Interfacing Coq + SSReflect with GAP

Vladimir Komendantsky[1]

School of Computer Science
St Andrews University
St Andrews, KY16 9SX, UK

Alexander Konovalov[1]

School of Computer Science
St Andrews University
St Andrews, KY16 9SX, UK

Steve Linton[1]

School of Computer Science
St Andrews University
St Andrews, KY16 9SX, UK

Abstract

We report on an extendable implementation of the communication interface connecting Coq proof assistant to the computational algebra system GAP using the Symbolic Computation Software Composability Protocol (SCSCP). It allows Coq to issue OpenMath requests to a local or remote GAP instances and represent server responses as Coq terms.

Keywords: Coq, GAP, Symbolic Computation Software Composability Protocol, OpenMath

1 Introduction

A theorem prover may significantly extend its functionality from the ability to communicate with computer algebra systems (CAS). Examples may include, but not limited to, retrieving objects from mathematical databases available in CAS, or computing results that can not be derived using the the theorem prover alone, but once known, may be verified in the prover or used as prover's

[1] {vk|alexk|sal}@cs.st-andrews.ac.uk

axioms for further proofs. Such combinations may not only speed up prover's work, but also allow getting results that can not be obtained within any single prover.

Developers of existing interfaces between theorem provers and CAS (we mention some of them in Sec. 8) may select various ways. For example, a prover may write CAS input files and then invoke it; the CAS will write prover's input to a file and exit; the prover will read it and perform further actions. This works, but has fairly serious limitations. A better setup might allow the prover to interact with other CAS while they run and provide a separate interface to each possible external CAS. However, achieving this is a major programming challenge, and an interface will be broken if the other system changes its I/O format, for example.

In the EU Framework VI project "SCIEnce – Symbolic Computation Infrastructure in Europe" (`www.symbolic-computation.org`) which is a major 5-year project that brings together CAS developers and experts in computational algebra, OpenMath, and parallel computations, a common standard interface that may be used for combining computer algebra systems and any other compatible software has been developed. It aims to provide an easy, robust and reliable way for users to create and consume services implemented in any compatible systems, ranging from generic services (e.g. evaluation of a string or an OpenMath object) to specialised (e.g. lookup in the database; executing certain procedure). This interface is in fact a lightweight XML-based remote procedure call protocol called *SCSCP* (*Symbolic Computation Software Composability Protocol*, [8]) in which both data and instructions are represented as OpenMath objects, what was an obvious choice as a common way of marshalling mathematical semantics. OpenMath [16] is a a well-established flexible language built from only twelve language elements (integers, doubles, variables, applications etc.). All the semantics is encapsulated in *symbols* which are defined in *Content Dictionaries* (CDs) and are strictly separate from the language itself. OpenMath was designed to be efficiently used by computers, and may be represented in several different encodings, the most commonly used of which is the XML representation.

SCSCP is now implemented in several computer algebra systems, including GAP [14], KANT, Macaulay2, Maple, MuPAD, TRIP (see [7,9] for details) and has APIs making it easy to add SCSCP interface to more systems. The advantage of this approach is that any system that implements SCSCP can immediately connect to all other systems that already support it. This avoids the need for special cases and minimises repeated effort. In addition, SCSCP allows remote objects to be handled by reference so that clients may work with objects of a type that do not exist in their own system at all. For example, to represent the number of conjugacy classes of a group, only knowledge of integers is required while knowledge of groups is not. The SCSCP protocol

(currently at version 1.3) is socket-based. It uses port number 26133, as assigned by the *Internet Assigned Numbers Authority* (IANA), and XML-format messages.

In the present paper we report about the prototype implementation of the SCSCP client in Coq which allows to send OpenMath requests from Coq to a local or remote GAP SCSCP server, receive back results and represent them as Coq terms. This implementation provides an extendable and flexible framework: In the future it may support more kinds of mathematical objects. Moreover, thanks to the SCSCP flexibility, remote procedures in specific applications may be designed in a way to avoid some OpenMath-related restrictions. Another direction may be adding features to Coq SCSCP server to allow handling requests from other applications.

We discuss an example involving computation in GAP of roots of a polynomial defined in SSReflect in Sec. 3. In Sec. 4 and Sec. 5, we give a presentation of two features of Coq that lay behind the algebraic hierarchy of SSReflect: coercions [19] and canonical structures [20]. In Sec. 6, we describe our approach according to which the user takes care about programming the "interface" for calls from Coq + SSReflect to GAP in the sense that they specify how OpenMath requests to GAP are formed from data in Coq and how incoming OpenMath objects will be translated to Coq terms. In particular, OpenMath objects are constructed using the subclass information given by coercions, while synthesis of a Coq term from a given OpenMath object re-uses the canonical structure mechanism. We allow for a manual definition of Coq terms that provide helper information for the automated tactic that performs data exchange between the two systems.

2 Translation from OpenMath to Coq

Translation from XML data respecting the DTD of the Predicative Calculus of (Co-)Inductive Constructions (pCic; see [20], Chapter 4) to Coq can be straightforward although routine [1]. On the contrary, there seems to be a central problem with the representation of data sent by the computer algebra system such as GAP [5,10]. Namely, the objects of the pCic are terms; even proofs are terms constructed from smaller terms. The question is how to interpret arbitrary non-pCic data that can come packaged in OpenMath objects. Such data may not correspond to terms for the reason that computer algebra data are not in general *constructed* but rather *given*, and we cannot establish a constructive proof of how the values in the OpenMath object were obtained.

One possible approach to such a representation would be to translate OpenMath data as axioms or values (the latter is the case of the command `coq_gap` in Sec. 8 due to S. Ould-Biha [17]). Therefore proofs of how values were obtained are not needed. An extension of Coq with such a facility does not add

convenience to proofs but rather presents another way to call a GAP server. If calls to GAP are made possible from inside the Coq proof mode, it can be a handy shortcut allowing to succeed with some proofs without respecting the constructivity requirement, that is, such proofs would be based on axioms.

One can formally assign mathematical meaning to objects of Strong OpenMath, a strict subset of OpenMath. Hence there is a chance that Strong OpenMath objects can be mapped to the pCic by means of formal analysis. However, we do not see a way to restrict or convert to Strong OpenMath the set of objects generated by GAP.

To faithfully translate a given OpenMath object to the pure pCic with no axioms one needs to represent that object in terms of the pCic. For example, if GAP computes a group of permutations, one must have a constructive definition of a group of permutations in the pCic rather than arrays of numbers that would likely be the external GAP representation of such a group. We are inclined to think that this is possible to satisfy at least partially if we take group-theoretic definitions that already exist in SSReflect [11]. However, many constructive objects of that kind depend on proofs (of algebraic properties, for instance), which renders a fully automated translation impossible. Therefore some form of user interaction is still required.

SSReflect is based on constructive solvability of problems over finite domains using two-valued boolean logic (as opposed to infinite-valued constructive one). Thus one can reduce a finite problem of sort `Prop` to an equivalent one of type `bool`, with the notion of reduction provided by the reflection relation below:

```
Inductive reflect (P : Prop) : bool -> Set :=
  | ReflectT : P -> reflect P true
  | ReflectF : ~P -> reflect P false.
```

This has a profound impact since this allows to reduce the complexity of proofs in the library of finite algebraic structures [11].

3 Working example: roots of a polynomial

In SSReflect 1.2, the definition of a polynomial over a ring is the following:

```
Record polynomial (R : ringType) : Type :=
  Polynomial {polyseq :> seq R; _ : last 1 polyseq != 0}.
```

A polynomial can then be seen as a sequence with non-zero last element. The coercion `polyseq` of type ∀ `R` : `ringType`, `polynomial R` → `seq R` takes polynomials over a ring to sequences (which are isomorphic to plain lists) on that ring.

For example, considering arithmetic modulo natural n on the finite set $\{0, 1, \ldots, n-1\}$, a polynomial

$$p(x) = x^3 - 1$$

over the field $\mathbb{Z}/3\mathbb{Z}$ is constructed using the field type `Fp_field` (inheriting from `ringType`) defined in the library `zmodp.v` of modular arithmetic. The polynomial $p(x)$ can be coerced into a sequence of ordinals which, in turn, can be coerced into a sequence of natural numbers

```
Cons 2 (Cons 0 (Cons 0 (Cons 1 Nil)))
```

On the other side, to construct such polynomial in GAP and find its roots one should perform the following steps:

```
gap> x:=Indeterminate(GF(3),"x");;
gap> f:=x^3-1;
x^3-Z(3)^0
gap> RootsOfUPol(f);
[ Z(3)^0, Z(3)^0, Z(3)^0 ]
```

We will return to this example in Sec. 6 and 7 to illustrate SCSCP procedure calls issued by Coq and corresponding responses from the GAP server.

4 Type inference with subclasses

In a type-theoretic proof system, type-checking is a problem of deciding whether a typing judgement is derivable according to the rules of the system. Although this problem is undecidable in general, it is decidable for most systems of interest, in particular, for injective ones [4]. In proof assistants, this job is delegated to the module known as *type-checker*.

Another related problem is type inference. It consists of inferring types that have not been explicitly specified by the user, that is, synthesising or constraining omitted subterms in such types. We are particularly interested in the ability of type theoretic proof assistants to recognise mathematical abuse of notations which is quite handy in situations when the same mathematical object has to be viewed at different levels (e.g., in a hierarchical implementation of a mathematical theory such as SSReflect). This ability is provided by the module known as *refiner*.

The *subtype relation* $\leq$ is derived in the pCic as a proof convenience. It can be characterised using the rule below:

$$\frac{E[\Gamma] \vdash t : A \qquad E[\Gamma] \vdash c_{A,B} : A \leq B}{E[\Gamma] \vdash (c_{A,B} t) : B}$$

This can be defined in terms of the pCic [19]. An *inheritance class* (class)

is either a term of type $\forall\,(x_1 : A_1)\ldots(x_n : A_n),\ s$ with n parameters or one of the abstract inheritance classes `SORT` and `FUN`, the classes of sorts and functions respectively. A partial function ClassOf from terms to inheritance classes is defined as follows:

$$\begin{array}{lcl}
\mathsf{ClassOf}\ s & = & \texttt{SORT} \\
\mathsf{ClassOf}\ (\forall\, x : A,\ B) & = & \texttt{FUN} \\
\mathsf{ClassOf}\ (C\ t_1 \ldots t_n) & = & C \text{ if } C \text{ is a class with } n \text{ parameters} \\
 & & \text{undefined otherwise}
\end{array}$$

Given classes C and D with n and m parameters respectively, a term f can be declared as a *coercion* with domain C and codomain D, denoted $f : C \rightarrowtail D$, if its type has form

$$\forall\ (x_1 : A_1)\ldots(x_n : A_n)\ (y : (C\ x_1 \ldots x_n)),\ (D\ u_1 \ldots u_m)$$

and C is neither `SORT` nor `FUN`. Let $t : C\,t_1 \ldots t_n$ be a well-typed term inhabiting the class C. We can define the application of f to t as $f@t = f\,t_1 \ldots t_n\,t$. The type of this application is denoted $f\{t\}$ and is $D\,u'_1 \ldots u'_m$ where $u'_i = u'_i[y := t][x_n := t_n]\ldots[x_1 := t_1]$, for $1 \leq i \leq m$.

A *class inheritance graph* Δ has classes as nodes and coercions between those classes as edges. A *coercion path* is given by the composition of k elementary coercions, for $k \geq 0$, that is, $f_1 \circ \cdots \circ f_k$. A class C is said to be a *subclass* of D in Δ whenever there is a coercion path in Δ from C to D. One also says that C *inherits* from D. The graph is represented as a list of coercions; the classes and paths are inferred.

The type-checking algorithm with inheritance takes as input some environment E, context Γ, coercion graph Δ and a term t and outputs the *explicit term* t' and T such that $t' : T$ obtained by application of appropriate coercions to t (which is also called the *implicit term*). A typing judgement for this can be written $E[\Gamma]\Delta \vdash t \Rightarrow t' : T$.

In order to transform a implicit terms to explicit ones, an algorithm is defined in [19] to insert appropriate coercions. The function is applied only if the term in question is not in the explicit form. The algorithm is partial and fails if case match is not successful. The following properties of the typing algorithm are known:

- *correctness* of the algorithm: If $E[\Gamma]\Delta \vdash t \Rightarrow t' : t'$ then t is a well-typed term and T' is its inferred type;
- *conservativeness* of extension with respect to implicit typing judgements $\_ \vdash_I \_$: If $E[\Gamma] \vdash_I t : T$ then $E[\Gamma]\Delta \vdash t \Rightarrow t : T$ for all coercion graphs Δ.

5 Canonical structure hints

A general approach to hints in unification was introduced in [2,18]. A general *unification hint* has kind

$$\frac{?_{x_1} := H_1 \qquad \ldots \qquad ?_{x_n} := H_n}{P \equiv Q}$$

where $P \equiv Q$ is a type equivalence pattern with free variables $\mathbf{?_v}$ such that $\{?_{x_1}, \ldots, ?_{x_n}\} \subseteq \mathbf{?_v}$, all of $?_{x_i}$ being distinct, and H_i cannot depend on any of the pattern variables $?_{x_j}$ for $1 \leq i \leq j \leq n$.

In Coq, there is a dedicated mechanism to deal with unification problems of the specific kind

$$\pi\ ?_1\ \ldots ?_n \equiv t$$

where π is one of the projectors created by the declaration of a record type, $?_1, \ldots, ?_n$ (partially) unknown arguments and t the known value of this projection applied to the arguments. This mechanism is provided by canonical structures. A *canonical structure* [20] is a purposely marked instance of a record (that possibly contains functional abstractions at the topmost level). The type checker employs such marked instances when attempting to solve unification problems. This can be expressed in a rule below:

$$\frac{\pi = \mathsf{HeadConstant}\ t \qquad ?_1 := t_1 \qquad \ldots \qquad ?_n := t_n}{\pi\ ?_1\ \ldots ?_n \equiv t}$$

where $t_1, \ldots, t_n$ are terms that appear as arguments of the head constant of t. The actual implementation in Coq has further aspects such as unification strategies (e.g., delayed expansion of defined constants) and treatment of functional records.

The following example quotes the standard equality type of SSReflect, whose definition follows the SSReflect class-mixin design pattern, and then declares unification hints for the equality structure on the standard type `nat`:

```
Module Equality.

Definition axiom T e := forall x y : T, reflect (x = y) (e x y).
Structure mixin_of (T : Type) : Type := Mixin {
  op : rel T;
  _  : axiom op}.
Notation class_of := mixin_of (only parsing).
Structure type : Type := Pack {
  sort :> Type;
  _    : class_of sort;
  _    : Type}.
```

```
...
Definition pack T c := @Pack T c T.

End Equality.

Definition eq_op T := Equality.op (Equality.class T).
Lemma eqnP : Equality.axiom eqn. ... Qed.

Canonical Structure nat_eqMixin := Equality.Mixin eqnP.
Canonical Structure nat_eqType :=
  Eval hnf in Equality.pack nat_eqMixin.
```

Thanks to the declaration of `nat_eqType`, the notation `@eq_op _ 0 1`, with the implicit type argument being omitted, will be typed as

```
@eq_op nat_eqType 0 1
```

The latter is $\beta\iota\delta$-convertible with `eqn 0 1` where `eqn` is a boolean equality on type `nat` defined in SSReflect.

6 User interface

Suppose we are submitting a request to the GAP server to compute the roots of the polynomial in Sec. 3. The process of obtaining the OpenMath object for the polynomial is rather straightforward thanks to the coercion graph. In this object, one has powers of primitive elements of the finite field which are essentially OpenMath integers. To represent them, the user provides the following mappings:

- from relevant SSReflect field operations such as ring exponentiation (inherited by the field structure) to pairs consisting of the appropriate content dictionary and dictionary field (in the case of exponentiation, `arith1` and `power` respectively);
- from SSReflect ordinals to OpenMath integers (trivially by providing the coercion from ordinals to the type `nat`).

A similar mapping from a polynomial to an OpenMath dictionary field should also be provided. Mappings are represented by Coq terms with names of OpenMath dictionaries and dictionary fields being qualified identifiers defined in Coq. These identifiers are processed by the tactic and respective string values are obtained for XML field names.

Having sent a request for the polynomial from the example in Sec. 3 to the GAP server as we describe in Sec. 7, the tactic should receive a response containing the following object, corresponding to the list of three multiplicative neutral elements of the field $\mathbb{Z}/3\mathbb{Z}$:

```
<OMOBJ>
  <OMA>
    <OMS cd="list1" name="list"/>
    <OMA>
      <OMS cd="arith1" name="power"/>
      <OMA>
        <OMS cd="finfield1" name="primitive_element"/><OMI>3</OMI>
      </OMA>
      <OMI>0</OMI>
    </OMA>
    ...
    <OMA>
      <OMS cd="arith1" name="power"/>
      <OMA>
        <OMS cd="finfield1" name="primitive_element"/><OMI>3</OMI>
      </OMA>
      <OMI>0</OMI>
    </OMA>
  </OMA>
</OMOBJ>
```

In order to represent this as a Coq term, the user should have defined translation rules for the corresponding entries in the OpenMath content dictionaries in Coq. In general, representation of an OpenMath object in Coq depends very much on the context. Therefore it is quite impossible to have a fixed set of translation rules. Instead, we should provide means to the user to define appropriate mappings.

For example, suppose that the context requires to represent this object as a `seq` containing elements of the field $\mathbb{Z}/3\mathbb{Z}$. For that purpose we can use the library `zmodp.v` of modular arithmetic, where such a sequence will be typed `seq (Fp_field pp)`, for `p:nat` and `pp:prime p`. The OpenMath entity `list` will be mapped to the type `forall (T:Type), seq T`. The user must then fulfil the proof obligation `prime p`, for the given `p`.

7 Tactic implementation

We are developing a tactic implementation for data exchange between Coq and the GAP server, where the latter should be started using GAP packages SCSCP and OpenMath [6,14]. The tactic is being implemented as a Coq plug-in module, with a possibility of dynamic loading by a Coq command language request. The main client function initiates a TCP/IP socket connection, performs the handshake according to the SCSCP specification and evaluates the client callback function. The latter function composes and sends through the output I/O channel the SCSCP request using data provided as arguments to the tactic, flushes the output channel and receives back the SCSCP packet containing server response. All SCSCP packets consist of a sequence of OpenMath XML objects and are enclosed in mandatory `start` and `end` tags. Therefore receipt amounts to reading from the input channel all the OpenMath data between these tags and parsing the data.

Request and response packages have similar structure. For example, a

request to find the roots of the polynomial from Sec. 3 may have the form

```
<OMOBJ>
  <OMATTR>
    <OMATP>
      <OMS cd="scscp1" name="call_id"/><OMSTR>host:port:pid:string</OMSTR>
    </OMATP>
    <OMA>
      <OMS cd="scscp1" name="procedure_call"/>
      <OMA>
        <OMS cd="scscp_transient_1" name="WS_RootsOfUpol"/>
        <OMA>
          <OMS cd="polyd1" name="DMP"/>
          <OMS cd="polyd1" name="poly_ring_d_named"/>
          <OMA><OMS cd="setname2" name="GFp"/><OMI>3</OMI></OMA>
          <OMV name="x"/>
        </OMA>
        <OMA>
          <OMS cd="polyd1" name="SDMP"/>
          <OMA>
            <OMS cd="polyd1" name="term"/>
            <OMA>
              <OMS cd="arith1" name="power"/>
              <OMA>
                <OMS cd="finfield1" name="primitive_element"/><OMI>3</OMI>
              </OMA>
              <OMI>0</OMI>
            </OMA>
            <OMI>3</OMI>
          </OMA>
          <OMA>
            <OMS cd="polyd1" name="term"/>
            <OMA>
              <OMS cd="arith1" name="times"/>
              <OMA>
                <OMS cd="finfield1" name="primitive_element"/><OMI>3</OMI>
              </OMA>
              <OMI>0</OMI>
            </OMA>
            <OMI>2</OMI>
          </OMA>
          <OMA>
            <OMS cd="polyd1" name="term"/>
            <OMA>
              <OMS cd="arith1" name="times"/>
              <OMA>
                <OMS cd="finfield1" name="primitive_element"/><OMI>3</OMI>
              </OMA>
              <OMI>0</OMI>
            </OMA>
            <OMI>1</OMI>
          </OMA>
          <OMA>
            <OMS cd="polyd1" name="term"/>
            <OMA>
              <OMS cd="arith1" name="power"/>
              <OMA>
                <OMS cd="finfield1" name="primitive_element"/><OMI>3</OMI>
              </OMA>
              <OMI>1</OMI>
            </OMA>
            <OMI>0</OMI>
          </OMA>
        </OMA>
      </OMA>
    </OMA>
  </OMATTR>
</OMOBJ>
```

Such objects always carry a call ID that has to be the same both in requests and responses. The server provides the most of this ID in the handshake tag at the start of the connection, the part known as *service ID*: the host, the port and the process ID. The client then chooses a random string and appends it to the service ID to form the call ID. This call ID is then checked for every received OpenMath object.

8 Related work

Harrison and Théry experimented with data exchange between the theorem prover HOL and the computer algebra system Maple [12,13], especially that involving powerful although obfuscated rewriting techniques implemented in Maple and not in HOL. The authors brought forwards the concept of a *degree of trust* of a prover to a computer algebra program. This degree was supposed to reflect the general attitude towards interpreting computational values returned by the computer algebra system. Remarkably, already in the case of HOL, the most appropriate degree of trust was the least one, that is, "no trust at all". This choice was motivated, first, by correctness considerations and, second, by constraints implicit in the term structure of the theorem prover. Since computational values had to be re-assembled as HOL terms anyway, it was perfectly sensible to issue some correctness proof goals and delegate them to the user of HOL. We follow a similar approach, making the dependency of terms on values and proofs slightly more explicit.

More recently, S. Ould-Biha wrote in C a limited external tactic [2] `coq_gap` using the library `xml2` that executes the GAP interpreter in a quiet mode and communicates via Unix pipes [17]. An example of a Coq script could be the following:

```
Definition gap_fun : nat -> nat.
intro n.
let x := external "coq_gap" "Fun" n in exact x.
Defined.
```

The third line is an existential proof that `gap_fun n` is a natural number. Therefore `gap_fun n` equals `x` that equals the string-to-natural converted value of the call to `Fun(n);` in the GAP interpreter. In fact, due to some internal limitations such as the restriction to functions of type `nat` $\rightarrow$ `nat`, it is difficult to see how this approach might be effectively generalised onto functions of more general type.

The tactic `external` [20] is ubiquitous in Coq interfaces. It's purpose is to run an executable outside the Coq executable. The syntax is the following:

[2] The authors of [13] might call such a tactic a *bridge*.

```
external "command" "request" arg_1 ... arg_n
```

An XML tree of the following form is sent to the standard input of the external command:

```
<REQUEST req="request">
the XML tree of the first argument
...
the XML tree of the last argument
</REQUEST>
```

The external command must send on its standard output an XML tree of the following form:

```
<TERM>
the XML tree of a term
</TERM>
```

or, if the response is a tactic call rather than a term,

```
<CALL uri="ltac_qualified_ident">
the XML tree of the first argument
...
the XML tree of the last argument
</CALL>
```

where `ltac_qualified_ident` is the name of a defined function in the Coq tactic language, and each XML subtree is recursively a `CALL` or a `TERM` node.

Based on the `external` infrastructure, there is another Maple interface for Coq due to H. Herbelin that updates an earlier version due to M. Mayero and D. Delahaye [15]. This interface relies on `external` in a similar way to S. Ould-Biha's tactic.

The tactic `external` requires an intermediate XML representation of external data before they are interpreted in Coq. This is uniform but not necessarily efficient in cases where intermediate data do not conform with the pCic DTD and can be passed to Coq naturally by other means. It this respect, everyday examples of communication with Coq such as native CoqIDE [20] and ProofGeneral [3], an Emacs front-end for proof assistants, can provide inspiration in a way communication data are represented there.

9 Conclusions

We discussed a design pattern that can be employed to interpret OpenMath data as objects in the SSReflect library of Coq. We are working on an implementation of this design pattern in a Coq tactic. At the moment the tactic has essential support for SCSCP I/O transport functions. More work is needed on

interpretation of OpenMath data received by Coq from GAP since the current implementation offers itself only to rendering the GAP response verbatim or at least close to verbatim. Such verbatim rendering is expected to become usable by June, 2010. The tactic will be maintained on the SCIEnce project website [3]. There are more related questions that we would like to address in the observable future, including use of GAP as a search engine for SSReflect, helping to locate useful group-theoretic objects or check properties without actually defining search algorithms in Coq.

References

[1] Asperti, A., L. Padovani, C. Sacerdoti Coen and I. Schena, *HELM and the semantic Math-Web*, in: *Proc. TPHOLS 2001*, LNCS **2512** (2001).

[2] Asperti, A., W. Ricciotti, C. Sacerdoti Coen and E. Tassi, *Hints in unification*, in: *Proc. TPHOLs 2009*, LNCS **5674**, Munich, 2009, pp. 84–98.

[3] Aspinall, D., *ProofGeneral*, http://proofgeneral.inf.ed.ac.uk/.

[4] Barthe, G., *Type-checking injective pure type systems*, J. Functional Programming **9** (1999), pp. 675–698.

[5] Breuer, T. and S. Linton, *The GAP 4 type system: organising algebraic algorithms*, in: *ISSAC '98: Proceedings of the 1998 international symposium on Symbolic and algebraic computation* (1998), pp. 38–45.

[6] Costantini, M., A. Konovalov and A. Solomon, "OpenMath – OpenMath functionality in GAP, Version 10.1," (2010), GAP package, http://www.cs.st-andrews.ac.uk/~alexk/openmath.htm.

[7] Freundt, S., P. Horn, A. Konovalov, S. Lesseni, S. Linton and D. Roozemond, *OpenMath in SCIEnce: Evolving of symbolic computation interaction*, in proceedings of OpenMath Workshop 2009 (to appear).

[8] Freundt, S., P. Horn, A. Konovalov, S. Linton and D. Roozemond, *Symbolic Computation Software Composability Protocol (SCSCP) specification*, http://www.symbolic-computation.org/scscp, Version 1.3, 2009.

[9] Freundt, S., P. Horn, A. Konovalov, S. Linton and D. Roozemond, *Symbolic computation software composability*, in: *AISC/MKM/Calculemus, Springer LNCS 5144*, 2008, pp. 285–295.

[10] The GAP Group, "GAP – Groups, Algorithms, and Programming, Version 4.4.12," (2008), http://www.gap-system.org.

[11] Garillot, F., G. Gonthier, A. Mahboubi and L. Rideau, *Packaging mathematical structures*, in: *Theorem Proving in Higher Order Logics (2009)*, LNCS **5674**, 2009.

[12] Harrison, J. and L. Théry, *Reasoning about the reals: the marriage of HOL*, in: A. Voronkov, editor, *Logic programming and automated reasoning: proceedings of the 4th international conference, LPAR '93*, Lecture Notes in Computer Science **698** (1993).

[13] Harrison, J. and L. Théry, *A skeptic's approach to combining HOL and Maple*, Journal of Automated Reasoning **21** (1998), pp. 279–294.

[14] Konovalov, A. and S. Linton, "SCSCP – Symbolic Computation Software Composability Protocol, Version 1.2," (2010), GAP package, http://www.cs.st-andrews.ac.uk/~alexk/scscp.htm.

[3] http://www.symbolic-computation.org/Downloads, starting from June/July, 2010

[15] Mayero, M. and D. Delahaye, *A Maple mode for Coq*, http://coq.inria.fr/contribs/MapleMode.html.

[16] *OpenMath*, http://www.openmath.org/.

[17] Ould-Biha, S., *Finite groups representation theory with Coq*, in: *8th International Conference on Mathematical Knowledge Management (2009)*, 2009.

[18] Sacerdoti Coen, C. and E. Tassi, *Working with mathematical structures in type theory*, in: *Proc. TYPES 2007*, LNCS **4941**, Cividale del Friuli, Udine, Italy, 2007, pp. 157–172.

[19] Saïbi, A., *Typing algorithm in type theory with inheritance*, in: *Proc. POPL'97*, 1997, pp. 292–301.

[20] The Coq development team, *The Coq proof assistant reference manual*, http://coq.inria.fr/refman/.