the parametric modular mechanisms in MMSs.

IMPS. IMPS [FGT93] is an interactive theorem prover designed to support mathematical reasoning. Two of its key concepts are little theories [FGT92] and theory interpretation [Far94]. An IMPS theory is constructed from a set of subtheories, a language, and a set of axioms. Theories can be related by the subtheory relation and theory interpretation. The subtheory relation can be expressed by the union and extension operations in Mei.

In IMPS, all theories are generic and defined over arbitrary types. They can be instantiated in a polymorphic manner via an explicitly defined theory interpretation. This is similar to our first-order functor except that: (1) There is no separate notion of parameterized theories. The subtheory used as interface is identified at the time of instantiation by building an interpretation from it

to the actual parameter theory, i.e. the source theory of the interpretation as in Specware. (2) Only one occurrence of a subtheory can be replaced for each instantiation.

PVS. PVS [ORS92] is an environment for specification and verification. A PVS specification is a collection of theories. A PVS theory consists of a theory name, a list of formal parameters, an export part, an assumption part, and a theory body. The theory body is the main part of the theory, consisting of imports, axioms, and theorems. Exporting is a mechanism to hide some names declared in the current theory, and can be simulated by the up-casts in Mei as shown in CASL. The assumption part gives constraints over the current theory, which have to hold for any instance of the theory. Internally, the assumptions are the same as axioms. Externally, they generate obligations which must be proved for each import of the theory [ORS92].

PVS provides a notion of parameterized theories that simulates theory interpretations [OS01]. An explicit theory interpretation mechanism is also supported in PVS [OS01]. What we called parameterized theories is termed as *theory as parameter* in PVS [OS01]. They are roughly first-order functors in Mei. Theory interpretations are used as a parameter passing mechanism as in Mei.

Isabelle. Isabelle [NPW02] is a generic theorem prover which supports user-defined logics. The primitive theory system of Isabelle provides only a theory importation mechanism, which can be expressed in Mei by the union and extension operations. Although L. Paulson [Pau91] proposed and D. Aspinall [Asp91] implemented an ML-style module system for Isabelle, it does not seem to be an official part of the current Isabelle distribution. In addition, Isabelle's meta-logic is a higher-order constructive logic with Π and Σ types in which dependent records can be modeled. Modules can then be represented as dependent records [CPT05]. This will resemble the ML-style module systems with the advantage that modules are first-class citizens. However, no module system based on dependent records is implemented in the current Isabelle distribution.

Isabelle provides two other notions of *axiomatic type classes* [Wen04] and *locales* [Bal04] for modular development of theories. The axiomatic type classes are quite simple and restricted, which resemble Haskell's type classes. Locales is a theory like mechanism encoding theory interpretations. Conceptually, locales can replace Isabelle theories since they possesses all functionalities of theories. Locale expressions provide a more flexible way for combining locales than theory constructions, in particular, locales can be renamed and merged. This is similar to the theory operations in Mei. However, neither of generic locales or generic theories, in the sense of functor, are supported in Isabelle.

Modules in Coq. Coq [Tea04] is a theorem prover based on the Curry-Howard isomorphism. Stating a theorem is writing a well-formed type, and proving this theorem is finding a term whose type corresponds to the theorem in question.

Consequently, proof checking is reduced to type checking. Although Coq's logic is strong enough to formalize a module system as dependently typed records, Jacek Chrzkszcz implements a stratified ML-style module system [Chr04] following Leroy's manifest type approach. The basic module expressions include structure, signature, (high-order) functor, and functor signature. Modules (structures) group together related concepts, and higher-order modules (functors) provide module abstraction.

The module system of Coq is similar to Mei Core, except for its support of translucent types. The major advantage of Mei over Coq's module system is the fitting morphism style parameter passing mechanism, which allows modules defined in different languages to instantiate a functor, as long as the modules share the same structure with the formal parameter of the functor.

5 Summary

We have presented a module system Mei in the style of the λ-calculus. As a result, it supports higher-order functors in a natural way.

The notion of view and its coercion semantics are novel. They allow the user to use a theory interpretation based parameter passing mechanism, while still keeping the simplicity of the λ-calculus. The coercion part of Mei is orthogonal to the module type checking and module expression evaluation of Mei Core, i.e. the module expressions of Mei are coerced to those in Mei Core and then type checked and evaluated. This separation of concerns enables Mei Core to have a simple formalization and implementation, which is the beauty of Mei. In addition, we propose and implement an amalgamation union operator with respect to module types, which solves naming conflicts with reasonable flexibility. Mei makes very weak assumptions over the underlying MMS, making it applicable to most MMSs.

We show that this simple system is powerful, in that many modular mechanisms can be simulated in Mei. Although most of the mechanisms of Mei are used in various module systems, as far as we know Mei is the first module system in which these mechanisms are fully integrated, via the new idea of coercion.

Acknowledgments

The paper is part of the auther's Ph.D. research. The author expresses sincere thanks to Dr. William Farmer for his insight and thoughtful guidance. The author is grateful to Jacques Carette, Tom Maibaum and Jeffery Zucker for many valuable discussions and comments. The author would also like to thank the referees for their suggestions on how to improve the paper.

References

[ABK+02] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: the common algebraic specification language. *Theor. Comput. Sci.*, 286(2):153–196, 2002.

[Asp91] D. R. Aspinall. Isabelle modules – a new theory mechanism for Isabelle. Master's thesis, University of Cambridge, 1991.

[Bal04] C. Ballarin. Locales and locale expressions in Isabelle/Isar. In *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, 2003*, volume 3085 of *LNCS*, pages 34–50. Springer, 2004.

[CDE+04] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude manual*, March 2004. Version 2.1.

[Chr04] J. Chrzkszcz. Modules in Coq are and will be correct. In *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, 2003*, volume 3085 of *LNCS*, pages 130–146. Springer, 2004.

[CPT05] T. Coquand, R. Pollack, and M. Takeyama. A Logical Framework with Dependently Typed Records. *Fundamenta Informaticae*, 65:112–134, 2005.

[DM00] F. Durán and J. Meseguer. Maude's module algebra, 2000.

[Far94] W. M. Farmer. Theory interpretation in simple type theory. In *Higher-Order Algebra, Logic, and Term Rewriting*, volume 816 of *Lecture Notes in Computer Science*, pages 96–123. Springer-Verlag, 1994.

[FGT92] W. M. Farmer, J.D. Guttman, and F. J. Thayer. Little theories. In *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 567–581. Springer-Verlag, 1992.

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

[Gro01] CoFI Language Design Task Group. *The common algebraic specification language (*CASL*) – summary*, March 2001. Version 1.0.1.

[JO99] R. M. Jiménez and F. Orejas. An algebraic framework for higher-order modules. In *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume II*, pages 1778–1797, London, UK, 1999. Springer-Verlag.

[Ler00] X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[Ore99] F. Orejas. Structuring and modularity. In *Algebraic Foundations of System Specification*, pages 159–200. Springer-Verlag, 1999.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, 1992. Springer-Verlag.

[OS01] S. Owre and N. Shankar. Theory interpretations in PVS. Technical Report SRI-CSL-01-01, SRI International, Menlo Park, CA, April 2001.

[Pau91] L. C. Paulson. Theories as ML structures, signatures, and functors. University of Cambridge, January 1991.

[SJ95] Y. V. Srinivas and R. Jullig. Specware: Formal support for composing software. In *Mathematics of Program Construction*, pages 399–422, 1995.

[Tea04] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2004. Version 8.0.

[Wen04] M. Wenzel. Using axiomatic type classes in Isabelle, 2004. Version 8.0.

[Xu07] J. Xu. Mei – A module system for mechanized mathematics systems. Technical Report CAS-07-01-WF, McMaster Univercity, Hamilton, Canada, 2007. Available at http://imps.mcmaster.ca/jianxu/.

Procedural Representation of CIC Proof Terms

Ferruccio Guidi*

Department of Computer Science
Mura Anteo Zamboni 7, 40127 Bologna, ITALY.
fguidi@cs.unibo.it

Abstract. In this paper we propose an effective procedure for translating a proof term of the Calculus of Inductive Constructions (CIC), which is very similar to a program written in a prototypal functional programming language, into a tactical expression of the high-level specification language of a CIC-based proof assistant like COQ [1] or MATITA [2]. As a use case, we report on our implementation of this procedure in MATITA and on the translation of 668 proofs generated by COQ 7.3.1 [3][1], from their logical representation as CIC proof terms to their high-level representation as tactical expressions of MATITA's user interface language.

1 Introduction

Proof assistants are increasingly used to build large-scale digital libraries of formalised mathematical knowledge. In principle, these libraries should allow users to benefit from a large set of basic units of knowledge (*i.e.* definitions, axioms, proofs, etc.) upon which to build up their own developments. However in practice, users often face difficulties in adapting existing formalisations to their needs and start their developments nearly from scratch, leading to duplicate work and unintegrated contributions.

The major difficulties are due to the fact that different proof assistants are very likely to use different knowledge representation formats both at the logical level and at the user interface level.

At the logical level the mathematical notions are expressed in a framework which usually is a type theory or a set theory, while at the user interface level these notions are expressed by a set of instructions (commands or tactics) that drive the proof assistant in the generation of the logical level contents. In some cases these instructions form a programming language.

* Partially supported by the Strategic Project DAMA (Dimostrazione Assistita per la Matematica e l'Apprendimento) of the University of Bologna.

[1] These proofs belong to a formal development of the author on λ-typed λ-calculus [4] that was finished when COQ 7.3.1 was outdated. Any user-level porting of the development to a newer version of COQ up to 8.1 failed or was unsatisfactory, so the author did not submit his proofs to COQ's library and, being a member of the MATITA development group, decided to migrate to MATITA implementing the procedure presented in this paper. We stress that the porting was done at the user level because the work in [4] is still evolving and the proofs need to be maintained accordingly.

Sharing knowledge among different proof assistants at the logical level is usually less difficult since the relationships between various logical frameworks are known and some techniques that facilitate knowledge reuse have been provided [5]. But in actual fact this is not always satisfactory because proof assistants allow to edit an existing piece of knowledge only starting from its user interface representation, so the adaptation and maintenance of this knowledge can not be performed at the logical level. Therefore sharing knowledge at the user interface level appears more desirable although much more difficult because in most cases the instructions used at this level do not have a formal semantics and their effects heavily depend on the execution context and on implementation details.

Nevertheless there is a good chance that a proof assistant can translate a knowledge item from its own logical level to its own user level by using its own interface instructions. So we propose to approach the problem of sharing knowledge at the user interface level by reducing it to the problem of sharing its counterpart at the logical level while leaving to each proof assistant the task of translating this knowledge from one level to the other.

Transforming a knowledge item from the user interface level to the logical level yields no problems apart from the possible loss of the information about the item that is not captured by its logical representation. In that event the logical representation of the item can be annotated [6].

In this paper we consider the opposite transformation in the case of the proof assistants COQ [1] and MATITA [2] when the knowledge item to be translated is a proof. This amounts to compiling a proof term of the Calculus of Inductive Constructions (CIC) [7] in a tactical expression of the user interface language used by these systems. In Section 2 we will overview CIC proof terms and we will discuss the tactical expressions we want to use in the transformation. The transformation itself will be presented in Section 3 and a use case will be outlined in Section 4. Section 5 contains our concluding remarks.

2 CIC Proof Terms and Tactical Expressions

In this section we introduce the domain and the codomain of our transformation, *i.e.* the CIC proof terms and tactical expressions we will use, while providing some related definitions and results.

Throughout the paper we assume the "Barendregt's convention" [8], *i.e.* the names of the bound variables and of the free variables are disjoint. Also notice that we will use the symbol $\equiv$ for definitional equality.

2.1 CIC Proof Terms and Related Definitions

CIC [7] is a powerful typed λ-calculus that, through the "propositions as types and proofs as terms" (PAT) interpretation [8] (*i.e.* the Curry-Howard isomorphism), serves as logical framework for some proof assistants like COQ and MATITA. In the following we recall some concepts about CIC just to set the

notation we will use in the paper (therefore this description of CIC is not meant to be complete). In particular the letters t, u, v, w will denote terms.

Here are some constructions that appear only in the CIC terms used as types:

Definition 1 (some type constructions).

1. *(type of a proposition according to the PAT interpretation)* `Prop`*;*
2. *(type of a local declaration of x of type v in t)* $\Pi x{:}v.t$*.*

The terms of CIC representing proofs are those whose type is of type `Prop` and they are built using the following constructions:

Definition 2 (proof terms).

1. *(reference to a global declaration or definition) c;*
   this includes a reference to a constructor of an (co)inductive type;
2. *(reference to a local declaration or definition) x;*
3. *(local declaration of x of type w in t)* $\lambda x{:}w.t$*;*
4. *(local definition of x as v in t)* $x{\leftarrow}v.t$*;*
5. *(application of the function t to the arguments $v_1 \cdots v_n$)* $(t\ v_1 \cdots v_n)$*;*
   when $n = 0$ it is convenient to set $(t) \equiv t$;
6. *(explicit cast of the type of t to w)* $(t : w)$*;*
7. *(type casted case analysis on v)* $(v \Rightarrow t_1 \cdots t_n : w)$*;*
   v inhabits a (co)inductive type [7] with constructors $c_1 \cdots c_n$ and t_i is the branch taken when v is an instance of c_i. Here the term w is the type of the whole expression.

Local definition by (co)recursion [7] is also available in the calculus but we do not consider it at the moment since it is rarely found inside proofs.

From now on $t[w_1 \cdots w_n/v_1 \cdots v_n]$ denotes the sequential replacement of $v_1 \cdots v_n$ with $w_1 \cdots w_n$ in t, $\Gamma \vdash t_1 \leftrightarrow t_2$ denotes the CIC conversion judgement (*i.e.* t_1 and t_2 are convertible according to the reduction rules of CIC under the premises in Γ, that can be local declarations or definitions of the form $x{:}u$ and $x{\leftarrow}v$ respectively), and $\Gamma \vdash t : u$ denotes the CIC type judgement (*i.e.* the type of t is u according to the type rules of CIC under the premises in Γ). Notice that we are not using CIC terms with meta-variables (*i.e.* placeholders).

2.2 Contents and structure of proofs

In general terms a proof has two main aspects: its contents, *i.e. what* is proved step by step, and its structure, *i.e. how* the proof is developed step by step. Evidently a representation of a proof focused on its structure is less readable for the human user, but appears to be easier to maintain. In fact adapting or maintaining a proof usually amounts to changing some aspects of its contents while trying to preserve its structure. Moreover in the perspective of identifying the common proof patterns occurring in a set of proofs, which can be useful in the design of automated proof procedures, we observe that such patterns are

more likely to concern the structure of these proofs rather than their contents. So we expect that a structure-oriented or *procedural* representation of the proofs would make these patterns more evident. On the contrary if focus our attention on proof readability at the user interface level, a contents-oriented or *declarative* representation of the proof is more desirable and effective especially if natural language rendering is exploited [9–11].

The distinction between the declarative aspect and the procedural aspect of a proof is captured at the logical level by the distinction between the parts of the proof term, *i.e.* the subterms, that denote types and the parts that do not.

2.3 Primitive Tactical Expressions

We recall that in general terms *tactical expressions* are a way of representing proofs at the user interface level. They are evaluated by the system in the context of a conjecture, *i.e.* the statement to prove (the conclusion) under a list of premises (local declarations and definitions) and the effect of the evaluation, *i.e.* the result of the expression, is the (virtual or real) construction of a proof for the conjecture. We also recall that the atomic tactical expressions are termed *tactics* and represent atomic proof steps at the user interface level.

The specification languages of COQ and MATITA include several classes of tactics each differing in the kind of information the user must provide to the system in order to perform the corresponding proof step This information appears as "arguments" (*i.e.* sub-expressions) inside the tactics themselves.

We can classify the tactics on the basis of their arguments as follows:

Definition 3 (classification of tactics according to their arguments).

1. *the* procedural arguments *are fragments of proof terms not including types; the* procedural tactics *are tactics having arguments of this kind;*
2. *the* declarative arguments *are types that, according to the PAT interpretation, may correspond to fragments of the statement to prove; the* declarative tactics *are tactics having at least one argument of this kind;*
3. *the* locative arguments *are pointers to parts of the conjecture (usually to its premises) used to localise the results of the expressions in which they appear; the* locative tactics *are tactics having arguments of this kind;*
4. *other arguments usually appear in* automated tactics*, i.e. tactics whose effect is to build fragments of proofs following some automated decision procedures; such arguments are used to tune these procedures.*

According to what we said in Subsection 2.2, if we represent a proof with declarative tactics, we obtain a view focused on its contents, while if we use procedural tactics, we obtain a view focused on its structure. Therefore in the perspective of adapting and maintaining proofs at the user interface level, we propose in this paper to limit the use of declarative tactics while pursuing the use of procedural tactics.

In the following we present the tactical expressions we will use in the transformation we are discussing, explaining their semantics in terms of their result,

so let $[\![\Gamma \vdash u, U]\!]$ be the CIC proof term produced by the evaluation of the expression U in the context of the conjecture $\Gamma \vdash u$, when the evaluation succeeds.

In the notation below we implicitly use the tactical $U;\ [W_1 \cdots W_n]$ that denotes the generalised sequential composition of $W_1 \cdots W_n$ after U.

Definition 4 (primitive tactical expressions).

1. `intro as` $x;\ [U]$ *builds a local declaration in front of the proof term resulting from the evaluation of* U*; formally we give the following semantics:*
   $[\![\Gamma \vdash \Pi x{:}v.u, \texttt{intro as}\ x;\ [U]]\!] \equiv \lambda x{:}v.[\![\Gamma.x{:}v \vdash u, U]\!]$*;*
   notice that we can expect the system to perform weak head reduction and α*-conversion on the conclusion of the conjecture if needed;*
   also notice that `intro` *is a procedural tactic according to our definition because its argument (i.e.* x*) is part of the constructed proof term;*
2. `pose` v `as` $x;\ [U]$ *builds a local definition in front of the proof term resulting from the evaluation of* U*; formally we give the following semantics:*
   $[\![\Gamma \vdash u, \texttt{pose}\ v\ \texttt{as}\ x;\ [U]]\!] \equiv x{\leftarrow}v.[\![\Gamma.x{\leftarrow}v \vdash u, U]\!]$*;*
   notice that `pose` *is a procedural tactic according to our definition because its arguments (i.e.* x *and* v*) are parts of the constructed proof term;*
3. `cut` w `as` $x;\ [U\ W]$ *is similar to* `pose` v `as` $x;\ [U]$ *but works under some restrictions; formally: if* $\Gamma \vdash w : \texttt{Prop}$ *then* $[\![\Gamma \vdash u, \texttt{cut}\ w\ \texttt{as}\ x;\ [U\ W]]\!] \equiv x{\leftarrow}[\![\Gamma \vdash w, W]\!].[\![\Gamma.x{:}w \vdash u, U]\!]$*;*
   notice that `cut` w `as` x *is a declarative tactic since* w *is a type;*
4. `apply` $t;\ [W_1 \cdots W_n]$ *builds an application in front of the proof terms resulting from the evaluation of* $W_1 \cdots W_n$*; formally we give the following semantics: if* $\Gamma \vdash t : \Pi x_1 \cdots x_n{:}w_1 \cdots w_n.u$ *and* $v_1 \equiv [\![\Gamma \vdash w_1, W_1]\!]$ *and ... and* $v_n \equiv [\![\Gamma \vdash w_n[v_1 \cdots v_{n-1}/x_1 \cdots x_{n-1}], W_n]\!]$ *then*
   $[\![\Gamma \vdash u[v_1 \cdots v_n/x_1 \cdots x_n], \texttt{apply}\ t;\ [W_1 \cdots W_n]]\!] \equiv (t\ v_1 \cdots v_n)$*;*
   notice that `apply` t *is a procedural tactic according to our definition because its argument (i.e.* t*) is part of the constructed proof term; also notice that in practice the system can infer some of the* $v_1 \cdots v_n$ *by unification, so the corresponding expression* W_i *must not be specified in the list* $[W_1 \cdots W_n]$ *that follows* `apply`*; furthermore the expressions in this list are evaluated respecting the order so when each* W_i *is evaluated the terms* $v_1 \cdots v_{i-1}$ *to place in the conclusion of the corresponding conjecture, i.e.* w_i*, are known;*
   Finally for $n = 0$ *we obtain the base case of the induction by which we are defining the tactical expressions: if* $\Gamma \vdash t : u$ *then* $[\![\Gamma \vdash u, \texttt{apply}\ t]\!] \equiv t$*;*
   Notice that in real implementations $\Gamma \vdash t : u$ *implies* $[\![\Gamma \vdash u, \texttt{apply}\ t]\!] \equiv t$ *also when* u *is a function type as an extension of the previous case.*
5. `cases` $v;\ [W_1 \cdots W_n]$ *builds a proof by cases on* v *whose branches result from the evaluation of* $W_1 \cdots W_n$*; formally we give the following semantics: if* v *belongs to an inductive type with constructors* $c_1 \cdots c_n$ *and if* $\Gamma \vdash c_i : w_i$*, then* $[\![\Gamma \vdash u, \texttt{cases}\ v;\ [W_1 \cdots W_n]]\!] \equiv (v \Rightarrow [\![\Gamma \vdash w_1, W_1]\!] \cdots [\![\Gamma \vdash w_n, W_n]\!] : u)$*;*
   notice that `cases` *is a procedural tactic according to our definition;*
6. `change` $u_2;\ [U]$ *changes the conclusion of the conjecture in which* U *is evaluated; formally if* $\Gamma \vdash u_2 \leftrightarrow u_1$ *then* $[\![\Gamma \vdash u_1, \texttt{change}\ u_2;\ [U]]\!] \equiv [\![\Gamma \vdash u_2, U]\!]$*;*
   notice that `change` *is a declarative tactic because its argument* u_2 *is a type;*

7. `change` w_2 `in` x; $[U]$ *changes the type of the local declaration of* x *in the premises of the conjecture in which* U *is evaluated; formally if* $\Gamma_1 \vdash w_2 \leftrightarrow w_1$ *then* $[\![\Gamma_1.x{:}w_1.\Gamma_2 \vdash u, \texttt{change}\ w_2\ \texttt{in}\ x;\ [U]]\!] \equiv [\![\Gamma_1.x{:}w_2.\Gamma_2 \vdash u, U]\!]$*;*
this is the locative version of the `change` *tactic.*
8. *We define* U *as "well-formed" in the context of* $\Gamma \vdash u$ *if* $[\![\Gamma \vdash u, U]\!]$ *exists.*

Some tactics of the above list are declarative. In COQ and MATITA `change` has some procedural counterparts (mainly `simplify` and `unfold`) whose results are not easily predictable, and `cut` can be replaced by `lapply` in some cases.

A result on the tactical expressions defined above is easily provable and explains in exact terms why such expressions represent proofs:

Theorem 1 (soundness).
If U *is well formed in* $\Gamma \vdash u$ *then* $\Gamma \vdash [\![\Gamma \vdash u, U]\!] : u$.

Proof. By induction on the structure of U using the type rules of CIC [1]. □

2.4 Induction Principles and Related Tactical Expressions

A CIC-based proof assistant allows to define data structures by means of inductive types and provides automatically defined lemmas proving forms of structural induction on the elements of such types. These lemmas are termed default induction principles. Induction principles have a particular shape, whose description goes beyond the scope of this paper, and the system provides specialised versions of the `apply` tactic as facilities for invoking these principles in proofs.

In particular we want to mention the following tactical expressions:

Definition 5 (tactical expressions related to induction principles).

1. *if* v *is an element of an inductive type* w *with* n *constructors and* t *is an elimination principle over* w *then* `elim` v `using` t; $[W_1 \cdots W_n]$ *has the semantics of* `apply` t; $[\cdots\ W_1 \cdots W_n\ \cdots\ \texttt{apply}\ v]$*; the extra arguments denoted by* $\cdots$ *are inferred by the system;*
2. `rewrite right` v; $[W]$ *and* `rewrite left` v; $[W]$ *are special cases of the above for two induction principles over the type denoting Leibniz equality;*
3. `elim` v `using` t `in` $y_1 \cdots y_n$ `as` x; $[U]$ *is the locative version of* `elim` *and has the semantics of* `pose` $(t\ \cdots\ y_1 \cdots y_n\ \cdots\ v)$ `as` x; $[U]$*; this represents structural induction applied in a forward reasoning manner;*
4. *the locative version of* `rewrite` *are also provided:*
`rewrite right` v `in` y `as` x; $[U]$ *and* `rewrite left` v `in` y `as` x; $[U]$.

This definition is informal because these tactical expressions are not strictly necessary since in principle they can be expressed in terms of other expressions). Nevertheless we decide to use them in our transformation since the user certainly expects to see them in the resulting description of the proof.

Moreover `elim` and `rewrite`, as implemented in MATITA, may take an argument, representing a CIC term pattern, that is not considered in the above syntax and that is involved in the construction of the resulting term.

3 From CIC Proof Terms to Tactical Expressions

The purpose of the transformation ϵ we are presenting in this section is to construct a tactical expression whose result is a given proof term. Formally speaking, given t such that $\Gamma \vdash t : u$ we seek $\epsilon(\Gamma, t)$ such that $[\![\Gamma \vdash u, \epsilon(\Gamma, t)]\!] = t$. The auxiliary precondition $\Gamma \vdash u : \texttt{Prop}$ ensures that t is really a proof term.

The expression $\epsilon(\Gamma, t)$ is computed in two steps: first we preprocess t with a proof term optimiser returning a term $\omega_1(\Gamma, t)$ such that $\Gamma \vdash \omega_1(\Gamma, t) \leftrightarrow t$, then we seek the tactical expression representing $\omega_1(\Gamma, t)$ using a function π. So in the end ϵ is π after ω_1. The subsections below illustrate these functions in detail.

3.1 CIC Proof Term Optimisation

Proof terms are optimised to improve the quality of the proofs represented at the user interface level in the perspective of an easier maintenance of these proofs.

The optimised version of a proof term t, which we denote by $\omega_1(\Gamma, t)$, is computed by repeatedly applying a number of conversion steps to t until no application is possible. The optimisation works on every sub-term of t but we apply it only to proof terms because normally a conversion performed elsewhere does not improve the the final representation of the proof significantly. Some conversion steps performed by ω_1 appear in [4] as part of a calculus termed $\lambda\delta$.

According to the "direct proof paradigm", *i.e.* the simplest proof strategy coming from standard mathematical practice, the part of a proof carried out by forward reasoning should precede the part of that proof carried out by backward reasoning. At the logical level this means that abbreviations (*i.e.* local definitions) should precede applications in the proof term (recall that the application of a lemma v in a backward reasoning manner is represented by the construction $(v\ \cdots)$ while the same lemma applied in a forward reasoning manner is represented as $x{\leftarrow}(v\ \cdots).t)$. Since in our experience exploiting this paradigm contributes to clarify the structure of the final proof, our optimisation procedure takes care of moving the abbreviations towards the top part of the proof term whenever possible while leaving the applications in the bottom part.

The sub-terms that are certainly shown as procedural arguments after the transformation π are the head of applications (appearing in `apply` expressions), the first argument of proofs by cases (appearing in `cases` expressions) and the last argument of the applications of an inductive principle (appearing in `elim` expressions or the like). In principle these sub-terms can be very complex and verbose so we choose to abbreviate them when they are not atomic (*i.e.* we perform an anticipation of these sub-terms by ζ-expansion [1]).

If t expects m formal parameters then `apply` t; $[W_1 \cdots W_n]$ can be well formed only if $n \leq m$. This means that an application of t represented using `apply` t is better handled if the number of its arguments is less or equal to m. In this paper an application with this property will be termed sober. To ensure that all applications are sober, ω_1 exploits anticipation by ζ-expansion [1].

Formally $\omega_1(\Gamma, t)$ is defined by iterating the following rules until a fixed point is reached. The main properties of ω_1 are listed below.

Definition 6 (the rules for ω_1).

1. *if t is not of sort* `Prop` *in Γ then $\omega_1(\Gamma, t) \equiv t$; else*
2. $\omega_1(\Gamma, c) \equiv c$;
3. $\omega_1(\Gamma, x) \equiv x$;
4. *if $\omega_1(\Gamma.x{\leftarrow}v, t_1) = t_2$ and x does not occur in t_2 then $\omega_1(\Gamma, x{\leftarrow}v.t_1) \equiv t_2$ (ζ-contraction of [4]); else*
5. *if v_1 is of sort* `Prop` *and $\omega_1(\Gamma, v_1) = y{\leftarrow}v.v_2$ then $\omega_1(\Gamma, x{\leftarrow}v_1.t) \equiv y{\leftarrow}v.\omega_1(\Gamma, x{\leftarrow}v_2.t)$; else*
6. *if $\omega_1(\Gamma, v) = (v_0\ \cdots\ v_1 \cdots v_{i-1}\ v_i\ v_{i+1} \cdots v_n\ \cdots)$ where v_0 is an inductive principle and $v_1 \cdots v_n$ are the proofs of the inductive cases, if v_i is not a local reference and if y is fresh in Γ then $\omega_1(\Gamma, x{\leftarrow}v.t) \equiv \omega_1(\Gamma, x{\leftarrow}y{\leftarrow}v_i.(v_0\ \cdots\ v_1 \cdots v_{i-1}\ y\ v_{i+1} \cdots v_n\ \cdots).t)$ (ζ-expansion of [1] or $\delta\zeta$-expansion of [4]); else*
7. $\omega_1(\Gamma, x{\leftarrow}v.t) \equiv x{\leftarrow}\omega_1(\Gamma, v).\omega_1(\Gamma.x{\leftarrow}v, t)$;
8. *if $\omega_1(\Gamma.x{:}w, t_1) = y{\leftarrow}v.t_2$ and x does not occur in v then $\omega_1(\Gamma, \lambda x{:}w.t_1) \equiv y{\leftarrow}v.\omega_1(\Gamma, \lambda x{:}w.t_2)$; else*
9. $\omega_1(\Gamma, \lambda x{:}w.t) \equiv \lambda x{:}\omega_1(\Gamma, w).\omega_1(\Gamma.x{:}w, t)$;
10. *if $\omega_1(\Gamma, t_1) = x{\leftarrow}v.t_2$ then $\omega_1(\Gamma, (t_1\ v_1 \cdots v_n)) \equiv x{\leftarrow}v.\omega_1(\Gamma, (t_2\ v_1 \cdots v_n))$ (υ-swap of [4]); else*
11. *if $\omega_1(\Gamma, t_1) = \lambda x{:}w.t_2$ and t_2 contains at most one free occurrence of x then $\omega_1(\Gamma, (t_1\ v\ v_1 \cdots v_n)) \equiv \omega_1(\Gamma, (t_2[v/x]\ v_1 \cdots v_n))$ (β-contraction of [1]); else*
12. *if $\omega_1(\Gamma, t_1) = \lambda x{:}w.t_2$ then $\omega_1(\Gamma, (t_1\ v\ v_1 \cdots v_n)) \equiv \omega_1(\Gamma, (x{\leftarrow}v.t_2\ v_1 \cdots v_n))$ (β-contraction of [4]); else*
13. *if $\omega_1(\Gamma, t_1) = (t_2\ v'_1 \cdots v'_m)$ then $\omega_1(\Gamma, (t_1\ v_1 \cdots v_n)) \equiv \omega_1(\Gamma, (t_2\ v'_1 \cdots v'_m\ v_1 \cdots v_n))$; else*
14. *if $\omega_1(\Gamma, t_1) = t_2$ and $\Gamma \vdash t_2 : \Pi x_1 \cdots x_m{:}w_1 \cdots w_m.u$ where u is not a function type and $m > 0$, if x is fresh in Γ and $n > 0$, then $\omega_1(\Gamma, (t_1\ v'_1 \cdots v'_m\ v_1 \cdots v_n)) \equiv \omega_1(\Gamma, x{\leftarrow}(t_1\ v'_1 \cdots v'_m).(x\ v_1 \cdots v_n))$ (ζ-expansion of [1] or $\delta\zeta$-expansion of [4]); else*
15. *if $\omega_1(\Gamma, v_i) = x{\leftarrow}v.v'_i$ and v_i is of sort* `Prop` *in Γ then $\omega_1(\Gamma, (t\ v_1 \cdots v_n)) \equiv x{\leftarrow}v.\omega_1(\Gamma, (t\ v_1 \cdots v_{i-1}\ v'_i\ v_{i+1} \cdots v_n))$; else*
16. *if $\omega_1(\Gamma, t_1) = t_2$ and $\Gamma \vdash t_2 : \Pi x_1 \cdots x_m{:}w_1 \cdots w_n.u$ where u is not a function type and $n > 0$, if t_2 is an induction principle and v_n is not atomic and x is fresh in Γ then $\omega_1(\Gamma, (t_1\ v_1 \cdots v_n)) \equiv \omega_1(\Gamma, x{\leftarrow}v_n.(t_1\ v_1 \cdots v_{n-1}\ x))$ (ζ-expansion of [1] or $\delta\zeta$-expansion of [4]); else*
17. $\omega_1(\Gamma, (t\ v_1 \cdots v_n)) \equiv (\omega_1(\Gamma, t)\ \omega_1(\Gamma, v_1) \cdots \omega_1(\Gamma, v_n))$;
18. $\omega_1(\Gamma, (t : v)) \equiv \omega_1(\Gamma, t)$ (ϵ-contraction of [4]);
19. *if $\omega_1(\Gamma, t_1) = x{\leftarrow}v.t_2$ with t_1 of sort* `Prop` *in Γ then $\omega_1(\Gamma, (t_1 \Rightarrow v_1 \cdots v_n : w)) \equiv x{\leftarrow}v.\omega_1(\Gamma, (t_2 \Rightarrow v_1 \cdots v_n : w))$*
20. *if $\omega_1(\Gamma, t_1) = (c_i\ v'_1 \cdots v'_m)$ then $\omega_1(\Gamma, (t_1 \Rightarrow v_1 \cdots v_n : w)) \equiv \omega_1(\Gamma, (v_i\ v'_1 \cdots v'_m))$ (ι-contraction of [1]); else*
21. *if $\omega_1(\Gamma, t_1) = t_2$ and t_2 is not atomic and x is fresh in Γ, then $\omega_1(\Gamma, (t_1 \Rightarrow v_1 \cdots v_n : w)) \equiv x{\leftarrow}t_2.\omega_1(\Gamma, (x \Rightarrow v_1 \cdots v_n : w))$ (ζ-expansion of [1] or $\delta\zeta$-expansion of [4]); else*
22. *if $\omega_1(\Gamma, v_i) = x{\leftarrow}v.v'_i$ and v_i is of sort* `Prop` *in Γ then $\omega_1(\Gamma, (t \Rightarrow v_1 \cdots v_n : w)) \equiv x{\leftarrow}v.\omega_1(\Gamma, (t \Rightarrow v_1 \cdots v_{i-1}\ v'_i\ v_{i+1} \cdots v_n : w))$;*

23. $\omega_1(\Gamma, (t \Rightarrow v_1 \cdots v_n : w)) \equiv (\omega_1(\Gamma, t) \Rightarrow \omega_1(\Gamma, v_1) \cdots \omega_1(\Gamma, v_n) : \omega_1(\Gamma, w))$.

Coming now to the issue of proving some properties of the function ω_1, we observe that any fully developed proof by induction on twenty-three cases can not be placed in the present paper, so the following proofs will be only sketched.

In particular we conjecture that the computation of $\omega_1(\Gamma, t)$ always terminates and that its complexity can be exponential in the size of the term t.

Theorem 2 (fixed point).

1. *(fixed point)* ω_1 *after* ω_1 *is* ω_1*;*
2. *(compatibility) if* $\Gamma \vdash t : u$ *then* $\Gamma \vdash \omega_1(\Gamma, t) : u$ *and* $\Gamma \vdash \omega_1(\Gamma, t) \leftrightarrow t$.

Proof.

1. $\omega_1(\Gamma, t)$ is defined as a fixed point;
2. ω_1 performs conversion steps that are known to preserve the type [7]. □

3.2 Representation of Optimised CIC Proof Terms

Here we present the function π that produces a tactical expression U starting from a CIC term t typable in a context Γ. If t is optimised in the sense of Subsection 3.1, we guarantee that U can be evaluated and that its result is t.

The expression U includes two sorts of proof steps: the construction steps corresponding to the constructors of t and the conversion steps that we detect by comparing the inferred type and the expected type of the subterms of t according to Coscoy's double type-inference [11]. Notice that the inferred type of a term v is always defined while the expected type of v is defined only in some cases, so when it is not defined, we set it as the inferred type of v for convenience.

The most delicate aspect of the design of π is the translation of an application $t' \equiv (t\ v_1 \cdots v_n)$ into an effective invocation of the `apply` t tactic. In particular we have to make sure that the conclusion u of the conjecture in the context of which this tactic is evaluated, can be unified with the inferred type of t'. Since the amount of conversion performed during unification depends on the particular system and is unpredictable in practice, we choose to assume that `apply` performs no conversion when unification is invoked, and we convert u by hand by an explicit invocation of the `change` tactic. Being u the expected type of t' [11], this conversion is needed only when the expected type of t' differs from the inferred type of t' (this can happen if t' is an argument of an application).

A similar problem arises when we detect a difference between the inferred type and the expected type of the argument, say v, of a construction that we want to render with a `cases` v or an `elim` v `using` $\cdots$, since evaluating such a tactic requires computing the inferred type of v. In this case v is a reference to a local premise x (after optimisation) so we invoke `change` $\ldots$ `in` x. In the other cases the explicit conversion can be safely omitted.

The rules defining the function π are given below with the rules of two auxiliary functions γ_1 and γ_2 used to handle conversion. Notice that we make no assumptions on how the inferred and expected types of a term are computed.

Definition 7 (the rules for γ_1).

1. *if the inferred type u of t differs from the expected type of t then* $\gamma_1(\Gamma, t, U) \equiv$ `change` u; $[U]$*; else*
2. $\gamma_1(\Gamma, t, U) \equiv U$.

Definition 8 (the rules for γ_2).

1. *if* $\Gamma \equiv \Gamma_1.x{:}w.\Gamma_2$ *and if w is not the inferred type of x then* $\gamma_2(\Gamma, x, U) \equiv$ `change` w `in` x; $[U]$*; else*
2. $\gamma_2(\Gamma, t, U) \equiv U$.

Definition 9 (the rules for π).

1. *if t is not of sort* `Prop` *then* $\pi(\Gamma, t) \equiv \gamma_1(\Gamma, t,$ `apply` $t)$*; else*
2. *if* $t \equiv c$ *or* $t \equiv x$ *then* $\pi(\Gamma, t) \equiv \gamma_1(\Gamma, t,$ `apply` $t)$*;*
3. $\pi(\Gamma, \lambda x{:}w.t) \equiv$ `intro as` x; $[\pi(\Gamma.x{:}w, t)]$;
4. *if* $v' \equiv (t' \cdots y_1 \cdots y_n \cdots v)$ *where t' is an induction principle and $y_1 \cdots y_n$ are the proofs of the inductive cases, if* $\Gamma \vdash v' : w'$ *and* $\Gamma \vdash w'$: `Prop` *then* $\pi(\Gamma, x{\leftarrow}v'.t) \equiv \gamma_2(\Gamma, v, \gamma_2(\Gamma, y_1, \cdots\ \gamma_2(\Gamma, y_n, U)\ \cdots))$ *for* $U \equiv$ `elim` v `using` t' `in` $y_1 \cdots y_n$ `as` x; $[\pi(\Gamma.x{:}w', t)]$ *when* $\Gamma.x{\leftarrow}v' \vdash t : t'$ *and x does not occur in t'; else*
5. *if* $v' \equiv (t' \cdots y_1 \cdots y_n \cdots v)$ *where t' is an induction principle and $y_1 \cdots y_n$ are the proofs of the inductive cases then* $\pi(\Gamma, x{\leftarrow}v'.t) \equiv \gamma_2(\Gamma, v, \gamma_2(\Gamma, y_1, \cdots\ \gamma_2(\Gamma, y_n, U)\ \cdots))$ *for* $U \equiv$ `elim` v `using` t' `in` $y_1 \cdots y_n$ `as` x; $[\pi(\Gamma.x{\leftarrow}v', t)]$*; else*
6. *If* $\Gamma \vdash v : w$ *and* $\Gamma \vdash w$: `Prop` *and* $\Gamma.x{\leftarrow}v \vdash t : t'$ *and x does not occur in t', then* $\pi(\Gamma, x{\leftarrow}v.t) \equiv$ `cut` w `as` x; $[\pi(\Gamma.x{:}w, t)\ \pi(\Gamma, v)]$*; else*
7. $\pi(\Gamma, x{\leftarrow}v.t) \equiv$ `pose` v `as` x; $[\pi(\Gamma.x{\leftarrow}v, t)]$;
8. *if* $t' \equiv (t \cdots v_1 \cdots v_n \cdots v)$ *where t is an induction principle and $v_1 \cdots v_n$ are the proofs of the inductive cases then* $\pi(\Gamma, t') \equiv \gamma_2(\Gamma, v, \gamma_1(\Gamma, t',$ `elim` v `using` t; $[\pi(\Gamma, v_1) \cdots \pi(\Gamma, v_n)]))$*; else*
9. *if* $t' \equiv (t\ v_1 \cdots v_n)$ *then* $\pi(\Gamma, t') \equiv \gamma_1(\Gamma, t',$ `apply` t; $[\pi(\Gamma, v_1) \cdots \pi(\Gamma, v_n)])$;
10. *if* $t \equiv (v \Rightarrow t_1 \cdots t_n : w)$ *then* $\pi(\Gamma, t) \equiv \gamma_2(\Gamma, v, \gamma_1(\Gamma, t, U))$*; for* $U \equiv$ `cases` v; $[\pi(\Gamma, v_1) \cdots \pi(\Gamma, v_n)]$;
11. $\pi(\Gamma, (t : w)) \equiv \pi(\Gamma, t)$*;*
12. `rewrite` *tactics are used in place of* `elim` *tactics when possible.*

We can set the following results about the transformation π:

Theorem 3 (correctness).

1. *(correctness) if* $\Gamma \vdash t : u$ *then* $[\![\Gamma \vdash u, \epsilon(\Gamma, t)]\!] = \omega_1(\Gamma, t)$*;*
2. *(round trip) if* $\Gamma \vdash t : u$ *and if* $U \equiv \epsilon(\Gamma, t)$ *then* $\epsilon(\Gamma, [\![\Gamma \vdash u, U]\!]) = U$.

Proof.

1. Once the statement is rephrased like $\Gamma \vdash t : u$ and $t' \equiv \omega_1(\Gamma, t)$ implies $[\![\Gamma \vdash u, \pi(\Gamma, t')]\!] = t'$, given that t' is optimised and that γ_1 and γ_2 do not affect the result of the expression they are applied to, the proof becomes straightforward;
2. unfolding t', clause 1 gives $[\![\Gamma \vdash u, \epsilon(\Gamma, t)]\!] = \omega_1(\Gamma, t)$ that, by Theorem 2.1, implies $\omega_1(\Gamma, [\![\Gamma \vdash u, \epsilon(\Gamma, t)]\!]) = \omega_1(\Gamma, t)$ by which we can conclude $\epsilon(\Gamma, [\![\Gamma \vdash u, \epsilon(\Gamma, t)]\!]) = \epsilon(\Gamma, t)$ meaning that ϵ after $[\![\ ]\!]$ after ϵ is ϵ. □

4 Implementation and Testing in a Real Case

In this section we discuss our own implementation of the transformation ϵ (see Section 3) in the proof assistant MATITA and the tests we used to verify it.

4.1 Implementation Issues

The current version of the proof assistant MATITA provides our transformation ϵ as an experimental feature whose implementation is still under development.

Given the uniform resource identifier (URI) of a theorem in the Hypertextual Electronic Library of Mathematics (HELM) [12], that is the digital library of MATITA, the tactical expression representing its proof is computed as follows:

Definition 10 (ϵ transformation pipeline).

1. *the proof is read from the library obtaining its representation at the logical level as a plain CIC proof term t_0, i.e. the initial version of the proof;*
2. *the transformation ω_1 (see Subsection 3.1) is applied to t_0 giving a plain CIC proof term t_1, i.e. the optimised version of t_0;*
3. *t_1 is type-checked and every sub-term is annotated with its inferred and expected types according to Coscoy's double type-inference algorithm [11]; this returns an annotated CIC proof term t_2;*
4. *the transformation π (see Subsection 3.2) is applied to t_2 giving a tactical expression T_3 represented in an ad hoc intermediate format designed by us;*
5. *T_3 is encoded into a tactical expression T_4 of Grafite, the knowledge representation language of MATITA at the user interface level; this format contains many details that are omitted in our intermediate format;*
6. *T_4, the final representation of the proof, is processed by the Grafite pretty printer and rendered as a script that the user can store or modify at will.*

We stress that the segmentation of the above pipeline decouples its transformation stages from its type checking and rendering stages allowing a factorised implementation of the whole procedure.

Currently Definition 6.6 and Definition 6.8 are not implemented while Definition 6.11 is implemented only in the case of t_2 containing no free occurrences of x; Definition 6.12 is used otherwise. Furthermore Definition 9.4 and Definition 9.5 are implemented only in the case `rewrite` tactics can be used, since MATITA does not implement the tactic `elim` $\cdots$ `in` $\cdots$ at the moment. In addition, the current implementation of `rewrite` $\cdots$ v `in` y `as` x in this system does not follow the semantics of Definition 5.3 when y refers to a local definition.

Taking this argument into account would force us to discuss the structure of inductive principles and this is not in the scope of the present paper.

Concerning Definition 9.3, we exploit a generalisation of `intro as` x; $[U]$ provided by MATITA in the form `intro as` $x_1 \cdots x_n$; $[U]$ with the semantics of a repeated `intro` before U. Concerning Definition 9.6 we use a variant of `cut` w `as` x; $[U\ W]$ provided by MATITA in the form `cut` w `as` x; [`id` W]; $[U]$

because we want to present W before U in the resulting proof. This approach forces us to introduce the `id` (identity) tactic having the meaning of a placeholder for a postponed expression. Formally: $[\![\Gamma \vdash u, \mathtt{id};\ [U]]\!] \equiv [\![\Gamma \vdash u, U]\!]$ and $[\![\Gamma \vdash u, T;\ [U_1 \cdots U_i\ \mathtt{id}\ U'_1 \cdots U'_j];\ [U]]\!] \equiv [\![\Gamma \vdash u, T;\ [U_1 \cdots U_i\ U\ U'_1 \cdots U'_j]]\!]$.

Finally, concerning Definition 9.9, we said in Subsection 2.3 that the system can infer some terms among $v_1 \cdots v_n$ by unification, but it is a matter of facts that higher order unification may fail in some cases.

To solve this inconvenient we take advantage of the possibility offered by MATITA to indicate the unifiers used by the `apply` tactic. In particular we detect the inferable functional terms among $v_1 \cdots v_n$ and we specify them as explicit unifiers in the `apply` tactical expression. Notice that although these terms appear as procedural arguments in this expression, we judge that anticipating them by ζ-expansion brings no benefit to the resulting proof.

Each proof by cases can be turned into a proof by induction carried out by applying a system-provided theorem denoting a default induction principle.

We may extend ω_1 with this transformation step for two reasons: firstly we may want to reuse the optimised proof term in those logical frameworks that support proofs by induction in place of proofs by cases, as the Minimal Type Theory [13]; secondly the set of constructions appearing in the optimised proof term is reduced. We strongly stress that this transformation is not a conversion according to CIC rules, so a term may not be well typed after this extended optimisation. Nevertheless it is well typed in the great majority of the real cases.

Definition 11 (critical optimisation steps).

1. *if $\Gamma \vdash v : w$ and c is the default induction principle of w for the sort* `Prop` *then $\omega_1(\Gamma, (v \Rightarrow t_1 \cdots t_n : w)) \equiv \omega_1(\Gamma, (c \cdots v))$.*

4.2 Testing Issues

We tested our implementation of the transformation ϵ on the 668 proofs of [4]. These proofs, originally appearing as tactical expressions of the Gallina specification language version 7 [3, 14], were processed by the proof assistant COQ, which turned them into CIC proof terms. These terms where processed by the proof assistant MATITA, which turned them into objects of HELM and they are available in this form inside the HELM directories cic:/matita/LAMBDA-TYPES/Base-1 and cic:/matita/LAMBDA-TYPES/LambdaDelta-1. The transformation ϵ was applied at this stage.

Three proofs of these were actually generated by COQ without evaluating a tactical expression[2] and make the development self-contained.

Figure 1 shows the frequency of the optimisations performed by ω_1. Definition 6.18 is never applied because the proofs do not contain any type cast construction. All proofs successfully type-check after optimisation.

Sixteen proofs do not reach stage 4 of Definition 10 because of problems in the current implementation of Coscoy's algorithm in MATITA.

[2] These proofs were produced by the GENERATE INVERSION directive of COQ 7.3.1 [3]

Optimisation	Action	Applied
Definition 6.4	information removal	250 times
Definition 6.5	definition lifting	429 times
Definition 6.10	definition lifting	2360 times
Definition 6.12	information removal	227 times
Definition 6.13	nested application	494 times
Definition 6.14	anticipation	645 times
Definition 6.15	definition lifting	3781 times
Definition 6.16	anticipation	2163 times
Definition 11.1	critical step	254 times
Any of the above		10603 times

Fig. 1. Frequency of the optimisations

Fourteen tactical expressions resulting from the transformation π applied to the remaining proofs, are not evaluated correctly by MATITA because of problems in the current implementation of the `elim` tactic.

We stress that the generated proofs should always be checked by MATITA especially if critical optimisation steps are allowed as in the case we are discussing.

Source (content)	Scripts size (type)	Tactics
Initial COQ input (668 proofs − 3)	0.4 Mbytes (Gallina)	9879
Output of COQ (668 CIC proof terms)	2.8 Mbytes (Gallina)	
Initial MATITA input (668 CIC proof terms)	4.1 Mbytes (Grafite)	
Output of π (668 proofs − 16)	2.1 Mbytes (Grafite)	51289

Fig. 2. Volume of the data

Figure 2 shows the volume of the data (in scripts size and complexity) involved in the whole transformation from initial COQ input to final π output.

Notice that the scripts are self-contained so they include the definitions not found in COQ's library. This additional content is considered in the calculation of size, but it is not considered in the calculation of complexity. Commented texts and unnecessary black characters are not considered in the computation of sizes.

Notice that the initial Gallina scripts are smaller than the final Grafite scripts both in size and in amount of tactics. This is because in the initial scripts we highly exploit automation tactics and code factorisation through macro tactics written in the LTAC language, while in the final scripts we use just primitive tactics without factorisation. In particular the scripts size increases by a factor 5.3 while the number of tactics, removing from the final scripts the `id` tactics that we could avoid and the other tactics of the 3 generated proofs (4912 all together), increases by a factor 4.7. We stress that these values are comparable with other values of the "de Bruijn loss factor" [15] found in the literature [16, 17]. This factor represents the increment in complexity occurring when the information hidden by automation and abstraction is fully displayed. The two factors we

give here can be considered the "apparent loss factor" and the "intrinsic loss factor" in the sense of [17]. Remarkably the difference of size between the final COQ output and the initial MATITA input (increment factor: 1.5) is entirely due to the greater verbosity of Grafite with respect to Gallina, since all data use the same concrete representation format. *i.e.* the ASCII-7 encoding.

It would be interesting to compare the evaluation times of the two sets of scripts once provided to the respective proof assistants (running in the same conditions) and to see if many primitive tactics are evaluated faster or slower than few high-level tactics producing the same proof terms. Unfortunately we can not perform tests like this one at the moment because COQ 7.3.1 and MATITA can not parse their own outputs in full at the moment.

4.3 A Running Example

In this section we present the result of the transformation ϵ applied to one of the 668 proofs mentioned in Subsection 4.2. Namely we consider the statement:

$$\text{theorem } le\_x\_pred\_y : \forall(y : nat).(\forall(x : nat).((lt\ x\ y) \rightarrow (le\ x\ (pred\ y))))$$

that formalises the property of the natural numbers: "$x < y$ implies $x \leq y-1$". Its HELM URI is cic:/matita/LAMBDA-TYPES/Base-1/ext/arith/le_x_pred_y.con and the constants occurring in it, *i.e.* *nat*, *lt*, *le*, *pred*, refer to entities defined in the standard library of COQ as included in HELM.

Figure 3 shows the initial proof term of the statement as produced by COQ and translated faithfully in Grafite. Notice the placeholder ?, added during the translation process, for a term that MATITA can infer.

The optimised version of the proof term is shown in Figure 4 and results from the application of the transformations described by Definition 6.10 (twice), Definition 6.13, Definition 6.14 and Definition 11.1. In particular by Definition 6.10 the construction $((\text{let } H2 \equiv \ldots \text{ in } (False\_ind\ \ldots\ H2))\ H0)$ becomes
$\text{let } H2 \equiv \ldots \text{ in } ((False\_ind\ \ldots\ H2)\ H0)$, then the nested application is detected and we get $\text{let } H2 \equiv \ldots \text{ in } (False\_ind\ \ldots\ H2\ H0)$ by Definition 6.13, then the non-sober application is detected and by Definition 6.14 we get
$\text{let } H2 \equiv \ldots \text{ in } ((\text{let } LOCAL \equiv (False\_ind\ \ldots\ H2) \text{ in } LOCAL)\ H0)$, which, by the second application of Definition 6.10, becomes
$\text{let } H2 \equiv \ldots \text{ in let } LOCAL \equiv (False\_ind\ \ldots\ H2) \text{ in } (LOCAL\ H0)$. Secondly by Definition 11.1 the construction match H in le ... (proof by cases) becomes $(le\_ind\ \ldots\ H)$ (corresponding proof by induction carried out by the application of a default induction principle). The other match constructions are not affected by this transformations because they do not represent proofs.

Figure 5 shows the Grafite proof script derived from the optimised version of the proof term. Notice that the tactics `change` and `elim` take a locative argument that denotes a conjecture pattern. This pattern contains the placeholders `%` and `?` representing the terms (or subterms) on which these tactics must act and the other terms (or subterms) respectively. Look at [2] Section 3.2 for details. Also notice that the `intros` tactic can take an `_` in place of a premise name

to mean that the introduced premise will not be used to complete the proof and is to be removed from the current conjecture context. More generally the author is working on an improved version of π that exploits the `clear` tactic to remove a premise from the current conjecture context as soon as the proof can be completed without it.

5 Conclusions and Future Work

In the previous sections we proposed an effective procedure for translating a proof term of the Calculus of Inductive Constructions (CIC), which is very similar to a program written in a prototypal functional programming language, into a tactical expression of a CIC-based proof assistant's user interface language.

This procedure allows to convert a proof encoded at the logical level and coming from any source, *i.e.* from a digital library or from another proof development system, into an equivalent proof presented in the proof assistant's editable high-level format. In particular we can improve the quality of user-provided proof scripts by regenerating them from their logical level content. In fact the scripts generated by the transformation we presented are clean (*i.e.* they contain no detours or unused information), they are specifically designed to be easily maintained and in the end they may contain many useful optimisations that perhaps the user would not introduce systematically in hand-written proof scripts, especially in the context of a large-scale formal development.

As a use case, we reported on our implementation of the procedure in the system MATITA [2] and on the translation of 668 proofs generated by the system COQ 7.3.1 [3], from their logical representation as CIC proof terms to their high-level representation as tactical expressions of MATITA's user interface language.

We noticed that the comparison between the initial COQ scripts producing the proofs and the final MATITA scripts resulting from the conversion, gives an increment factor in size and complexity that is compatible with other values of the "de Bruijn loss factor" [15] found in the literature. This increment occurs because the initial scripts are based on sophisticated tactics providing automation and code factorisation, whereas the final scripts are based on primitive tactics.

Our next objective is to improve the conversion procedure so that the increment factor is reduced as much as possible. In order to achieve this goal, the current output of our procedure needs further processing (before stage 5 of Definition 10) aimed at replacing groups of primitive tactics with advanced tactics having the same semantics. This means that advanced tactics with a formal semantics must be provided. We are also interested in limiting the use of declarative tactics in Definition 9 favouring their procedural counterparts.

Notice that the measure of complexity we used for the scripts is the number of tactics they contain. This measure does not take into account the complexity of the CIC terms appearing in the scripts as tactical arguments. We can estimate such complexity by counting the number of nodes occurring in the tree representation of these terms. So we plan to improve the accuracy of our tests by including this measure as well.

References

1. Coq development team: The Coq Proof Assistant Reference Manual Version 8.1. INRIA, Orsay (Feb 2007)
2. Asperti, A., Sacerdoti Coen, C., Tassi, E., Zacchiroli, S.: User Interaction with the Matita Proof Assistant. Journal of Automated Reasoning, Special Issue on User Interfaces for Theorem Proving. To appear (2006)
3. Coq development team: The Coq Proof Assistant Reference Manual Version 7.3.1. INRIA, Orsay (Oct 2002)
4. Guidi, F.: Lambda-Types on the Lambda-Calculus with Abbreviations. Submitted to ACM TOCL (Nov 2006) http://arxiv.org/cs.LO/0611040.
5. Barthe, G., Pons, O.: Type Isomorphisms and Proof Reuse in Dependent Type Theory. In Honsell, F., Miculan, M., eds.: 4th International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2001). Volume 2030 of LNCS., Springer (2001) 57–71
6. Sacerdoti Coen, C.: From Proof-Assistants to Distributed Libraries of Mathematics: Tips and Pitfalls. In: Mathematical Knowledge Management 2003. Volume 2594 of LNCS., Springer (2003) 30–44
7. Coquand, T., Paulin-Mohring, C.: Inductively defined types. In Martin-Löf, P., Mints, G., eds.: Proceedings of the International Conference on Computer Logic (Colog'88). Volume 417 of LNCS., Springer (1990)
8. Kamareddine, F., Laan, T., Nederpelt, R.: A Modern Perspective on Type Theory From its Origins Until Today. Volume 29 of Applied Logic Series. Kluwer Academic Publishers, Norwell (May 2004)
9. Sacerdoti Coen, C.: Declarative Representation of Proof Terms. Submitted to: Programming Languages for Mechanised Mathematics Workshop (PLMMS07) (2007)
10. Sacerdoti Coen, C.: Explanation in Natural Language of $\bar{\lambda}\mu\tilde{\mu}$-terms. In: 4th International Conference on Mathematical Knowledge Management (MKM2005). Volume 3863 of LNAI., Springer (2006) 234–249
11. Coscoy, Y.: A Natural Language Explanation for Formal Proofs. In Retoré, C., ed.: Int. Conf. on Logical Aspects of Computational Linguistics (LACL). Volume 1328 of LNAI., Springer (Sept 1996) 149–167
12. Asperti, A., Padovani, L., Sacerdoti Coen, C., Guidi, F., Schena, I.: Mathematical Knowledge Management in HELM. Annals of Mathematics and Artificial Intelligence **38**(1) (May 2003) 27–46
13. Maietti, M., Sambin, G.: Towards a minimalist foundation for constructive mathematics. In Crosilla, L., Schuster, P., eds.: From Sets and Types to Topology and Analysis: Practicable Foundations for Constructive Mathematics. Oxford University Press, Oxford (2005) Forthcoming.
14. Guidi, F.: λ-Types on the λ-Calculus with Abbreviations. Formal development with the proof assistant COQ (Jan 2006) http://www.cs.unibo.it/~fguidi/download/LAMBDA-TYPES.tgz.
15. de Bruijn, N.: A survey of the project Automath. In: Selected Papers on Automath. North-Holland, Amsterdam (1994) 141–161
16. van Benthem Jutting, L.: Checking Landau's Grundlagen in the Automath System. In: Selected Papers on Automath. North-Holland, Amsterdam (1994) 299–301,701–720,721–732,763–799,805–808
17. Wiedijk, F.: The De Bruijn Factor. Typescript note (2000) http://citeseer.ist.psu.edu/wiedijk00de.html.

$\lambda(y : nat).(nat_ind\ (\lambda(n : nat).(\forall(x : nat).((lt\ x\ n) \to (le\ x\ (pred\ n))))))\ (\lambda(x : nat).(\lambda(H : lt\ x\ O)).$let $H0 \equiv$ (match **H** in **le** return $(\lambda(n : nat).(\lambda(\_ : (le\ ?\ n)).((eq\ nat\ n\ O) \to (le\ x\ O))))$ with $[le_n \Rightarrow (\lambda(H0 : (eq\ nat\ (S\ x)\ O)).$let $H1 \equiv (eq_ind\ nat\ (S\ x)\ (\lambda(e : nat).($match e in nat return $(\lambda(\_ : nat).Prop)$ with $[O \Rightarrow False \mid (S\ \_) \Rightarrow True]))\ I\ O\ H0)$ in $(False\_ind\ (le\ x\ O)\ H1)) \mid (le\_S\ m\ H0) \Rightarrow (\lambda(H1 : (eq\ nat\ (S\ m)\ O)).((let$ **H2** $\equiv (eq_ind\ nat\ (S\ m)\ (\lambda(e : nat).($match e in nat return $(\lambda(\_ : nat).Prop)$ with $[O \Rightarrow False \mid (S\ \_) \Rightarrow True]))\ I\ O\ H1)$ in (**False_ind** $((le\ (S\ x)\ m) \to (le\ x\ O))$ **H2**)) **H0**)]) in $(H0\ (refl\_equal\ nat\ O)))\ (\lambda(n : nat).(\lambda(\_ : ((\forall(x : nat).((lt\ x\ n) \to (le\ x\ (pred\ n))))))).(\lambda(x : nat).(\lambda(H0 : (lt\ x\ (S\ n))).(le\_S\_n\ x\ n\ H0)))))\ y)$

Fig. 3. The initial proof term

$\lambda(y : nat).(nat_ind\ (\lambda(n : nat).(\forall(x : nat).((lt\ x\ n) \to (le\ x\ (pred\ n))))))\ (\lambda(x : nat).(\lambda(H : (lt\ x\ O)).$let $H0 \equiv$ (**le_ind** $(S\ x)\ (\lambda(n : nat).((eq\ nat\ n\ O) \to (le\ x\ O)))\ (\lambda(H0 : (eq\ nat\ (S\ x)\ O)).$let $H1 \equiv (eq_ind\ nat\ (Sx)\ (\lambda(e : nat).$match e return $(\lambda\_ : nat.Prop)$ with $[O \Rightarrow False \mid (S(\_ : nat)) \Rightarrow True])\ I\ O\ H0)$ in $(False_ind\ (le\ x\ O)\ H1))\ (\lambda(m : nat).(\lambda(H0 : (le\ (S\ x)\ m)).(\lambda(_ : ((eq\ nat\ m\ O) \to (le\ x\ O))).(\lambda(H1 : (eq\ nat\ (S\ m)\ O)).$let **H2** $\equiv (eq_ind\ nat\ (Sm)\ (\lambda e : nat.$match e return $(\lambda\_ : nat.Prop)$ with $[O \Rightarrow False \mid (S(\_ : nat)) \Rightarrow True])\ I\ O\ H1)$ in let **LOCAL** $\equiv$ (**False_ind** $((le\ (S\ x)\ m) \to (le\ x\ O))$ **H2**) in (**LOCAL H0**))))) O **H**) in $(H0\ (refl\_equal\ nat\ O))))\ (\lambda(n : nat).(\lambda(\_ : (\forall(x : nat).((lt\ x\ n) \to (le\ x\ (pred\ n))))))).(\lambda(x : nat).(\lambda(H0 : (lt\ x\ (S\ n))).(le\_S\_n\ x\ n\ H0)))))\ y)$

Fig. 4. The intermediate optimised proof term

```
theorem le_x_pred_y:
 (\forall y:nat.\forall x:nat.x<y -> x<=pred y).
intros 1 (y);
elim y using nat_ind in |- ((? -> ? ? % -> ? ? (? %))) names 0; [
intros 2 (x H); change in |- (%) with (x<=0);
cut (0=0 -> x<=0) as H0; [ id | change in H:(%) with (S x<=0);
elim H using le_ind in |- ((? ? % ? -> ?)) names 0; [
intros 1 (H0); cut match 0 in nat return \lambda _:nat.Prop with
 [0 -> False|S (_:nat) -> True] as H1; [ id |
rewrite < H0 in |- (%); change in |- (%) with True; apply I];
change in H1:(%) with False;
elim H1 using False_ind in |- (?) names 0 | intros 4 (m H0 _ H1);
cut match 0 in nat return \lambda _:nat.Prop with
 [0 -> False|S (_:nat) -> True] as H2; [ id |
rewrite < H1 in |- (%); change in |- (%) with True; apply I];
cut (S x<=m -> x<=0) as LOCAL; [ id | change in H2:(%) with False;
elim H2 using False_ind in |- (?) names 0];
apply LOCAL; apply H0]]; apply H0; apply refl_equal |
intros 4 (n _ x H0); change in |- (%) with (x<=n);
apply le_S_n; change in |- (%) with (x<S n); apply H0];
qed.
```

Fig. 5. The final proof script

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*).

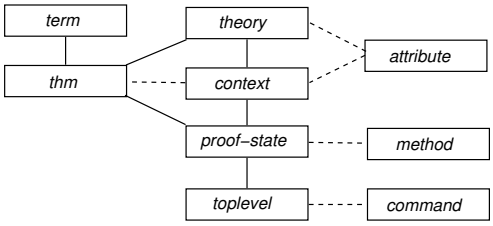


Fig. 1. Isabelle/Isar concepts

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.

Computer Algebra and the three 'E's: Efficiency, Elegance and Expressiveness

James H. Davenport & John Fitch
Department of Computer Science
University of Bath, Bath BA2 7AY
United Kingdom
{J.H.Davenport,J.P.Fitch}@bath.ac.uk

June 18, 2007

1 Introduction

What author of a programming language would not claim that the 3 'E's were the goals? Nevertheless, we claim that computer algebra does lead to particular emphases, and constraints, in these areas.

We restrict "efficiency" to mean machine efficiency, since the other 'E's cover programmer efficiency. For the sake of clarity, we describe as "expressiveness", what can be expressed in the language, and "elegance" as how it can be expressed.

2 Efficiency

Most programming languages claim efficiency, even when their authors are dead[1]. Times have moved on, but there is still a requirement for efficiency in terms of time and space in computer algebra.

Large data structures and the need for efficiency lead to techniques such as only working modulo word-sized primes, or packing exponents several to a word. These techniques may, and generally do, lead to high-performance, but require specialised, software components, such as polynomials modulo a small prime[2], which do not generalise to cases beyond those envisaged by the designers [6].

Another illustration of the difficulties of genericity comes when we consider Gaussian elimination in sparse matrices. Here we clearly want to use a variant of Dodgson/Bareiss [2, 7] to avoid intermediate expression swell, but also some

[1] "Fortran was also extremely efficient, running as fast as programs painstakingly hand-coded by the programming elite, who worked in arcane machine languages. This was a feat considered impossible before Fortran." [20]

[2] Before these were available in Maple, non-standard techniques [3] were required to get performance.

sparsity heuristics. But the coefficients can range from large expressions to integers, most of them very small, where efficiency of storage and operation dispatch are critical [13], to the point where we would like to store (most) integers in a single byte, and avoid any dispatch overhead altogether.

Axiom [14] supported a concept of "special-case compilation", where the same code could be compiled generically and for special values of the (type) parameters, leading to in-lining etc. This or some equivalent technique is needed to bridge the genericity/efficiency gap. Furthermore, it must be (essentially) automatic, otherwise one is liable to see the code bloat that apparently bedevilled Reduce 4 [12].

However, we must admit that automated support can only go so far in providing efficiency: at some point one has to "get one's hands dirty". For example, the variety of exponent representations supported automatically by Singular [17] can only be provided by a fairly intricate piece of C/machine code. The trick then is, as Singular does, to hide this from the user. We should note that Singular is 'sound', in the sense that the choice Singular makes is in terms of the number of variables in the input, which fixes the number of variables throughout the calculation. Similarly, it would be possible (though the authors do not know of any instance) to imagine a Gröbner-oriented engine which chose a representation based on the input total degree, and (globally) changed representation if this increased, which can only happen at fixed points in Buchberger's algorithm.

3 Elegance

Compare the following.

$$\frac{-b+\sqrt{b^2-4ac}}{2a} \tag{1}$$

```
\frac{-b+\sqrt{b^2-4ac}}{2a}

(-b+SQRT(b^2-4*a*c))/(2*a)

(/ (+ (- b) (SQRT (- (^ b 2) (* 4 a c)))) (* 2 a))

(divide (plus (minus b) (sqrt (minus (power b 2) (times 4 a c))))
        (times 2 a))
```

Of course, equation (1) is what we would like to see. Given the developments in mathematical editing [1, 8, 19, and many others], we could reasonably expect the front end to mathematical languages of the future to present (and probably edit) in the format of (1), so we would argue that the problem of elegance *of appearance* has largely been moved elsewhere, while elegance of functionality is effectively expressiveness.

If this is true of *input* expressiveness, it is certainly not true of *output* expressiveness, which is related to the looser meanings of "simplication" [15]. Despite

numerous attempts [1, 11, for example], there are still large gaps between what we see and what we would like to see.

Part of the problem is that input beauty is "in the eyes of the writer", an envisionable character, whereas output beauty is "in the eyes of the (unknown) beholder" — some early examples of the challenges this poses are in [10]. It could certainly be argued that this is an area where computer algebra ought to re-discover its roots in artificial intelligence.

4 Expressiveness

However, equation (1) is not what we see in most textbooks: we see

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{2}$$

This in itself does not look too dangerous, we merely need to widen the addition and division operators to be set-valued, as for example in SETL [18]. This looks like a road paved with a good intention, but, at least naïvely, it can lead to hell *in this case* (it *is* a useful facility in many other circumstances, such as dividing a vector by its norm).

The generally-given solution to the cubic $x^3 + bc + c$,

$$\frac{1}{6}\sqrt[3]{-108\,c + 12\sqrt{12\,b^3 + 81\,c^2}} - \frac{2b}{\sqrt[3]{-108\,c + 12\sqrt{12\,b^3 + 81\,c^2}}}, \tag{3}$$

is not three-valued at all [16], and this is a manifestation of the general problem of associating the "right" choices among multi-valued expressions. A computer scientist might feel tempted to 'solve' the problem by writing equation 3 as

$$\left(\lambda x.\frac{1}{6}x - \frac{2b}{x}\right)\sqrt[3]{-108\,c + 12\sqrt{12\,b^3 + 81\,c^2}}, \tag{4}$$

but this still appears to be six-valued (as indeed any similar expression

$$\left(\lambda x.\frac{1}{6}x - \frac{2b}{x}\right)\sqrt[3]{A + \sqrt{B}}$$

would be).

A similar problem is seen with interval arithmetic, where to get the "correct" results we need to know the dependencies between objects. A simple example (see [5] for more) is that of $x(1 - y)$ when $x, y \in [0, 1]$. This is also in $[0, 1]$ but its specialisation $x(1 - x) \in [0, 0.25]$.

For these reasons and others, extensions to computer algebra systems to support multi-valued objects have generally been of limited appeal. [4, 9]

A different problem, only partly within the scope of this workshop, concerns mathematical expressiveness. If a variable x appears, most systems have an in-built prejudice as to its range of values. This is rarely documented but seems to be as in table 1. Of these, only Maple, with its `assume` facility [21], lets the user change this prejudice.

Table 1: Systems and their views of Domains

System	Reduce	Macsyma	Maple	Mathematica
Domain	$\mathbf{R}^{\geq 0}$	$\mathbf{R}$	$\mathbf{C}$	$\mathbf{C}$

5 Conclusions

Computer algebra has made significant advances in the 3 E's since its early days in the 1960s. We have moved from simple hope of getting an answer to wishing to make it easy to use, and hence Elegance, useful in its results (Expressiveness) and capable of solving the large problems which still requires Efficiency, some of which can be obtained by a suitably expressive programming language, and some of which still requires hard work.

References

[1] O. Arsac, S. Dalmas, and M. Gaëtano. The Design of a Customizable Component to Display and Edit Formulae. In S. Dooley, editor, *Proceedings ISSAC '99*, pages 283–290, 1999.

[2] E.H. Bareiss. Sylvester's Identity and Multistep Integer-preserving Gaussian Elimination. *Math. Comp.*, 22:565–578, 1968.

[3] B.W. Char, K.O. Geddes, and G.H. Gonnet. GCDHEU: Heuristic Polynomial GCD Algorithm Based on Integer GCD Computation. In J.P. Fitch, editor, *Proceedings EUROSAM 84*, pages 285–296, 1984.

[4] J.H. Davenport and C.R. Faure. The "Unknown" in Computer Algebra. *Programmirovanie (Jan. 1994)*, pages 4–10, 1994.

[5] J.H. Davenport and H.-C. Fischer. Manipulation of Expressions. *Improving Floating-Point Programming*, pages 149–167, 1990.

[6] J.H. Davenport, J.A. Padget, and J.P. Fitch. On Symbolic Mathematical Computation. *Comm. ACM (ACM Forum)*, 28:1273–1274, 1985.

[7] C.L. Dodgson. Condensation of determinants, being a new and brief method for computing their algebraic value. *Proc. Roy. Soc. Ser. A*, 15:150–155, 1866.

[8] S.S. Dooley. Editing Mathematical Content and Presentation Markup in Interactive Mathematical Documents. In T. Mora, editor, *Proceedings ISSAC 2002*, pages 55–62, 2002.

[9] C. Faure, J.H. Davenport, and H. Naciri. Multi-valued Computer Algebra. Technical Report 4001, INRIA, 2000.

[10] R. Fenichel. An On-line System for Algebraic Manipulation. Technical Report MAC-TR-35, M.I.T., 1966.

[11] A.C. Hearn. Structure: the Key to Improved Algebraic Computation. In N. Inada and T. Soma, editors, *Proceedings 2nd. RIKEN Symp. Symbolic and Algebraic Computation*, pages 215–230, 1985.

[12] A.C. Hearn and E. Schrüfer. A Computer Algebra System based on Order-sorted Algebra. *J. Symbolic Comp.*, 19:65–77, 1995.

[13] A.J. Holt and J.H. Davenport. Resolving Large Prime(s) Variants for Discrete Logarithm Computation. In P.G. Farrell, editor, *Proceedings 9th IMA Conf. Coding and Cryptography*, pages 207–222, 2003.

[14] R.D. Jenks and R.S. Sutor. AXIOM: The Scientific Computation System. *Springer-Verlag*, 1992.

[15] J. Moses. Algebraic Simplification - A Guide for the Perplexed. *Comm. ACM*, 14:527–537, 1971.

[16] R.W.D. Nickalls. A new approach to solving the cubic: Cardan's solution revealed. *Math. Gazette*, 77:354–359, 1993.

[17] H. Schönemann. Singular in a Framework for Polynomial Computations. *Algebra*, pages 163–176, 2003.

[18] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. Programming with Sets: An Introduction to SETL. *Springer-Verlag*, 1986.

[19] E. Smirnova and S.M. Watt. Notation Selection in Mathematical Computing Environments. In *Proceedings Transgressive Computing 2006*, pages 339–355, 2006.

[20] New York Times. Computing Pioneer Backus Dies. `http://www.nytimes.com/2007/03/20/business/20backus.html?_r=1&adxnnl=1&oref=slogin&adxnnlx=1174472570-nB7vbQKJb5SeT53vVfnlSg`, 2007.

[21] T. Weibel and G.H. Gonnet. An Assume Facility for CAS with a Sample Implementation for Maple. In *Proceedings DISCO '92*, 1993.

Algebraic structures in Axiom and Isabelle: attempt at a comparison

Clemens Ballarin

Institut für Informatik
Universität Innsbruck
6020 Innsbruck, Austria
http://www4.in.tum.de/~ballarin

Abstract. The hierarchic structures of abstract algebra pose challenges to the module systems of both programming and specification languages. We relate two existing module systems that are designed for this purpose: the type system of the computer algebra system Axiom, and the module system of the theorem prover Isabelle.

Abstract algebra has been developed at the beginning of the 20th century and has enabled a high degree of reuse of mathematical, in particular algebraic, knowledge in various branches of mathematics. The same kind of reuse is desirable also in mathematical software, both computer algebra systems and provers.

Various facilities to enable code reuse are known for programming languages. These include polymorphism, and module and class systems, for example. Reuse in abstract algebra is particularly demanding and surpasses what programming languages offer normally. The module system of the computer algebra system Axiom [7] is particularly powerful and was designed to accommodate these needs.

Locales are the theory development modules of the theorem prover Isabelle [9, 10]. They too were developed to provide support for abstract algebra. Locales enable to manage specifications and derived theorems of modules effectively. Initial ideas were drawn from the sectioning concept of Coq [8]. Later extensions integrated concepts from algebraic specification, in particular a language to compose specifications [1], and a facility to declare and maintain derived relations between modules [2] (interpretation).

Although modules for programming languages and provers have rather differing requirements — the former deal with reuse of code, the latter with reuse of theorems — interesting insight can be gained from a comparison of both. In the following an attempt at such a comparison is made, by expressing a small fragment of mathematics in both of the systems. We proceed by presenting a piece of abstract algebra first in the usual mathematical notation and then formally in both Axiom and Isabelle (using locales). Both formalisations are then compared.

1 Some Group Theory

Definition 1. *A* semi group *is a tuple* $(G, *)$ *where* G *is a set (the* carrier set*) and* $*$ *is a binary associative operation on* G*.*

Definition 2. *A* monoid *is a triple* $(G, *, 1)$ *where* $(G, *)$ *is a semi group, and* $1 \in G$ *such that for all* $x \in G$

$$1 * x = x \tag{1}$$
$$x * 1 = x \tag{2}$$

Normally, in order to simplify notation, carrier set and algebraic structure are identified. We do so in the following definitions.

Definition 3. *A* group G *is a semi group with* $1 \in G$ *and an operation* $\mathrm{inv} : G \to G$ *such that for all* $x \in G$

$$1 * x = x \tag{3}$$
$$\mathrm{inv}(x) * x = 1 \tag{4}$$

It can be shown that (2) holds also for groups, hence

Theorem 1. *Every group is a monoid.*

The following algebraic structure, whose definition is taken from Jacobson's text book on Algebra [6] involves two carrier sets. While not all module systems are able to deal with such structures, they occur frequently in algebra. A more prominent example involving two carrier sets are vector spaces.

Definition 4. *A group* G *is said to* act on the set S *if there exists a map* $\circ : G \times S \to S$ *satisfying for all* $x, y \in G$ *and* $s \in S$

$$1 \circ s = s \tag{5}$$
$$(x * y) \circ s = x \circ (y \circ s) \tag{6}$$

2 Axiom: Categories and Domains

Axiom is, to a large extent, implemented in a functional programming language whose type system is unusually rich. It supports dependent types, and its dynamic type discipline has a distinct object-oriented flavour. *Categories* resemble abstract classes, and *domains* provide implementations. A category is asserted to Axiom through a category constructor — that is, a function returning the signature of that category. Multiple inheritance is supported, and the hierarchy of classes forms a directed acyclic graph.

Likewise, a domain constructor is a function, into a category, returning a domain implementing the category. Hence, which domain belongs to which categories is asserted. Categories and domains are parametric. For instance, the category constructor for vector spaces takes as an argument the coefficient field.

Figure 1 shows category declarations for the algebraic structures from Section 1. Each of these declares a function returning the signature of the algebraic structure, which is an object of type `Category`. The function bodies of these declarations are of the form

import `with` *export*.

```
SemiGroup(): Category == SetCategory with
    "*": (%, %) -> %

Monoid(): Category == SemiGroup with
    1: -> %

Group(): Category == Monoid with
    inv: % -> %

Action(G: Group): Category == SetCategory with
    "o": (G, %) -> %
```

Fig. 1. Group category declarations in Axiom

The declarations closely follow Section 1, with the exception that Theorem 1 has been incorporated in the definition of `Group`. Its import is `Monoid` rather than `SemiGroup`. The % sign denotes the carrier type. In Category `Action` the group and its carrier is introduced as an argument of the category constructor.

3 Locales: Morphisms and Interpretation

Locales are an extra-logical concept built on top of the meta-logic of Isabelle, which is available for all object-logics of Isabelle, including higher-order logic and ZF set theory. A locale contains a specification and can be seen as a storage container for theorems implied by that specification. Theorems may be added to a locale at any time, not just during the creation of the locale.

Locales are parametric. The parameters, which occur in both specification and theorems, can be replaced by terms, thus creating instances of the locale. This mapping from parameters to terms is called a *morphism*. By a (simple) property of the meta-logic images of locales under morphisms are consistent. This means that the instantiated specification implies the instantiated theorems.

Locales are, like to Axiom's categories, hierarchic. A locale may import one or several other locales, with the effect that the parameters and specifications of the imported locales become part of the importing locale. The same applies to theorems. Import may also be through morphisms. Then, instantiated specification and instantiated theorems become part of the locale.

Like with Axiom's categories, the import hierarchy of locales is a directed acyclic graph, where additionally, edges are labelled with morphisms. Besides of the import hierarchy, *interpretation* relations between locales are maintained. If the specification of one locale implies (an instance of) the specification of another locale, the corresponding instances of the theorems of that locale are also theorems of the locale. Locales provide an interpretation command through which interpretation relations between locales can be stated. Unlike import, which is declared, interpretation is a consequence of the specification of the involved locales and must be accompanied by a proof.

```
locale semigroup =
  fixes mult :: "'a => 'a => 'a" (infixl "*" 60)
  assumes assoc: "(x * y) * z = x * (y * z)"

locale monoid = semigroup +
  fixes one :: "'a" ("1")
  assumes lone: "1 * x = x"
    and rone: "x * 1 = x"

locale group = semigroup +
  fixes one :: "'a" ("1")
    and inv :: "'a => 'a"
  assumes lone: "1 * x = x"
    and linv: "inv x * x = 1"

interpretation group < monoid

locale action = group +
  fixes op :: "'a => 'b => 'b" (infixl "o" 60)
  assumes op_one: "1 o s = s"
    and op_assoc: "(x * y) o s = x o (y o s)"
```

Fig. 2. Group locale declarations in Isabelle

The management of hierarchic information in locales is similar to development graphs [5], and indeed they can be phrased in terms of development graphs [2, 3].

Figure 2 shows locale declarations corresponding to Section 1. Its structure is similar to the category declarations from Figure 1 with the exception that `group` directly extends `semigroup`. Instead, Theorem 1 is present in the declarations, as an interpretation[1] making theorems a user may add to `monoid` available in `group`. Carrier sets are represented by types, and are denoted by the type variables `'a` and `'b`.

4 Comparison

A comparison between the module system of a programming language and that of a theorem prover may appear an unfeasible endeavour. But both systems are designed to reflect hierarchies of abstract algebraic structures, and this is a natural starting point for a comparison. Axiom's categories correspond to abstract algebraic structures, and so do locales. The following comparison is restricted to aspects related to the representation of hierarchies of algebraic structures.

Programming vs. specification language. Since Axiom is a programming language, a category only represents the signature, while locales exhibit both signature and specification.[2]

[1] In Figure 2 the proof justifying the interpretation is omitted.

[2] An extension of Axiom by specifications was presented by Poll and Thompson [11].

Role of parameters. The parameters of a category are the arguments of the category. They are distinct from the signature, which is its result. This distinction is not present in locales: here the parameters *are* the signature, and the locale can be seen as a relation between the parameters. Hence, while in Axiom the category constructor for action groups takes the group signature as an argument and returns the signature of the set structure the group acts upon, the corresponding locale has parameters for both the operations of the group and the set. We may call Axiom's categories *functorial* and Isabelle's locales *relational* representations of algebraic structures.

Morphisms. Morphisms are not present in Axiom and they are not fully developed in locales.[3] Figure 2 contains no explicit use of morphisms. An example of their use is the derivation of an additive group from a multiplicative one.

```
locale additive_group =
  group add (infixl "+" 55) zero ("0") uminus ("- _")
```

The parameters, which have a canonical positional order, are renamed and receive new syntex. Morphisms are convenient for the construction of rings, or when defining the notion of a group homomorphism. In order to achieve such structures in Axiom, two separate hierarchies are implemented.

Interpretation. Categories only model the declared import hierarchy between abstract structures. Derived dependencies (interpretations) cannot be asserted later, while with locales this is possible. Interpretation is a standard device in a specification language, since it permits to establish relations between modules that were not anticipated when these modules were declared. Interpretation does not make sense in the context of a programming language, since there are no specifications and consequently no relations between them can be proved. Nevertheless, a facility to declare module dependencies not anticipated by a library designer might be a useful asset in a programming language. The algebraic hierarchy implemented in Axiom has been criticised as hard to extend. This might be an indication that similar means are missing in Axiom's module system.

5 Conclusions and Directions for Further Research

We have presented and compared aspects of two module systems for the representation of abstract algebraic structures. Both are based on strong foundations: Axiom's programming language has dependent types, which is unusual for functional languages. The hierarchic structure of locales can be described with development graphs. This is combined with Isabelle's meta-logic, which is a variant of higher-order logic. In contrast, specification languages are normally based on variants of first-order logic.

The comparison is preliminary and has highlighted interesting differences. Some of these differences are rooted in the fact that we have compared a programming and a specification language: Axiom's categories are functorial, while Isabelle's locales resemble relations with additional infrastructure for the management of hierarchies.

[3] The current implementation is restricted to parameter renamings, not arbitrary instances.

In the author's view this is the fundamental difference between Axiom's type system and locales. Once a concise characterisation has been obtained by abstracting away from technical aspects of both systems, two goals may be addressed:

- Compare the notions of functorial vs. relational approach to specifications with respect to their strengths.
- Classify other module systems, like the one for Coq [4], or FoCal [12].

Additionally, concepts from specification languages might be fruitfully carried over to programming languages. While interpretation *per se* is only meaningful in the presence of specifications, a similar mechanism to modify an existing module hierarchy might also be useful in programming languages. Similarly, morphisms might enable more complex import operations than inheritance.

References

1. C. Ballarin. Locales and locale expressions in Isabelle/Isar. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs, TYPES 2003, Torino, Italy*, LNCS 3085, pages 34–50. Springer, 2004.
2. C. Ballarin. Interpretation of locales in Isabelle: Managing dependencies between locales. Technical Report TUM-I0607, Technische Universität München, 2006.
3. 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, Wokingham, UK*, LNCS 4108, pages 31–43. Springer, 2006.
4. J. Chrzaszcz. Implementing modules in the Coq system. In D. Basin and B. Wolff, editors, *Theorem proving in higher order logics: TPHOLs 2003, Rome, Italy*, LNCS 2758, pages 270–286. Springer, 2003.
5. D. Hutter. Management of change in structured verification. In *Automated Software Engineering, ASE 2000, Grenoble, France*, pages 23–31. IEEE Computer Society, 2000.
6. N. Jacobson. *Basic Algebra*, volume I. Freeman, 2nd edition, 1985.
7. R. D. Jenks and R. S. Sutor. *AXIOM. The scientific computation system*. Numerical Algorithms Group, Ltd. and Springer-Verlag, Oxford and New York, 1992.
8. 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, Nice, France*, LNCS 1690, pages 149–165. Springer, 1999.
9. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.
10. L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, 1989.
11. E. Poll and S. Thompson. Adding the axioms to Axiom: Towards a system of automated reasoning in Aldor. Technical Report 6–98, Computing Laboratory, University of Kent, May 1998. Presented at the workshop Calculemus and Types, Eindhoven, Netherlands, July 1998.
12. V. Prevosto. Certified mathematical hierarchies: the FoCal system. In T. Coquand, H. Lombardi, and M.-F. Roy, editors, *Mathematics, Algorithms, Proofs*, number 05021 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005.

What Happened to Languages for Symbolic Mathematical Computation?

Stephen M. Watt
Ontario Research Centre for Computer Algebra
University of Western Ontario
London Ontario, Canada N6A 5B7
watt@csd.uwo.ca

Abstract

While the state of the art is relatively sophisticated in programming language support for computer algebra, there has been less development in programming language support for symbolic computation over the past two decades. We summarize certain advances in programming languages for computer algebra and propose a set of directions and challenges for programming languages for symbolic computation.

1 Introduction

Digital computers have been used to solve symbolic mathematical problems now for more than half a century. While early work considered algebraic expressions and the operations of differential and integral calculus (*e.g.* [11, 17, 27]), now the full range of mathematics is considered from an algorithmic point of view. Also, where early work used computers to perform the same basic algorithms as used by people at elementary levels, now highly sophisticated techniques are used in mathematical algorithms, in mathematical experimentation and in automated proof.

There have been some remarkable advances, both theoretical and practical, extending the scope of problems that computer algebra can treat. Moreover, there are commercial and research systems that have brought computer algebra into the mainstream, where it may be used by anyone who deals with mathematical formulæ. The state of the art in symbolic algorithms for linear, polynomial and differential systems has evolved and continues to evolve rapidly. The ranks of researchers working in the area have grown considerably, now with many mathematicians contributing to the main algorithmic advances. It is now sometimes feasible to treat large problems exactly by symbolic means where approximate numerical methods fail.

What is wrong with this rosy state of affairs? The successes of symbolic mathematical computation have also been, to a certain extent, its undoing.

As algorithms in some areas have been developed, problems have been re-cast using the specialized languages of those areas. This has moved the focus *away from general techniques* for doing mathematics by computer, and toward the quest for *better algorithms in central areas*. Our focus on these areas has led to the exclusion of others and, moreover, to the tendency to see all problems only in these terms. To give just one example, it has been common practice for computer algebra systems to provide formal antiderivatives where integrals have been requested. Because system developers became so used to thinking in terms of differential algebra, they would miss the fact that (or even argue that it was correct that), as functions, the "integrals" could have spurious discontinuities and jumps.

While it is important to pursue technical development on the algebraic algorithms for core domains, there are embarrassing lacunae in the repertoire of what computer algebra can do. To give a simple example, although we may now factor multivariate polynomials of high degree in many variables over algebraic function fields, computer algebra systems cannot presently factor the simple expression $d^2 - 2^{n^2+n}$, which is a difference of squares for any integer n. This is not a contrived problem: perhaps every reader has had to simplify similar expressions when analyzing algorithms.

Corresponding to the nearly exclusive focus on core problem domains, there has been diminished attention on how programming languages can best support symbolic mathematical computation. Indeed, where there used to be a host of programming languages specializing in symbolic mathematical computation, there are now only a few. Moreover, there seems to be little current attention as to how they can be made more effective. To a first approximation, today we no longer have languages *for* symbolic mathematical computation. Instead we have general purpose languages that happen to be *applied to* symbolic mathematical computation.

Many ideas that later find their way into main-stream programming languages have appeared first in programming languages for mathematical computation. These ideas include programming with algebraic expressions, the use of arrays, arbitrary precision integers, automatic memory management and dependent types. We argue that there is still a lot left to learn: By looking harder at the problem domain of symbolic mathematical computation we can develop new language ideas that will first help symbolic mathematical computation, and then perhaps also be more generally applicable. This note is intended to encourage discussion on this topic.

We start with a discussion of the relation between computer algebra and symbolic computation. While this gives a computer algebra bias to the paper, we find it to be a useful example of the solution changing the problem. We then give a short summary of our own biases and outline why we think that programming language technology for computer algebra is in a useful state. We then consider the state programming languages for symbolic computation and identify a host of issues where we believe that language support could be both interesting and useful.

2 Computer algebra and symbolic computation

The term "symbolic computation" has different meanings to different audiences. Depending on the setting, it can mean: any computation that is not primarily arithmetical, any computation involving text, any computation on trees with free and bound variables, or computation involving codes over a set of symbols. In fact the first volume of the *Journal of Lisp and Symbolic Computation* (Kluwer 1991) and the first volume of the *Journal of Symbolic Computation* (Academic Press 1985) have almost disjoint subject matter. For our purposes, "symbolic computation" or "symbolic mathematical computation" will relate to mathematical computation with expressions, exact quantities or approximate quantities involving variables or indeterminates. We could adopt a broader or narrower definition, but this would not change the main points we wish to make.

In the early days of symbolic mathematical computing, much of what was done could be described as expression manipulation. This point of view is nicely captured in the survey by Jean Sammet [25]. For several years there was varying terminology involving combinations of "symbolic", "algebraic", "computation" and "manipulation" as well as the term "computer algebra." During this naming uncertainty the field itself was changing: First, broader classes of problems were addressed and specialized communities grew up around particular areas. Second, within many of these areas, problems were solved in specific algebraic structures rather than in terms of general expressions. Thus, to a certain extent, part of symbolic mathematical computing grew into the name "computer algebra."

Many working in the area of computer algebra use the terms "computer algebra" and "symbolic computation" synonymously. This author, however, finds it very useful to make a distinction between the two, even when working on the same types of objects. By "computer algebra," I mean the treatment of mathematical objects as values in some algebraic domain, for example performing ring operations on polynomials. In this view, $x^2 - 1$ and $(x - 1) \times (x + 1)$ are the same. By "symbolic computation," I mean working with terms over some set of symbols. In this view, the two expressions $x^2 - 1$ and $(x - 1) \times (x + 1)$ are different. Computing a gcd is computer algebra; completing a square is symbolic computation.

It is difficult in computer algebra to treat problems with unknown values (*e.g.* with a matrix with unknown size, a polynomial of unknown degree or with coefficients in a field of unknown characteristic). It is difficult in symbolic computation to use any but the most straightforward mathematical algorithms before falling back on general term re-writing. We must explore what can be done to bridge the gap between these two views. We can make computer algebra *more symbolic*, providing effective algorithms for a broader class of mathematical expressions. We can likewise make symbolic computation *more algebraic*, by restricting classes of admissible expressions, for example using typed terms.

As an example of this rapprochement, the papers [14, 33, 34] begin a discussion of the relationship between the arithmetic view of computer algebra and the term-rewriting view of symbolic computation, concentrating on polynomials of

unknown degree. The first objective is to formalize the notion of symbolic polynomials. The approach has been to take symbolic polynomials as expressions in finite terms of ring operations, allowing variable and coefficient exponents to be multivariate integer-valued polynomials, e.g. $x^{n^4-6n^3+11n^2-6(n+2m-3)} - 1000000^m$. This gives a group-ring structure, which, for suitable coefficient rings, can be shown to be a unique factorization domain. Factorizations in this UFD are uniform factorizations under evaluation of the exponent variables.

The cited papers begin to explore two approaches for algorithms to compute greatest common divisors, factorizations, *etc.* The first is to use the algebraic independence of x_i, $x_i^{n_1}$, $x_i^{{n_1}^2}$, $x_i^{n_1 n_2}$, *etc.* By introducing a number of new variables, we reduce the problem to one on usual multivariate polynomials. Avoiding fixed divisors of the exponent polynomials requires expressing them in a binomial basis. This, however, makes exponent polynomials dense, leading to doubly exponential algorithms. The second approach is to use an evaluation/interpolation scheme on the exponent variables. This has combinatorial problems when the polynomials have a limited set of distinct coefficients. Careful use of evaluation, and solving equations in symmetric polynomials, seems to point to a direction that avoids exponential combinations. Early experiments seem to show that this approach is amenable to the use of sparse interpolation techniques in the exponents. A simple application of Baker's transcendence theory allows us to treat logarithms of primes as algebraically independent so we can handle polynomial exponents on integer coefficients.

A number of authors have extended the domain of computer algebra in particular directions, *e.g.*, in the areas of mathematical theory exploration [1] and quantifier elimination [29]. More closely related to the current discussion are recent work on parametric [37] and uniform [23] Gröbner bases. Part of the motivation for the current direction for algorithms on symbolic polynomials comes from earlier work that extended the domain of polynomials to supersparse polynomials [12] and the ring of exponential polynomials [9]. A novel current direction is the algebraic treatment of elision ($\cdots$) in symbolic matrices [26]. There are isolated examples of early work on simplifying matrix-vector expressions [28] and the relation of symbolic computing to computer algebra [4].

3 Prejudices

For context, I should say that I have been involved in the design of several languages used in computer algebra, including Maple, Axiom, Aldor, OpenMath and MathML. The first three are programming languages, or systems with programming languages, and the last two are data languages.

Both Maple and the Axiom interpreter language try to provide a relatively simple to use language designed for scripting and interactive use. The Maple system uses its scripting language for most algebraic library development, while Axiom provides a distinct (but related) language for this purpose.

The Axiom library programming language had its initial version described in 1981 [10]. It provided an early implementation of qualified parametric poly-

morphism, which has decades later been adopted as an essential technique in the main-stream programming world.

Aldor [8, 31] extended the ideas of the Axiom programming language, adopting dependent types as its foundation and reconstructing Axiom's categories and domains, as well as object-oriented constructs from them. The use of *post-facto extensions* foreshadowed the separation of concerns by aspect-oriented programming, as reported in [35]. In addition, Aldor's use of abstract iteration [36] was an early example of the renaissance of control-based iterators (with `yield`), pioneered by CLU and Alphard, and now appearing in limited forms in languages such as Ruby and `C#`.

4 Language support for computer algebra

Computer algebra libraries tend to require "programming in the large," with precisely-related composable libraries, while at the same time requiring access to efficient arithmetic. In my opinion, the necessary programming language support for computer algebra is now quite mature and very good. Quite reasonable computer algebra libraries can be obtained using

- automated memory management,
- access to machine-level arithmetic and bit operations,
- parametric polymorphism, preferably with qualified parameters.

In some cases, mechanisms for very good support is available, *e.g.* to specify precise relationships among type parameters (Aldor), or to efficiently compile special cases of templates (`C++`).

Computer algebra is one of the few domains that provide rich and complex relationships among components that are at the same time well-defined. We therefore sometimes see programming language issues arising in area of computer algebra before they are seen in more popular contexts. We continue to see this today. Some of our recent work has included techniques for memory management [3, 30], performance analysis of generics [5, 6, 7], optimization of iterators and generics [5, 32, 36], and interoperability in heterogeneous environments [18, 19, 20, 21, 30, 38].

I see the following as interesting programming language problems whose investigation will benefit computer algebra:

(1) how to improve the efficiency of deeply nested generic types (templates, domain towers),

(2) techniques for efficient use of parametric polymorphism in multi-language environments,

(3) language support for objects that become read-only after an initial, extended construction phase,

(4) to use type categories to specify machine characteristics for re-configurable high-performance codes,

(5) a framework in which program-defined generic (parameterized) type constructors can form dependent types, e.g. `HashTable(k: Key, Entry(k))`, analogous to `(a: A) -> B(a)`,

(6) the use of *post facto* functor extensions as an alternative to open classes and aspect-oriented programming,

We have made some progress on some of these items: (1) and (2) have been the subject of PhD theses of two recent students. Some preliminary investigations on (3) have been undertaken together with a current postdoc. The topic (4) has been the subject of a collaboration with a group at U. Edinburgh that uses Aldor for Quantum Chromodynamics computations. The question (5) is motivated by the desire to treat all type constructors on an equal footing, an idea which has returned a lot of benefit in Aldor. There the mapping and product type constructors are still special only inasmuch as they provide dependent types. A model for generic dependency would allow mathematical programs to represent mappings either as functions, tables or other structures and allow all of these alternatives equal expressive power. Although the ideas behind (6) date back to work at IBM research in the early 90s, the idea of post facto extension of functors seems to just now be coming of age in the programming language world.

For problem (1), the Stepanov benchmarks provide a very simple measure for C++, which we have generalized to our SciGMark benchmark. Post-facto extensions are closely related to the notions of Aspect-Oriented Programming [13] and open classes [16], however they seem to provide more structure than the first and more opportunity for optimization than the second. Dependent types have for the longest time remained a secret of theorem provers and boutique theoretical languages, despite their early prominent occurrence AUTOMATH in the 1960s. The Aldor community has found them extremely useful. A limited form of dependency has been popularized by F-bounded polymorphism [2] and, more recently, dependent types have been receiving increased attention [15, 22].

5 Language support for symbolic computation

In contrast to the ample language support for computer algebra, there has been relatively little fundamental advance in language support for symbolic mathematical computation. In fact, with so many of the main advances being in algebraic algorithms, our software for symbolic mathematical computation have become popularly known as "Computer Algebra Systems."

Most computer algebra systems do in fact provide some level of support for expression manipulation. This typically includes expression traversal, expression reorganization into different forms (e.g. Horner form *vs* expanded form *vs* factored form), substitution, evaluation and various simplifications. Macsyma was an early system providing very good support in this area, and Mathematica is a more modern example of a system with good support in this area.

Despite this level of support for expression manipulation, I would suggest that programming language support for symbolic computation is still at an early stage. Some of the language directions that could fruitfully be explored include:

(1) better support for typed terms, *i.e.* variables that admit substitutions of only certain types and typed function symbols,

(2) construction of domains of symbolic terms (initial objects) by functorial operations on categories of concrete domains,

(3) smooth and extensible inter operation of program expressions on concrete domains and typed symbolic expressions as data,

(4) anti-unification modulo equational theories, use of adjoint functors for expression simplification, for example as in [24],

(5) use of empirical measures on expression spaces to define preferred forms of expressions under simplification,

(6) constructing expression transformations automatically from re-ordering of composed algebraic domain constructors ,

(7) generalizing the functional programming techniques of monads or arrows to move algorithms over commutative diagrams,

(8) more robust support for symbolic expressions on particular domains, including vector algebra and algebras of structured matrices,

(9) tools for encoding and using results from universal algebra,

(10) tools for better working with rule-sets, *e.g.* to determine noetherianity or divergence in certain settings,

(11) well-defined interfaces between automated theorem provers and computer algebra systems,

(12) technical matters, such as better support for hygienic treatment of free and bound variables in expressions.

These are a few of the immediate directions where symbolic computation could be better supported by linguistic mechanisms. It is a nice problem in language design to take full advantage of data/program duality so that meaningful composition of mathematical expressions can be constructed from composition of the corresponding programs. This is a fundamental area to get right if our symbolic mathematics systems are to be able to scale up. The "wild west" days of computer algebra systems working with ill-defined classes of expressions must be put behind us.

6 Conclusions

We have argued that the "computer algebra" and "symbolic mathematical computation" are two very different things and that, while computer algebra has been flourishing, symbolic computation on the same domain has been languishing. This is true both at the level of mathematical algorithms and in programming language support.

We have argued that the gap between these two areas must be bridged in order to have more useful systems for mechanized mathematics. These bridges can occur at the mathematical level, both by algebratizing the treatment of symbolic expressions and by developing algorithms for broader classes of algebraic objects. We gave the treatment of polynomials of symbolic degree as an example. These bridges should also occur at the programming language level, and we have presented a number of directions in which better language support could make symbolic computing richer.

What has happened to languages for symbolic mathematical computation? At least to this author, it appears that after the initial developments there have been many avenues left unexplored. Building sophisticated mathematical software that works, and that scales, is a difficult problem. If we are to succeed at building modular, extensible symbolic mathematics systems, then we should think harder about what we should ask of our programming languages.

References

[1] B. Buchberger. Algorithm-supported mathematical theory exploration: A personal view and strategy. In *Proc AISC*, pages 236–250. Springer, LNAI 3249, 2004.

[2] Peter Canning, William Cook, Walter Hill, and Walter Olthoff. F-Bounded Polymorphism for Object Oriented Programming. In *Proc. FPCA*, pages 273–280. ACM, 1989.

[3] Yannis Chicha and Stephen M. Watt. A localized tracing scheme applied to garbage collection. In *Proc. The Fourth Asian Symposium on Programming Languages and Systems*, pages 323–339. Springer Verlag LNCS 4279, 2006.

[4] J.H. Davenport and Ch. Faure. The "unknown" in computer algebra. *Programmirovanie*, 1:4–10, 1994.

[5] Laurentiu Dragan and Stephen Watt. Parametric polymorphism optimization for deeply nested types in computer algebra. In *Proc. Maple Conference 2005*, pages 243–259. Maplesoft, 2005.

[6] Laurentiu Dragan and Stephen M. Watt. Performance analysis of generics for scientific computing. In *Proc. 7th Internatioanl Symposium on Symbolic and Numeric Algorithms in Scientific Computing*, pages 93–100. IEEE Press, 2005.

[7] Laurentiu Dragan and Stephen M. Watt. On the performance of parametric polymorphism in maple. In *Proc. Maple Conference 2006*, pages 35–42. Maplesoft, 2006.

[8] S.M. Watt *et al. Aldor User Guide*. Aldor.org, 2003.

[9] C.W. Henson, L. Rubel, and M. Singer. Algebraic properties of the ring of general exponential polynomials. *Complex Variables Theory and Application*, 13:1–20, 1989.

[10] R. Jenks and B. Trager. A language for computational algebra. In *Proc. 1981 ACM Symposium on Symbolic and Algebraic Computation*, pages 6–13. ACM Press, 1981.

[11] H. G. Kahrimanian. *Analytical differntation by a digital computer*. MA Thesis, Temple University, Philadelphia PA, 1953.

[12] E. Kaltofen and P. Koiran. On the complexity of factoring bivariate supersparse (lacunary) polynomials. In *Proc. ISSAC'05*, pages 208–215. ACM, 2005.

[13] G. Kiczales, J. Lamping, and A. Mendhekar *et al.* Aspect-oriented programming. In *Proc. ECOOP*, pages 220–242. Springer LNCS 1241, 1997.

[14] Matthew Malenfant and Stephen M. Watt. Sparse exponents in symbolic polynomials. In *Proc. Symposium on Algebraic Geometry and Its Applications: in honor of the 60th birthday of Gilles Lachaud*. (to appear), 2007.

[15] James McKinna. Why dependent types matter. In *Proc. POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–1. ACM, 2006. Extended draft by Altenkirch, McBride, McKinna available on-line www.dcs.st-andrews.ac.uk.

[16] T. Millstein and C. Chambers. Modular statically typed multimethods. In *Proc. ECOOP*, pages 279–303. Sprinter LNCS 1628, 1999.

[17] J. Nolan. *Analytical differentiation on a digital computer*. MA Thesis, Math Dept MIT, Cambridge MA, 1953.

[18] C. Oancea and S.M. Watt. A framework for using aldor libraries with maple. In *Actas de los Encuentros de Algebra Computacional y Aplicaciones (EACA) 2004*, pages 219–224. Universitad de Cantabria, ISBN 84-688-6988-04, 2004.

[19] C. Oancea and S.M. Watt. Domains and expressions: An interface between two approaches to computer algebra. In *Proc. International Symposium on Symbolic and Algebraic Computation*, pages 261–268. ACM Press, 2005.

[20] Cosmin Oancea and Stephen M. Watt. Parametric polymorphism for software component architectures. In *Proc. 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 147–166. ACM Press, 2005.

[21] Cosmin E. Oancea and Stephen M. Watt. Generic library extension in a heterogeneous environment. In *Proc. Library Centric Software Design*, 2006.

[22] M. Odersky, V. Cremet, C. Röcl, and M. Zenger. A nominal theory of objects with dependent types. In *Proc. ECCOP'03*, pages 201–224. Sprinter LNCS 2743, 2003.

[23] W. Pan and D. Wang. Uniform Gröbner bases for ideals generated by polynomials with parametric exponents. In *Proc. ISSAC'06*, pages 269–276. ACM, 2006.

[24] L. Pottier. Generalisation de termes en theorie equationnelle. Cas associatif-commutatif. Technical Report RR-1056, INRIA, 1989.

[25] J. Sammet. A survey of formula manipulation. *C. ACM*, 9(8):555–569, 1966.

[26] A. Sexton and V. Sorge. Abstract matrices in symbolic computation. In *Proc. ISSAC'06*, pages 318–325. ACM, 2006.

[27] J. R. Slagle. A hueristic program that solves symbolic integration problems in freshman calculus. *J. ACM*, 10(4):507–520, 1963.

[28] D. R. Stoutemyer. Symbolic computer vector analysis. *Computers and Mathematics with Applications*, 5(1):1–9, 1979.

[29] Th. Sturm. New domains for applied quantifier elimination. In *Computer Algebra in Scientific Computing, CASC 2006*, pages 295–301. Springer, LNCS 4194, 2006.

[30] S.M. Watt. A study in the integration of computer algebra systems: Memory management in a maple-aldor environment. In *Proc. International Congress of Mathematical Software*, pages 405–410. World Scientific 2002, 2002.

[31] S.M. Watt. Aldor. In V. Weispfenning J. Grabmeier, E. Kaltofen, editor, *Handbook of Computer Algebra*, chapter 4.1.2, pages 265–270. Springer Verlag, Heidelberg 2003 ISBN 3-540-65466-6, 2003.

[32] S.M. Watt. Optimizing compilation for symbolic-numeric computing. In *Proc. 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computation*, pages 18–. MITRON Press ISBN 973-661-441-7, 2004.

[33] Stephen M. Watt. Algorithms for symbolic polynomials. In *Proc. 9th International Workshop on Computer Algebra in Scientific Computing*, pages 302–. Springer Verlag LNCS 4194, 2006.

[34] Stephen M. Watt. Making computer algebra more symbolic. In *Proc. Transgressive Computing 2006: A conference in honor or Jean Della Dora*, pages 43–49, 2006.

[35] Stephen M. Watt. Post facto type extension for mathematical programming. In *Proc. ACM SIGSAM/SIGSOFT Workshop on Domain-Specific Aspect Languages*, 2006.

[36] Stephen M. Watt. A technique for generic iteration and its optimization. In *Proc. ACM SIGPLAN Workshop on Generic Programming 2006*, pages 76–86. ACM, 2006.

[37] V. Weispfenning. Gröbner bases for binomials with parametric exponents. Technical report, Universität Passau, Germany, 2004.

[38] C. Oancea Y. Chicha, M. Lloyd and S.M. Watt. Parametric polymorphism for computer algebra software components. In *Proc. 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computation*, pages 119–130. MITRON Press ISBN 973-661-441-7, 2004.