The Soft Type System of GAP

Florian Rabe

LRI Paris, University Erlangen-Nuremberg

Abstract

The question of how to design a good type system for mathematics remains open and challenging. Despite many proposals, no best solution has emerged. Of particular interest are soft type systems, which can be a compromise between sophisticated type theories and untyped systems. We contribute to the discussion by giving an easily accessible high-level overview of GAP's soft type system.

1 Introduction

Both computation-oriented systems (like computer algebra system) and deduction-oriented systems (like proof assistants) are developed in adjacent but almost disjoint communities. Not surprisingly, the distinction between computation and deduction causes many differences between these systems. But more interestingly, we find major differences also at the level of *type systems* even though these are needed in both kinds of systems.

As a particular example, we consider the type systems of GAP [**?**]. It uses a **soft** type system, where — akin to the elementhood relation of mathematics — typing is an undecidable binary relation on a collection of not-inherently-typed objects. But with the notable exception of Mizar [Wie07], deduction-based languages for mathematics tend to use **hard** type systems, where the type is an inherent property of an expression. Moreover, GAP uses a *hyper-dynamic* type system, where the type of an object is not only *determined* at run time (as in all dynamic type systems) but can even *change* at run time as more information is discovered about the object. But deduction-based languages tend to use **static** type systems where the type is a decidable property of the expression.

The above factors have led to fundamental differences between GAP and other type systems, already starting with the GAP keywords being unintuitive to many type theorists. It is not obvious at all whether and to what extent these differences are due to (i) a lack of awareness by the GAP community of formal type theories, or (ii) the inadequacy of the latter for practical computational applications.

The author suspects the truth is somewhere in between. This paper is meant to contribute to help inform the discussion by providing an easily accessible overview of the essential qualities of the GAP type system presented from the perspective of type theory. It does not provide an authoritative account of GAP's type system: Firstly, some subtleties and possibly even a few important features remain beyond the scope of this paper. Secondly, the author has —

knowingly or unknowingly — abstracted away some idiosyncrasies in order to make this account more accessible.

Additionally, this paper can be seen as a working document that can be refined over time with the goal of obtaining a rigorous, formal description of the entire GAP type system. The author is hopeful that this process may even feed back into the GAP design when discussing whether inconsistencies between the author's understanding and the actual implementation should result in changes to the former or the latter.

In any case, none of the type system design or the GAP implementation is due to the author, and this paper should not be construed as claiming such a contribution. The author's sole contribution is describing the existing type system from a specific outsider's perspective. This, however, is still difficult enough to be worthwhile.

2 Concepts

2.1 Overview

GAP uses a single flat namespace where every declared entity is identified by its **name**.

Three kinds of named **declarations** exists: categories, operations, and methods. Additionally, constructors, attributes, and properties are distinguished special cases of operations.

Operations and categories introduce new objects and thus must have fresh names. Methods, on the other hand, introduce unnamed implementations of a previously declared named operation; thus, the name of a method is the same as that of an operation. (At the meta-level, a specific method may be referenced by combining the operation name with the method's documentation string.)

Three kinds of anonymous **complex** entities exist: objects, families, and filters. GAP objects represent mathematical objects and are the primary interest. Families and filters provides a type system on objects: a type consists of a family (the base type) and a filter F, which provides a unary predicate on objects of family F. The family of an object O is a hard type: it is unique, computable, and fixed. The filter is a soft type: O can satisfy any number of filters, filters may be undecidable, and the type of O can be refined at run-time as more filters become known that O satisfies.

2.2 Complex Entities

In deduction systems, it is possible, even typical to build all or most expressions ex nihilo, typically via inductive types or axiomatic specifications. But such representations are usually efficient and therefore problematic in computation systems. Therefore, GAP allows arbitrary primitive objects backed by concrete representations in the underlying run-time environment. These are provided by the **families**: each family introduces a set of primitive objects.

Users can implement new families. But the following families with their respective primitive objects are built into GAP:

- one each for a few types of built-in literals:

  - cyclotomic numbers (elements of the algebraic closure of the rationals),
  - booleans,
  - strings,
- one each for several built-in operators that form complex objects
  - homogeneous lists (called **collections**): lists of objects that have the same family,
  - heterogeneous lists: lists of arbitrary objects,
  - functions on objects.

The objects are the primitive objects introduced by the families and any application of an operation to objects.

A **filter** is one of the following:

- the universal filter `IsObject`,
- a category C,
- a property P,
- a conjunction $F \wedge G$ of filters.

We call categories and properties **atomic filters**. By convention, their names are of the form `IsXXX`.

Filters can be normalized into a set of atomic filters (with `IsObject` corresponding to the empty set and $\wedge$ to union). Therefore, types are essentially pairs of a family and a set of atomic filters, and a type can be efficiently stored as a bitvector indexed by the known atomic filters. GAP stores this bitvector together with every object.

Because the family is an inherent property of an objects anyway, the **typing relation** reduces to a relation $O : F$ between objects O and filters F. It is defined as follows:

- O : `IsObject` always holds
- $O : F \wedge G$ holds if $O : F$ and $O : G$.
- $O : C$ holds if O was returned by a constructor of category C,
- $O : P$ if evaluating property P on O returns `true`,
- In addition to the above rules, $O : F$ holds for an atomic filter F if the corresponding bit was set when O was constructed. This is used in particular by the constructors of categories (see below).

Types are hyper-dynamic: whenever a property is evaluated for an object at run-time, its bit in the cached bitvector type is updated. Thus, the type changes dynamically as more properties are evaluated.

2.3 Declarations

Categories A category declaration consists of

- a name,
- a filter (called the superfilter).

The concrete syntax is `DeclareCategory(name: String, superfilter: Filter)`.

A **category** declaration introduces a primitive filter. All categories are created empty. The objects satisfying this filter are introduced by declaring constructors. These are operations whose implementation explicitly marks the returned objects as having the category as a filter.

When a constructor of category C is run, the returned object automatically has all filter bits set that correspond to the atomic filters in the superfilter of C.

Operations An **operation** declaration introduces an n-ary[1] function on objects. Operations are softly typed: each n-ary operations provides a list of length n providing the input filters of the respective argument. Operations may also carry an optional return type, which defaults to `IsObject` if omitted.[2]

The concrete syntax is

```
DeclareOperation(name: String, inputfilters: Filter*, outputfilter: Filter?).
```

An **attribute** declaration is the special case of an operation that is unary. The special treatment of attributes is important only for efficiency reasons: The values of attributes are cached with each object. The concrete syntax is `DeclareAttribute(name: String, inputfilter: Filter, outputfilter: Filter?)`.

A **property** declaration is the special case of an attribute that returns a boolean. The special treatment of properties is important only because properties can be used as filters. The concrete syntax is `DeclareProperty(name: String, inputfilter: Filter)`.

A **constructor** declaration is the special case of an operation that returns an object of a given category. The concrete syntax is `DeclareConstructor(name: String, inputfilters: Filter*, outputfilter: Filter?)`.[3]

Conceptually, all operations are defined. But the definiens is never part of the declaration and instead provided separately in method declarations.

Methods Every operation can have multiple definitions, which are provided by methods. A method declaration consists of

- the name of the operation,
- the input and output filters,
- the actual definition, as a function in the underlying programming language.

The concrete syntax of a method declaration is `InstallMethod(operationname: String, inputfilters: Filter*, outputfilter: Filter?, definition: function)`.

The input and return filters of a method may be more restrictive than the filters used in the operation declaration. More restrictive input filters can be used to represent overloading of operations or run-time polymorphism. A more restrictive output filter can be used to indicate a sharper type than required by the operations.

When evaluating the application of an operation to arguments, GAP selects a specific method executes its definition. If more than one method exists, whose input filters type the operation arguments, an internal ranking is used to disambiguate.[4]

[1] GAP has an implementation restriction of $n \leq 6$.

[2] This is a recent feature motivated by the discussions that also led to this paper.

[3] The return argument is a recent feature. More generally, the current implementation of constructors is somewhat awkward and may be subject to change. Currently, a constructor's first argument is special: It must be the expected return filter (rather than an object). This is used to allow method selection to choose a different method for different special cases. A more elegant solution would be to allow every operation to declare that some of its arguments must be filters. This would yield an untyped version of bounded polymorphism with filter arguments corresponding to type arguments.

[4] In particular, if a property of O is evaluated in between two calls of the same operation on O, a different method may be selected the second time. This is often desirable, particularly when the second method is more efficient.

3 Conclusion

References

[Wie07] F. Wiedijk. Mizar's Soft Type System. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 383–399. Springer, 2007.