Abstract

Computer algebra systems have to deal with the confusion between "programming variables" and "mathematical symbols". We claim that they should also deal with "unknowns", i.e. elements whose values are unknown, but whose type is known. For example, $x^p \neq x$ if x is a symbol, but $x^p = x$ if $x \in \mathrm{GF}(p)$. We show how we have extended Axiom to deal with this concept.

The "Unknown" in Computer Algebra

James Davenport*
jhd@mathbath.ac.uk

Christèle Faure†
crf@maths.bath.ac.uk

School of Mathematical Sciences
University of Bath
Bath BA2 7AY
England

1 Introduction

Computer Algebra and Numerical Computation are two means of computational mathematics which are different in the way they use and treat symbols.
In numerical computation, symbols are considered as programming variables and are used to store intermediate results.
Computer algebra systems take into account in polynomials etc. a second class of symbols: uninterpreted variables.
But in mathematics, a third class of symbols is also involved in computation. One often begins mathematical texts with sentences like "let p be a polynomial" or "let n be an integer" ... In those sentences, n and p are neither programming variables, nor uninterpreted variables, but are names of mathematical objects, with types but without known values, called **unknowns** — in future we will refer to such symbols as n as **basic unknowns**.
One begins a mathematical text by "let p be a polynomial" to make the following text more precise. For example, the expression "$n+m$" can mean a lot of things: the sum of two matrices, the application of the concatenation operator to two lists etc., whereas this expression is clear if one declares that n and m are two "integers".
Such symbols must be treated in computer algebra in a specific way because

*This research, and the access to `Axiom` used in this project, were supported by the UK's Science and Engineering Research Council under grant GR/H/11587.

†Supported by an INRIA scholarship and EEC Project POSSO Esprit Basic Research Action 6846.

they don't have values as programming variables, nor do they mean a position like uninterpreted variables, and the only thing known about them is a sort of "type" like "integer", "polynomial" ...

It is quite easy to confuse the concept of uninterpreted or formal variable with the concept of , but the two are in fact different concepts. Over $\mathbf{Z}_p$, the polynomials x and x^p are different if x is a formal variable, but if x is an (unknown) element of $\mathbf{Z}_p$, then $x^p = x$ by Fermat's Little Theorem.

Traditionally, computer algebra systems cannot handle such symbols properly because they have no mechanism for the user to associate such "types" with symbols. Classical computer algebra systems must be considerably modified to be able to treat such symbols. An example of a system able to take them into account is *Ulysse* ([**?**]), a system based on an interpreter enriched with a type checker ([Fau92]). A non-classical system such as `Axiom` is, roughly, a library built from high-level mathematical descriptions by a compiler. Introducing unknowns in this kind of system is quite different because one does not have to modify the computing part of the system (interpreter) as in classical systems. Instead, one need only introduce a new class of mathematical expressions.

We will use `Axiom` to explain the concepts defined in this paper. The user is able to define new mathematical objects in `Axiom`. We will describe this aspect of the system here: other descriptions can be found in [**?**]. A set of objects and the operators used to manipulate them is called a `domain` (see [JS92a]). Each `domain` can be constructed through an existing functor, or a new one defined by the user [Dav92]. A new `domain` must first be defined in terms of more abstract concepts called `categories` (see [JS92b]) which belong to a hierarchical graph based on algebraic properties (see [**?**]). One is obviously allowed to define new categories and introduce them into the hierarchy. This leads to a two-level typing mechanism in Axiom:

$$\text{Object} \in \text{Domain} \in \text{Category}.$$

To make the specification of the domain complete, all the operators specific to this domain must be declared. Then one has to implement it, i.e. define the representation of the object in the domain and implement all the functions allowed over those objects. For more information about the creation of new domains, the reader can refer to ([Dav92]).

2 Unknowns: specification

We want to introduce unknowns through the definition of a domain in `Axiom`. For this purpose, we will define a functor of domains of unknowns. We first

have to answer a few questions to specify it clearly.

1. What is a domain of unknowns?
   Basically, it contains symbols with their "types", called **basic unknowns**. But obviously one wants to built expressions that contain them, therefore valid operators must be defined. Those expressions are purely symbolic, when one also needs to use partially evaluated expressions such as "$n+2$" where n is a basic unknown integer. Then a second class of constants are necessary: they are values belonging to some particular sets.
   An expression is a valid unknown if, whenever each basic unknown is replaced by a value from the appropriate set, the whole expression can be evaluated to a well-defined object.
   The parameters necessary to define a constructor of domains of unknowns are a list of typed symbols (called basic unknowns), and a list of domains for the values (called value domains).

2. Type of basic unknowns and value domains?
   To be able to check the validity of unknown expressions, we have chosen to use domains as types for basic unknowns. That means that, if a basic unknown is replaced by an element of its type domain, the expression is still valid. We can only talk of a domain if all operators are valid over all elements of the corresponding set and particularly over all basic unknowns. For example, if n and 2 are in a domain which is a ring, then not only must $2+2$ be allowed, but also $n+2$, and indeed $n+n$. This would seem unworkable if the type domains of those basic unknowns are all different. Therefore we decide that basic unknowns always belong to the same type domain, all values belong to the same value domain, and that those two types are the same. These restrictions are not as strong as it seems to be because one is able in Axiom to create several domains of unknown, and use `coercions`.

   Then the constructor of such domains needs two arguments: a list of symbols, the type domain (same as value domain). If we call the type domain and the value domain `D` and the list of basic unknowns `BasicUnknowns`, the declaration of this constructor in `Axiom` is:
   `Unknown (D,BasicUnknowns:List Symbol)`

3. One question we should ask ourselves is: should there be one or more functors to create such domains? The algebraic type of a domain defined by `Unknown (D,BasicUnknowns:List Symbol)` depends on the type of `D`. For example, the domains of integer or polynomial unknowns may be specified to be rings whereas the matricial or vectorial unknowns cannot be so specified. If `X'` is a weak form of the category `X`, that is only suitable properties and operators from `X`, the definition of the most general

constructor seems to have a head like:
`Unknown (D:X,BasicUnknowns:List Symbol):X'`

In order to show how such a constructor can be defined, we will work with the example of the integers, whilst keeping the plan as general as possible.

3 Pure Unknown Integers

From the specifications studied in the previous section, a domain of unknown integers is completely defined if one gives the basic unknowns `BasicUnknowns` and the domain of integer values `D`. Amongst the current operations over integers the operations from `Ring` seem to be the essential ones. We first define the functor which implements all those operations.

In order to define the functor of domains of unknown integers, we first have to specify those domains.
The category `IntegerNumberSystem` covers all the implementations of integers, so the type domain `D` must belong to this category.
We have now to define the category of the domains thus constructed. In this section, we take the point of view that the most important operations over the integers are those of a ring, therefore we declare the category `Ring` as the category of the resulting domain.

We chose to represent the element of this domain as `LocalAlgebra (Polynomial D,D,D)` which is a domain of fractions $\frac{p}{n}$ where `p` is an element of `Polynomial D` and `n` is an integer from `D`. Only a few fractions belonging to the type `LocalAlgebra (Polynomial D,D,D)` are actually unknown integer. The function ``/'' must then check (see [**?**]) that the expression that they built really belong to this subset.

Then the functor creating domains of unknown integer (as rings) is declared as:

```
PureUnknownInteger (D,BasicUnknown):Ring == Implementation where

 D:IntegerNumberSystem
 BasicUnknown:List Symbol

 Implementation ==> LocalAlgebra (Polynomial D,D,D) with
   "/" : ($,D) -> $
   ++ a/n computes the expression whose value is a/n if it
   ++ is actually an unknown integer
```

4 Conditional Unknown Integers

However, we also need to use other operators such as `abs, =, < ...` These operators cannot be added to the previous functor because values such as `abs(n)` cannot be represented as a polynomial over `D` whose variables are basic unknowns. In order to motivate this discussion, we focus on the example of the operator `abs`. A first solution would be to change the representation to `Expression D`. But this solution does not permit any real *computation* over unknown expressions, and the high-level computation we wanted to introduce through unknowns would not be available. For example, an expression such as $2 * abs(n) - abs(2 * n)$ is only 0 if the computer algebra system knows about `abs`. A more complex example would be $abs(n) - n$, which is zero for non-negative n.

We chose another solution. We introduced a new kind of objects called **Conditional objects** to enable more complete computation. Those conditional objects are expression with several values depending on some boolean conditions.

Example 1 *The expression* `abs(n)` *may be translated to this conditional unknown integer:*

$$\begin{array}{lll} if & n \geq 0 & then \quad n \\ & & else \quad -n \end{array}$$

and the expression `n<m` *may be translated to:*

$$\begin{array}{lll} if & n < m & then \quad true \\ & & else \quad false \end{array}$$

The kind of conditions necessary to express the previous expressions are quite simple, but one can see that this is not always the case. The treatment of conditions will certainly be very complicated if one wants to use congruence relations.

Thus before defining unknown integers, we have to define conditional domains: i.e. we have to specify the functor that builds them.

1. What is a conditional domain?
   The objects that belong to this domain may have different values depending on certain boolean conditions such as "if C_1, it is exp_1, if C_2, it is exp_2 ..."
2. What about conditions?
   All the conditions describe a unique set, they are a partition of this set.

But each set can be split with a lot of partitioning operators. For example, $<, >, =$ or congruence relations split the set of integers ... We decided that all the conditions of a conditional object must belong to the same domain, i.e. the set of operators involved is "homogeneous". So if one wants to use several kinds of conditions over the same set S, one has to define several domains of conditions over S.

3. What about values?
   Values obviously belong to the same domain because conditional objects define functions from a domain S to another domain V, the domain of values.

4. The functor:
   A conditional domain is defined from a domain of conditions and a domain of values. We hope that the conditional domain will be as powerful as the value domain. In fact all the functions which still mean something over conditional objects must be extended, so the category of the constructed domain `Y'` must be a weak form of the category of `Val` called `Y`. The declaration of the functor constructing such domains is:

   ```
   Conditional (Cond:X,Val:Y):Y'
   ```

At this point, a question may strike us: are conditional objects partial or complete? Indeed, we want to know if the list of conditions (over S) of a conditional object applies to the full set S. For example, the following expression is partial:

$$f(n) = \begin{array}{ll} n & \text{if} \quad 0 \leq n \\ -n & \text{if} \quad n < -3, \end{array}$$

whereas this one is complete:

$$f(n) = \begin{array}{ll} n & \text{if} \quad 0 \leq n \\ n+3 & \text{if} \quad -3 \leq n < 0 \\ -n & \text{if} \quad n < -3. \end{array}$$

We chose to consider all the conditional objects as complete and all the functions defined over this domain must keep this completeness.

In order to define the domain of unknown integers, two conditional domains must be created: the conditional unknown integer (for the operator `abs`) and the domain of conditional boolean (for `inf?,eq?`). For this purpose two functors have been defined: the first for conditional rings (called `ConditionalRing`), and the second for conditional domains similar to boolean (called `Conditional-ExtendedBoolean`).

The functor of domains of conditional unknowns is defined as follows :

```
ConditionalUnknownInteger (D,BasicUnknown):Exports
   == Implementation where

 D : IntegerNumberSystem
 BasicUnknown : List Symbol

 C ==> ElementaryIntegerConditions BasicUnknown
 V1 ==> PureUnknownInteger (D,BasicUnknown)
 V2 ==> CopyBoolean
 CV2 ==> ConditionalExtendedBoolean (C,V2)

 Exports ==> Join(ConditionalCategory (C,V1),Ring) with
   abs : $ -> $
   ++ abs(a) computes : if a>0 then a, if a=0 then 0, else -a,
   inf? : ($,$) -> CV2
   ++ inf? (a,b) computes : if a<b then True else False
   eq? : ($,$) -> CV2
   ++ eq? (a,b) computes : if a=b then True else False
   "/" : ($,D) -> $
   ++ a/n computes the conditional object whose value
   ++ is a/n if it is an element of V1 else error
```

The implementation of this functor is `ConditionalRing (C,V1)` with the definitions of the functions (not belonging to `Ring`) `abs, inf?, eq?, "/"` added.

5 Conclusion

The definition of conditional pure unknown integers fits perfectly the formal definition of unknown integers previously given, so the two functors are the same. With this new type, one can compute some expressions in a way quite different from the usual one:

```
(1) -> cu := (n*(n+1)/2)::ConditionalUnknownInteger([n,m,p],Integer)

                  2
                 n  + n
   (1)  (True -> ------)
                   2
             Type: ConditionalUnknownInteger([n,m,p],Integer)
(2) -> abs cu

                           2
                        - n  - n
   (2)  ((n(n + 1)<0) -> --------)
```

```
                      2
((n(n + 1)=0) -> 0)
                     2
                    n  + n
((n(n + 1)>0) -> ------)
                      2
     Type: ConditionalUnknownInteger([n,m,p],Integer)
```

Such technology can be applied to other problems, as far as conditional results are needed. For example, it is useful in some computations of limits ($lim_{x\to\infty} ae^x$ depends on the sign of a), integrals ...

References

[Dav92] J.H. Davenport. How does one program in the axiom system. 1992.

[Fau92] C. Faure. *Quelques aspects de la Simplification en Calcul Formel*. PhD thesis, Université de Nice Sophia-Antipolis, 1992.

[JS92a] R.D. Jenks and R.S. Sutor. *Axiom The Scientific Computation System*, chapter 13, pages 527–546. Springer-Verlag, 1992.

[JS92b] R.D. Jenks and R.S. Sutor. *Axiom The Scientific Computation System*, chapter 12, pages 515–526. Springer-Verlag, 1992.

The Authors

Prof. James Davenport took his degrees at Cambridge University (England). He is now at Bath University, where he is Hebron & Medlock Professor of Information Technology, and Head of the School of Mathematical Sciences. He has worked extensively on the Reduce and Axiom computer algebra systems, and has written the books *On the Integration of Algebraic Functions* and *Calcul Formel* (with Y. Siret and E. Tournier). Both books have been translated into Russian.

Dr. Christèle Faure took her degrees at the University of Nice–Sophia Antipolis (France). She spent 1992–93 as a post-doctoral researcher at the University of Bath, working on the subject of this paper and its implications.