Packed views of pre-structured data

Vladimir Komendantsky*
School of Computer Science
University of St Andrews
St Andrews
KY16 9SX, UK
vk10@st-andrews.ac.uk

We propose a technique of packed view for interactive communication between heterogeneous software such as theorem provers and computer algebra systems. A packed view is a method to infer, with a possible share of user interaction, the type-theoretic meaning of a given pre-structured object, as well as a method to display the underlying syntactic structure of a semantic term as a pre-structured object (essentially, an abstract syntax tree) corresponding to it. With this notion of view in hand, it is relatively straightforward to program non-trivial parts of communication interfaces between a theorem prover and a computer algebra using the logic of the prover as the programming language.

1 Introduction

In scientific as well as in educational applications there is a common problem of communication between different components of heterogeneous software. For example, we may consider communication links between two theorem provers, in which case we are likely interested in communicating a proof goal and its context, or a theory while preserving as much logic contents of terms as possible. A rather different kind of situation arises when a theorem prover is communicating with a non-logic based computational system. Such systems operate according to their own, possibly informal semantics, and may accept communications in a structure-oriented format based on the XML technology. A circumstance that we can use to our benefit is that the type of communicated structure depends on the domain of application of a given program rather on particularities of implementation of that program, such as operational semantics, whether it is formal or not. Therefore a semi-formal specification of the communication format should exist. Existence of a specification means that the type of communicated structure, or at least its faithful subpart, can be presented as an inductive datatype in a prover.

Given an inductive datatype representing the class of communicated abstract syntax trees (ASTs) recognised by the prover, we can think about an embedding of these ASTs into the logic of the prover. For such an embedding, we can consider theories or type-theoretic hierarchies of mathematical objects. It is sensible to allow this embedding to be partial since not all syntactic expressions have a meaning and therefore not all possible ASTs can be rendered as objects in the corresponding theory.

The inductive type of AST is our type of *pre-structured data*. Pre-structuring can be done at the level of implementation language of the prover. It essentially consists of taking a representation of an AST in an XML-based format as input, parsing it, and producing a well-typed term of the prover's type of AST, and vice versa. All these operations are rather straightforward and only require some input/output capabilities. The most interesting and important part is to assign meaning to ASTs in the type theory of the prover by defining functions that take ASTs as input and construct objects in the respecting theory.

*Supported by the research fellowship EU FP7 Marie Curie IEF 253162 'SImPL'.

Submitted to:
THedu'11

At the pre-structuring stage, nothing is known about the meaning of a given AST, even whether such a meaning exists at all. By finding a meaning for the AST, we view the syntactic object corresponding to it as a semantic object. Assignment of a meaning to an object is done by a semantic interpretation function. The converse process performed by a display function that translates a semantic object to the syntactic form for the purpose of displaying it either to a user or to another program. The former process infers the semantic interpretation of an object, while the latter process forgets it.

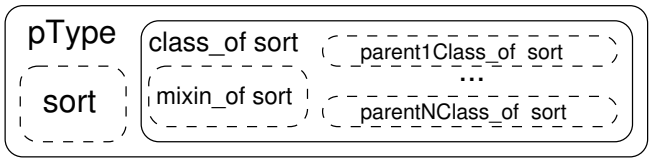
Contribution. We introduce a notion of packed view of a pre-structured object. A packed view is a method to infer, with a significant share of user interaction, the type-theoretic meaning of a given pre-structured object, as well as a method to display the underlying syntactic structure of a semantic term as a pre-structured object (essentially, an abstract syntax tree) corresponding to it. With this notion of view in hand, it is relatively straightforward to program non-trivial parts of communication interfaces between a theorem prover and a computer algebra system inside the theorem prover itself, as an alternative to having a complicated and difficult to extend interface at the level of implementation language of the theorem prover.

Outline. This short paper discusses generic packed views in Section 2. An example of pre-structured data, a subtype of the OpenMath standard, is given in Section 3. Our conclusions are contained in Section 4.

Notational conventions. Our meta-language, Coq [10], has dependent products of the form $\forall\ (x : A).\ B$ where x is a variable which is bound in B; the case when x is not free in B is denoted $A \to B$ which is a simple, non-dependent type. Also, Coq features inductive and coinductive type definitions. For the sake of presentation, we do not provide listings of Coq code in Sec. 2. We use a human-oriented type-theoretic notation, where $\star$ denotes the predicative universe of types, Type, and inductive and coinductive definitions are displayed in natural-deduction style with single and, respectively, double lines.

Related work. Views were considered by Wadler in [11] as a method to convert from concrete data with free structure to abstract data. The problem was to provide a formal concept of definitional correspondence between structured data that allows pattern matching and data with any structure being missing, e.g., for the purpose of efficiency. The user of a programming language supporting views receives a method to access features specific either to concrete or to abstract data (resp., pattern matching and representation hiding) working with one and the same object by viewing that object as either concrete or abstract.

Our method is based on the design pattern of *packed types* introduced in [3]. A packed type pType can be depicted as follows, in an object-oriented programming style:



Enclosed in boxes are types. Dashed borders indicate a type to which the containing type can be coerced. A *packed class* is a container inductive structure supplied with pack and unpack operations that, together with unification hints implemented in Coq (that is, implicit coercions and canonical structures), allows to be organised in a consistent and extendible hierarchy of structures and has been employed in [3, 5] to define a mathematical type hierarchy. The authors of [3] mentioned that, unlike the straightforward telescopic model of inheritance, packed types allow natural multiple inheritance and a reduced size of terms thanks to putting separate components of mathematical structures in records called *mixins*, and reducing structure nesting depth in general. The latter is crucial as a workaround to cope with the type inference algorithm of Coq that is exponential in the size of term which, if hierarchies are concerned,

should be of the order C^n, where C is the number of the components of the structure, and n is the structure nesting depth. There is also a current effort to apply packed types to theorem proving automation in [4].

The problem of finding a meaning of pre-structured data in Coq was previously approached in the context of requesting computer algebra system computations from the prover in [8, 7]. Other approaches to computer algebra are also practised in the theorem proving community. There are two major related trends which however have quite different aims to ours: 1) building a computer algebra system on top of a proof assistant [6, 1], and 2) creating a programing environment for development of certified computation [2]. Unlike any of these developments, we do not approach the nature of computational algorithms but rather concern ourselves with the task of representing given computer algebra data in a way acceptable in a theorem prover.

2 A view

The base type of view. *Mixins* are introduced as elementary building blocks for containers such as viewType below. A mixin is an inductive type together with projections to its constructors. Below is the base view mixin for a type of pre-structured data called OM:

$$\mathsf{mixin\_of} : \star \to \star$$

$$\frac{\mathit{viewin} : \mathsf{OM} \to \mathsf{option}\ \mathit{iT} \qquad \mathit{viewout} : \mathit{iT} \to \mathsf{option}\ \mathsf{OM}}{\mathsf{ViewMixin}\ \mathit{viewin}\ \mathit{viewout} : \mathsf{viewMixin\_of}\ \mathit{iT}}$$

Here, option is the polymorphic type of fan ordering with values either of the kind Some of a value of a given type, or None, with the latter representing the absence of translation. Field projections are the following:

$$\begin{aligned} \mathsf{viewin} &= \lambda\ (\mathit{iT} : \star)\ (m : \mathsf{viewMixin\_of}\ \mathit{iT}).\ \mathbf{let}\ (\mathit{viewin},\ \_) := m\ \mathbf{in}\ \mathit{viewin} \\ \mathsf{viewout} &= \lambda\ (\mathit{iT} : \star)\ (m : \mathsf{viewMixin\_of}\ \mathit{iT}).\ \mathbf{let}\ (\_,\ \mathit{viewout}) := m\ \mathbf{in}\ \mathit{viewout} \end{aligned}$$

The *class* of the base type is simply its mixin.

$$\mathsf{viewClass\_of} = \mathsf{viewMixin\_of}$$

In the packed type methodology, containers $*$type pack a given representation type with given base classes and the underlying mixin. In the base case, we have only a mixin. The container viewType is given below. The third argument U is required for the purpose of unification.

$$\mathsf{type} : \star$$

$$\frac{\dfrac{\mathit{viewSort} : \star}{\mathit{viewClass} : \mathsf{viewClass\_of}\ \mathit{viewSort}} \qquad U : \star}{\mathsf{ViewPack}\ \mathit{viewSort}\ \mathit{viewClass}\ U : \mathsf{viewType}}$$

Now we have to provide a hint for the unification algorithm that allows to coerce viewType to its representation type. This function is delegated to the field projection viewSort.

$$\mathsf{viewSort} = \lambda\ (t : \mathsf{viewType}).\ \mathbf{let}\ (\mathit{viewSort},\ \_,\ \_) := t\ \mathbf{in}\ \mathit{viewSort}$$

The second field projection, viewClass, is not associated with a coercion.

$$\begin{aligned} \mathsf{viewClass} &: \forall\ \mathit{cT} : \mathsf{viewType}.\ \mathsf{viewClass\_of}\ (\mathsf{viewSort}\ \mathit{cT}) \\ \mathsf{viewClass}\ \mathit{cT} &= \mathbf{let}\ \mathsf{ViewPack}\ \_\ c\ \_ := \mathit{cT}\ \mathbf{in}\ c \end{aligned}$$

Finally, we define the constructor ViewType for the type viewType, and input and output view functions:

$$\begin{aligned}\mathsf{ViewType}\ T\ m &= \mathsf{ViewPack}\ T\ m\ T\\ \mathsf{In} &= \lambda\ (iT : \mathsf{viewType}).\ \mathsf{viewin}\ iT\ (\mathsf{viewClass}\ iT)\\ \mathsf{Out} &= \lambda\ (iT : \mathsf{viewType}).\ \mathsf{viewout}\ iT\ (\mathsf{viewClass}\ iT)\end{aligned}$$

Higher-level types. The base type of view can be extended, for example, with a notion of correctness of a view for the purpose of certification of computer algebra computations, or by adding other kinds of conversion functions for extended usability. Construction of view types given other view types is reminiscent of construction of objects in object-oriented programming, and can be depicted diagrammatically. A generic diagram is shown in Introduction. The main difference with the base type of view is in the class of the higher-level type that can contain a number of mixins. In the extended class, the respective field projections coerce it to the base classes and the underlying mixin but *not* to the representation type iT, which facilitates multiple inheritance.

3 Instantiated views

In this section we consider a concrete type of semi-structured data. This is a datatype called OM representing a certain subset of the OpenMath standard[9]. The definition is by induction as follows:

$$\mathsf{OM} : \star$$

$$\frac{x : \mathbb{Z}}{\mathsf{OMInt}\ x : \mathsf{OM}} \qquad \frac{s : \mathsf{string}}{\mathsf{OMVar}\ s : \mathsf{OM}} \qquad \frac{s_1\ s_2 : \mathsf{string}}{\mathsf{OMSym}\ s_1\ s_2 : \mathsf{OM}} \qquad \frac{o : \mathsf{OM} \quad os : \mathsf{list\ OM}}{\mathsf{OMApp}\ o\ os : \mathsf{OM}}$$

The above definition can be programmed as an inductive datatype in Coq:

```
Inductive OM : Type := OMInt : Z → OM | OMVar : string → OM
  | OMSym : string → string → OM | OMApp : OM → list OM → OM.
```

Polymorphism. Given a canonical structure T : viewType of view, we can construct polymorphic views. For example, a view for polymorphic lists can be constructed below. The *canonical structure* list_viewType is a unification hint that allows to unify function arguments of type list T with the view of lists and hence construct views by unification.

```
Fixpoint om_list : OM → option (list T) := ...
Fixpoint list_om : list T → option OM := ...
Definition list_viewMixin := ViewMixin _ om_list list_om.
Canonical Structure list_viewType := Eval hnf in ViewType (list T) list_viewMixin.
```

Dependent types. Dependent types can be treated using the inductive type of dependent pairs in Coq, which allows to delay proof construction until the stage of interactive proof. For example, one of possible representations of polynomials (a list of coefficients with a proof that the last element is not 0, unless the list is empty) can be treated as follows, again, for a given T : viewType. For this, however, we also provide a mapping from the type poly T of polynomial over T to the corresponding dependent pair type for the view to be useful in a context.

```
Fixpoint om_poly : OM → option {c : list T & last 1 c ≠ 0 → poly T} := ...
Fixpoint poly_om : {c : list T & last 1 c ≠ 0 → poly T} → option OM := ...
Definition poly_viewMixin := ViewMixin _ om_poly poly_om.
Canonical Structure poly_viewType := Eval hnf in ViewType _ poly_viewMixin.
```

4 Conclusions

Views can be constructed piecewise by providing views for 1) simple types, 2) polymorphic types and 3) dependent types, and then providing hints for the unification algorithm of the theorem prover to infer the type of view given the type of argument of functions In and Out that perform viewing.

The views paradigm can be especially useful in the prototype Coq-to-GAP communication interface currently available for download at `http://www.cs.st-andrews.ac.uk/~vk/Coq+GAP/`. One of possible projects can be connected with visualisation of the object-oriented structure of packed types outside the theorem prover, and interacting with the communication between the prover and the computer algebra tool by means of graphical construction of views.

References

[1] Yves Bertot, Frédérique Guilhot & Assia Mahboubi (2011): *A formal study of Bernstein coefficients and polynomials*. *Mathematical Structures in Computer Science* Available at `http://hal.inria.fr/inria-00503017/en/`.

[2] S. Boulmé, T. Hardin, D. Hirschkoff, V. Ménissier-Morain & R. Rioboo (1999): *On the way to certify Computer Algebra Systems*. *Electronic Notes in Theoretical Computer Science* 23(3), pp. 370–385, doi:10.1016/S1571-0661(05)80609-7. CALCULEMUS 99, Systems for Integrated Computation and Deduction (associated to FLoC'99, the 1999 Federated Logic Conference).

[3] François Garillot, Georges Gonthier, Assia Mahboubi & Laurence Rideau (2009): *Packaging mathematical structures*. In: *Theorem Proving in Higher Order Logics (2009), LNCS* 5674. Available at `http://hal.inria.fr/inria-00368403_v2/en/`.

[4] G. Gonthier, B. Ziliani, A. Nanevski & D. Dreyer (2011): *How to Make Ad Hoc Proof Automation Less Ad Hoc*. Manuscript.

[5] Georges Gonthier, Assia Mahboubi & Enrico Tassi (2011): *A Small Scale Reflection Extension for the Coq system*. Research Report RR-6455, INRIA. Available at `http://hal.inria.fr/inria-00258384/en/`.

[6] Cezary Kaliszyk & Freek Wiedijk (2007): *Certified Computer Algebra on Top of an Interactive Theorem Prover*. In: *Proceedings of the 14th symposium on Towards Mechanized Mathematical Assistants: 6th International Conference*, Calculemus '07 / MKM '07, Springer-Verlag, Berlin, Heidelberg, pp. 94–105, doi:10.1007/978-3-540-73086-6_8.

[7] Vladimir Komendantsky, Alexander Konovalov & Steve Linton (To appear): *Interfacing Coq + Ssreflect with GAP*. In: *Proc. User Interfaces for Theorem Provers (UITP) 2010*, ENTCS, Elsevier.

[8] Vladimir Komendantsky, Alexander Konovalov & Steve Linton (To appear): *View of computer algebra data from Coq*. In: *Proc. Conference on Intelligent Computer Mathematics (CICM) 2011*, LNAI, Springer.

[9] OpenMath: `http://www.openmath.org/`.

[10] The Coq development team: *The Coq Proof Assistant Reference Manual*. `http://coq.inria.fr/refman/`.

[11] Philip Wadler (1987): *Views: A Way for Pattern Matching to Cohabit with Data Abstraction*. In: *POPL'87*, pp. 307–313, doi:10.1145/41625.41653.