Order Sorted Computer Algebra and Coercions

submitted by

Nicolas James Doye

for the degree of PhD

of the

University of Bath

1997

Abstract

Computer algebra systems are large collections of routines for solving mathematical problems algorithmically, efficiently and above all, symbolically. The more advanced and rigorous computer algebra systems (for example, Axiom) use the concept of strong types based on order-sorted algebra and category theory to ensure that operations are only applied to expressions when they “make sense”.

In cases where Axiom uses notions which are not covered by current mathematics we shall present new mathematics which will allow us to prove that all such cases are reducible to cases covered by the current theory. On the other hand, we shall also point out all the cases where Axiom deviates undesirably from the mathematical ideal. Furthermore we shall propose solutions to these deviations.

Strongly typed systems (especially of mathematics) become unusable unless the system can change the type in a way a user expects. We wish any type change expected by a user to be automated, “natural”, and unique. “Coercions” are normally viewed as “natural type changing maps” — this thesis shall rigorously define the word “coercion” in the context of computer algebra systems

We shall list some assumptions so that we may prove new results so that all coercions are unique — this concept is called “coherence”.

We shall give an algorithm for automatically creating all coercions in type system which adheres to a set of assumptions. We shall prove that this is an algorithm and that it always returns a coercion when one exists. Finally, we present a demonstration implementation of this automated coercion algorithm in Axiom.

Contents

Structure of the Thesis

The thesis can be viewed as decomposable into the following topics:

Background and existing theory

Chapter 1 introduces computer algebra systems in general, strong typing and abstract datatyping. We introduce some of the problems that can occur in a strongly typed language which uses abstract datatypes. We present an analogy of the solution to one of the problems in such a system.

In chapter 2 we discuss and compare the type systems in several modern computer algebra systems. We also mention OBJ, a computer language which implements a type system based on abstract datatypes.

Both category theory (chapter 3) and universal algebra (chapter 4) may be used as the basis for systems of abstract datatypes. In these chapters we state all the current theory which we shall require in the later chapters.

We shall also compare how many of these theoretical notions are represented in the computer algebra system, Axiom.

New theory

Chapter 5 details some extra (mostly) new mathematics which allows for certain extensions to the theory of abstract datatypes (as currently implemented in Axiom) to be formalised in the language of order-sorted algebra.

We briefly look at the correspondence between the two approaches to abstract datatyping. This allows us to use either terminology when one may be more appropriate than the other.

Finally we formalise the meaning of the word "coercion" which is used widely in Axiom without any strict definition.

In chapter 6 we look at the work of Weber in the area of coherence. We adjust and extend his work so that we may sensibly restrict the type system of Axiom and still have unique coercions.

Chapter 7 introduces new definitions and assumptions on the type system which allow us to create coercions automatically. We provide a function to do so and prove that it is an algorithm which creates a unique coercion if and only if one exists.

Demonstration implementation

In chapter 8 we detail how the automated coercion algorithm may be implemented in Axiom. We mention where this prototype implementation disagrees with the theoretical algorithm.

Conclusions and future extensions

In chapter 9 we list the differences between Axiom's current type system and the formal theoretical type system we have specified.

We draw conclusions from this work in chapter 10. We also state future directions research may take towards the aim of creating the perfect computer algebra system based on abstract datatypes.

Acknowledgements

None of this work would have been possible without the original input and ideas of my supervisor, Professor James H. Davenport.

The implementation would have been impossible without the help from the following Lisp and/or Axiom-internals experts: Professor James H. Davenport (again); Mike Dewar; Russell Bradford; Pete Broadberry.

Mathematical issues were discussed at length with Professor James H. Davenport (yet again); Tracy Gardner; Russell Bradford; Mark “Beanz” Hindess.

My thanks go out to both Professor James H. Davenport and Russell Bradford for proof reading various versions of this thesis.

In addition to all the intellectual support, there are two people without whom none of this would have been possible. I refer, of course to my wife Deborah and my father, Mr. Geoffrey Doye.

I’d like to thank Professor John ffitch for giving me this chance in the first place. I’d also better thank all the postgrads in the computing group in the last few years for putting up with me.

My thanks go to NAG for supplying the Axiom Boot source code. Also, to the following computing services departments for the use of their systems: University of Bath, Department of Mathematical Sciences computing services; BUCS (University of Bath Computing Services); NISS (National Information Services and Systems).

This research was funded by the EPSRC (previously known as SERC). Much needed personal financial assistance came from my father and part-time employment from both BUCS and the Department of Mathematical Sciences. In the overrun, I have been employed by both the Department of Mathematical Sciences and more recently by NISS.

Finally, I ought to thank the authors of XEmacs for making editing everything just *so* easy and the authors of LaTeX 2_ε for the layout of this thesis. Diagrams were produced using Paul Taylor’s diagrams package.

For Deborah.

Chapter 1

Introduction

1.1 Introduction

This chapter introduces the reader to the main ideas behind this thesis. We give some background information, explain the context of the work, and say why the work is a "good idea".

This work centres on types in computer algebra.

1.2 Computer algebra

Computer algebra [DST92] is an all-encompassing term that covers performing any sort of mathematics on a computer. Numerical analysis (mathematics involving floating point numbers and a certain degree of error) is often excluded from this term, though most modern computer algebra systems are (at the very least) competent in this area.

Computer algebra systems may excel in various areas of mathematics. Some are masters of discrete mathematics (groups, rings, monoids, etc.); the more common systems deal with more mainstream concerns such as: indefinite integration; solving differential equations; matrices etc.

The important feature of all computer algebra systems is their ability to solve equations (like differential equations) *symbolically*. In other words, they may solve equations for the general case; they need not substitute values for the variables.

Modern computer algebra systems are large collections of routines and algorithms. These algebra systems often come with their own language(s) for user input. The user is usually supplied with an interpreter which can usually read both input from the keyboard and from files. The more advanced computer algebra systems come with a compiler, too.

Common computer algebra systems are Axiom [JS92], Derive [Der97], GAP [GAP97], Macsyma [Mac97], Magma [Pla96], Maple [CGG$^+$92], Mathematica [Wol92] and Reduce [Hea91]. We discuss the relevant details of some of these systems in chapter 2.

The user of a computer algebra system is usually someone who wishes to solve a mathematical problem. Mathematics is a formal subject and as such there is a need for rigour. Hence computer algebra systems need a rigorously defined language to avoid confusion and incorrect results.

Many of the more popular commercial computer algebra systems do not provide this rigour. Instead they trade rigour and strictness for ease-of-use (and hence mass-market appeal). This is not to say that they consistently supply incorrect results — merely that if the user has not given all the input that they can (or are allowed to by constraints of the system) then the system may misinterpret their input.

Being able to restrict the domain over which a variable may range, or from where a function may take its arguments can reduce the amount of confusion and ambiguity in any mathematics a user may enter.

For example, if a user enters a `1` then a computer algebra system has to judge whether this is the 1 from the integers, a constant polynomial 1, the identity of some group, a 1 from the integers modulo some n, the distance which is not zero in a discrete metric space, etc.

Restricting variables to various "sets" of values is a useful concept and is called typing. For example, if one says that `x` is an integer and then declares `x` to be `1` then there is no confusion as to what `x` is. We are free to add `x` to `x`, safe in the knowledge that the answer is the integer `2`. If we did not know that `x` were an integer, then the answer could have been `0` (in $\mathbf{Z}_2$), `1` (in a group where `1` is the identity and the binary operator is `+`), or any other value in some contrived example.

Such restrictions on variables are usually called types. For example, declaring `x` to be an integer, is the same as saying that the type of `x` is an integer.

Mathematics written in books is usually stated formally, with all the context and meaning of certain symbols stated in the text. When mathematicians talk to one-another they may (and in practice, do) drop some of this formality since they can infer the meaning of certain symbols and the context from extra-mathematical factors (such as: with whom they are holding a conversation; previous conversations they may have had; etc.). They may even correct each other.

Computer algebra systems are not people and can fail to infer the type and meaning of terms successfully. They also can not understand paragraphs of English (or other natural languages) to gain the context from there. They are highly unlikely to correct the user. This is why their languages should be strictly defined, and yet fully expressive.

Guessing the type (intelligently) is called type inference. All the untyped systems effectively guess the "type" internally whenever a function is applied to any data. Entering `1+1` will result in `2`, simply this is because this is the most common case.

A more intelligent approach (like type inference in Axiom) is to allow the user to say which `+` function they require. Axiom will then infer the types of the two `1`s from the fact it knows which types that `+` is expecting. Alternatively the user may state which `1`s they have inputted and what type they expect in return; Axiom will then use the `+` function uniquely determined by this information.

1.3 Strong types, categories, varieties and theories

In this thesis we are mainly concerned with the typing mechanisms in modern computer algebra systems. However this work is also relevant to type systems in general. Many languages (principally the so-called object-oriented languages) use a type system which may occasionally be viewed in a mathematical fashion.

Axiom and Magma are two of the best computer algebra systems and what sets them apart from their peers is the concept of categories (chapter 3) or varieties (specified by theories) (section 4.6). Some other algebra systems do possess the concept of types, and Reduce [Hea91] will allow new types to be introduced (section 2.4). GAP [GAP97] also claims to have a concept of categories, though they are not so deeply ingrained in the type system as they are in Axiom (section 2.5) and Magma (section 2.7). GAP will also allow new types to be introduced.

Certain forms of inheritance found in object-oriented languages follow the paradigms used in Axiom and Magma. However, most object-oriented languages (C++ [Str97], Eiffel [Eif97], Java [Gra97], youtoo [Kin96], etc.) are more flexible and allow many different ways for types to re-use bits of code from each other. However, automated coercion (in any sense) is *not* a feature of these languages. OBJ [GWM$^+$93] is not a computer algebra system, however this work is wholly suited to it since it uses a similar type paradigm to Axiom and Magma. We discuss OBJ further in sections 2.8 and 2.9.

The category (or variety) mechanism found in Axiom and Magma is a way of abstracting a further level away from mere types. The categories are used to type the types, and provide a "standard" interface and/or implementation for each type in a category.

Magma also abstracts one further level away, by allowing the existence of varieties of categories, in a similar way.

The concept of strong typing, is that every thing must have a type. Axiom and Magma are both strongly typed. Most of the other systems are not strongly typed. Indeed some do not even have any notion of types at all.

The categories (or varieties) of Axiom, Magma and OBJ are abstract datatypes. We shall detail this more in the next section.

1.4 Abstract datatypes in general

Abstract datatypes provide a way of specifying a type. As a (useful) side effect an abstract datatype can express how a whole collection of types may act.

An incredibly specific specification will only specify types which are all "the same" (isomorphic). A less exacting specification may result in types which are similar, but not the same.

This "side effect" of using a less exacting specification to collect similar types together forms the basis of strict categorical type systems like that in Axiom and many object oriented languages.

For example, we may say which operations are available on a certain family of types. We may also state certain facts about the structure of all types in a certain family. Most importantly of all, we can enforce relationships between various types (of different families), how they interact, and how they may depend upon each other (or not).

See section 4.2 for how we may specify types and what operations are available to them. Look at section 4.7 for the relationships between different types. Section 4.6 details how we may restrict the structure of certain types using sets of equations.

As a simple example, a polynomial ring depends for its specification on the underlying ring (from which the coefficients are taken).

As a more complicated example, a polynomial ring could also depend upon: the type from which it takes its variables; the type of the exponents; a boolean algebra; an ordered free monoid with one generator (the natural numbers adjoin $\{0\}$ is often a good choice); and maybe a few others.

Another factor is the actual relationship between the various types. In our above example of the polynomial ring, we know that:

1. there is a ring monomorphism from the underlying ring to the polynomial ring.
2. there is an injection from the variable type[1] to the polynomial ring. (In fact, an ordered-set monomorphism[2].)

Other types often have more complicated lattices of relationships. These relationships are often able to be abstracted out to apply all the instance types of a particular abstract data type.

1.5 The problem

A commonly encountered difficulty in languages which utilise types is the following:

1. Given `thing`, of type `Type1`, can we change the type of `thing` to be that of `Type2`? More accurately, can we create an element of `Type2`, which corresponds, in some natural way to `thing` of type `Type1`?
2. If so, how do we go about performing such an operation? Can we perform this task algorithmically?
3. Does there exist a way of abstracting this question or must every (`Type1`,`Type2`) pair be considered? Moreover, can we abstract such an algorithm (as mooted in 2, above) out to cover all cases?

Such type changes are often called *coercions*, *conversions*, or *castings*, with all three words having slightly different meanings.

In this work, we shall consider castings and conversions to mean any type change, regardless of mathematical rigour. We shall call a type change a coercion if it is in some way, “natural”. We will define this more rigorously later (in definition 5.5.2).

[1]The type from which all the variables are taken. In Axiom, the variables are usually elements such as `X`, `Y` and `Z` which are from the type, `Symbol`.

[2]This is an injective function, ϕ, which is both a set-homomorphism (which is just a total function) and preserves order. So if $a < b$ then $\phi(a) < \phi(b)$. Mathematicians often utilise the order on the variables and extend it to an order on polynomials.

Computer algebraists often need to use coercions since informally most mathematicians alter the domain of computation without saying phrases like, "in the monomorphic image of A in B" or, "forgetting that x is a T and viewing it as a U".

For example, we often view the integers ($\mathbf{Z}$) as polynomials (in $\mathbf{Z}[x]$) rather than considering the constant polynomials of $\mathbf{Z}[x]$ as being a monomorphic copy of $\mathbf{Z}$. This is the coercion $\mathbf{Z} \to \mathbf{Z}[x]$.

Other simple examples of this sort of coercion are,

$$\mathbf{Z} \to \mathbf{Q}, \quad \mathbf{Z}[x] \to \mathbf{Z}[y,x], \quad \mathbf{Z} \to \mathbf{Q}[x,y], \quad S(2) \to S(5), \quad \mathbf{Z}[x] \to \mathbf{Z}(x)$$

where $S(n)$ is the symmetric group on a set of n symbols.

Coercions are useful when creating elements of quotient structures, such as $\mathbf{Z} \to \mathbf{Z}_n \cong \mathbf{Z}/n\mathbf{Z}$ (where $n \in \mathbf{N}$). Other epimorphic (surjective) examples include,

$$\mathbf{Z}_{25} \to \mathbf{Z}_5, \quad G \to G/G'$$

where G is any group and G' is the commutator subgroup of G.

Under certain circumstances, coercions can be "lifted" into other constructors. We refer to these as "structural coercions" later in this work. As examples, consider,

$$\begin{gathered}\mathtt{List}(\mathbf{Z}) \to \mathtt{List}(\mathbf{Q}), \quad \mathtt{List}(\mathbf{Z}_{25}) \to \mathtt{List}(\mathbf{Z}_5), \quad \mathbf{Z}_{25}[x] \to \mathbf{Z}_5[x], \\ \mathcal{M}_{2,2}(\mathbf{Z}) \to \mathcal{M}_{2,2}(\mathbf{Z}(x))\end{gathered}$$

Examples of type-conversions which are *not* coercions include,

$$\mathbf{Q} \to \mathtt{Float}, \quad \mathtt{List}(\mathbf{N}) \to S(n), \quad \mathbf{Z}_5 \to \mathbf{Z}, \quad \mathbf{Z}_5 \to \mathbf{Z}_3, \quad \mathbf{Z}[x] \to \mathbf{Z}[y]$$

Natural type changes (coercions) can be created in the Axiom interpreter at the moment using a series of *ad hoc* measures. All of this occurs transparently to the user and works for most common types. However, it is *not* algorithmic. One of the aims of this thesis is to provide an algorithm (algorithm 7.4.1) to create unique (theorems 6.5.4, 7.6.7) coercions.

1.6 Examples of how Axiom coerces

Axiom knows (or believes) that all functions called `coerce` are coercions and will use them to "build" other coercions using certain rules, and special cases defined in the interpreter. Here we give examples of how Axiom coerces between certain common types.

Simple examples

Example 1.6.1 $\mathbf{Z} \to \mathbf{Q}$:

In Axiom, $\mathbf{Z}$ is the type `Integer` and $\mathbf{Q}$ is the type `Fraction(Integer)`. The type constructor `Fraction(R : Ring)` exports a function[3],

```
coerce :  R -> %
```

which is the natural monomorphism including the ring `R` in its fractional field `Fraction(R)`.

Example 1.6.2 $\mathbf{Z} \to \mathbf{Z}[X]$:

In Axiom, the type `UnivariatePolynomial(X,Integer)` typically represents $\mathbf{Z}[X]$. This is constructed by the type constructor,

```
UnivariatePolynomial :  (S: Symbol, R: Ring) -> PolynomialCategory(R,
                        NonNegativeInteger, Symbol)
```

`UnivariatePolynomial` exports a function,

```
coerce :  R -> %
```

which is the natural monomorphism including the ring `R` in the polynomial ring as the constant polynomials.

Example 1.6.3 $\mathbf{Z}[X] \to \mathbf{Z}[X, Y]$ (where $\mathbf{Z}[X, Y] = \mathbf{Z}[X][Y]$):

This is the same as the previous example, $[Y]$ is acting as a morphism $\mathbf{Z}[X] \to \mathbf{Z}[X][Y]$.

[3]Actually, defined in a `Category` to which `Fraction` belongs.

Example 1.6.4 $\mathbf{Z}[X] \to \mathbf{Z}[Y,X]$ (where $\mathbf{Z}[Y,X] = \mathbf{Z}[Y][X]$):

Axiom knows that $\mathbf{Z} \to \mathbf{Z}[Y]$ is a coercion, and that the map $[X] : \mathbf{Z}[Y] \to \mathbf{Z}[Y][X]$ (ie. `UnivariatePolynomial`) lifts this coercion.

Example 1.6.5 $\mathbf{Z} \to \mathbf{Q}[X,Y]$:

This is built by the chain of coercions,

$$\mathbf{Z} \to \mathbf{Q} \to \mathbf{Q}[X] \to \mathbf{Q}[X,Y]$$

Example 1.6.6 $S(2) \to S(5)$:

These do not exist as domains in Axiom.

Example 1.6.7 $\mathbf{Z}[X] \to \mathbf{Z}(X)$:

This uses the coercion from `Fraction` discussed above, since $\mathbf{Z}(X)$ is represented in Axiom by, `Fraction(UnivariatePolynomial(X,Integer))`.

Example 1.6.8 $\mathbf{Z} \to \mathbf{Z}_5$:

This uses the function,

```
coerce :  Z -> %
```

defined in `IntegerMod(n :  PositiveInteger)`

Example 1.6.9 $\mathbf{Z}_{25} \to \mathbf{Z}_5$:

Axiom cannot coerce from `IntgerMod(25)` to `IntegerMod(5)`!

Structural coercions

The structural coercions are "lifts" of other coercions. In the case of `Lists`, one may perform a structural coercion as follows.

```
coerce( x : List(A) ) : List(B) ==
  map( coerce, x )@List(B)

map( f : List(A) -> List(B) , x : List(A) ) : List(B) ==
  nullp(x) => nil()$List(B)
  cons( f(car(x)) , map( f , cdr(x)))
```

In the case of `Polynomials`, a similar `map` function exists which uses `+` instead of `cons`.

In the case of `Matrix` a similar map function can be used (though since matrices are of fixed sizes - unlike sets, lists and polynomials - a default valued matrix (0) has to be created, then all the values substituted into (added to) it).

1.7 Mathematical solution overview

The solution to the questions raised in 3 of section 1.5 rely on abstract datatypes. These "collect" large numbers of types into collections of similar types. We shall be considering abstract datatypes in the language of universal algebra; specifically, order-sorted algebra (section 4.4).

Universal algebra has close links with the ideas of category theory. (See chapter 3 for category theory and section 5.4 for the links between the two theories). Indeed, Axiom uses the language of category theory to describe its algebraic design. Thus we shall also discuss the fundamentals of category theory in relation to this work.

Using the order-sorted algebra framework we shall show that there is indeed a strict mathematical definition (definition 5.5.2) which we may use to tell us which type changes are natural, and therefore, in our definition, coercions.

Moreover, we shall show that this approach allows us to create such coercions, algorithmically (algorithm 7.4.1).

In section 1.8 we shall show an analogy with moving buildings to the solution of changing the type of (ie. coercing) something.

1.8 Constructing coercions algorithmically

Consider the following analogy.

Suppose you have been given the job of moving London Bridge from one location to another. The bridge is too large to move in one piece.

You will merely be supervising the work and will not have to carry any of the bricks yourself, hence efficiency is not your concern.

You are also planning to move other different bridges in the future, and once you have a method for moving one bridge, you would like to be able to apply that method to the next. This means that next time a bridge needs moving, you won't have to do much work at all.

Luckily for you someone has already worked out a way of moving any of the bricks at the very bottom of the bridge to their new location, regardless of any obstruction, which would normally leave civil engineers crying. This person was clearly a magician.

So here is your algorithm for moving London bridge, and hence any future bridge.

```
If the original bridge has no bricks left then finished.
Else,
      take a brick from the top of the original bridge.
      If this brick is a bottom brick then,
            use the magical bottom brick mover, to place
            the brick in the corresponding location of
            the new bridge.
      Else
            hold the brick in place, in the corresponding
            location in the new bridge.
      Endif.
Endif.
Repeat.
```

Indeed, you realize that this method will work for other brick-based constructions, or indeed anything that is "built". (Obviously, there are other considerations, is the new location for the bridge a "natural" one? For example: Is it long or tall enough?)

The analogue of the bridge is a "thing" or "item" in our computer algebra system. The original location of the bridge is the original type of the "item" and the new location for the bridge is the analogue of the type to which we are coercing the "item".

Thus what we need to know are:

1. what are the foundations of a bridge?
   (**Analogue**: what are the constants of our type?)

2. what is the length, width, strength of a bridge?
   (**Analogue**: what are the parameters of the type?)

3. how do you alter the bricks to be of a new shape/material?
   (**Analogue**: we need to be able to recursively call the coercion algorithm on types which are "recursively required for coercion". (cf. Definition 7.6.5.) That is types which form the bits of the "item" once we have chopped it up.)

This is our analogy (to algorithm 7.4.1) and we may consider many data types to be constructed (or built) similarly. For example, lists (in the Lisp-sense) are always constructed either by being the empty list, `()`, or by being the `cons` of something on to the front of another list.

As a more complicated example, polynomials are constructed via one of the following mechanisms:

1. by being a member of (the included monomorphic copy of) the underlying ring (a constant polynomial);
2. by being a symbol exponentiated to some positive power (a univariate monic monomial);
3. by being the product of a monic monomial and a univariate monic monomial (a monic monomial);
4. by being the product of a monic monomial and a member of the underlying ring (a monomial);
5. by adding a monomial to a polynomial.

So we see that at least two of the most basic types in computer algebra are constructed in this way.

1.9 Background of Axiom

Computer algebra is a subject which dates back to 1953, and has been described as "constructive algebra plus efficient algorithms" [Bra92]. Towards the late 1960s, large collections of routines or programs were starting to become what one might think of as the first "computer algebra systems".

Scratchpad I [GJ71] was born in about 1971, the principal architect being R.D. Jenks [JS92]. Scratchpad I took the best ideas from Reduce 2 [Hea71], and Mathlab, and with new ideas and a new language became IBM's research computer algebra system.

In 1977 Jenks initiated the Scratchpad II project. Over the next couple of years, he, along with J.H. Davenport and Barton designed and implemented prototypes of a "sufficiently clever algebra system" - one which used the concept of "categories". An idea which, (apart from research projects such as Newspeak [Fod83]) up until the release of Magma [Pla96] (November 1, 1994) made Scratchpad II unique amongst computer algebra systems.

The category inheritance mechanism is based on an original "abstract datatype" design by Jenks and B.M. Trager. The algebraic category hierarchy has subsequently been redesigned by Davenport with Trager.

The implementation of Scratchpad II commenced in 1982, and in 1991, support for the system transfered to NAG, and the name of the system changed to Axiom.

1.10 OBJ

The subject of data abstraction has not been monopolised by the computer algebra community. Indeed, much of the major work has been done by those who are almost completely disjoint from the computer algebra field.

A key player in the area has been Joseph A. Goguen. In 1976, Goguen designed OBJ as a language for dealing with algebraic abstract data types, errors, and partial functions in a simple, and more importantly, uniform way. (See [GWM$^+$93] for more details.)

The current version of OBJ is OBJ3 Release 2 and is implemented in Common Lisp. OBJ3 is based on order-sorted equational logic, which provides a notion of "subsort". OBJ3 also provides "parameterised programming". Both of these notions are found in Axiom and Magma, but OBJ3's idea of subsort is far more rigorous that that of Axiom. We shall discuss OBJ in more detail in section 2.8.

1.11 Aims of the thesis

We shall lay down a strict mathematical foundation for a type system in a computer algebra system like Axiom. We state the mathematics which inspired Axiom's type system (chapter 4).

Axiom's type system uses ideas which are not explained using this basic mathematics. In section 5.2 we explain some current mathematics which models how we shall be viewing Axiom's partial functions. In 5.3 we state some new mathematics to deal with Axiom's conditional categories. In section 5.5 we formalise what is meant by a coercion. In chapter 9 we state the differences between Axiom and the mathematical theory.

However, the greater part of this work is dedicated to stating (sections 6.2 and 6.3) and creating (sections 6.4, 6.5 and chapter 7) enough mathematics to state an algorithm for automatically creating mathematically correct coercions.

In Axiom coercions are defined by:

- being a (rare) special case, hand-coded in the interpreter;
- being an explicitly written `coerce` function, from a `Category`, `Domain` or occasionally a `Package`; or
- being created via special interpreter tricks, which combine the above in an *ad hoc* manner.

These special interpreter[4] tricks[5] cover most cases, but not all. Users can not guarantee that coercions entered in the interpreter will exist.

It is also important to note that these special interpreter tricks are not available in the Axiom compilers (for Spad and Aldor).

We shall present an algorithm which builds coercions (for a very large subclass of types) which should exist from a (potentially, but not necessarily) minimal set. This algorithm is implementable (chapter 8) in the Axiom interpreter. Subject to a set of conditions being met by the type system these coercions are unique.

We draw some conclusions about this work in chapter 10.

[4]The interpreter is the interactive interface to the Axiom system.
[5]Special hand-written pieces of code which execute "behind the scenes" in the interpreter.

Chapter 2

Types in computer algebra

2.1 Introduction

In this chapter we shall go into more detail about types in various computer algebra systems. We shall discuss how the different systems view the concept of typing information; from the simplest non-typed languages in the so-called "M&M-systems"[1] through Reduce's type system[2], to the full-blown mathematical beauty of systems such as Magma and Axiom.

2.2 Mathematica

Mathematica is one of the most popular computer algebra systems on the market, today. Priced very reasonably, and with an attractive user interface, it is easy to see why ordinary PC owners are attracted to this product.

In chapter 4 of "Mathematica, a System for Doing Mathematics by Computer" [Wol92] Wolfram states,

> "At a fundamental level, there are no data types in Mathematica. Every object you use is an *expression*."

Types may be created in Mathematica, but they are merely tags on which one can perform switches. (For example, the tag "`List`".) This is an obvious approach, since Mathematica uses rule-based programming.

[1]In [JT94] Mathematica and Maple are referred to as M&M.

[2]The ability to extend the type system of Reduce was inspired by Axiom (then called Scratchpad).

Abstract data types for Mathematica are discussed in [Mae92], and inheritance for such a system in [Har94]. However, this approach is only marginally more sophisticated than that used in Reduce (described in [BHPS86]). There is some form of abstraction in this Mathematica abstract data type model, but function method selection is just as primitive as in the Reduce model.

2.3 Maple

Maple [CGG+92] is a small, fast, popular computer algebra system, and is well respected both by users and experts in the area of symbolic computation. Maple (version V release 3)[3] does have some concept of "type", but as you can see, this is not particularly well founded.

```
> x:=array(0 .. 1, 0 .. 1, [[a,b],[c,d]]);

                          x := array(0 .. 1,0 .. 1,, [
                                   (0, 0) = a
                                   (0, 1) = b
                                   (1, 0) = c
                                   (1, 1) = d
                               ])

> m:=linalg[matrix](2,3,[x,y,z,a,b,c]);

                                    [ x  y  z ]
                               m := [         ]
                                    [ a  b  c ]

> eval(m[1,1]);

                            array(0 .. 1,0 .. 1,, [
                                (0, 0) = a
                                (0, 1) = b
                                (1, 0) = c
                                (1, 1) = d
```

[3]Maple V release 4 is now available.

```
                              ])

> type(m,matrix);

                                          true
```

Notice that `m` is now a 2×3 "matrix" (in Maple's view) whose (1,1)th element is an `array` of symbols. Three of these symbols are also in the second row of `m`. The array type does not form a ring and moreover, the elements of `m` are from different types.

It is ridiculous that one may create matrices over any type which does not form a ring! One learns at school that given two matrices A and B, with the same number of rows, p, and the same number of columns, q, one may calculate

1. the $p \times q$ additive identity matrix
2. if $p = q$, I_p - the $p \times p$ multiplicative identity matrix
3. the negation of a matrix, $-A$
4. their sum, $A + B$
5. the product AB^T (where B^T represents the transpose of T).

However, should one create matrices over types which do not form rings, then one of the above may not be well defined ($\forall A$ in case 3, and $\forall A, B$ in cases 4 and 5). This is the situation in Maple — it lets one create objects of the type `matrix` which are not matrices in the mathematical sense[4].

Maple will now also let one "assume" certain properties about, or types for elements, but these are entirely at the user's discretion. This mechanism is based on "the algebra of properties" as described in [WG91][WG92]. One can think of this assumption mechanism as an advanced way of declaring the type of a variable.

The algebra of properties is a fascinating and extremely clever way of assuming certain facts about a given symbol, and works very well in the context of saying, "We assume that α is real and greater than zero", and then deducing various facts about $f(\alpha)$.

[4]Admittedly, Axiom does not always get these things perfectly correct. Martin [Mar97] points out that Axiom allows `Matrix(Float)` as a type, yet the `Float` type does not really form a ring.

However, the algebra of properties does not work particularly well for typing symbols/data. Type inference, for example, is problematic — given α, real and greater than zero, what are the types of $\alpha - 1$, $1 - \alpha$, $\alpha^2 - \alpha$ and $\sqrt{\log(\alpha)}$?

Inserting new elements in the property lattice is inherently difficult, and in a dynamic system means that binding operations to implementations is impossible at compilation time.

It could be argued that this method could run along side of Axiom's more advanced notion of types. Part of the problem with the Maple system is a confusion between different classes of "unknowns". There is no distinction between the importance of being a real number, and that of being greater than zero[5]. Surely "being a real" should be far more important than some statement regarding a symbol's possible values.

Much work has been done on this subject with special emphasis on Axiom, and the definitive work is [DF94] which defines the different classes of "unknowns" and provides details of an implementation for Axiom.

2.4 Reduce

Reduce 3 (current release is 3.6 [Str95][ffi98][Neu96]) is a well-established Lisp based computer algebra system, and although the user interface is archaic, until recently, its algebra system was still one of the very best around[6].

Reduce has a well-defined concept of "type" and indeed, new datatypes may be added and included at will. [BHPS86]

Reduce prototypes have also been developed [HS95] which take an order-sorted approach to algebra. This implementation seems to take some of the Axiom ideas and some of the algebra of properties ideas. It appears to be far cleaner and clearer than the model used in Maple.

However, in this prototype, it is still stated (in [HS95]) that:

> "Mathematical structures can have several isomorphic representations. A good example is the representation of polynomials in a distributed or recursive form. [...] For the simultaneous use of multiple representations,

[5] Being "greater than zero" is not just a proposition restricted to real numbers (or subsets thereof). Polynomial types are often ordered, and zero is a constant polynomial.

[6] The commercial success of some of the other systems has forced them (and gained them the capital) to correct many of the bizarre bugs that used to abound in earlier releases. Thus the commercial systems now work as well as Reduce, have a friendly GUI and may well have consigned Reduce to history.

explicitly defined coercions are necessary." (My emphasis.)

This is not currently the case in Axiom where coercions can be built (in the interpreter) by clever tricks in the underlying code. Furthermore, it is the aim of this thesis to demonstrate that, using an order-sorted approach to algebra, one can "build" many of these coercion functions *without* explicitly defining the coercions themselves.

2.5 Axiom

Axiom [JS92] (née Scratchpad II) is a Lisp-based, general computer algebra system. Axiom's main view of things is that every object has a type, and that there are ostensibly four layers.

$$\texttt{items} \in \texttt{Domains} \in \texttt{Categories} \in \texttt{Category}$$

For example,

$$\texttt{3} \in \texttt{Integer} \in \texttt{Ring} \in \texttt{Category}$$

and the top layer is the unique distinguished symbol, "`Category`". In general, this top layer is never referred to, and the three lower layers are all that are ever considered.

Strictly speaking, if one considers Axiom's `Categories` to be categories, and Axiom's `Category` symbol, to signify the category of all the `Categories`[7], then one should write,

$$\texttt{3} \in \texttt{Integer},\ \texttt{Integer} \in \mathrm{Obj}(\texttt{Ring}),\ \texttt{Ring} \in \mathrm{Obj}(\texttt{Category}).$$

where `3` $\in$ `Integer` means `3` is an element of the carrier of the principal sort of the signature to which `Integer` belongs.

Users may extend the Axiom system by writing new `Domains` and `Categories` in the Axiom extension language, called Aldor[8] [WBD+94a]. (They may also be written in an older Axiom language, called Spad. This is the language described in [JS92].)

Aldor can be complied into many languages: for use inside Axiom; to Lisp (currently AKCL [AKC97] or CCL [ffi97]); to C [WBD+94b]. This makes Axiom far more than

[7]`Category` is not one of the `Categories` in Axiom.

[8]This language was sometimes referred to as A♯ in internal documents, and has often been referred to as Axiom-XL.

just an algebra system, Aldor is also a first class language for writing fast, efficient, stand-alone applications.

A category is a collection of domains all of which are "similar". In [DT90] the authors state that they designed the Scratchpad system of categories or "abstract algebras"[9] (qv. chapter 4) for the following reasons:

1. economy of effort (section 2.5.1);
2. interest (section 2.5.2);
3. functoriality (section 2.5.3).

2.5.1 Economy of effort

The most obvious reason for this is the view of inheritance. A category (in Axiom) is said to extend another if it inherits all the other's functions, attributes and equations. This allows for a significant amount of re-use.

A more important issue in an algebra system is that if one can prove a theorem in some generality, ie. for all the objects in a category, then one does not have to go around proving the theorem for each object of the category.

2.5.2 Interest

Some categories are interesting and some are not. For example, `Ring` and `Field` are particularly interesting. Many theorems can be proved *for all* `Fields`.

However (to borrow an example from [DT90]), the category of all rings which when viewed as Abelian groups, have an involution with precisely one fixed point, is not particularly interesting.

One may think of the designers' concept of "interest" as congruent to "useful" (usually to the user, but occasionally to the designers).

2.5.3 Functoriality

This is called "higher order Polymorphism" in [Cro93]. There are operators which given objects of a category can create new objects of a given (potentially different)

[9] Axiom's `Categories` can also be thought of as order-sorted algebras, but the order on sorts is never explicitly defined and there are some difficulties in the definition of the operator symbols. This is discussed in more detail in section 9.2

category. For example `List` is a functor from `SetCategory` to `ListAggregate`.

$$\begin{aligned} \texttt{List} \; : \; \texttt{SetCategory} &\to \texttt{ListAggregate} \\ \texttt{S} &\mapsto \texttt{List(S)} \end{aligned}$$

Indeed, this target category may even be the distinguished symbol `Category`.

In Axiom, the `Category` constructors (often called functors) are functors from

$$\prod_{n \in \mathbf{N} \cup 0} \texttt{Category} \to \texttt{Category}$$

For example, the `Ring` functor takes no arguments and returns the `Category` of `Ring`s.

$$\begin{aligned} \texttt{Ring} \; : \; () &\to \texttt{Category} \\ () &\mapsto \texttt{Ring} \end{aligned}$$

The special `Category` creation operation, `Join` is also a functor. `Join` is used to declare a new `Category` to be a subcategory of the intersection of two or more `Categories`. Here we illustrate the case of `Join` acting on two `Categories`.

$$\begin{aligned} \texttt{Join} \; : \; (\texttt{Category}, \texttt{Category}) &\to \texttt{Category} \\ (\texttt{A}, \texttt{B}) &\mapsto \texttt{A} \cap \texttt{B} \end{aligned}$$

More commonly in Axiom, and Axiom's own use of the word "functor" covers cases as follows. This is similar to the case of Ring, only less trivial. `ListAggregate` is the `Category` of all types of finite lists.

$$\begin{aligned} \texttt{ListAggregate} \; : \; (\texttt{SetCategory}) &\to \texttt{Category} \\ \texttt{S} &\mapsto \texttt{ListAggregate(S)} \end{aligned}$$

Here we see the subcategory `ListAggregate(S)` of the `Category`, `ListAggregate`. When using Axiom, one usually thinks of one's type as belonging to the smaller, concrete (sub)`Category`, `ListAggregate(S)`. Mathematically, one usually thinks of one's type as being in the larger, more abstract `ListAggregate`.

2.6 Newspeak

As an aside from the main computer algebra systems listed in this chapter, a research system called "Newspeak" [Fod83] (or more correctly, NEWSPEAK) will now be discussed.

Newspeak was a language written at the University of California at Berkeley in the early 1980s.

It differed from Axiom (then in the early stages of being Scratchpad II, otherwise called Newspad) by discarding the category mechanism and making a list of improvements, most of which are now in Axiom.

The differences are listed in [Fod83] and are summarised below. We shall also point out whether the differences still hold or not.

It should be pointed out that Barton (of the Scratchpad II design team) also had an implementation of Scratchpad II called Andante. Andante grew in parallel with Scratchpad II, but this chapter will refer mainly to Scratchpad II.

Specific advantages of Newspeak over Axiom which have not yet been incorporated into Axiom are,

1. Axiom's implementation language is not itself. Newspeak, however was written in Newspeak. This allowed the authors of Newspeak to write dedicated garbage collection, and fast function selection routines.

2. Function lifting: If one writes a `Category` in Axiom which has parameters from a very generic `Category`, such as `Ring`, then for certain functions to be available in the type one has to write code such as the following (from the definition of `FiniteAbelianMonoidRing(R:Ring, E:OrderedAbelianMonoid)`)

```
if R has IntegralDomain then
   "exquo": (%,R) -> Union(%,"failed")
   ++ exquo(p,r) returns the exact quotient of
   ++ polynomial p by r, or "failed"
   ++ if there is none exists.
if R has GcdDomain then
   content: % -> R
     ++ content(p) gives the gcd of the
     ++ coefficients of polynomial p.
   primitivePart: % -> %
     ++ primitivePart(p) returns the unit
     ++ normalized form of polynomial p
     ++ divided by the content of p.
```

Newspeak's "parameterised representationless types" (equivalent to Axiom's

`Categories`) get around this problem by extending (or in their doublethink wording, "restricting") the Polynomial constructor < `Poly` > to < `PolyId` > (Polynomials over Integral domains) etc.

To this author, this does not appear to solve the problem of having to know the category hierarchy of one's parameters. It merely abstracts it and moves the problem to a less satisfactory place.

Problems Scratchpad II/Axiom then had which have since been fixed,

1. A domain could only belong to one `Category`. (This was not true even then for Andante.)

2. Functions declared in `Categories` were only able to have their sources and targets from: a specific domain; an arbitrary type of *this* `Category`; the value of a parameter of *this* `Category`.

   Although Newspeak allowed functions between categories (which it called, "parameterised representationless types") which cannot be written in Axiom's Spad or Aldor languages; one could do this in Axiom's Boot language, but this is not a desirable option.

3. Axiom's items did not know their own type or that type's position in the hierarchy.

In conclusion, Newspeak was a brave attempt at rewriting Scratchpad II/Axiom without explicit categories, and solved many of the problems that the language then had. Fortunately (maybe thanks to Newspeak) these problems have since been redressed, and Axiom is all the more flexible and easier to use for it.

2.7 Magma

Magma, the system to replace Cayley, is a new and powerful computer algebra system. As with Cayley, the main purpose of Magma is to deal with discrete algebraic objects. While Cayley was useful as a tool for investigating groups, Magma can deal with many differing discrete algebraic objects.

Magma users are people more interested in branches of mathematics such as Semigroup theory, Group theory, Galois theory or other areas of modern algebra[10]. Magma does

[10]It should be noted that Axiom has some competence in all these areas, but for discrete mathematics, it can not match the pure speed of Magma.

have some ability to deal with the mainstream areas of computer algebra (polynomials, etc.) but this is not Magma's *raison d'être*.

The design philosophy of Magma [BCM94] takes its cues from Axiom, but goes further, supporting algebraic structures as first class objects. This makes Magma far more powerful than Axiom when dealing with algebraic structures. Indeed, manipulating algebraic structures is as easy as dealing with their elements in Magma. In Axiom, this is simply not the case.

A fundamental design difference between Axiom and Magma is the following. Axiom's designers took their cue from order-sorted algebra. Thus their definitions of types and their method of collecting them (`Categories`) focus on the operations which are valid on those types.

The Magma methodology is more representation centred. A category in Magma consists of all objects with a similar representation — the category of all permutation groups, for example. However, all the categories of different representations of an algebraic idea are collected together in a variety — the variety of groups, for example.

This is a subtle difference between Axiom and Magma, since Axiom views varieties as categories. At some future date, should Axiom ever support algebraic structures as first class objects, then the noticeable difference would be minimal.

Magma also supports the idea of order-sorted algebra more concretely than Axiom does. From the Magma WWW page [Pla96]:

> "The primary concept in the design of the Magma system is 'magma'. Following Bourbaki, a magma can be defined as a set with a law of composition.
>
> "Thus, types correspond to magmas; a collection of magmas sharing a common representation forms a category (e.g. the category of permutation groups); a collection of categories satisfying the same set of identical relations forms a variety (e.g. the variety of groups). Functors may be used to move between categories, and the variety operations (substructure, homomorphic image, and Cartesian product) are available as uniform constructors across all categories.
>
> "While every value in Magma belongs to a unique parent magma, the system provides a mechanism for automatic and forced coercion, to move a value from one magma to another. To take a simple example, when an integer is added to a rational, Magma automatically coerces the integer into the rational field, so that the addition operation can be carried out on two

values with the same parent."

This automatic coercion (also implemented in Axiom) is from the order on the sorts `R:Ring` $\prec$ `Fraction(R)`, in the theory of fractional (quotient) fields. ($\prec$, as we shall define in section 4.4 roughly means that there is some sort of "natural way" of mapping every element of `R` to one in `Fraction(R)`.)

2.8 OBJ

In this section we shall discuss how OBJ implements the ideas of order sorted algebra in a more concrete way than Axiom or Magma. Much of the information in this section can be found in [GWM$^+$93].

It should be noted that OBJ is not marketed as a computer algebra system, but as a comprehensive language with a strong mathematical foundation.

OBJ provides a strongly typed language based more closely on order sorted algebra than any other.

At OBJ's top level, there exist three types of entity,

1. objects: These are described in the literature as an encapsulation of executable code, or an algebra. Though it should be noted that an OBJ algebra need not be a model of any theory. (Note: OBJ documentation sometimes refers to the objects as Modules.)
2. theories: In the literature, theories are described as varieties, but in our language, OBJ's theories are indeed theories.
3. views: Views show how objects (algebras) satisfy or model certain theories.

In the next few sections we shall investigate these entities in more detail, so that we may compare them with Axiom's top level forms.

2.8.1 Objects

An OBJ-object is an executable algebraic specification for an abstract data type. That is to say, an OBJ-object defines a sorted signature (or indeed a theory, since equations are allowed) and provides the initial object of that theory[11].

[11] The initial object of a theory is the term algebra factored out by the equations. This is equivalent to the category theory notion of initiality.

The parameters of an OBJ-object can be typed, and forced to be models of particular theories. Indeed, OBJ-objects can be viewed as algebras modelling particular theories themselves, as we shall see in section 2.8.3.

2.8.2 Theories

OBJ-theories are indeed order sorted signatures with sets of equations. Semantically, OBJ-objects and OBJ-theories are very similar. However, one important difference is that the equations presented in a theory do not have to have to satisfy the same restrictions as those in an OBJ-object.

OBJ-objects must have equations which can be converted into rewrite rules, however, this restriction does not apply to OBJ-theory equations. This is because the OBJ-objects are executable, whereas OBJ-theories are not. (Remember that OBJ3[12] is based upon a rewriting system.)

2.8.3 Views

Views are the most exciting feature of OBJ. A view in OBJ is a method for stating that a particular OBJ-object models a particular OBJ-theory.

What makes OBJ unique is not the fact that the carriers may have different names to the sorts, but that the operator names may be different to the operator symbols.

2.9 Comparison

So how do our definitions of higher order entities, OBJ's and the other systems' compare?

[12]The current implementation of OBJ is called OBJ3.

Universal Algebra	**Category Theory**	**OBJ**		**Magma**	**Axiom**
		(abstract)	**(concrete)**		
Variety				Variety	
Theory	Category	Theory	Theory defined in an object		
Signature				Category	Category
Algebra	Object	View	Object	Magma	Domain

In the Universal Algebra column, we have written variety in a different row to theory. This is not quite correct, yet the two are different. A theory is a specification of a type, and a variety is the collection of all types with satisfy or model that specification.

There is a difference between theories and signatures, since signatures contain no information about equations. However, the difference is small; a theory with no equations is a signature.

We have slotted categories in as the equivalent of universal algebra's theories. Yet since everything may be thought of as a category, this decision is merely arbitrary. The specification of a category could be the same as a theory (indeed this is the case in Axiom), but the collection of all objects of that category would form a variety.

In the OBJ (abstract) column we refer to genuine OBJ theories declared to be theories (as opposed to those mentioned in the OBJ (concrete) column). OBJ's theories are most certainly theories (and the collection of views of a particular theory forms a variety). A view of an OBJ-theory is most certainly a model satisfying that theory, and hence an algebra of the signature of the theory, by definition.

In the OBJ (concrete) column the theories are those which exist by inference from an object definition. OBJ-objects define a singleton variety (defined by the inferred theory which the object models).This inferred theory is not an OBJ-theory. However, it is still a mathematical theory.

Magma has a view of varieties and categories matching in with the mathematical variety and signature definition. It should be noted that Magma's categories neither have an explicit notion of sortedness, nor one for equations. Though certain assertions, assumptions and statements may be made.

Axiom's categories (written `Categories` elsewhere in this thesis) are discussed at length in other chapters of this work.

Chapter 3

Category theory

3.1 Introduction

As we have already hinted at, category theory is one of the main foundations of both Magma and Axiom. In this chapter, we shall give the necessary definitions to investigate this claim. We shall also investigate the claim in itself.

The definitive text on Category Theory is Mac Lane [ML71]. The amount of theory we require is relatively small. We define categories without worrying about Russell's Paradox [Men87]. In appendix B we have a brief look at the complications which arise when one does.

3.2 Category theory

Definition 3.2.1 *A* category*, C, consists of two collections. The first collection is called the* objects *of the category, or* $\mathrm{Obj}(C)$*. The second collection is called the* arrows *of the category, or* $\mathrm{Arr}(C)$*.*

Also, for each arrow, f, there exists two special objects with which it is associated. The first is the source *of f, called* $\mathrm{source}(f)$*. The second is called the* target *of f, called* $\mathrm{target}(f)$*.*

*There also exists a "*law of composition*" for arrows:*

$$(\forall g, f \text{ arrows})((\mathrm{source}(g) = \mathrm{target}(f)) \Rightarrow$$
$$(\exists g \circ f \text{ arrow})((\mathrm{source}(g \circ f) = \mathrm{source}(f)) \wedge (\mathrm{target}(g \circ f) = \mathrm{target}(g))))$$

Next, for each object, c, there exists a unique arrow, called the identity arrow on c, *or* id_c.

Finally the following two axioms must hold:

$$(\forall k, g, f \text{ arrows})((g \circ f, k \circ g \text{ arrows}) \Rightarrow \\ ((k \circ (g \circ f), (k \circ g) \circ f \text{ arrows}) \wedge (k \circ (g \circ f) = (k \circ g) \circ f))) \\ (\forall f \text{ arrows})((\mathrm{id}_{\mathrm{target}(f)} \circ f = f) \wedge (f \circ \mathrm{id}_{\mathrm{source}(f)} = f))$$

Here endeth the definition.

To introduce the concept, here are some simple finite categories.

Example 3.2.2 **0** is the empty category, it has no objects and no arrows.

Example 3.2.3 **1** is the category with one object, and one arrow.

Example 3.2.4 **2** is the category with two objects, a, b and one non-identity arrow, $f : a \to b$.

Example 3.2.5 $\downarrow\downarrow$ is the category with two objects, a, b and two non-identity arrows, $f, g : a \to b$.

Now, let is consider some more useful categories. The collection of objects in the following examples do not always form a set [ML71][Ber91][Dev79][BHFL73].

Example 3.2.6 **Set** is the category which has as objects, all sets, and has as arrows, all total functions between sets.

Example 3.2.7 **Grp** is the category which has as objects, all groups, and has as arrows, all group homomorphisms between them.

Example 3.2.8 **Ring** is the category which has as objects, all rings, and has as arrows, all ring homomorphisms between them.

Example 3.2.9 **Poly** is the following category. An object of **Poly** is a set of all polynomials with: variables from a fixed ordered set, V; coefficients from a fixed ring, R; and the exponents of the variables from an ordered free monoid with one generator, E.

An arrow of **Poly** is a Polynomial-homomorphism. That is, a Ring-homomorphism which also acts homomorphically on a certain set of functions which act on polynomials.

Some texts call the arrows of a category the *morphisms* of a category. Examples 3.2.6 - 3.2.9 show us that for some naturally occurring categories, the arrows (or morphisms) are just the homomorphisms of the category. Indeed, an arrow of **Set** (a total function) is really just a homomorphism between sets.

So we see that the arrows preserve a certain set of properties for each element of the object.

We also see that every object of **Grp** is an object of **Set**; every object of **Ring** is an object of **Grp**; and every object of **Poly** is an object of **Ring**.

Now similarly, replace the word "object" with the word "arrow" in the previous paragraph and it still holds true.

Hence we see that in an algebra system, there is a certain amount of "inheritance" amongst the categories. The categories are the so-called *abstract datatypes* since they type the usual datatypes.

We can also see how the categories can collect together all similar types. This functionality can be used for the three design goals of Axiom: economy of effort — the code for many similar types need only be written once; interest — collecting similar types together gives the user an identical interface[1] to similar types; and functoriality which is best left to be discussed after the following definition.

Definition 3.2.10 *Let B, C be categories, then a* functor, *$T : C \to B$ consists of two functions:*

1. $T : \mathrm{Obj}(C) \to \mathrm{Obj}(B)$ *(the* object *function);*
2. $T : \mathrm{Arr}(C) \to \mathrm{Arr}(B)$ *(the* arrow *function).*

These must obey the following:

[1] Which has an entirely different, yet identical meaning in Java [Gra97]. Java does not allow multiple inheritance from classes, due to the usual problems. To work around this, Java provides "interfaces".

An interface is just like a class (although declared using different syntax) except that it can not provide a method for any operation. It may declare many (signatures of) operations available to any class which implements the interface.

This is similar to abstract types (classes with all functions "pure virtual" and no data members) in C++ [Str97].

$$\begin{gathered}(\forall f \text{ arrow})(T(\text{source}(f)) = \text{source}(T(f)));\\ (\forall f \text{ arrow})(T(\text{target}(f)) = \text{target}(T(f)));\\ (\forall c \text{ object})(T(\text{id}_c) = \text{id}_{T(c)});\\ (\forall g, f \text{ arrows})((\text{source}(g) = \text{target}(f)) \Rightarrow (T(g \circ f) = T(g) \circ T(f))).\end{gathered}$$

This is a very useful definition. It shows us that when we have two objects and an arrow between them in one category, then a "sensible" map of these objects to another category will induce the obvious map between these image objects.

In fact, one can see that if one were to define a category which had as objects "all categories", and as arrows "all functors" then we would (set theoretical concerns aside) have a well-defined category.

Categories, just like many other mathematical constructs, may form products.

Definition 3.2.11 *For two categories B and C we may construct a new category denoted $B \times C$ called the* product of B and C*.*

An object of $B \times C$ is a pair $\langle b, c\rangle$ where b is an object of B and c an object of C.

An arrow of $B \times C$ is a pair $\langle f, g\rangle$ where $f : b \to b'$ is an arrow of B, $g : c \to c'$ is an arrow of C, the source of $\langle f, g\rangle$ is $\langle b, c\rangle$ and the target of $\langle f, g\rangle$ is $\langle b', c'\rangle$.

Composition of arrows $\langle f', g'\rangle : \langle b', c'\rangle \to \langle b'', c''\rangle$ and $\langle f, g\rangle : \langle b, c\rangle \to \langle b', c'\rangle$ is defined via

$$\langle f', g'\rangle \circ \langle f, g\rangle = \langle f' \circ f, g' \circ g\rangle$$

Axiom makes use of such products implicitly in its functor definitions. For example, see the discussion of `PolynomialCategory`, below.

"Natural transformations" are another important part of category theory. They are to functors, as functors are to categories. Here is a more formal definition.

Definition 3.2.12 *For two functors $S, T : C \to B$ a* natural transformation $\tau : S \dot{\to} T$ *is a function which assigns to every $c \in \text{Obj}(C)$ an arrow $\tau_c = \tau c : Sc \to Tc$ of B such that*

$$(\forall c \in \text{Obj}(C))(\forall f : c \to c' \in \text{Arr}(C))(Tf \circ \tau c = \tau c' \circ Sf)$$

Now we may define the following interesting category.

Definition 3.2.13 *For two categories B, C, the* functor category B^C *is the category* *with objects the functors $T : C \to B$ and arrows, the natural transformations between two such functors.*

As another fairly abstract definition, consider the following.

Definition 3.2.14 *Let $S : D \to C$ be a functor and $c \in \mathrm{Obj}(C)$. An* universal arrow *$c \to S$ is a pair $\langle r, u\rangle \in \mathrm{Obj}(D) \times \mathrm{Arr}(C)$ where $u : c \to Sr$ such that*

$$(\forall \langle d, f\rangle \in \mathrm{Obj}(D) \times \mathrm{Arr}(C) \mathrm{where} f : c \to Sd)(\exists! f' : r \to d \in \mathrm{Arr}(D))(Sf' \circ u = f)$$

Functoriality in a computer algebra system allows us to view objects of one category as object of another.

As we have already seen, an object of **Poly** is an object of **Ring** and hence an object of **Grp** and thus an object of **Set**. We have also seen this is true for their arrows. This relationship is called the subcategory relationship, defined formally as follows.

Definition 3.2.15 *A category C is a* subcategory *of a category B iff every object of C is an object of B and every arrow of C is an arrow of B.*

Functors from subcategories to the categories of which they are subcategories, are often called "forgetful functors". This is more often true when the target of the functor is **Set**.

Axiom's designers also use functors to create instances of abstract datatypes. `PolynomialCategory(.,.,.)` (Axiom's equivalent of **Poly**) is in Axiom's view a functor,

$$\begin{aligned}\mathtt{Ring} \times \mathtt{OrderedAbelianMonoid} \times \mathtt{OrderedSet} &\to \mathtt{PolynomialCategory(.,.,.)}\\ \mathtt{(R,E,V)} &\mapsto \mathtt{PolynomialCategory(R,E,V)}\end{aligned}$$

This functoriality provides the "glue" for Axiom's type mechanism.

Now, trivially for categories A, B if $\exists F : A \to B$ a forgetful functor, then B is a subcategory of A. Equally trivially, a concrete instance of `PolynomialCategory(R,E,V)` is a subcategory of `PolynomialCategory(R,.,.)` which is a subcategory of **Poly**.

It is this first form of subcategory relation that provides Axiom's inheritance mechanism.

3.3 Categories and Axiom

Notation 3.3.1 *To distinguish between Axiom's internal structures and those commonly used in mathematics, things which belong to Axiom will be written in* `this font`*. Specifically,*

- `Category` *will always refer to Axiom's distinguished symbol, to which all Axiom's* `Categories` *belong.*
- *Hence, a* `Category` *is an Axiom object declared to be such an object. eg.* `Ring`, `PolynomialCategory(R,E,V)`.
- *A* `Domain` *is an Axiom object declared to be a member of a particular* `Category`*. eg.* `Integer`, `Polynomial(Integer)`.
- *An* `item` *is an element of a* `Domain`*. eg.* `1`, `5*x**2 +1`.

As stated in section 2.5, Axiom's main view of things is that every object has a type, and that there are four layers.

$$\texttt{items} \in \texttt{Domains} \in \texttt{Categories} \in \texttt{Category}$$

`Categories` also may inherit from or extend other `Categories`, forming an inheritance lattice. Thanks to the higher order polymorphism available in Axiom, `Categories` may also be parameterised by `items`[2] or `Domains`.

This parameterisation may cause some confusion. For example,

```
List(S) : ListAggregate(S)
```

declares for each and every `S`, `List(S)` is in the category `ListAggregate(S)`. For example `List(Integer)` is an object of `ListAggregate(Integer)`. However, `ListAggregate(Integer)` is a mere subcategory of "the category of all objects which are domains of linked lists". This is `ListAggregate(S)`.

So `ListAggregate(S)` contains, for example both `List(Integer)` and `List(Fraction(Integer))` as objects, but what are the arrows of this category? This

[2]One may view a domain as a category, whose objects are the items, and whose arrows are either trivial (ie. solely the identity arrows) or some other natural occurring meaning. eg. In `PositiveInteger`, one may think that there is a natural map $35 \to 5$, since $5 \mid 35$. This would then mean that this map could be "lifted" to `IntegerMod 35 -> IntegerMod 5`.

is something which is not made explicit in the Axiom literature, and is indeed the core of this thesis.

We shall discuss what it means to be a "natural map" and how this relates to categorical arrows. (section 5.4.)

3.4 Functors and Axiom

Axiom describes its domain constructors as functors, and this is true. After all, (neglecting difficulties with constructors which take domain elements for arguments) these constructors are maps from a cross product of categories to another category.

The difficulties with constructors which take domain elements for arguments disappear when considering the argument used in footnote 2 of section 3.3.

3.5 Coercion and category theory

If `coerce : A → B`, then we are going to have that `A` and `B` are objects of the same category, `ACat`, and that `coerce` is an arrow of that category. (See definition 5.5.2.)

Also, if `T` is a functor from `ACat` to `TCat`, which in Axiom would look like,

```
T(A:ACat) : TCat
```

then `T` lifts the arrows of `ACat` to `TCat`, and in particular,

```
T : coerce:A → B ↦ coerce:TCat(A) → TCat(B)
```

In many types (and some other languages) `T` is acting like the familiar "`map`" operator on the `coerce` function.

3.6 Conclusion

We have seen in this chapter how category theory and abstract datatyping especially with respect to Axiom's `Category` mechanism are ideologically similar.

We have also shown how Axiom's functors interact with coercions from a category theoretical perspective.

Chapter 4

Order sorted algebra

4.1 Introduction

In this chapter we shall introduce the concepts of universal algebra. We shall look at the unsorted case to start with and then move on to the sorted case.

We then follow up with the equational calculus which allows us to consider sets of equations which must hold in a concrete instance of an algebra.

All the work in this chapter is taken from [Dav93] except: example 4.2.8; the Perl example in section 4.4; and section 4.7.

4.2 Universal Algebra

Universal algebra will provide us with a natural way of representing categories of types which possess certain functions. In the following lengthy section of definitions, keep in mind that what we shall define as a "signature" will be equivalent (in some sense) to our notion of a category (or abstract datatype). An algebra will be the ideological equivalent of an object (or type).

Definition 4.2.1 *A* sort-list S*, is a (finite) sequence of symbols (called* sorts*) normally denoted* $(s_1, \ldots, s_m)$*.*

Definition 4.2.2 *Given a sort-list of size* m*, a set* A *of* S-carriers *is an ordered* m*-tuple of sets* A_{s_i} *indexed by* S*.*

Example 4.2.3 *One may consider the sort-list of a vector space to be (K, V) where K is to be identified with the underlying field and V is to be identified with the set of all points in the vector space. Then considering* $\mathbf{C}$ *to be a vector space over* R, *the (K, V)-carriers of* $\mathbf{C}$ *are* $(\mathbf{R}, \mathbf{C})$.

So we see that the carriers of a particular type are a list of types which "have something to do with" the type in question. The sort list is a list of the same length where the ith element is a symbol corresponding to a particular abstract datatype to which the ith carrier belongs[1].

Definition 4.2.4 *Given a sort-list S and $n \in \mathbf{N} \cup \{0\}$, an S-*arity* of* rank n *is an ordered n-tuple of elements of S.*

An arity is merely a list of elements of the sort list. This will be useful when we wish to type polymorphic (or abstract) functions.

Definition 4.2.5 *Given a sort-list S, $n \in \mathbf{N} \cup \{0\}$, $q = (q_1, \dots, q_n)$ an S-arity of rank n, and a set A of S-carriers, we define*

$$A^q := \prod_{i \in \{1, \dots, n\}} A_{q_i}$$

This is the map of an arity to the list of carriers.

Definition 4.2.6 *An S-*operator of arity q *is a function from A^q to one of the elements of A.*

Definition 4.2.7 *An S-*operator set *or S-*sorted signature *is a set Σ of sets $\Sigma_{n,q,s}$ indexed by $n \in \mathbf{N} \cup \{0\}$, q an S-arity of rank n, and $s \in S$, such that $\bigcup_{n,q,s} \Sigma_{n,q,s}$ is a subset of some alphabet. An element of some $\Sigma_{n,q,s}$ is called operator symbol.*

The usual notation for such a signature is $\langle \Sigma, S \rangle$.

So this defines us a set of sets of polymorphic functions[2] for a particular sort-list. A signature is like a category in that it is an abstract datatype.

A signature collects together all types which share a similar family of operators.

[1]Notice that the sort list is defined first and that the carriers depend on the sort list. One does not define a list of carriers and then fix a list of abstract datatypes *post facto*.

[2]Functions without methods.

Example 4.2.8 *Monoids:* $\mathbf{N}\cup\{0\}$ *is an additive monoid, whereas* $\mathbf{N}$ *is a multiplicative monoid.*

The signature for monoids could be viewed as being

$$\langle \Upsilon, U\rangle = \langle (M,B),((e),(),(id),(ep),(),(),(b),()\ldots)\rangle$$

where M *is the monoid sort,* B *is the sort of boolean logic types. The operator symbol* e *is a member of* $\Upsilon_{(0,(),M)}$ *and corresponds to the function which always returns the identity constant*[3] *of the monoid.*

id *is the operator symbol in* $\Upsilon_{(1,(M),M)}$ *which corresponds to the identity function.*

ep *is the operator symbol in* $\Upsilon_{(1,(M),B)}$ *which corresponds to the "is this the identity element?" function.*

Last (in our example) but by no means least, b *is the monoid's binary operator from* $\Upsilon_{(1,(M,M),M)}$.

So in $\mathbf{N}\cup\{0\}$, e, id, ep, b *correspond to* $0, id, 0?, +$ *respectively. Whereas in* $\mathbf{N}$, e, id, ep, b *correspond to* $1, id, 1?, \times$ *respectively.*

Definition 4.2.9 *A* (multi-sorted, total) Σ-algebra *is an ordered pair* $\langle A,\alpha\rangle$ *where* A *is an* S*-carrier set and*

$$\alpha = \{\alpha_{n,q,s} | n\in \mathbf{N}\cup\{0\}, q \text{ an arity of rank } n, s\in S\}$$

$$\alpha_{n,q,s} = \{\alpha_{n,q,s,\sigma} : A^q \to A_s\}_{\sigma\in\Sigma_{n,q,s}}$$

So if $\langle\Upsilon, U\rangle$ is the monoidal signature, then

$$\langle(\mathbf{N},\texttt{Boolean}),((1),(),(id),(1?),(),(),(\times),\ldots)\rangle$$

is an Υ-algebra.

Notation 4.2.10 *Let* $\langle A,\alpha\rangle$ *be a* Σ*-algebra. For an operator symbol* $\sigma_{n,q,s}$ *of the signature* $\langle\Sigma, S\rangle$ *the function associated with this symbol in* $\langle A,\alpha\rangle$ *is represented by either* $\alpha_{n,q,s,\sigma}$ *or* $\alpha_{\sigma_{n,q,s}}$.

[3] Constants are often represented by (if not compiled in the same way as) functions in programming languages, such as Axiom's Spad language. This gives a homogeneous interface for providing constants.

This second form of notation is useful for when we refer to operator symbols by names other than those in the form $\sigma_{n,q,s}$. For example, if $\tau = \sigma_{n,q,s}$ then

$$\alpha_{n,q,s,\sigma} = \alpha_{\sigma_{n,q,s}} = \alpha_\tau$$

Definition 4.2.11 *Given $\langle A, \alpha\rangle$, $\langle B, \beta\rangle$ both Σ-algebras, then a* Σ-homomorphism $\phi : \langle A, \alpha\rangle \to \langle B, \beta\rangle$ *is an S-indexed family of functions $\phi_s : A_s \to B_s$ (for each s in S) such that:*

$$(\forall n \in \mathbf{N} \cup \{0\})(\forall q \text{ arities of rank } n)(\forall (a_1, \ldots, a_n) \in A^q)(\forall s \in S)(\forall \sigma \in \Sigma_{n,q,s})$$

we have,

$$\phi_s(\alpha_{n,q,s,\sigma}(a_1, \ldots, a_n)) = \beta_{n,q,s,\sigma}(\phi_{q_1}(a_1), \ldots, \phi_{q_n}(a_n)).$$

As an example of a homomorphism, let us consider our monoidal case once more. The map ψ from $\mathbf{N} \cup \{0\} \to \mathbf{N}$ which maps[4] $n \mapsto 2^n$ is a homomorphism.

4.3 Term Algebras

Term algebras provide us with at least one example of an algebra for each signature. In some sense, it is the "freest" algebra of the signature and all other algebras are isomorphic to quotients of the term algebra.

Notice that given $\langle \Upsilon, U\rangle$ the definition of the monoidal signature from example 4.2.8, the term algebra is not the free monoid, since we have not added in any "laws" or "equations" to the algebra. Thus we do not have associativity in the term algebra, whereas we do in the free monoid.

Thus we see that, as yet, universal algebra does not model real mathematics perfectly. However this situation will be remedied somewhat in section 4.6.

Notation 4.3.1 *We shall define the set Δ to be the set containing three special symbols.*

$$\Delta := \{(\} \cup \{)\} \cup \{,\}$$

Δ is a just a piece of notation that will make the following definition less verbose.

[4]We haven't provided an exponentiation function in our monoidal algebra, but adding an extra sort and a function it is possible. Otherwise think of the map as taking 0 to 1; 1 to 2; 2 to 2×2; 3 to $2 \times 2 \times 2$; and so on.

Definition 4.3.2 *Let X be an S-indexed family of sets disjoint from each other, from the set Δ and from $\bigcup_{n,q,s} \Sigma_{n,q,s}$. We define $T_\Sigma(X)$ to be the S-indexed family of sets of strings of symbols from $\bigcup_{s \in S} X_s \cup \Delta \cup \bigcup_{n,q,s} \Sigma_{n,q,s}$, each set as small as possible satisfying these conditions:*

1. $(\forall s \in S)(\Sigma_{0,(),s} \subseteq T_\Sigma(X)_s)$
2. $(\forall s \in S)(X_s \subseteq T_\Sigma(X)_s)$
3. $(\forall \sigma \in \Sigma_{n,q,s})(\forall i \in \{1,\ldots,n\})(\forall t_i \in T_\Sigma(X)_{q_i})(\sigma(t_1,\ldots,t_n) \in T_\Sigma(X)_s)$

We make $T_\Sigma(X)$ into a Σ-algebra by defining operators σ_T on $T_\Sigma(X)$, for each $\sigma \in \Sigma_{n,q,s}$ via:

- *If $n = 0$ then $\sigma_T := \sigma$. (Guaranteed to be in $T_\Sigma(X)_s$ by (1)).*
- *Else, define $\sigma_T(t_1,\ldots,t_n)$ to be the* string $\sigma(t_1,\ldots,t_n)$

$T_\Sigma(X)$ is called the term algebra*, and an element of $T_\Sigma(X)$ is called a* term*.*

4.4 Order-sorted algebras

Order sorted algebras extend the concept of of universal algebras by imposing an order on the elements of the sort-list. This can be useful if we know that all algebras of our signature $\langle \Sigma, S \rangle$ have the carrier of S_2 as a subset of S_4, say.

The ordering of sorts imposes a subtype lattice on the sorts (and hence their carriers). This can then be used in any algebra of the signature to either restrict a function already defined on one type to a subtype of that type, or extend a function on a type to a partial function on a supertype. Partial functions and order sorted algebra are discussed in section 5.2.

Firstly, we had better define what we mean by an "order".

Definition 4.4.1 *A* strict partial order *on a set S is a relation $\prec$ on S which is transitive, antisymmetric and irreflexive.*

A weak partial order *on a set S is a relation $\preceq$ such that, $a \preceq b \Leftrightarrow (a \prec b) \vee (a = b)$. Such a relation is transitive.*

Notation 4.4.2 *Let S be a set of sort symbols, such that there is a partial order $\prec$ defined on S, and a "top" element u of S such that $s \preceq u$ for all s in S.*

u provides us with a universe (see appendix B.2) in which to work. Equivalently, we could set u to be any class$_\omega$ which is "big enough". (The term class$_\omega$ is defined in appendix B.2).

Definition 4.4.3 *Extend $\preceq$ from S to the S-arities of rank n by defining $(s_1, \ldots, s_n) \preceq (t_1, \ldots, t_n)$ iff $(\forall i \in \{1, \ldots, n\})(s_i \preceq t_i)$.*

Definition 4.4.4 *An* order-sorted, total Σ-algebra *is an ordered triple $\langle A, \{A_s : s \in S\}, \alpha$, where A is a class known as the* universe, *$\{A_s : s \in S\}$ is an S-indexed family of subsets of A, known as the* carriers *of the algebra, and α is a set of sets of functions $\alpha_{n,q,s} = \bigcup_{\sigma \in \Sigma_{n,q,s}} \{\alpha_{n,q,s,\sigma} : A^q \to A_s\}$, such that:*

1. *$A_u = A$;*
2. *If $s \preceq s'$ in S, then $A_s \subseteq A_{s'}$;*
3. *If $\sigma \in \Sigma_{n,q,s} \cap \Sigma_{n,q',s'}$, with $s \preceq s'$ and $q' \preceq q$,then $\alpha_{n,q,s,\sigma} \mid_{A^{q'}} = \alpha_{n,q',s',\sigma}$*

Strictly speaking, that last condition should be,

$$\iota_{A_s \to A_{s'}} \circ \alpha_{n,q,s,\sigma} \mid_{A^{q'}} = \alpha_{n,q',s',\sigma}$$

(where $\iota_{A_s \to A_{s'}}$ is the inclusion operator $A_s \to A'_s$) since otherwise the target of the left hand side would be A^s and of the right hand side would be $A^{s'}$.

This is typical of the sort of "abuse of notation" that computer systems often have to implement.

In a language like C [KR88], there is not much scope for such abuse. The compiler will complain if we attempt to change the type of any variable.

However, in a language like Perl [WCS96] where type changes are performed automatically, abuse is everything. For example to convert the string `"12"` to a number, one simply adds `0` to it.

The following command line session shows the abuse Perl allows.

```
[nic@pangur-ban Tmp]$ perl -e '
> $a_string = "12";
> $a_number = 12;
> print $a_string;
> print "\n";
> print $a_number;
> print "\n";
> print $a_string + $a_number;
> print "\n";
> $a_string .= "text";
> print $a_string;
> print "\n";
> '
12
12
24
12text
```

`print "\n";` means print carriage return, linefeed (as in C). See how `$a_string` is treated as a number when added to `$a_number` and a string when we append (using the `.=` operator) some text to it.

Notice that the opposite can be performed, too. We may treat numbers as strings. We have already seen this with the line `print $a_number;` but as a more explicit example, we can append text to a number.

```
[nic@pangur-ban Tmp]$ perl -e '
> $a_number = 12;
> $a_number .= "text";
> print $a_number;
> print "\n";
> '
12text
```

Perl is incredibly good at all such built in type changes. However, Perl is not a computer algebra system and does not have the vast number of types that Axiom has.

Perl also has a (fixed) universe (see appendix B.2 and definition 4.4.4) and knows how to retract any variable back to the universe type. From this it can see how to coerce any two objects to types over which an operator (such as `print`, `+` or `.`) is valid.

The definition of order sortedness ensures some confluence[5] amongst operators on subtypes. However, it does not provide enough for most sensible applications. The following definition ensures a more confluent system.

Definition 4.4.5 *The order-sorted signature Σ is* regular *if whenever $\tilde{q}$ is an arity and $\sigma \in \Sigma_{n,q,s}$ with $\tilde{q} \preceq q$, there is a least pair q', s' such that $\sigma \in \Sigma_{n,q',s'}$ and $\tilde{q} \preceq q'$ and $s' \preceq s$.*

Now, we shall extend the definition of term algebras to the order sorted case.

Definition 4.4.6 *Let X be an S-indexed family of sets disjoint from each other, from the set Δ and from $\bigcup_{n,q,s} \Sigma_{n,q,s}$. We define $T_\Sigma(X)$ to be the S-indexed family of sets of strings of symbols from $\bigcup_{s \in S} X_s \cup \Delta \cup \bigcup_{n,q,s} \Sigma_{n,q,s}$, each set as small as possible satisfying these conditions:*

1. $(\forall s \in S)(\Sigma_{0,(),s} \subseteq T_\Sigma(X)_s)$
2. $(\forall s \in S)(X_s \subseteq T_\Sigma(X)_s)$
3. $(\forall s, s' \in S)((s \preceq s') \Rightarrow (T_\Sigma(X)_s \subseteq T_\Sigma(X)_{s'}))$
4. $(\forall \sigma \in \Sigma_{n,q,s})(\forall i \in \{1, \ldots, n\})(\forall t_i \in T_\Sigma(X)_{q_i})(\sigma(t_1, \ldots, t_n) \in T_\Sigma(X)_s)$

We make $T_\Sigma(X)$ into a Σ-algebra by letting the first component be $T_\Sigma(X)_u$ and defining operators σ_T on $T_\Sigma(X)$, for each $\sigma \in \Sigma_{n,q,s}$ via:

- *If $n = 0$ then $\sigma_T := \sigma$. (Guaranteed to be in $T_\Sigma(X)_s$ by (1)).*
- *Else, define $\sigma_T(t_1, \ldots, t_n)$ to be the* string $\sigma(t_1, \ldots, t_n)$

$T_\Sigma(X)$ is called the term algebra, *and an element of $T_\Sigma(X)$ is called a* term.

The following theorem proves the "freeness" of term algebras. That is to say all algebras are isomorphic to a quotient of the term algebra. In this way we see that all algebras, once represented as a term algebra and a set of rewrite rules are easily implementable in a rewrite system, such as OBJ[6].

[5]Perhaps this is why OBJ is a rewriting system.

[6]With the usual caveats on being able to check whether two elements are equal or not and other unsolvable problems.

It also says something about the constructibility of types. That is, if $\bigcup_{n,q,s} \Sigma_{n,q,s}$ is finite then clearly only a finite number of functions in each and every Σ-algebra construct the whole algebra.

Moreover, suppose in every real algebra which we wish to study, a certain set of equations hold (see section 4.6). Then all our algebras are isomorphic to factors of the term algebra factored out by that set of equations.

Then if every element of this freest factor algebra is equal to one constructed by a (potentially very small, finite) subset of a (potentially infinite) $\bigcup_{n,q,s} \Sigma_{n,q,s}$, we may utilise this to construct elements of our algebra.

In the automated coercion algorithm (section 7.3) we utilise a small (but not necessarily minimal) set to construct all (or some) of the elements of one of the sorts of an algebra.

Theorem 4.4.7 (First universality theorem) *Let $\langle A, \alpha\rangle$ be any Σ-algebra, θ any map (S-indexed family of maps) from X into A. Then there exists a unique Σ-homomorphism θ^* from $T_\Sigma(X)$ to A such that $(\forall s \in S)(\forall x \in X_s)$*

$$\theta^*_s(\iota(x)) = \theta_s(x)$$

The proof may be found in [Dav93].

4.5 Extension of signatures

In this section we shall formalise what we mean for one algebra to be an extension of another, or more importantly, from our point of view, for one algebra to be a portion of another.

More formally we are saying how the abstract datatypes (or categories or signatures) may inherit from each other. This sometimes corresponds to algebras depending on each other.

First, a piece of notation.

Notation 4.5.1 *If S and T are two sets of sets both indexed by the same set, I, we say that $S \vec{\subseteq} T$ iff $(\forall i \in I)(S_i \subseteq T_i)$.*

This extends the definition of subsets to n-tuples of sets.

Definition 4.5.2 *Let S, S' be two sort-lists, such that, as sets of symbols $S \subseteq S'$, and* *that* $(\forall s, t \in S)((s \preceq_S t) \Leftrightarrow (s \preceq_{S'} t))$. *Let* Σ *be an* S*-sorted signature and* T *be a* S'*-sorted signature. If* $\Sigma \vec{\subseteq} \mathrm{T}$, *we say that* Σ *is a* sub-signature *of* T *and that* T *is a* super-signature *of* Σ.

Thus we have the obvious definition of sub-signature. As an example, the monoidal signature is a sub-signature of the signature for groups. The notion of sub-signature corresponds directly with that of sub-category.

Definition 4.5.3 *With* $S, S', \Sigma, \mathrm{T}$ *as in the previous definition, let* $\langle A, \alpha\rangle$ *be an* S'*-sorted* T *algebra. Define* $\langle A, \alpha\rangle \mid_{\Sigma}$, *called* $\langle A, \alpha\rangle$ restricted to Σ, *to be the* S*-sorted* Σ*-algebra with carriers, those carriers of* $\langle A, \alpha\rangle$ *which are indexed by sort symbols from* S, *and operators,* n*-ary operators* α_τ *of arity* q, *and result sort* s, *for every* $\tau \in \Sigma_{n,q,s}$.

This is applying the forgetful functor (from the category (corresponding to) T to that (corresponding to) Σ) to $\langle A, \alpha\rangle$.

4.6 The equational calculus

Again, we borrow heavily from Davenport's lecture notes [Dav93]. Throughout this section, we assume that $S = \{s_1, \ldots, s_n\}$ is our indexing family of sort symbols, and $\Sigma = \{\Sigma_{n,q,s}\}$ is a S-sorted signature.

The equational calculus presented here applies to multi-sorted algebra. The reader will see that it clearly may be extended to the order-sorted case.

The equational calculus allows us to add "equations" to our signatures (and hence, algebras). This will allow us to assert facts about all the algebra in a signature. For example, we may wish note that one of the binary operators is always associative. Other more complicated expressions are available also.

The equational calculus does not allow us to define everything that we need: only those things that are easily definable as equations.

Definition 4.6.1 *A* S*-indexed family of relations* $R = \{R_{s_1}, \ldots, R_{s_n}\}$ *on a* Σ*-algebra,* $\langle A, \alpha\rangle$, *is called a* Σ-congruence *if it satisfies the following four families of conditions:*

$$\begin{array}{rcll} a \in A_{s_i} & \Rightarrow & aR_{s_i}a & (R) \\ aR_{s_i}b & \Rightarrow & bR_{s_i}a & (S) \\ aR_{s_i}b \text{ and } bR_{s_i}c & \Rightarrow & aR_{s_i}c & (T) \\ (\forall \sigma \in \Sigma_{n,q,s})(a_1R_{q_1}b_1, \ldots . a_nR_{q_n}b_n & \Rightarrow & \sigma(a_1, \ldots, a_n)R_s\sigma(b_1, \ldots, b_n)) & (C_\sigma) \end{array}$$

The first three conditions imply that R_{s_i} is an *equivalence relation* on the set A_{s_i}, while the conditions (C) explain how R relates to the various operators of Σ. The operators of $\Sigma_{n,q,s}$ are well-defined on the equivalence classes.

Definition 4.6.2 *Let $\langle A, \alpha\rangle$ be a Σ-algebra, and $\equiv$ be a Σ-congruence on $\langle A, \alpha\rangle$. Define $\langle A, \alpha\rangle / \equiv$ to be the Σ-algebra $\langle B, \beta\rangle$, where the carrier set B_{s_i} of B is the set of equivalence classes of A_{s_i} under the equivalence relation $\equiv_{s_i}$, and β_σ is the operator defined by*

$$\beta_\sigma([a_1],[a_2],\ldots,[a_n]) = [\alpha_\sigma(a_1,a_2,\ldots,a_n)]$$

for every operator symbol σ in every $\Sigma_{n,q,s}$.

This is merely the quotient algebra, and to be sure, the following is a theorem.

Theorem 4.6.3 *The operators β_σ in the above definition are well-defined.*

The following definition actually turns out to be very important.

Definition 4.6.4 *We say that a sort s is* void *in the signature Σ if $T_\Sigma(\emptyset)_s = \emptyset$.*

Basically, having a void sort $\tilde{s}$ in the signature means that there are no constants of that type ($\Sigma_{0,(),\tilde{s}} = \emptyset$) and that one of the following is true,

- no operators have $\tilde{s}$ as a return type: $((\forall q, n)(\Sigma_{n,q,\tilde{s}} = \emptyset)$ where q is an arity of rank n)
- every operator with $\tilde{s}$ as a return type has an argument whose sort is either void or $\tilde{s}$: $((\forall q, n)((\Sigma_{n,q,\tilde{s}} \neq \emptyset) \Rightarrow (\exists i \in \{1,\ldots,n\})(q_i$ is a void sort or $\tilde{s}))$ where q is an arity of rank n)

Lemma 4.6.5 *If s is not void in the signature Σ, then in every Σ-algebra$\langle A, \alpha\rangle$, $A_s \neq \emptyset$.*

Goguen & Meseguer suggest the following rules of deduction for a sound multi-sorted equational calculus.

Notation 4.6.6 *Let X be a S-indexed family of sets of variable symbols, such that each X_{s_i} is disjoint from all the others, from the operator symbols of Σ, and from any symbols in any particular algebras we may be reasoning over.*

We will be reasoning frequently with S-indexed families of sets, and wishing to perform operations on them. Let $X = \langle X_1, \ldots, X_n \rangle$ and $Y = \langle Y_1, \ldots, Y_n \rangle$ be two S-indexed families of sets, and define $X \vec{\cup} Y$ to be the S-indexed family $\langle X_1 \cup Y_1, \ldots, X_n \cup Y_n \rangle$. Similarly, we will write $X \vec{\subseteq} Y$ to indicate that each component of X is a subset of the corresponding element of Y.

A final piece of notation is that the indexed family of empty sets will be denoted by $\vec{\emptyset}$.

Definition 4.6.7 *A* multi-sorted equation *for the signature Σ consists of a triple (Y, t_1, t_2), where $Y \vec{\subseteq} X$, t_1 and t_2 are terms from the same carrier set of $T_\Sigma(X)$ (or $T_\Sigma(X \vec{\cup} A)$ if we are dealing with equations in a particular algebra $\langle A, \alpha \rangle$), and every variable occurring in t_1 and t_2 occurs in the appropriate member of Y: $t_1, t_2 \in T_\Sigma(Y)$. Such equations are written $\forall Y \quad t_1 = t_2$.*

If t_1 and t_2 come from the same carrier set of $T_\Sigma(X)$ we say that the equation is Σ-generic.

One should read the symbol $\forall Y$ as meaning "for all values of all the variables of Y in the appropriate sorts (and there had better be some values in those sorts)". It is this interpretation that will solve the paradox mentioned earlier.

Definition 4.6.8 *An* equational system *for the signature Σ is a set of equations for $T_\Sigma(X)$ (or $T_\Sigma(X \vec{\cup} A)$ if we are dealing with equations in a particular algebra $\langle A, \alpha \rangle$).*

We will tend to write $e = f$ for an equation from an equational system, meaning $\forall Y \quad e = f$ where Y is the S-indexed family of sets of variables consisting *precisely* of those variables occurring in e and f.

Notation 4.6.9 *x_i/t_i means substitute the variable x_i with the term t_i. We call this a* substitution instance.

Definition 4.6.10 *A* proof in the multi-sorted equational calculus *of the equation $\forall Y \quad e = f$ from the equational system $\mathcal{E}$ is a finite set of equations $\forall Y_i \quad e_i = f_i$ such that each equation is justified by one of the seven following rules of inference:*

$$\vec{\forall} Y \quad e[x_1/t_1, \ldots, x_n/t_n] = f[x_1/t_1, \ldots, x_n/t_n] \qquad (E)$$

where $\forall X \quad e = f$ is an equation of $\mathcal{E}$, the t_i are terms of the appropriate sort of $T_\Sigma(X)$ and Y is the S-indexed family of sets whose s-th component is the set of all variables *of sort s in all the t_i and those variables of X_s which have not been substituted for, i.e. which are not one of the x_i;*

$$\overline{\forall} Y \quad e = e \tag{R}$$

where Y is the S-indexed family of sets whose s-th component is the set of all variables of sort s in e;

$$\forall Y \quad e = f \quad \overline{\forall} Y \quad f = e \tag{S}$$

$$\forall Y \quad e = f \quad \forall Y' \quad f = g \quad \overline{\forall} Y \vec{\cup} Y' \quad e = g \tag{T}$$

$$\forall Y_1 \quad e_1 = f_1 \quad \ldots \quad \forall Y_n \quad e_n = f_n \quad \overline{\forall} Y_1 \vec{\cup} \ldots \vec{\cup} Y_n \quad \sigma(e_1, \ldots, e_n) = \sigma(f_1, \ldots, f_n) \tag{C_σ}$$

where σ is any symbol of $\Sigma_{n,q,s}$, and each e_i is a term of sort q_i;

$$\forall Y \quad e = f \quad \overline{\forall} Y' \quad e = f \tag{A}$$

where $Y \vec{\subseteq} Y'$;

$$\forall Y \quad e = f \quad \overline{\forall} \langle Y_1, \ldots, Y_{i-1}, Y_i \setminus \{y\}, Y_{i_1}, \ldots, Y_n \rangle \quad e = f \tag{Q}$$

where y does not occur in e or f, and the sort i is non-void for Σ.

We use the notation $\vdash \forall Y \quad e = f$ (or $\vdash_{\mathcal{E}} \forall Y \quad e = f$ if we wish to make clear which equational system is being considered) to mean that $e = f$ is provable in the equational system using the above rules of inference.

Definition 4.6.11 *If $\forall Y \quad e = f$ is an S-sorted Σ-equation (call it E), and $\langle A, \alpha \rangle$ is a Σ-algebra, then we say that $\langle A, \alpha \rangle$* satisfies *$E$, or that $\langle A, \alpha \rangle$is a* model *for E, if for all S-sorted maps θ from Y to A, $\theta^*(e) =_A \theta^*(f)$, where θ^* is the map from $T_\Sigma(X \vec{\cup} A)$ to $\langle A, \alpha \rangle$, whose existence is guaranteed by theorem 4.4.7. We extend the notation to sets of equation $\mathcal{E}$ by insisting that $\langle A, \alpha \rangle$ be a model for each equation in $\mathcal{E}$.*

Theorem 4.6.12 (The Soundness Theorem) *If $\langle A, \alpha \rangle$ is a model for $\mathcal{E}$, and $\vdash_{\mathcal{E}}$ $\forall Y \quad e = f$, then $\langle A, \alpha \rangle$ is a model for $\mathcal{E} \cup \{\forall Y \quad e = f\}$*

The proof of the soundness theorem may be found in [GM82].

Definition 4.6.13 *Let $\langle A, \alpha \rangle$ be a Σ-algebra, and let $\mathcal{E}$ be an equational system for $\langle A, \alpha \rangle$. The* congruence induced by *$\mathcal{E}$, denoted $\equiv_{\mathcal{E}}$, on $\langle A, \alpha \rangle$ is defined by $A \equiv_{\mathcal{E}} B$ if, and only if, $\vdash_{\mathcal{E}} \forall Y \quad a = b$, where Y is the S-indexed family of empty sets.*

The following two definitions are very important. A theory specifies a type, and we define a variety to be the collection of all types which model that theory.

Definition 4.6.14 *We define a* theory *to be the ordered pair $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$ where Σ is an S-sorted signature and $\mathcal{S}$ is a set of S-sorted Σ-equations. We say that a Σ-algebra* models *or* satisfies *the theory iff it is a model for $\mathcal{S}$.*

Definition 4.6.15 *The collection of all models of a particular theory is called a* variety.

Clearly, since a signature $\langle\Sigma, S\rangle$ may be viewed as the theory $\langle\langle\Sigma, S\rangle, \emptyset\rangle$ the collection of all Σ-algebras forms a variety.

When referring to the variety of all models of a particular signature, (for example, $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$) we usually say the "variety defined by (or specified by) the signature $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$."

Theorem 4.6.16 (Second universality theorem) *Let X be a S-sorted set of variables, Σ an S-sorted operator set, $\mathcal{E}$ a set of equations for Σ, $\langle A, \alpha\rangle$ a Σ-algebra which is a model for $\mathcal{E}$, and θ a S-sorted mapping from X to A. Then there is a unique Σ-homomorphism θ^{**} from $T_\Sigma(X)/\equiv_{\mathcal{E}}$ to A such that*

$$\theta^{**}(\iota^*(x)) = \theta(x) \qquad (**)$$

for all $x \in X$, where ι^* is the map from X into $T_\Sigma(X)/\equiv_{\mathcal{E}}$ defined by $x \mapsto [x]$.

Theorem 4.6.17 (The Completeness Theorem) *If every Σ-algebra which satisfies the equation $\mathcal{E}$ also satisfies the equation $\forall Y \quad e = f$, then $\vdash_{\mathcal{E}} \forall Y \quad e = f$.*

Again, the proof of this may be found in [GM82].

Finally, we state the definition of an extension and a protecting extension. Extensions are self-explanatory.

Definition 4.6.18 *Suppose that Σ and $\Sigma\vec{\cup}\mathrm{T}$ are signatures where Σ is S-sorted and $\Sigma\vec{\cup}\mathrm{T}$ is S'-sorted where $S \subseteq S'$.*

Then $\Sigma\vec{\cup}\mathrm{T}$ is said to be an extension *of Σ.*

A protecting extension is an extension which preserves the equational system for a theory. Combining the second universality theorem 4.6.16 with the definition of protecting extension allows us to view a model of a theory as a model of a protecting extension of that theory.

More importantly, if we have an algebra which is a model for a protecting extension of a theory, then we may view the algebra as a model of that theory (unextended).

For example, if the `Ring` theory is defined to be a protecting extension of the `Group` theory, then we may view any ring (`Ring`-algebra) as a group (`Group`-algebra).

Definition 4.6.19 *With Σ,* T*, S and S' as in definition 4.6.18.*

Let $\mathcal{E}$ be an equational system on Σ. Also let $\mathcal{E} \cup \mathcal{F}$ be an equational system on $\Sigma\vec{\cup}\mathrm{T}$. Such an extension is called a protecting extension *if*

$$T_{\Sigma\vec{\cup}\mathrm{T}}(X)/\equiv_{\mathcal{E}\cup\mathcal{F}}|_{\Sigma}$$

is isomorphic to $T_{\Sigma}(X)/\equiv_{\mathcal{E}}$ (isomorphism meaning "isomorphism as Σ-algebras").

Notice that the definition of protecting extension is equivalent to the notion of *enrichment* given in [Pad80] (although, this does not allow for the existence of S'). Thus a theory Ω_1 is a protecting extension of another Ω_0 iff Ω_1 is complete and consistent with respect to Ω_0.

4.7 Signatures, theories, varieties and Axiom

Axiom's type system uses the terminology from category theory, yet its design is based on order sorted signatures.

A `Category` definition in Axiom (ie. the source code that defines the `Category`) is equivalent to a signature or theory, being a specification of a type or types.

The `Category` viewed as a collection of objects (`Domains`) is the variety specified by the theory which defines the `Category` in the `Category` definition.

The sorts are `%` for the (principal) sort (see definition 7.2.1), and the argument and return types of all the operator symbols of the `Category`[7].

For example, `PolynomialCategory(R,E,V)` is a `Category`. `%` is the (principal) sort. Other sorts include `R`, `E`, `V` a sort each for the `Boolean` and `PositiveInteger` types.

The equations of a theory in an Axiom `Category` definition are either defined in the

[7]Unless they are concrete types — `Domains`. See section 9.6.

comments or as certain "attributes" of operators. For example `commutative(*)` means that `*` is commutative and the traditional commutativity equation (for the operator `*`) is an equation of the theory. (There is currently no method for enforcing the equations to hold in any model.)

An algebra (or model for a theory) in Axiom is a `Domain`. Declaring a `Domain` to be in a `Category` is equivalent to saying that is an algebra of that signature (or model of that theory). Or in other words, a member of the variety defined by the theory which specifies the `Category`.

For example, `Polynomial(Integer)` is a model for `PolynomialCategory(R,E,V)`.

An operator symbol in Axiom is a function declaration in a `Category`. An operator name in Axiom always corresponds to the operator symbol.

For example, `+` is an operator symbol in `Ring`. In `Integer`, a model for `Ring` the operator name of `+` is (and has to be) `+`.

A carrier is the concrete type substituted for a parameter in any instantiation of a `Domain`.

In `Polynomial(Integer)`, `Polynomial(Integer)` is the carrier `%`. Whereas `Integer`, `NonNegativeInteger` and `Symbol` are the carriers of `R`, `E` and `V`, respectively. (No sort is given for the `Boolean` and `PositiveInteger` types.)

All of Axiom's extensions are protecting extensions. That is, if a `Category` is declared to extend another then it is always a protecting extension of that `Category`.

In chapter 9 we shall discuss how Axiom differs from order sorted algebra and how we may remedy this situation.

4.8 Conclusion

In this chapter we have introduced all the basics of order sorted algebra and the equational calculus. We have also shown how these notions are represented in Axiom.

We have demonstrated that order sorted algebra combined with the equational calculus provides a sound basis for a computer algebraic type system.

A multi sorted signature is a specification for a type and hence is an abstract datatype.

Adding an order on the sorts to obtain an order sorted signature enforces relations between the sorts. It also guarantees sensible interaction between these carriers of these sorts and operators thereon in any algebra of the signature.

Combining signatures with the equational calculus yields theories. Theories further enhance the usefulness of signatures by ensuring certain equations will hold in any model (algebra) of the theory (signature).

Chapter 5

Extending order sorted algebra

5.1 Introduction

In this chapter we shall discuss how partial functions and conditional signatures may "tie-in" with traditional order sorted algebra. This will mean that any facts that we may prove for or use from traditional order sorted algebra will also apply when partial functions and/or conditional signatures are considered, too provided certain extra facts hold.

Next, we shall look at the similarities between order sorted algebra and category theory. This will demonstrate why computer algebra systems such as Axiom use category theory terminology, whereas most of this thesis uses order sorted algebra as its framework.

Lastly, we define what we mean by a coercion. This definition is fundamental to the rest of this thesis.

All the work in this chapter is the author's own except for section 5.2 which is based almost entirely on [Bro88] after a suggestion by [Ric97] and [Mar97].

5.2 Partial Functions

In this section we ask the question, "How do partial functions interact with classical universal algebra?" We need this in case some of the functions used to create our coercions later are only partial. This can happen (see definition 7.3.3). For example, the division operator in any quotient field is partial, but could be viewed as a constructor function.

In our previous chapter on universal algebra, we defined an S-operator of arity q, to be a (total) function from some A^q to some element of A. (Where A is a set of S-carriers.) Is it possible to redefine this so that the S-operators are partial? Indeed, is this necessary? One of the original reasons for the "invention" of order-sorted algebra (definition 4.4.4) (rather than multi-sorted algebra (definition 4.2.9)) was so that all functions could be considered total. See, for example, [GM92].

For example, if $\alpha_{\sigma_{1,(P)},T} : A_P \to A_T$ were a partial function in an Σ-algebra, $\langle A, \alpha\rangle$, then we could insert a new sort $N \prec P$, such that $\alpha_{\sigma_{1,(N)},T} : A_N \to A_T$ were a total function.

In the example of a quotient field, one would introduce a subsort of the integral domain which would represent all the non-zero elements of the domain.

We may proceed by either attempting to redefine all of universal algebra using partial functions, or through some different mechanism. The following mechanism which uses "virtual sorts" turns out to be flawed. Virtual sorts are an on-the-fly way of generating sorts to represent things like, the non-zero elements of an integral domain, or non-empty stacks.

Definition 5.2.1 *Let $\langle\Sigma, S\rangle$ define an order-sorted signature. If we define $\langle\Lambda, L\rangle$ to be the signature where*

1. $L = S \cup \{u_1, \ldots, u_m\}$ *(a set of symbols distinct from all those in S)*
2. $(\forall i \in \{1, \ldots, m\})(\exists s_{u_i} \in S)(u_i \prec_L s_{u_i})$
3. $(\forall n \in \mathbf{N} \cup \{0\})(\forall s \in S)(\forall q \in S^n)(\Lambda_{n,q,s} = \Sigma_{n,q,s})$
4. $(\exists n \in \mathbf{N})(\exists q \in L^n \setminus S^n)(\exists s \in S)(\Lambda_{n,q,s} \neq \emptyset)$ *(Note that there may be more than one such triple $\langle n, q, s\rangle$)*

then we say that the u_i are virtual sorts *of $\langle\Sigma, S\rangle$ and $\lambda_{n,q,s} \in \Lambda_{n,q,s}(n \in \mathbf{N})(q \in L^n \setminus S^n)(s \in S)$ a* virtual operator symbol *of $\langle\Sigma, S\rangle$.*

Unfortunately, virtual sorts, virtual operators and homomorphism do not interact in a satisfactory manner. The introduction of a virtual sort in the source of a homomorphism may be meaningless in the target or vice versa.

For traditional examples, like the non-empty stack they are fine, since this has some meaning in all stack-algebras. But for types like, "all the elements which do not map to 5 under a particular coercion," then it may be meaningless to create a virtual sort which attempts to represent this.

Broy [Bro88] defines Σ-algebras as having either partial or total operators. A (partial) Σ-homomorphism is then defined as follows:

Definition 5.2.2 ((Partial) homomorphism) *If $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ are $\langle \Sigma, S\rangle$ algebras*[1]*, and ψ is a family of partial maps $\psi_s : A_s \to B_s$ then ψ is a* (partial) Σ-homomorphism $\langle A, \alpha\rangle \to \langle B, \beta\rangle$ *iff the follow two conditions are fulfilled:*
Firstly,

$$(\forall n \in \mathbf{N} \cup \{0\})(\forall q \text{ arities of rank } n)(\forall s \in S)(\forall \sigma \in \Sigma_{n,q,s})$$

$$(\forall (a_1, \ldots, a_n) \in A^q)$$

if both $\psi_s(\alpha_{\sigma_{n,q,s}}(a_1, \ldots, a_n))$ and $\beta_{\sigma_{n,q,s}}(\psi_{q_1}(a_1), \ldots, \psi_{q_n}(a_n))$ are defined, then

$$\psi_s(\alpha_{\sigma_{n,q,s}}(a_1, \ldots, a_n)) = \beta_{\sigma_{n,q,s}}(\psi_{q_1}(a_1), \ldots, \psi_{q_n}(a_n)).$$

Secondly,

$$(\forall n \in \mathbf{N} \cup \{0\})(\forall q \text{ arities of rank } n)(\forall s \in S)(\forall \sigma \in \Sigma_{n,q,s})$$

$$((\forall a_1, a'_1 \in A_{q_1}) \ldots (\forall a_n, a'_n \in A_{q_n})$$

$$(\bigwedge_{i \in \{1,\ldots,n\}} (\phi_{q_i}(a_i) =_{\text{strong}} \phi_{q_i}(a'_i))))$$

$$\Rightarrow \phi_s(\alpha_{\sigma_{n,q,s}}(a_1, \ldots, a_n)) =_{\text{strong}} \phi_s(\alpha_{\sigma_{n,q,s}}(a'_1, \ldots, a'_n))$$

Where "$=_{\text{strong}}$" is the strong equality defined via

$$(a =_{\text{strong}} b) \Leftrightarrow ((a \text{ defined } \Leftrightarrow b \text{ defined }) \wedge (a \text{ defined } \Rightarrow a = b))$$

Broy's definition of homomorphism (definition 5.2.2) is (as he states) rather liberal. A special factor which he notes is that the composition of partial homomorphisms need not be a homomorphism again.

The following two definitions turn out to be useful in distinguishing between types of partial homomorphism.

Definition 5.2.3 *Let $\langle \Sigma', S'\rangle$ be a sub-signature of $\langle \Sigma, S\rangle$. A Σ'-algebra $\langle A, \alpha\rangle$ is said to be a* Σ'-subalgebra *of the Σ-algebra $\langle B, \beta\rangle$ iff*

[1] with partial operators, in our terminology

1. $(\forall s \in S')(A_s = B_s)$
2. $(\forall n \in \mathbf{N} \cup \{0\})(\forall q \text{ arities of rank } n)(\forall s \in S)(\forall \sigma \in \Sigma_{n,q,s})(\alpha_\sigma = \beta_\sigma |_{A^q})$

Definition 5.2.4 *Let $\langle \Sigma', S' \rangle$ be a sub-signature of $\langle \Sigma, S \rangle$. A Σ'-algebra $\langle A, \alpha \rangle$ is said to be a* weak Σ'-subalgebra *of the Σ-algebra $\langle B, \beta \rangle$ iff*

1. $(\forall s \in S')(A_s = B_s)$
2. $(\forall n \in \mathbf{N} \cup \{0\})(\forall q \text{ arities of rank } n)(\forall s \in S)(\forall \sigma \in \Sigma_{n,q,s})(\alpha_\sigma \leq_\perp \beta_\sigma |_{A^q})$

where "$\leq_\perp$ is defined via,

$$f \leq_\perp g \Leftrightarrow (\forall x)((f(x) \text{ is not defined}) \vee (f(x) = g(x)))$$

Broy notes that any partial homomorphism may be decomposed using the following methodology. A partial homomorphism $\phi : A \to B$ defines a weak subalgebra A' of A defined via,

$$(\forall s \in S)(A'_s = \{a \in A_s : \psi(a) \text{ defined}\})$$

with the functions taking the meanings $\alpha'_{\sigma_{n,q,s}}(a_1, \ldots, a_n) = \alpha_{\sigma_{n,q,s}}(a_1, \ldots, a_n)$ if $(\forall i \in \{1, \ldots, n\})(\phi(a_i)$ are defined) and $\phi(\alpha'_{\sigma_{n,q,s}}(a_1, \ldots, a_n))$ is defined. Otherwise, $\alpha'_{\sigma_{n,q,s}}(a_1, \ldots, a_n)$ is not defined.

By ϕ' we denote the weak partial identity function $A \to A'$. Then we define $\tilde{\phi} : A' \to B$ to be the total homomorphism which is the restriction of ϕ to A'.

Being total, $\tilde{\phi}$ induces a weak subalgebra B' of B defined via,

$$(\forall s \in S)(B'_s = \{b \in B_s : (\exists a \in A_s)(\tilde{\psi}(a) = b)\})$$

with the functions defined via $\beta'_{\sigma_{n,q,s}}(a_1, \ldots, a_n) = \tilde{\phi}(\alpha_{\sigma_{n,q,s}}(a_1, \ldots, a_n))$ where $b_i = \tilde{\phi}(a_i)$ and $\beta_{\sigma_{n,q,s}}(a_1, \ldots, a_n)$ is defined.

This then induces a total surjective homomorphism $\phi'' : A' \to B'$. Also, by construction B' is a weak subalgebra of B. Therefore there also exists a total injective homomorphism $\phi''' : B' \to B$ which is the natural inclusion operator.

As examples consider the following coercions. (We are only looking at the carrier of the principal sort (definition 7.2.1) in these examples.

Example 5.2.5 $\mathbf{Q} \to \mathbf{Z}_5$:

`Fraction Integer -> PrimeField 5`

$$A' = \mathbf{Q} \setminus \{q\frac{1}{5} \mid q \in \mathbf{Q} \setminus \{0\}\}$$

$$B' = B$$

Example 5.2.6 $\mathbf{Q}[x] \to \mathbf{Z}(x) = \mathbf{Q}(x)$:

`Polynomial Fraction Integer -> Fraction Polynomial Integer`

$$A' = A$$

$$B' = \mathbf{Q}[x] = \{p \mid p \in \mathbf{Q}(x) \wedge (\exists q \in \mathbf{Q}(x))(p = q \wedge \text{denom}(q) \in \mathbf{Q})\}$$

Example 5.2.7 $\mathbf{Q}[x] \to \mathbf{Z}_5(x)$:

`Polynomial Fraction Integer -> Fraction Polynomial PrimeField 5`

$$A' = \mathbf{Q}[x] \setminus \{\Sigma_{i \in \mathbf{N} \cup \{0\}}(q_i x^i) \in \mathbf{Q}[x] \mid (\exists i \in \mathbf{N} \cup \{0\})(s \in \mathbf{Q} \setminus \{0\})(q_i = s\frac{1}{5})\}$$

$$B' = \mathbf{Z}_5[x] = \{p \mid p \in \mathbf{Z}_5(x) \wedge (\exists q \in \mathbf{Z}_5(x))(p = q \wedge \text{denom}(q) \in \mathbf{Z}_5)\}$$

These show that real examples of coercions may indeed cause A and A' to differ as well as B and B'.

Notice that in example 5.2.5 above, we are considering a field homomorphism, and that A' is a weak subfield of A. (For example / is undefined on the pair $(1, 5)$ in A'.) Clearly, B' is a subfield of B.

In example 5.2.6, it is an integral domain homomorphism that concerns us. So although B' is only a weak subfield of B this does not matter, as it is sub-integral domain of B.

Similarly, in example 5.2.7 we are again considering an integral domain homomorphism, and both A' and B' are sub-integral domains of A and B, respectively.

As Broy notes, this definition of partial homomorphism allows the everywhere undefined function to be a partial homomorphism. Thus this definition is too weak for our purposes.

Definition 5.2.8 *A partial homomorphism, ϕ is called* strict *iff,*

$$(\forall n \in \mathbf{N} \cup \{0\})(\forall q \text{ arities of rank } n)(\forall s \in S)(\forall \sigma \in \Sigma_{n,q,s})(\forall (a_1, \ldots, a_n) \in A^q)$$

$$\phi(\alpha_\sigma(a_1, \ldots, a_n)) =_{\text{strong}} \beta_\sigma(\phi(a_1), \ldots, \phi(a_n))$$

Broy notes that a strict homomorphism ensures that B' is a Σ-subalgebra of B (and not just a weak Σ-subalgebra). ϕ'' is always strict, and if ϕ is strict, then B' is indeed a Σ-subalgebra of B.

Our requirements for homomorphism are that it act strictly on a certain set of partial functions (the constructors, definition 7.3.3). In other words, we shall require that our coercion be a strict partial Σ^C-homomorphism (again, see definition 7.3.3).

In the rest of this work we shall not in general concern ourselves with the partiality of homomorphisms, operators or strictness. Since, provided our homomorphisms act strictly on the constructors of the type (and any type recursively required for construction (definition 7.6.5)) then the strong equality in the definition of strictness puts us in good shape.

5.3 Conditional varieties

Axiom contains the notion of "conditional `Categories`" or in the language of universal algebra, "conditional varieties". In this section we define and reconcile the notions of conditional and non-conditional[2] varieties. This means that in future sections we shall be able to ignore the existence of conditional varieties in Axiom.

Notice that our choice of the word conditional here is similar to that used in the phrase, "conditional algebraic specifications". The difference being that our conditions are predicates evaluated over arities not terms. Ours is a higher order notion and should not be confused with the ordinary lower order work.

We extend the definition of signature to conditional signature, via the following definition:

Definition 5.3.1 *A S-sorted* conditional signature *is a set $C\Sigma$ of sets $C\Sigma_{n,q,s}$ indexed by $n \in \mathbf{N} \cup \{0\}$, q an S-arity of rank n, and $s \in S$. Each element of $C\Sigma$ is of the form*

$$\textit{if } P(w) \textit{ then } \sigma_{n,q,s}$$

where P is a well-formed proposition in some language, and w is an arity of finite

[2]Unconditional may have been a better choice of word from an English language point of view. However, we are not saying that an algebra is unconditionally an algebra of a variety; we are saying that it is an algebra of a variety which has no conditions.

rank (if it is a tautology, it is always represented by T). The $\sigma_{n,q,s}$ are the conditional operator symbols.

Firstly, we need to define a conditional term algebra.

Definition 5.3.2 *Let X be an S-indexed family of sets disjoint from each other, from the set Δ, the set $\{if, P, then\}$ and from $\{\sigma \mid (\exists \text{ arity } w)(\text{"}if\ P(w)\ then\ \sigma\text{"} \in \bigcup_{n,q,s} \Sigma_{n,q,s})\}$. We define $T_{C\Sigma}(X)$ to be the S-indexed family of sets of strings of symbols from $\{if, P, then\} \cup \bigcup_{s\in S} X_s \cup \Delta \cup \bigcup_{n,q,s} \Sigma_{n,q,s}$, each set as small as possible satisfying these conditions:*

1. $(\forall s \in S)(\Sigma_{0,(),s} \subseteq T_{C\Sigma}(X)_s)$
2. $(\forall s \in S)(X_s \subseteq T_{C\Sigma}(X)_s)$
3. $(\forall \text{"}if\ P(w)\ then\ \sigma\text{"} \in C\Sigma_{n,q,s})(\forall i \in \{1,\ldots,n\})(\forall t_i \in T_{C\Sigma}(X)_{q_i})(\text{"}if\ P(w)\ then\ \sigma\text{"}\ (t_1,\ldots,t_n) \in T_{C\Sigma}(X)_s)$

We make $T_{C\Sigma}(X)$ into a conditional $C\Sigma$-algebra by defining conditional operators "if $P(w)$ then σ_T on $T_{C\Sigma}(X)$, for each "if $P(w)$ then σ" $\in C\Sigma_{n,q,s}$ via:

- *If $n = 0$ then "if $P(w)$ then σ_T":= "if $P(w)$ then σ". (Guaranteed to be in $T_{C\Sigma}(X)_s$ by (1)).*
- *Else, define "if $P(w)$ then $\sigma_T(t_1,\ldots,t_n)$" to be the* string *"if $P(w)$ then $\sigma(t_1,\ldots,t_n)$".*

$T_{C\Sigma}(X)$ is called the conditional term algebra*, and an element of $T_{C\Sigma}(X)$ is called a* conditional term.

Associated with the notion of conditional signature are the notions of conditional theory and conditional variety. However, we require a new definition for equations and equational systems.

Definition 5.3.3 *A* conditional (multi-sorted) equation *for the conditional signature $C\Sigma$ consists of a quadruple $(P(w), Y, t_1, t_2)$, where P is a well-formed proposition in some language, and w is an arity of finite rank, (if it is a tautology, it is always represented by T) where $Y \vec{\subseteq} X$, t_1 and t_2 are terms from the same carrier set of $T_{C\Sigma}(X)$ (or $T_{C\Sigma}(X \vec{\cup} A)$ if we are dealing with equations in a particular algebra $\langle A, \alpha\rangle$), and every* *variable occurring in* t_1 *and* t_2 *occurs in the appropriate member of* Y*:* $t_1, t_2 \in T_{C\Sigma}(Y)$*. Such equations are more usually written* (if $P(w))(\forall Y \quad t_1 = t_2)$.

If t_1 *and* t_2 *come from the same carrier set of* $T_{C\Sigma}(X)$ *we say that the equation is* $C\Sigma$-generic*.*

Note that this definition of conditional equation should not be confused with that found in work such as [KR91]. In that context, a conditional equation is a clause such as (if $P(t))(\forall Y \quad t_1 = t_2)$ where t is a subterm of t_1, for example.

Definition 5.3.4 *An* conditional equational system *for the signature* Σ *is a set of conditional equations for* $T_{C\Sigma}(X)$ *(or* $T_{C\Sigma}(X\vec{\cup}A)$ *if we are dealing with equations in a particular algebra* $\langle A, \alpha\rangle$*).*

Associated with definition 5.3.1 are two special signatures Σ and $\bar{\Sigma}$. The former is the extension of all the $C\Sigma$s; the latter is extended by all $C\Sigma$s. More formally,

Definition 5.3.5 *Using the definitions of definition 5.3.1, we define* Σ *to be the* S*-sorted signature where* $\forall n, q, s$ *(*$n \in \mathbf{N} \cup \{0\}$*,* q *an* S*-arity of rank* n*, and* $s \in S$*)*

$$\Sigma_{n,q,s} = \{\sigma_{n,q,s} \mid \text{“}\mathit{if}\ P(w)\ \mathit{then}\ \sigma_{n,q,s}\text{”} \in C\Sigma_{n,q,s}\}$$

Now similarly, we define $\bar{\Sigma}$ *to be the* $\bar{S}$*-sorted signature where* $\forall n, q, s$ *(*$n \in \mathbf{N} \cup \{0\}$*,* q *an* $\bar{S}$*-arity of rank* n*, and* $s \in \bar{S}$*)*

$$\bar{\Sigma}_{n,q,s} = \{\sigma_{n,q,s} \mid \text{“}\mathit{if}\ T\ \mathit{then}\ \sigma_{n,q,s}\text{”} \in C\Sigma_{n,q,s}\}$$

where $\bar{S}$ *is the largest subset of* S *such that* $\bar{\Sigma}$ *has no void sorts.*

The algebras of conditional signatures are the same as ordinary algebras, except that the proposition P is evaluated over A^w, and iff this is true, there is an operator $\alpha_{n,q,s,\sigma}$ in that algebra. More formally,

Definition 5.3.6 *An* order-sorted, total, conditional $C\Sigma$-algebra *is an ordered triple* $\langle A, \{A_s : s \in S\}, \alpha\rangle$*, where* A *is a class known as the* universe*,* $\{A_s : s \in S\}$ *is an* S*-indexed family of subsets of* A*, known as the* carriers *of the algebra, and* α *is a set of sets of functions* $\alpha_{n,q,s} = \bigcup_{\sigma \in \Sigma_{n,q,s}} \{\alpha_{n,q,s,\sigma} : A^q \to A_s \mid P(A^w)\}$*, such that:*

1. $A_u = A$*;*

2. *If $s \preceq s'$ in S, then $A_s \subseteq A_{s'}$;*

3. *If "if $P(w)$ then σ" $\in C\Sigma_{n,q,s}$ and "if $P'(w')$ then σ" $\in C\Sigma_{n,q',s'}$, with $s \preceq s'$ and $q' \preceq q$, and also $P(A^w)$ and $P'(A^{w'})$, then $\alpha_{n,q,s,\sigma} \mid_{A^{q'}} = \alpha_{n,q',s',\sigma}$*

Clearly, if all the P are tautologies then a conditional signature $C\Sigma$ is identical to its non-conditional partner, Σ. Therefore all conditional $C\Sigma$-algebras are (non-conditional) Σ algebras in this case.

In any case a $C\Sigma$-algebra is a $\bar{\Sigma}$-algebra. Thus we can see that $\bar{\Sigma}$ is a minimal[3] signature for $C\Sigma$.

Definition 5.3.7 *We define a* conditional theory *to be the ordered pair $\langle\langle C\Sigma, S\rangle, \mathcal{S}\rangle$ where $C\Sigma$ is an S-sorted conditional signature and $\mathcal{S}$ is a set of S-sorted conditional Σ-equations. We say that a $C\Sigma$-algebra $\langle A, \alpha\rangle$* models *or* satisfies *the conditional theory iff it is a model for $\mathcal{S} \mid_{\langle A,\alpha\rangle}$, where $\mathcal{S} \mid_{\langle A,\alpha\rangle}$ is defined as follows,*

$$\mathcal{S} \mid_{\langle A,\alpha\rangle} := \{(Y, t_1, t_2) \mid ((P(w), Y, t_1, t_2) \in S) \wedge P(A^w)\}$$

A theoretical interpretation *of a conditional theory $\langle\langle C\Sigma, S\rangle, \mathcal{S}\rangle$ is a theory $\langle\langle \Sigma_\tau, S\rangle, \mathcal{S}_\tau\rangle$ equivalent to $\langle\langle C\Sigma, S\rangle, \mathcal{S}\rangle$ with all the propositions $P(w)$ evaluated in some consistent manner.*

Finally we demand that for all possible pairs of theoretical interpretations $\langle\langle \Sigma_0, S\rangle, \mathcal{S}_0\rangle$, $\langle\langle \Sigma_1, S\rangle, \mathcal{S}_1\rangle$ where $\Sigma_0 \vec{\subseteq} \Sigma_1$ and $\mathcal{S}_0 \subseteq \mathcal{S}_1$ that $\langle\langle \Sigma_1, S\rangle, \mathcal{S}_1\rangle$ is a protecting extension of $\langle\langle \Sigma_0, S\rangle, \mathcal{S}_0\rangle$.

Definition 5.3.8 *The collection of all models of a particular conditional theory is called a* conditional variety.

In fact the conditional variety defined by $\langle\langle C\Sigma, S\rangle, \mathcal{S}\rangle$ is equivalent to the variety defined by $\langle\langle \bar{\Sigma}, \bar{S}\rangle, \mathcal{S}\rangle$.

Also, the variety specified by any theoretical interpretation of the conditional theory $\langle\langle C\Sigma, S\rangle, \mathcal{S}\rangle$ forms a subclass of the the conditional variety. Moreover, for a pair of theoretical interpretations $\langle\langle \Sigma_0, S\rangle, \mathcal{S}_0\rangle$, $\langle\langle \Sigma_1, S\rangle, \mathcal{S}_1\rangle$ where $\Sigma_0 \vec{\subseteq} \Sigma_1$ and $\mathcal{S}_0 \subseteq \mathcal{S}_1$, the variety specified by $\langle\langle \Sigma_1, S\rangle, \mathcal{S}_1\rangle$ forms a subclass of the variety specified by $\langle\langle \Sigma_0, S\rangle, \mathcal{S}_0\rangle$.

[3]Minimal in the sense that it has fewest operator symbols and sorts. The variety it specifies is maximal, if you consider the number of algebras that model it.

If we consider two conditional algebras from the same conditional variety, then we wish to know whether there is a homomorphism from one to the other. For this we need to define "conditional homomorphism".

A technical definition which will allow us to define homomorphisms more easily.

Definition 5.3.9 *Let $\langle A, \alpha\rangle$, and $\langle B, \beta\rangle$ be $C\Sigma$-algebras. For all n, q, s, we define*

$$\mu_{\alpha,n,q,s} : \alpha_{n,q,s} \to \Sigma_{n,q,s}$$

to be the map,

$$\alpha_{n,q,s,\sigma} \mapsto \sigma_{n,q,s}$$

Similarly, define $\mu_{\beta,n,q,s}$ for $\langle B, \beta\rangle$.

The family of maps, μ_α (as we show in notation 5.3.11) map an algebra, $\langle A, \alpha\rangle$ to a non-conditional signature which it models.

Definition 5.3.10 *$\phi : \langle A, \alpha\rangle \to \langle B, \beta\rangle$ is a* conditional $C\Sigma$-homomorphism *iff it is a non-conditional Σ^{AB}-homomorphism, where*

$$\left(\begin{matrix}\Sigma^{AB} = \{\{\mu_{\alpha,n,q,s}(\alpha_{n,q,s,\sigma}) \mid \alpha_{n,q,s,\sigma} \in \alpha_{n,q,s}\} \cap \{\mu_{\beta,n,q,s}(\beta_{n,q,s,\sigma}) \mid \beta_{n,q,s,\sigma} \in \beta_{n,q,s}\} \mid \\ n \in \mathbf{N} \cup \{0\}, q \text{ an } \breve{S}-\text{arity of rank } n, s \in \breve{S}\}\end{matrix}\right)$$

Σ^{AB} is a $\breve{S}$-sorted signature where $\breve{S}$ is the largest subset of S such that Σ^{AB} has no void sorts.

In fact, if we use the following piece of notation,

Notation 5.3.11 *If $\langle A, \alpha\rangle$ is a conditional $C\Sigma$-algebra, we define $\Sigma^{NC(A)}$ to be the non-conditional signature*

$$\{\{\mu_{\alpha,n,q,s}(\alpha_{n,q,s,\sigma}) \mid \alpha_{n,q,s,\sigma} \in \alpha_{n,q,s}\} \mid n \in \mathbf{N} \cup \{0\}, q \text{ an } \hat{S}-\text{arity of rank } n, s \in \hat{S}\}$$

$\Sigma^{NC(A)}$ is a $\hat{S}$-sorted signature where $\hat{S}$ is the largest subset of S such that $\Sigma^{NC(A)}$ has no void sorts.

As an aside, notice that for each $C\Sigma$-algebra $\langle A, \alpha\rangle$, there is an associated non-conditional term algebra $T_{\Sigma^{NC(A)}}(X)$. This term algebra is the equivalent of evaluating each $P(w)$ over A in $T_{C\Sigma}(X)$.

Then $\langle A, \alpha \rangle$ is a non-conditional $\Sigma^{NC(A)}$-algebra, and

$$\Sigma^{AB} = \{\Sigma^{NC(A)}_{n,q,s} \cap \Sigma^{NC(B)}_{n,q,s} \mid n \in \mathbf{N} \cup \{0\}, q \text{ an } \breve{S}\text{−arity of rank } n, s \in \breve{S}\}$$

So, for any two conditional algebras of the same conditional signature, there exists a fixed (maximal) non-conditional signature of which they are both non-conditional algebras. (We may have to forget some operators and even some sorts, which may only exist in conditions).

Our definition of conditional homomorphism is that of the non-conditional homomorphism from this fixed non-conditional signature.

Hence, the concept of conditional signatures can always be reduced to one for non-conditional signatures and therefore any results we prove or use need only be for non-conditional signatures.

As we have seen in definition 5.3.7 this concept may also be extended to theories. In addition to adding new operators when certain propositions hold, we also allow for new equations to be added (conditionally). However, we demand that any such extension is a protecting extension of the minimal non-conditional theory (the theory over the signature $\langle \bar{\Sigma}, \bar{S} \rangle$).

More explicitly, let us define $\hat{\mathcal{S}}$ to be all those equations from $\mathcal{S}$ which only involve arities from $\Sigma^{NC(A)}$ (as well as all equations not involving arities) [4]. Similarly, define $\bar{\mathcal{S}}$ to be all those equations from $\mathcal{S}$ which only involve arities from $\bar{\Sigma}$ (as well as all equations not involving arities).

If $\langle A, \alpha \rangle$ is a conditional $C\Sigma$-model then $\langle \langle \Sigma^{NC(A)}, \hat{S} \rangle, \hat{\mathcal{S}} \rangle$ must be a protecting extension of $\langle \langle \bar{\Sigma}, \bar{S} \rangle, \bar{\mathcal{S}} \rangle$.

5.4 A Category theory approach

This section is an aside from the rest of the chapter. We are not extending the theory of order sorted algebra nor category theory, merely explaining why we cover both.

There is a great deal of correspondence between category theory (specifically, categorical type theory) and universal algebra. There is not enough room here to cover this huge topic, but the reader is pointed to [Cro93], which covers categorical type theory in great depth.

[4] Alternatively, define $\hat{\mathcal{S}}$ to be those equations from $\mathcal{S}$ for which $\langle A, \alpha \rangle$ is a model, since it may not model all the equations in the definition of $\hat{\mathcal{S}}$ in the main text.

As usual, Mac Lane [ML71] also contains a great deal of information on the correspondence between universal algebra and adjoint functors in the chapter on "Monads and Algebras".

All the ideas from multi-sorted algebra theory may be represented in the categorical type theory framework, although an equivalent for order-sorted algebra appears not to be covered. However, the author believes that the notion could be introduced.

One area which we shall discuss here is the concept of algebraic homomorphism and the arrows (morphisms) of a given category.

The correspondence between a categorical type theory C and its associated algebraic type theory $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$ is the following,

$$\begin{aligned}\mathrm{Obj}(C) &= \text{all the } \Sigma\text{-algebras for a given } \Sigma \text{ which model } \mathcal{S}.\\ \mathrm{Arr}(C) &= \text{all (or a fixed sub-collection of) the } \Sigma\text{-homomorphisms.}\end{aligned}$$

The category of all algebras of a given signature $\langle\Sigma, S\rangle$ has as an initial object the term algebra, $T_\Sigma(X)$.

The category of all algebras which model a given theory $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$ has as an initial object the free algebra $T_\Sigma(X)/\equiv_{\mathcal{E}}$. (This is why the second universality theorem 4.6.16 holds).

So we see that both categories and theories collect together similar types. This abstraction of information forms the basis of abstract datatyping.

5.5 Coercion

In this work we are interested in coercions which so far have been explained as being natural, type-changing maps. We may define them in a far more strict fashion.

The following would seem to be a good definition for coercion. However, in practice this definition is not well-defined enough since it does not state which theory a coercion should come from.

Definition 5.5.1 *Let $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ be algebras from the variety specified by some theory $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$. The map $\phi : \langle A, \alpha\rangle \to \langle B, \beta\rangle$ is a coercion if it is a homomorphism of that theory.*

Using the category theory correspondence above, we see that a function between two types of the category of all algebras which model a particular theory is a coercion iff it is an arrow of the category.

The only problem with this definition of coercion is that it does not specify from which theory (or equivalently, category) from which we should demand the homomorphism be taken. In general, we would like this to be the most specific, or smallest category to which both algebras belong.

If we just used the definition of coercion above (definition 5.5.1) then any total map between types would be a coercion! Any small category may be forgotten back to **Set** via the forgetful functor and any total function is an arrow of **Set**. Most types in computer algebra system are objects of **Set** (which is equivalent to `SetCategory` in Axiom - the second most basic `Category` in Axiom).

Richardson [Ric97] notes that we also require a fixed framework in which we define our coercions. If we were to attempt to define a coercion $\langle A, \alpha\rangle$ to $\langle B, \beta\rangle$ to be "any map which is a homomorphism for all theories which both algebras model" then the definition would not be well-defined. We need to state a context.

We are attempting to reflect the situation that appears in a system like Axiom, Thus we must state that we have been given *a priori* a fixed collection of theories, and that given any type, we know precisely which theories it models.

This precludes a user from adding another theory which the algebras model which could redefine what "coercion" means for those algebras.

In the category theory correspondence, we say that Axiom's `Category` is a fixed collection of categories, and given any type we know of which categories it is an object.

Definition 5.5.2 (Coercion) *Let Υ be a fixed collection of theories.*

Let $\langle A, \alpha\rangle$ be a model for theories Θ_i (for i in some indexing set I) where $(\forall i \in I)(\Theta_i$ is a theory from $\Upsilon)$.

Similarly, let $\langle B, \beta\rangle$ be a model for theories Ω_j (for j in some indexing set J) where $(\forall j \in J)(\Omega_j$ is a theory from $\Upsilon)$.

Then we call a map $\langle A, \alpha\rangle$ to $\langle B, \beta\rangle$ a coercion *iff it is a homomorphism for all the theories in*

$$\{\Theta_i \mid i \in I\} \cap \{\Omega_j \mid j \in J\}$$

In Axiom, `Domains` are only members of a fixed set of `Categories`[5].

[5] A user could re-implement one of these `Categories` and ruin everything.

In fact, in Axiom we are in a slightly better position. If a `Domain` is an object of two `Categories` then there must exist a `Category` which extends (potentially trivially) both these `Categories` of which the `Domain` is an object.

Thus, in Axiom, definition 5.5.2 reduces to,

> “A coercion in Axiom from a type `A` to a type `B` is a homomorphism of the most restrictive `Category` to which both `A` and `B` belong.”

It would be useful to have a name for maps which are “coercions” in the sense of definition 5.5.1. These maps are in a sense natural, and may be the next-most natural map between two types. However, as we have shown in the example using **Set** the map need not particularly “natural” from a realistic point of view.

As it stands, with our current terminology maps which satisfy definition 5.5.1 which are not coercions are merely “homomorphisms which are not coercions”.

If no coercion existed between two types, but a “homomorphism which is not a coercion” existed and a user required that homomorphism then either the user only required a conversion or the theory lattice for the algebra system has not been designed correctly.

5.6 Conclusion

In this chapter we have shown how classical order sorted algebra may be extended to encompass the notion of partial functions and conditional signatures. We have also stated how these notions interact with the equational calculus. These are important extensions due to the fact that these notions are used extensively in Axiom.

We have also mentioned some of the correlation between category theory and universal algebra. Finally we have made the important definition of a coercion and demonstrated why the definition is necessarily strict.

Chapter 6

Coherence

6.1 Introduction

In this chapter we will look at a conjecture of Weber which is important to our work. We shall state his assumptions and proof (which is incorrect).

We shall then add in some extra assumptions and truly prove the theorem. Finally we shall relax one of Weber's assumptions and prove that the theorem still holds.

The assumptions made by Weber provide a strict formal setting for types in an algebra system. The theorem in itself proves confluence for coercions in this setting.

All the work in section 6.2 (except the explanation of "n-ary type constructor" which is by this author) and section 6.3 is taken from [Web93b][Web93a][Web95] . All the rest of the work is the author's own.

6.2 Weber's work I: definitions

This section contains the definitions from Weber's thesis [Web93b] [Web93a] [Web95] required for the statement and Weber's "proof" of Weber's coercion conjecture 6.3.7. His assumptions are in the next section.

These statements will also be used when we correctly prove the coherence theorem 6.4.7 in section 6.4.

It should be noted that Weber's coercion conjecture only makes up part of one chapter of his thesis [Web93b] which also covers various areas of type classes, type inference and coercion in great depth and detail.

Weber uses the phrase "type class" where we would use the terms "category" or "variety".

Definition 6.2.1 *A* base type *is any type which is not a parametrically defined type. (i.e. a 0-ary operator in the term algebra of type classes)*

So for example, the `Integer` and `Boolean` types are base types.

Definition 6.2.2 *A* ground type *is any type within the system which is either a base type or a parametrically defined type with all the parameters present. Any non-ground type is called a* polymorphic type.

As examples, we have `Integer` and `Polynomial(Fraction(Integer))`. As a non-example, we have `Polynomial(R:Ring)`.

Definition 6.2.3 *If there exists a coercion from t to t' we say that $t \trianglelefteq t'$.*

This definition places an order on the ground types.

Weber uses the phrase "n-ary type constructor" to mean a functor from the product of n categories to a specific category.

Equivalently, it is a function from the product of n varieties (specified respectively by the theories $\Omega_1, \ldots, \Omega_n$) to a variety (specified by the theory $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$). This is a function which, for all i maps the carrier of the principal sort (and potentially the carriers of some of the non-principal sorts) of a model of Ω_i to one (for each sort mapped) of the non-principal sorts of a model of $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$.

If the model returned is $\langle A, \alpha\rangle$, this function must map one and only one sort-carrier to each and every member of $A \setminus \{A_{S_1}\}$ where S_1 is the principal sort.

Definition 6.2.4 *For a ground type t we define* $\mathrm{com}(t)$ *to be* 1*, if t is a base type, or if $t = f(t_1, \ldots, t_n)$ (an n-ary type constructor) then* $\mathrm{com}(t)$ *is defined to be* $1 + \max(\{\mathrm{com}(t_i) | i \in \{1, \ldots, n\}\})$.

This defines the "complexity" of a ground type to be how far up the type lattice it is.

Definition 6.2.5 (Coherence) *A type system is* coherent *if the following condition is satisfied:*

$$(\forall \textit{ ground types } t_1, t_2)((\phi, \psi : t_1 \to t_2 \textit{ coercions}) \Rightarrow (\phi = \psi))$$

This guarantees that there only exist one coercion from one ground type to another. This is a highly desirable feature of any type system. The main results of this chapter (the coherence theorem 6.4.7 and the extended coherence theorem 6.5.4) are that we may be able to guarantee that our type system is coherent providing some sensible assumptions hold true.

In the following definitions, all the σ and σ' are type classes.

Notation 6.2.6 *$t : \sigma$ means that the type t is an object of the type class σ.*

Definition 6.2.7 *The n-ary type constructor f ($n \in \mathbf{N}$) induces a* structural coercion *if there are sets $\mathcal{A}_f \subseteq \{1, \ldots, n\}$ and $\mathcal{M}_f \subseteq \{1, \ldots, n\}$ such that the following condition is satisfied:*

> *If $f : (\sigma_1, \ldots, \sigma_n) \to \sigma$ and $f : (\sigma'_1, \ldots, \sigma'_n) \to \sigma'$, and $(\forall i \in \{1, \ldots, n\})(\forall$ ground types $t_i : \sigma_i$ and $t'_i : \sigma'_i)(i \notin \mathcal{A}_f \cup \mathcal{M}_f \Rightarrow t_i = t'_i)$ and there exist coercions:*
>
> $$\begin{array}{rcl} \phi_i : t_i \to t'_i & \text{if} & i \in \mathcal{M}_f \\ \phi_i : t'_i \to t_i & \text{if} & i \in \mathcal{A}_f \\ \phi_i = \mathrm{id}_{t_i} = \mathrm{id}_{t'_i} & \text{if} & i \notin \mathcal{A}_f \cup \mathcal{M}_f \end{array}$$
>
> *then there exists a **uniquely defined** coercion*
>
> $$\mathcal{F}_f(t_1, \ldots, t_n, t'_1, \ldots, t'_n, \phi_1, \ldots, \phi_n) : f(t_1, \ldots, t_n) \to f(t'_1, \ldots, t'_n).$$

The type constructor f is covariant (or monotonic) *in its i-th argument if $i \in \mathcal{M}_f$. f is* contravariant (or antimonotonic) *in its i-th argument if $i \in \mathcal{A}_f$*

Note that if $i \in \mathcal{A}_f \cap \mathcal{M}_f$ then $t_i \cong t'_i$

As an example of covariance, the list constructor in Axiom, `List` (a functor `Set` $\to$ `ListAggregate()`) takes one argument in which it is covariant. Given types `A` and `B`, such that there exists a coercion $\phi_1 :$ `A` $\to$ `B` then

$$\mathcal{F}_{\mathtt{List}}(\mathtt{A}, \mathtt{B}, \phi_1) : \mathtt{List}(\mathtt{A}) \to \mathtt{List}(\mathtt{B})$$

Since Axiom's type constructors are functors, then category theory states this more simply as

$$\mathcal{F}_{\texttt{List}}(\texttt{A}, \texttt{B}, \phi_1) = \texttt{List}(\phi_1)$$

Contravariance is a rarer case. However, Axiom's `Mapping` functor is contravariant in its first argument and covariant in its second. `Mapping` takes two types `A` and `B` and returns the type of all mappings from `A` to `B`.

As a concrete example for `Mapping`, suppose we wish to find the uniquely defined coercion

$$\texttt{Mapping(Fraction(Integer),Fraction(Integer))} \rightarrow \\ \texttt{Mapping(Integer,Fraction(Integer))}$$

There exists a coercion

$$\iota : \texttt{Integer} \rightarrow \texttt{Fraction}(\texttt{Integer})$$

the inclusion operator, There also exists a coercion

$$\text{id} : \texttt{Fraction}(\texttt{Integer}) \rightarrow \texttt{Fraction}(\texttt{Integer})$$

which is the identity operation. The uniquely defined coercion is as follows,

$$\mathcal{F}_{\texttt{Mapping}}(\texttt{Fraction}(\texttt{Integer}), \texttt{Fraction}(\texttt{Integer}), \texttt{Integer}, \texttt{Fraction}(\texttt{Integer}), \iota, \text{id}) : \\ \texttt{Mapping}(\texttt{Fraction}(\texttt{Integer}), \texttt{Fraction}(\texttt{Integer})) \rightarrow \\ \texttt{Mapping}(\texttt{Integer}, \texttt{Fraction}(\texttt{Integer}))$$

which sends $f \mapsto \text{id} \circ f \circ \iota$.

The following definitions shows that there is a homomorphic image of a parameter in the created type. For example there is a homomorphic image of the underlying ring in any polynomial ring.

Definition 6.2.8 *Let $f : (\sigma_1, \ldots, \sigma_n) \rightarrow \sigma$ be an n-ary type constructor. If $(\forall i \in \{1, \ldots, n\})($ for some ground types $t_i : \sigma_i)$ such that there exists a coercion*

$$\Phi^i_{f,t_1,\ldots,t_n} : t_i \rightarrow f(t_1, \ldots, t_n)$$

then we say that f has a direct embedding at its i-th position.

Moreover, let

$$\mathcal{D}_f = \{i | f \textit{ has a direct embedding at its } i\text{−}th \textit{ position}\}$$

be the set of direct embedding positions of f.

The definition of $\mathcal{D}_f$ is a technical definition of Weber's, needed for one of his assumptions (6.3.5).

6.3 Weber's work II: assumptions and a conjecture

This section provides all the assumptions and results which Weber uses in his "proof" of Weber's coherence conjecture 6.3.7 which we shall state and at the end of this section. The assumptions and results are also required in our proof of the coherence theorem 6.4.7.

Assumption 6.3.1 *For any ground type t, the identity on t will be a coercion. The (well-defined) composition of two coercions is also a coercion.*

This is clearly a sensible (if not-often implemented) statement. Since our coercions are always to be arrows of a category, then the above assumption must hold.

Lemma 6.3.2 *If assumption 6.3.1 holds, then the set of ground types as objects together with their coercions as arrows form a category.*

Proof. Immediate. □

The following assumption will provide us with the basis for a coherent type system. Our coherence is built by ensuring confluence amongst different paths leading to the same coercion. If we do not have coherence at the base types then we shall not have coherence amongst the general types.

Assumption 6.3.3 *The subcategory of base types and coercions between base types forms a preorder, i.e. if t_1, t_2 are base types and $\phi, \psi : t_1 \to t_2$ are coercions, then $\phi = \psi$.*

The following condition states that $\mathcal{F}_f$ is a functor over the category of all $f(\cdot, \ldots, \cdot)$s.

Assumption 6.3.4 *Let f be an n-ary type constructor which induces a structural coercion and let $f(t_1, \dots, t_n), f(t'_1, \dots, t'_n), f(t''_1, \dots, t''_n)$ be ground types. Assume that the following are coercions.*

$$i \in \mathcal{M}_f \Rightarrow \phi_i : t_i \to t'_i, \phi'_i : t'_i \to t''_i$$

$$i \in \mathcal{A}_f \Rightarrow \phi'_i : t''_i \to t'_i, \phi_i : t'_i \to t_i$$

$$i \notin \mathcal{A}_f \cup \mathcal{M}_f \Rightarrow t_i = t'_i = t''_i \text{ and } \phi_i = \phi'_i = \mathrm{id}_{t_i}$$

Then the following conditions are satisfied:

1. $\mathcal{F}_f(t_1, \dots, t_n, t_1, \dots, t_n, \mathrm{id}_{t_1}, \dots, \mathrm{id}_{t_n}) = \mathrm{id}_{f(t_1,\dots,t_n)}$
2. $\mathcal{F}_f(t_1, \dots, t_n, t''_1, \dots, t''_n, \phi_1 \circ \phi'_1, \dots, \phi_n \circ \phi'_n) =$
   $\mathcal{F}_f(t_1, \dots, t_n, t'_1, \dots, t'_n, \phi_1, \dots, \phi_n) \circ \mathcal{F}_f(t'_1, \dots, t'_n, t''_1, \dots, t''_n, \phi'_1, \dots, \phi'_n)$

This is a condition which stops direct embeddings "becoming confused". Firstly, Weber declares that any type constructor can only have one direct embedding. (We shall show how to relax this condition in a later section (6.5).) Secondly, he states that direct embeddings, where they exist, are unique.

Assumption 6.3.5 *Let f be an n-ary type constructor. Then the following conditions hold:*

1. $|\mathcal{D}_f| = 1$
2. *Direct embedding coercions are unique. i.e. if $\Phi^i_{f,t_1,\dots,t_n} : t_i \to f(t_1, \dots, t_n)$ and $\Psi^i_{f,t_1,\dots,t_n} : t_i \to f(t_1, \dots, t_n)$ then*

$$\Phi^i_{f,t_1,\dots,t_n} = \Psi^i_{f,t_1,\dots,t_n}.$$

The following assumption is highly technical and shows how direct embeddings interact with structural coercions. Basically, they commute.

Assumption 6.3.6 *Let f be an n-ary type constructor which induces a structural coercion and has a direct embedding at its r-th position. Assume that $f : (\sigma_1, \dots, \sigma_n) \to \sigma$ and $f : (\sigma'_1, \dots, \sigma'_n) \to \sigma$, and $(\forall i \in \{1, \dots, n\})(\exists t_i : \sigma_i$ and $t'_i : \sigma'_i)$. If there are coercions $\psi_r : t_r \to t'_r$, if the coercions $\Phi^r_{f,t_1,\dots,t_n}$ and $\Phi^r_{f,t'_1,\dots,t'_n}$ are defined, and if f is covariant in its r-th argument, then the following holds:*

$$\Phi^r_{f,t'_1,\dots,t'_n} \circ \psi_r = \mathcal{F}_f(t_1, \dots, t_n, t'_1, \dots, t'_n, \psi_1, \dots, \psi_n) \circ \Phi^r_{f,t_1,\dots,t_n}$$

or equivalently, the following diagram commutes:

$$\begin{tikzcd}
t_r \arrow[r, "\psi_r"] \arrow[d, "\Phi^r_{f,t_1,\ldots,t_n}"'] & t'_r \arrow[d, "\Phi^r_{f,t'_1,\ldots,t'_n}"] \\
f(t_1,\ldots,t_n) \arrow[r, "\mathcal{F}_f(t_1,\ldots,t_n,t'_1,\ldots,t'_n,\psi_1,\ldots,\psi_n)"] & f(t'_1,\ldots,t'_n)
\end{tikzcd}$$

However, if f is contravariant in its r-th argument then:

$$\mathcal{F}_f(t_1,\ldots,t_n,t'_1,\ldots,t'_n,\psi_1,\ldots,\psi_n) \circ \Phi^r_{f,t'_1,\ldots,t'_n} \circ \psi_r = \Phi^r_{f,t_1,\ldots,t_n}$$

or equivalently, the following diagram commutes:

$$\begin{tikzcd}
t_r \arrow[r, "\psi_r"] \arrow[d, "\Phi^r_{f,t_1,\ldots,t_n}"'] & t'_r \arrow[d, "\Phi^r_{f,t'_1,\ldots,t'_n}"] \\
f(t_1,\ldots,t_n) & f(t'_1,\ldots,t'_n) \arrow[l, "\mathcal{F}_f(t_1,\ldots,t_n,t'_1,\ldots,t'_n,\psi_1,\ldots,\psi_n)"']
\end{tikzcd}$$

We are now in a position to state Weber's coherence conjecture and his "proof". This attempts to show that when the aforementioned assumptions hold true, then we have a coherent type system. We shall give more assumptions and a proper proof in section 6.4.

Conjecture 6.3.7 (Weber's coherence conjecture) *Assume that all coercions between ground types are only built by one of the following mechanisms:*

1. *coercions between base types;*
2. *coercions induced by structural coercions;*
3. *direct embeddings in a type constructor;*
4. *composition of coercions;*
5. *identity function on ground types as coercions.*

If assumptions 6.3.1, 6.3.3, 6.3.4, 6.3.5 and 6.3.6 are satisfied, then the set of ground types as objects, and the coercions between them as arrows form a category which is a preorder.

This is the "proof" of this conjecture given in [Web93b] [Web93a] [Web95].

WEBER'S "PROOF". By assumption 6.3.1 and lemma 6.3.2, the set of ground types as objects and the coercions between them form a category.

For any two ground types t and t' we will prove by induction on $\max(\mathrm{com}(t), \mathrm{com}(t'))$ that if $\phi, \psi : t \to t'$ are coercions then $\phi = \psi$.

If $\max(\mathrm{com}(t), \mathrm{com}(t')) = 1$ then the claim follows by assumption 6.3.3. Now assume that the inductive hypothesis holds for k, and let $\max(\mathrm{com}(t), \mathrm{com}(t')) = k+1$. Assume w.l.o.g. that $t \trianglelefteq t'$ and that $\phi, \psi : t \to t'$ are coercions.

Now $t \trianglelefteq t' \Rightarrow \mathrm{com}(t) \leq \mathrm{com}(t')$. So we may assume that $t' = f(u_1, \ldots, u_n)$ for some n-ary type constructor f.

By assumption 6.3.4 and the induction hypothesis, we can assume that there are ground types s_1, s_2 and unique coercions $\psi_1 : t \to s_1$ and $\psi_2 : t \to s_2$ such that either

$$\phi = \mathcal{F}_f(\ldots, t, \ldots, s_1, \ldots, \psi_1, \ldots) \tag{6.1}$$

or

$$\phi = \psi_1 \circ \Phi^i_{f,\ldots,s_1,\ldots} \tag{6.2}$$

Similarly either,

$$\psi = \mathcal{F}_f(\ldots, t, \ldots, s_2, \ldots, \psi_2, \ldots) \tag{6.3}$$

or

$$\psi = \psi_2 \circ \Phi^j_{f,\ldots,s_2,\ldots} \tag{6.4}$$

If ϕ is of form 6.1 and ψ is of form 6.3 then $\phi = \psi$ by assumption 6.3.4 and the uniqueness of $\mathcal{F}_f$.

If ϕ is of form 6.2 and ψ is of form 6.3 then $\phi = \psi$ by assumption 6.3.6.

Analogously if ϕ is of form 6.1 and ψ is of form 6.4.

If ϕ is of form 6.2 and ψ is of form 6.4 then assumption 6.3.5 implies that $i = j$ and $s_1 = s_2$. Because of the induction hypothesis we have $\psi_1 = \psi_2$ and hence $\phi = \psi$ again by assumption 6.3.5. □

6.4 The coherence theorem

In the above proof, there seem to be some irregularities.

On coercibility and complexity

Weber states in the proof that

$$t \trianglelefteq t' \Rightarrow \mathrm{com}(t) \leq \mathrm{com}(t')$$

Weber does not prove this, and indeed it is not true.

Suppose that f is an n-ary type constructor and that it is contravariant in its i-th position. If

$$\mathcal{F}_f(s_1, \ldots, s_n, t_1, \ldots, t_n, \phi_1, \ldots, \phi_n) : f(s_1, \ldots, s_n) \to f(t_1, \ldots, t_n)$$

and $\phi_i : t_i \to s_i$ with

$$\mathrm{com}(s_i) > \max(\mathrm{com}(s_1), \ldots, \mathrm{com}(s_{i-1}), \mathrm{com}(s_{i+1}), \ldots, \mathrm{com}(s_n))$$

In other words, $\mathrm{com}(s_i)$ is the *unique*, maximum member of the set

$$\{\mathrm{com}(s_j) \mid j \in \{1, \ldots, n\}\}$$

Then if $\mathrm{com}(t_i) < \mathrm{com}(s_i)$ and for all j in $\{1, \ldots, n\} \setminus \{i\}$ we have that $s_j = t_j$, we have a counterexample Weber's claim, that

$$\mathrm{com}(f(s_1, \ldots, s_n)) > \mathrm{com}(f(t_1, \ldots, t_n))$$

Thus Weber's assertion is invalid.

Structural coercions (in the "proof")

In equation 6.2 (and similarly equation 6.4) ϕ is given as a function

$$f(\ldots, t, \ldots) \to f(\ldots, s_1, \ldots)$$

however, ϕ is supposed to be a function from t.

Structural coercions (syntax)

Weber calls the structural coercion function from $f(s_1, \ldots, s_n)$ to $f(t_1, \ldots, t_n)$

$$\mathcal{F}_f(s_1, \ldots, s_n, t_1, \ldots, t_n, \phi_1, \ldots, \phi_n)$$

where ϕ_i is from s_i to t_i or t_i to s_i depending on whether f is covariant or contravariant in its ith argument, respectively.

However, this is merely the functorial action of f on the maps $\phi_1, \ldots, \phi_n$ and could be represented more compactly as

$$f(\phi_1, \ldots, \phi_n)$$

The source and target of each ϕ_i and knowledge of the sets $\mathcal{A}_f$ and $\mathcal{M}_f$ uniquely determine the source and target of $f(\phi_1, \ldots, \phi_n)$. This also demonstrates the uniqueness of $f(\phi_1, \ldots, \phi_n)$ and guarantees that assumption 6.3.4 holds.

Identity coercions

Weber states in assumption 6.3.1 that the identity function is a coercion. However, he never proves this to be unique. Indeed, automorphisms are perfectly natural maps $t \to t$.

In a computer algebra system like Axiom, many automorphisms are not automorphisms of the smallest category to which a type belongs.

For example, the ring-automorphism $\mathbf{Z}[X, Y] \to \mathbf{Z}[Y, X]$ is not a `PolynomialCategory`-homomorphism since, for instance, the `leadingMonomial` function is not preserved under the map.

For some categories, like the category of groups, this may not be so easy to implement.

We add the following sensible assumption.

Assumption 6.4.1 *The only coercion from a type to itself is the identity function.*

We shall of course allow any number of functions from a type to itself, including conversions. It is merely the number of coercions which we are restricting.

Composition of coercions

It is possibly symptomatic of the previous errors that Weber has neglected to cover all the possible cases of ϕ and ψ.

All our coercions are compositions of coercions (or just a single coercion) from the list

1. coercions between base types;
2. coercions induced by structural coercions;
3. direct embeddings in a type constructor;
4. identity function on ground types as coercions.

Suppose that $\phi = \tau_1 \circ \phi'$ and that $\psi = \tau_2 \circ \psi'$ where τ_1 and τ_2 are single coercions, and ϕ' and ψ' are also (compositions of) coercions. ϕ' may be the identity coercion, as may be ψ'.

For a proof by induction on com, we need to cover all the cases of (τ_1, τ_2) pairs. Thus the list which we need to consider is

1. τ_1 is a coercion between base types. τ_2 is a coercion between base types.
2. τ_1 is a coercion between base types. τ_2 is a structural coercion.
3. τ_1 is a coercion between base types. τ_2 is a direct embedding.
4. τ_1 is a coercion between base types. τ_2 is an identity function.
5. τ_1 is a structural coercion. τ_2 is a coercion between base types.
6. τ_1 is a structural coercion. τ_2 is a structural coercion.
7. τ_1 is a structural coercion. τ_2 is a direct embedding.
8. τ_1 is a structural coercion. τ_2 is an identity function.
9. τ_1 is a direct embedding. τ_2 is a coercion between base types.
10. τ_1 is a direct embedding. τ_2 is a structural coercion.
11. τ_1 is a direct embedding. τ_2 is a direct embedding.
12. τ_1 is a direct embedding. τ_2 is an identity function.
13. τ_1 is an identity function. τ_2 is a coercion between base types.

14. τ_1 is an identity function. τ_2 is a structural coercion.

15. τ_1 is an identity function. τ_2 is a direct embedding.

16. τ_1 is an identity function. τ_2 is an identity function.

Firstly notice we are only interested in the pairs as unordered entities. Thus some of these cases are duplicates.

Indeed: case 5 is case 2; case 9 is case 3; case 10 is case 7; case 13 is case 4; case 14 is case 8; case 15 is case 12. So we may discard cases 5, 9, 10, 13, 14 and 15.

Now notice that a base type can not have a direct embedding, neither can it induce a structural coercion. If either τ_1 or τ_2 is coercion between base types then the target of ϕ and ψ is a base type. Hence the other τ_i can not be a direct embedding nor can it be a structural coercion.

All the cases of this form are 2 (equivalently 5) and 3 (equivalently 9). Thus we may ignore these too.

Our remaining cases are 1, 4, 6, 7, 8, 11, 12 and 16.

To really prove the coherence theorem we are going to require some more assumptions. Only one of them (6.4.5) is difficult to justify.

We shall replace this first assumption which uses Weber's definition 6.2.8 of a direct embedding presently with our own definition 6.4.3. In our definition assumption 6.4.2 shall always hold (trivially).

Assumption 6.4.2 *If a type constructor f has a direct embedding at its i-th position and $f(t_1, \ldots, t_n)$ exists then $\Phi^i_{f,t_1,\ldots,t_n} : t_i \to f(t_1, \ldots, t_n)$.*

Weber defines a f to have a direct embedding (definition 6.2.8) at a particular position if there exists some n-tuple of types $(t_1, \ldots, t_n)$ for which t_i directly embeds in $f(t_1, \ldots, t_n)$.

This seems sensible if we consider a type constructor to be a function which returns the carrier of the principal sort of the algebra. We are saying that direct embedding of a type in a constructor is equivalent to saying that a sort $\preceq$ the principal sort.

It is true that an Axiom type constructor returns the carrier of the principal sort of the algebra but Axiom is more specific than that. The type constructor furnished with full complement of arguments *is* the principal sort of the algebra.

In fact it would be better to replace definition 6.2.8 and assumption 6.4.2 with the following definition.

Definition 6.4.3 *Let $f : (\sigma_1, \ldots, \sigma_n) \to \sigma$ be an n-ary type constructor. If $(\forall i \in \{1, \ldots, n\})(\forall$ ground types $t_i : \sigma_i)$ there exists a coercion*

$$\Phi^i_{f,t_1,\ldots,t_n} : t_i \to f(t_1, \ldots, t_n)$$

then we say that f has a direct embedding at its i-th position*.*

Moreover, let

$$\mathcal{D}_f = \{i | f \textit{ has a direct embedding at its } i\textit{−th position}\}$$

be the set of direct embedding positions of f*.*

The next assumption is undesirable due to its difficulty to guarantee in any implementation. However, it is provable in the more common covariant case (and we shall prove it in the proof of the coherence theorem 6.4.7).

We are assuming that structural coercions behave confluently. The assumption below (6.4.4) is stated "too strongly". We shall state the assumption we really need (6.4.5 — which is more complicated) immediately after. Assumption 6.4.4 gives the idea of what we require, whereas assumption 6.4.5 gives us what is necessary. It is an assumption about type constructors.

Assumption 6.4.4 *Let the type constructor f induce a structural coercion. If $\phi : t \to f(s_1, \ldots, s_n)$, $\psi : t \to f(u_1, \ldots, u_n)$ are coercions and $t' = f(t'_1, \ldots, t'_n)$ then if there exists*

$$\mathcal{F}_f(s_1, \ldots, s_n, t'_1, \ldots, t'_n, \eta_1, \ldots, \eta_n) : f(s_1, \ldots, s_n) \to t'$$

$$\mathcal{F}_f(u_1, \ldots, u_n, t'_1, \ldots, t'_n, \zeta_1, \ldots, \zeta_n) : f(u_1, \ldots, u_n) \to t'$$

the following holds

$$\mathcal{F}_f(s_1, \ldots, s_n, t'_1, \ldots, t'_n, \eta_1, \ldots, \eta_n) \circ \phi = \mathcal{F}_f(u_1, \ldots, u_n, t'_1, \ldots, t'_n, \zeta_1, \ldots, \zeta_n) \circ \psi$$

or equivalently, the following diagrams commute

$$\begin{tikzcd}
t \arrow[r, "\phi"] \arrow[d, "\psi"'] & f(s_1,\ldots,s_n) \arrow[d, "{\mathcal{F}_f(s_1,\ldots,s_n,t'_1,\ldots,t'_n,\eta_1,\ldots,\eta_n)}"'] \\
f(u_1,\ldots,u_n) \arrow[r, "{\mathcal{F}_f(u_1,\ldots,u_n,t'_1,\ldots,t'_n,\zeta_1,\ldots,\zeta_n)}"] & t' = f(t'_1,\ldots,t'_n)
\end{tikzcd}$$

Now, what we actually require. Suppose that a type t is not constructed by the n-ary type constructor f which is contravariant in its ith position at which it also has a direct embedding.

Suppose also that t is coercible to two types which may be directly embedded in f (at the ith position). Let us directly embed them to gain two new types (constructed by f). If these new types (constructed by f) can both be structurally coerced to the same type t' (also constructed by f) then the two compositions of coercions $t \to t'$ are the same.

Succinctly: A structural coercion of a direct embedding of any other coercion is unique.

Assumption 6.4.5 *Let f be a type constructor contravariant at its i-th position at which it also has a direct embedding, and let t be a type not constructed by f.*

If $\phi'_a : t \to a_i$ and $\psi'_b : t \to b_i$ are coercions and $\phi_a : f(a_1,\ldots,a_n) \to f(t'_1,\ldots,t'_n)$ and $\psi_b : f(b_1,\ldots,b_n) \to f(t'_1,\ldots,t'_n)$ are structural coercions then

$$\phi_a \circ \Phi^i_{f,a_1,\ldots,a_n} \circ \phi'_a = \psi_b \circ \Phi^i_{f,b_1,\ldots,b_n} \circ \psi'_b$$

or equivalently, the following diagram commutes

$$\begin{tikzcd}
 & t \arrow[dl, "\phi'_a"'] \arrow[dr, "\psi'_b"] & \\
a_i \arrow[d, "\Phi^i_{f,a_1,\ldots,a_n}"'] & & b_i \arrow[d, "\Phi^i_{f,b_1,\ldots,b_n}"] \\
f(a_1,\ldots,a_n) \arrow[dr, "\phi_a"] & & f(b_1,\ldots,b_n) \arrow[dl, "\psi_b"'] \\
 & f(t'_1,\ldots,t'_n) &
\end{tikzcd}$$

Notice that assumption 6.4.5 is provable in the case that f is covariant at i (and we shall prove this in the proof of the coherence theorem 6.4.7). In the contravariant case, there is no link between t and t'_i and it is this which make it an assumption.

The following assumption is more easily ensurable. We merely require that for two types constructed by the same type constructor, if there exists a coercion between them then it equals the structural coercion between them which we now assume to exist. This is also an assumption on type constructors.

Assumption 6.4.6 *If $f(s_1,\ldots,s_n) \trianglelefteq f(t_1,\ldots,t_n)$ then there exists a structural coercion $f(s_1,\ldots,s_n)$ to $f(t_1,\ldots,t_n)$ and it is the unique coercion $f(s_1,\ldots,s_n)$ to $f(t_1,\ldots,t_n)$.*

We are now finally in a position to state and prove the coercion theorem.

Theorem 6.4.7 (Coherence theorem) *Assume that all coercions between ground types are only built by one of the following mechanisms:*

1. *coercions between base types;*
2. *coercions induced by structural coercions;*
3. *direct embeddings (definition 6.4.3) in a type constructor;*
4. *composition of coercions;*
5. *identity function on ground types as coercions.*

If assumptions 6.3.1, 6.3.3, 6.3.4, 6.3.5, 6.3.6, 6.4.1, 6.4.5 and 6.4.6 are satisfied, then the set of ground types as objects, and the coercions between them as arrows form a category which is a preorder.

PROOF. By assumption 6.3.1 and lemma 6.3.2, the set of ground types as objects and the coercions between them form a category.

All our coercions are either an individual coercion or a composition of finitely many coercions from the list: coercions between base types; structural coercions; direct embeddings (definition 6.4.3) in a type constructor; and identity functions.

Thus we may decompose any coercion into such a finite composition. We define len(ϕ) (the *length* of ϕ) to be the minimum number of coercions in any decomposition of ϕ. Clearly, there are infinitely many values of len(ϕ) but there is a minimum value.

For any two ground types t and t' we will prove by induction on the length of the coercions between them that any coercions between them are unique.

Let $\phi, \psi : t \to t'$ be coercions.

If the length of ϕ and ψ is 1 then (replacing the unordered pair (ϕ, ψ) with (τ_1, τ_2)) we need to look at our eight cases 1, 4, 6, 7, 8, 11, 12 and 16 defined above.

Cases 1 and 4: If ϕ is a base type coercion, then t and t' are base types and coercions between base types are unique by assumption 6.3.3

Case 8, 12 and 16: If ϕ is an identity function on a ground type then $t = t'$ then by assumption 6.4.1 $\psi = \phi$.

Case 6: If ϕ and ψ are structural coercions then ϕ and ψ are equal by assumption 6.3.4.

Case 7: If ϕ is a direct embedding and ψ is a structural coercion then there are two cases.

Case 7a: ψ is covariant. Then by assumption 6.4.6 $\psi = \phi$.

Case 7b: ψ is contravariant. Thus $t = f(t_1, \ldots, t_n)$ and there exists i and a coercion $t \to t_i$. This can not happen since no composition of our four basic coercions can create such a coercion.

Case 11: If ϕ and ψ are direct embeddings then by assumption 6.3.5 $\psi = \phi$.

We now may assume that any coercions of length less than or equal to k are unique.

Suppose that $\phi, \psi : t \to t'$ are coercions and $\max(\text{len}(\phi), \text{len}(\psi)) = k + 1$. Also suppose that $\phi = \tau_1 \circ \phi'$ and $\psi = \tau_2 \circ \psi'$ where for i in $\{1, 2\}$ we have $\tau_i : s_i \to t'$ are single atomic coercions. Also $\phi' : t \to s_1$ and $\psi' : t \to s_2$ are unique coercions since their lengths are less than or equal to k.

If the length of ϕ or ψ is 1 then we define ϕ' or ψ' respectively to be the identity function on t.

As stated before, there are eight cases 1, 4, 6, 7, 8, 11, 12 and 16. (We may transpose τ_1 and τ_2 if we wish since their order is unimportant.)

Case 1: τ_1 is a coercion between base types. τ_2 is a coercion between base types. Then t', s_1, s_2 are all base types. Base types may only be the targets of base type coercions hence t is a base type. Thus ϕ and ψ are coercions between base types and are thus equal by assumption 6.3.3.

Case 4: τ_1 is a coercion between base types. τ_2 is an identity function. Again t and t' must both be base types and ϕ and ψ are equal by assumption 6.3.3.

Case 6: τ_1 is a structural coercion. τ_2 is a structural coercion. Let $t' = f(t'_1, \ldots, t'_n)$.

If t is equal to some $f(t_1, \ldots, t_n)$ then ϕ and ψ are structural coercions $f(t_1, \ldots, t_n)$ to $t' = f(t'_1, \ldots, t'_n)$ and hence are equal by assumption 6.3.4.

Else, we may assume that $t \neq f(t_1, \ldots, t_n)$. Since all our coercions are built from compositions of the four basic types of coercion we may consider ϕ and ψ to be chains of compositions. Without loss of generality, assume that the length of ϕ is $k+1$ and the length of ψ is l where $l \leq k+1$.

$$t = h_0 \overset{\phi_0}{\to} h_1 \cdots h_k \overset{\phi_k}{\to} h_{k+1} = t'$$

and

$$t = s_0 \overset{\psi_0}{\to} s_1 \cdots s_{l-1} \overset{\psi_{l-1}}{\to} s_l = t'$$

In both the composition chains of coercions from t to t' there must exist a minimal element of the chain which is constructed by f and this (by assumption) can not be s_0 or h_0. Also because τ_1 and τ_2 are structural coercions, these minimal elements are not h_{k+1} or s_l.

In the ϕ-chain we shall assume that this element is h_a. In the ψ-chain we shall assume that this element is s_b.

Since h_a is constructed by f and h_{a-1} is not and ϕ_{a-1} is one of the four basic coercions, it must be a direct embedding. (It can not be a structural coercion since h_{a-1} is not constructed by f. It is not a coercion between base types because h_a is constructed by f. It is not an identity coercion since h_{a-1} is not constructed by f whereas h_a is.)

Similarly ψ_{b-1} is also a direct embedding.

Now by assumption 6.3.5 the direct embeddings must be at the same position. We shall assume that this position is the ith.

Let $h_a = f(a_1, \dots, a_n)$ and $s_b = f(b_1, \dots, b_n)$. Call the unique structural coercions $\phi_a : h_a \to t'$ and $\psi_b : s_b \to t'$. (Notice that $h_{a-1} = a_i$ and $s_{b-1} = b_i$.)

Call the maps $\phi'_a : t \to h_{a-1}$ and $\psi'_b : t \to s_{b-1}$ where

$$\phi'_a = \phi_0 \circ \cdots \circ \phi_{a-2}$$

and

$$\psi'_b = \psi_0 \circ \cdots \circ \psi_{b-2}$$

(If a or b equal 1 then we may insert an initial identity coercion and lengthen the chain by one. This does not affect our induction on length since we are always in good shape with identity coercions. Neither a nor b may be 0 by our assumption on t.)

Thus we have the following diagram, and we wish to prove that it commutes.

$$\begin{tikzcd}
& t \arrow[dl, "\phi'_a"'] \arrow[dr, "\psi'_b"] & \\
h_{a-1} = a_i \arrow[d, "\Phi^i_{f,a_1,\dots,a_n}"'] & & s_{b-1} = b_i \arrow[d, "\Phi^i_{f,b_1,\dots,b_n}"] \\
h_a = f(a_1, \dots, a_n) \arrow[dr, "\phi_a"] & & s_b = f(b_1, \dots, b_n) \arrow[dl, "\psi_b"'] \\
& f(t'_1, \dots, t'_n) &
\end{tikzcd}$$

Case 6a: f is covariant in the ith position.

Consider the coercions $\alpha : a_i \to t'_i$ and $\beta : b_i \to t'_i$. (Recall that $t' = f(t'_1, \dots, t'_n)$). These must exist because ϕ_a and ψ_b lift them covariantly respectively.

The definition of direct embeddings 6.4.3 guarantees the existence of $\Phi^i_{f,t'_1,\dots,t'_n}$.

Now by the definitions above

$$\phi = \phi_a \circ \Phi^i_{f,a_1,\ldots,a_n} \circ \phi'_a$$

By assumption 6.3.6

$$\phi_a \circ \Phi^i_{f,a_1,\ldots,a_n} = \Phi^i_{f,t'_1,\ldots,t'_n} \circ \alpha$$

Thus

$$\phi = \Phi^i_{f,t'_1,\ldots,t'_n} \circ \alpha \circ \phi'_a$$

By induction on length we know that the coercion t to t'_i is unique. Hence

$$\alpha \circ \phi'_a = \beta \circ \psi'_b$$

So

$$\phi = \Phi^i_{f,t'_1,\ldots,t'_n} \circ \beta \circ \psi'_b$$

Assumption 6.3.6 gives us

$$\Phi^i_{f,t'_1,\ldots,t'_n} \circ \beta = \psi_b \circ \Phi^i_{f,b_1,\ldots,b_n}$$

Thus

$$\phi = \psi_b \circ \Phi^i_{f,b_1,\ldots,b_n} \circ \psi'_b$$

which by definition means $\phi = \psi$.

Graphically, the following is a commutative diagram.

$$\begin{tikzcd}
 & & t \arrow[dll, "\phi'_a"'] \arrow[drr, "\psi'_b"] & & \\
h_{a-1} = a_i \arrow[drr, "\alpha"] \arrow[dd, "\Phi^i_{f,a_1,\ldots,a_n}"'] & & & & s_{b-1} = b_i \arrow[dll, "\beta"'] \arrow[dd, "\Phi^i_{f,b_1,\ldots,b_n}"] \\
 & & t'_i \arrow[dd, "\Phi^i_{f,t'_1,\ldots,t'_n}"] & & \\
h_a = f(a_1,\ldots,a_n) \arrow[drr, "\phi_a"] & & & & s_b = f(b_1,\ldots,b_n) \arrow[dll, "\psi_b"'] \\
 & & t' = f(t'_1,\ldots,t'_n) & &
\end{tikzcd}$$

Case 6b: f is contravariant in its ith position

There is nothing to link t and t'_i in this case and we must resort to assumption 6.4.5.

$$\begin{tikzcd}
 & t \arrow[ddl, "\phi'_a"'] \arrow[ddr, "\psi'_b"] & \\
 & t'_i \arrow[dl] \arrow[dr] & \\
h_{a-1} = a_i \arrow[d, "\Phi^i_{f,a_1,\ldots,a_n}"'] & & s_{b-1} = b_i \arrow[d, "\Phi^i_{f,b_1,\ldots,b_n}"] \\
h_a = f(a_1, \ldots, a_n) \arrow[dr, "\phi_a"] & & s_b = f(b_1, \ldots, b_n) \arrow[dl, "\psi_b"'] \\
 & t' = f(t'_1, \ldots, t'_n) &
\end{tikzcd}$$

Case 7: τ_1 is a direct embedding. τ_2 is a structural coercion.

Case 7a: τ_2 is covariant. Then $\phi = \psi$ by case 6a above.

Case 7b: τ_2 is contravariant. Let $s_2 = f(s_{2_1}, \ldots, s_{2_n})$ and $t' = f(t'_1, \ldots, t'_n)$. If $\kappa : t'_i \to s_{2_i}$ is the coercion lifted (contravariantly) by τ_2 where

$$\tau_2 = \mathcal{F}_f(\ldots, s_{2_i}, \ldots, t'_i, \ldots, \kappa, \ldots)$$

Also

$$\tau_1 = \Phi^i_{f,t'_1,\ldots,t'_n}$$

The uniqueness of ϕ' implies that

$$\phi' = \Phi^i_{f,s_{2_1},\ldots,s_{2_n}} \circ \kappa \circ \psi'$$

By assumption 6.3.6 we have

$$\mathcal{F}_f(\ldots, s_{2_i}, \ldots, t'_i, \ldots, \kappa, \ldots) \circ \Phi^i_{f,s_{2_1},\ldots,s_{2_n}} \circ \kappa = \Phi^i_{f,t'_1,\ldots,t'_n}$$

Thus

$$\mathcal{F}_f(\ldots, s_{2_i}, \ldots, t'_i, \ldots, \kappa, \ldots) \circ \Phi^i_{f,s_{2_1},\ldots,s_{2_n}} \circ \kappa \circ \psi' = \Phi^i_{f,t'_1,\ldots,t'_n} \circ \psi'$$

and therefore

$$\mathcal{F}_f(\ldots, s_{2_i}, \ldots, t'_i, \ldots, \kappa, \ldots) \circ \phi' = \Phi^i_{f,t'_1,\ldots,t'_n} \circ \psi'$$

Hence, $\phi = \psi$.

Case 8: τ_1 is a structural coercion. τ_2 is an identity function. Let $t' = f(t'_1, \ldots, t'_n)$. By viewing τ_2 as the identity function

$$\mathcal{F}_f(t'_1, \ldots, t'_n, t'_1, \ldots, t'_n, \mathrm{id}_{t'_1}, \ldots, \mathrm{id}_{t'_n})$$

Thus $\phi = \psi$ by case 6.

Case 11: τ_1 is a direct embedding. τ_2 is a direct embedding. Assumption 6.3.5 implies that the direct embeddings must be at the same position, i. Therefore $s_1 = s_2$. So by the inductive hypothesis $\phi' = \psi'$. By assumption 6.3.5 $\tau_1 = \tau_2$. Hence $\phi = \psi$.

Case 12: τ_1 is a direct embedding. τ_2 is an identity function. Let $t' = f(t'_1, \ldots, t'_n)$. By viewing τ_2 as the identity structural coercion

$$\mathcal{F}_f(t'_1, \ldots, t'_n, t'_1, \ldots, t'_n, \mathrm{id}_{t'_1}, \ldots, \mathrm{id}_{t'_n})$$

Thus $\phi = \psi$ by case 6, above.

Case 16: τ_1 is an identity function. τ_2 is an identity function. Thus $\phi = \phi'$ and $\psi = \psi'$. However, since $s_1 = s_2$, $\phi' = \psi'$ by the inductive hypothesis. Hence $\phi = \psi$.

Thus we have proved $\phi = \psi$ for all coercions of length less than or equal to $k+1$. Hence the result hold for all coercions, by induction. □

In case 10b of the case when the length is 1 we claimed that there can not be built a coercion from a type constructor to one of its arguments.

From our description so far of Axiom, one might think that Axiom would permit the construction of the following type

```
Fraction(Fraction(Integer))
```

which is the quotient field of the quotient field of integers[1]. However, the quotient field of a quotient field is itself and hence,

$$\texttt{Fraction(Fraction(Integer))} \cong \texttt{Fraction(Integer)}$$

[1] The quotient field of the integers ($\mathbf{Z}$) is more commonly known as the rationals ($\mathbf{Q}$).

For this reason, Axiom contains special code in the interpreter to stop such pathological types being instantiated. A Boot function (`isValidType`) checks to see if one is trying to create a type like the one above, or

```
Polynomial(Polynomial(Integer))
```

But this is hand-crafted, special code covering a few cases and mentions the types by name. It is conceivable that a user could create a new type called `MyFraction` which is identical to `Fraction`. This would not be picked up be `isValidType` and thus

```
MyFraction(MyFraction(Integer))
```

could be instantiated. Since the type is then isomorphic to one of its arguments, it is feasible that a coercion between the two could be defined, contradicting our claim in case 10b of the case when the length is 1.

This coercion can still not be built from our four basic types, thus defining such a coercion contradicts our assumptions for the coherence theorem.

6.5 Extending the coherence theorem

Firstly, Weber's "proof" of his conjecture 6.3.7 relies on induction on com(t). This assumes that there are no types of infinite complexity in our system. This is not the case in Axiom, since one could define the following types (in one file):

```
R(r : Ring) : Ring == r
D1 : Ring == R(D2)
D2 : Ring == R(D1)
```

(though calling `1()$D1` would be disastrous[2]!) So we add the following extra assumption.

Assumption 6.5.1 *There do not exist any types of non-finite complexity.*

The proof of the coherence theorem 6.4.7 does not rely on type having finite complexity. However it is still a sensible assumption to make and can be easily guaranteed in a real implementation. It is also imperative that this assumption holds if the automated coercion algorithm 7.4.1 is to terminate.

[2]This would try to create the constant `1` from the ring `D1` which can not be evaluated.

Assumption 6.3.5 states that we may have only one direct embedding into an algebra. However, in `Polynomial Integer` one would wish to perform the direct embeddings of both of `Integer` and `Symbol`, yet this violates assumption 6.3.5 and thus we would not be able to prove that our algebra system is coherent.

The reason for Weber's assumption is to stop coercions like the following occurring. If A is a group, then being able to coerce $A \to A \times A$ whilst potentially useful, is ambiguous. As he points out in [Web93b] [Web93a] [Web95], A can be coerced into $A \times A$ via the isomorphisms $A \cong A \times I$ or $A \cong I \times A$ (where I is the trivial subgroup of A).

In this example the inclusions are ambiguous, since using either coercion, the target of the inclusion is a group. However, in many cases the types are "incomparable", (eg. `Integer` and `Symbol`) and thus the assumption seems to be too strong. The question is what do we mean by "incomparable"?

Certainly, there is no coercion function `Integer` $\to$ `Symbol` or `Symbol` $\to$ `Integer`. But, for two distinct non-trivial, proper normal subgroups G, H of A such that $G \cap H = I$, then there is no coercion function $A/G \to A/H$ or $A/H \to A/G$, yet there then exists two distinct coercion functions $A \to A/G \times A/H$, and coherency is lost. (A/G is the quotient group "A factored out by G" and is the set $\{aG \mid a \in A\}$ where $aG = a'G$ iff $a^{-1}a' \in G$). Thus, the condition of there not existing a coercion function between our two types is not sufficient for our definition of "incomparability".

However, we notice that there exists no type which can be coerced to both `Symbol` and `Integer`, but there does exist homomorphisms $A \to A/G$ and $A \to A/H$. So if we choose the statement "Types `A` and `B` are incomparable if there does not exist a type `S` which can be coerced to both `A` and `B`" as a definition of incomparability then we are in good shape.

We state this in the language of Weber as follows. To replace the assumption we first need to alter definition 6.4.3 (our previous replacement of definition 6.2.8) to the following.

Definition 6.5.2 *Let $f : (\sigma_1, \ldots, \sigma_n) \to \sigma$ be an n-ary type constructor. If $(\forall i \in \{1, \ldots, n\})(\forall$ ground types $t_i : \sigma_i)$ there exists a coercion*

$$\Phi^i_{f,t_1,\ldots,t_n} : t_i \to f(t_1, \ldots, t_n)$$

then we say that f has a direct embedding at its i-th position.

Moreover, let s be a ground type and $i \in \{1, \ldots, n\}$ and define,

$$\mathcal{P}(i, s) := ((\exists \psi : s \to t_i) \wedge (\exists \Phi^i_{f,t_1,\ldots,t_n} : t_i \to f(t_1, \ldots, t_n)))$$

where ψ is a coercion, and also define,

$$\bar{D}_f := \{(i, s) | \mathcal{P}(i, s)\}$$

Note then that $D_f = \{i | (\exists s)(\mathcal{P}(i, s))\}$. We now alter assumption 6.3.5 to the following

Assumption 6.5.3 *Let f be an n-ary type constructor. Then the following conditions hold:*

1. $(i, s), (j, s) \in \bar{D}_f \to i = j$
2. *Direct embedding coercions are unique. i.e. if $\Phi^i_{f,t_1,\ldots,t_n} : t_i \to f(t_1, \ldots, t_n)$ and $\Psi^i_{f,t_1,\ldots,t_n} : t_i \to f(t_1, \ldots, t_n)$ then*

$$\Phi^i_{f,t_1,\ldots,t_n} = \Psi^i_{f,t_1,\ldots,t_n}.$$

So we may now extend the coherence theorem (theorem 6.4.7) by altering the assumption list to our new relaxed set of assumptions. It is not necessary to reprove the entire theorem, but merely the cases which involved assumption 6.3.5

Theorem 6.5.4 (Extended coherence theorem) *Assume that all coercions between ground types are only built by one of the following mechanisms:*

1. *coercions between base types;*
2. *coercions induced by structural coercions;*
3. *direct embeddings (definition 6.5.2) in a type constructor;*
4. *composition of coercions;*
5. *identity function on ground types as coercions.*

If assumptions 6.3.1, 6.3.3, 6.3.4, 6.3.6, 6.5.1, 6.4.6, 6.4.1, 6.4.5 and 6.5.3 are satisfied, then the set of ground types as objects, and the coercions between them as arrows form a category which is a preorder.

PROOF. The entire proof of theorem 6.4.7 is valid except where assumption 6.3.5 was used.

First we deal with the length of ϕ and ψ being 1. The only case which relied on assumption 6.3.5 was case 11.

Case 11: If ϕ and ψ are direct embeddings then by assumption 6.5.3 $\psi = \phi$.

Now, the induction case. The only cases which relied on assumption 6.3.5 were cases 6 and 11.

Case 6: In the proof of theorem 6.4.7 in this case we relied on assumption 6.3.5 to show that the two direct embeddings $h_{a-1} \to h_a$ and $s_{b-1} \to s_b$ were at the same position.

Now since $t \trianglelefteq h_{a-1}$ and $t \trianglelefteq s_{b-1}$, by assumption 6.5.3 the direct embeddings must be at the same position.

Thus the rest of the proof of this case holds.

Case 11: τ_1 is a direct embedding. τ_2 is a direct embedding. Since $t \trianglelefteq s_1$ and $t \trianglelefteq s_2$ assumption 6.5.3 implies that the direct embeddings must be at the same position, i. Therefore $s_1 = s_2$. So by the inductive hypothesis $\phi' = \psi'$. By assumption 6.5.3 $\tau_1 = \tau_2$. Hence $\phi = \psi$.

Inserting the rest of the proof of theorem 6.4.7 completes the proof. □

Thus we have relaxed an important one of the conditions of the coherence theorem 6.4.7 and proved that the theorem still holds.

6.6 Conclusion

In this chapter we stated all the mathematics needed to state Weber's coherence conjecture and give his proof.

We have also stated extra mathematics to correct the statement of the conjecture and then prove it; hence promoting it to a theorem. We have then relaxed one of the conditions and shown that the theorem still holds.

This theorem is the cornerstone for ensuring correct coercions.

By having a coherent type system, then provided we are careful in how we implement our type constructors (the must satisfy the assumptions) and how we create our coercions (compositions of the four basic types) all our coercions are then unique.

Chapter 7

The automated coercion algorithm

7.1 Introduction

In this chapter, we shall introduce some extra mathematics which will allow us to state the automated coercion algorithm. Furthermore, this sound foundation will allow us to demonstrate that all coercions generated by the algorithm are, in fact, homomorphisms. Indeed, we may even guarantee that these are coercions in the sense of definition 5.5.2.

All the work in this chapter is the author's own.

7.2 Finitely generated algebras

In this section we shall define what it means to say that an algebra is finitely generated, and also what it means to say that a finitely generated algebra is decomposable.

Definition 7.2.1 *Let Σ be an order-sorted S-signature. We say that S has a* principal sort *if $\langle \Sigma, S \rangle$ defines an algebra which has one sort (which without loss of generality, we shall always assume to be S_1) which is the "most interesting" of the algebra.*

By "most interesting", we mean that a more naïve algebraist would consider this sort to define the entire algebra.

For example, if we define Γ to be the Group algebra with sorts,

("the group", "a boolean sort", "an integer number system")

then the "most interesting" sort is the one for *"the group"* which would be carried by a set of all the group elements.

All Axiom algebras must have a principal sort, referred to as `%`, in the source code. One can think of $S_1 =$`%` as the sort that the Axiom code "constructs". In the terminology of chapter 6 it is the type constructed by the type constructor. (Axiom views base types as type constructors with no arguments.)

Definition 7.2.2 *We say that a theory $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$, is* finitely generated *iff S has a principal sort, S_1, and the following set is finite,*

$$\ddot{\Sigma} := \bigcup_{n,q} \Sigma_{n,q,S_1}$$

This states that given a finitely generated Σ-algebra, $\langle A, \alpha\rangle$ (which is a model for $\mathcal{S}$) then there are a finite number of operators which can return an element of the carrier of the principal sort.

This means that only finitely many functions (directly) construct (the carrier of) the principal sort.

The next definition allows us to decompose any element of such a Σ-algebra into at least two pieces. Moreover, there exists such a decomposition for each one of the (finitely many) constructors. Furthermore, decomposing such an element and recombining using the corresponding constructor is equivalent to the identity function.

Definition 7.2.3 *A finitely generated theory $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$, is* decomposable *iff*

$$(\forall \sigma \in \ddot{\Sigma})((\sigma \in \Sigma_{n,q,S_1}) \Rightarrow$$

$$(\forall i \in \{1, \ldots, n\})(\exists \pi_{\sigma,i})(\pi_{\sigma,i} \in \Sigma_{1,S_1,q_i}) \wedge ((\sigma(\pi_{\sigma,1}(\cdot), \ldots, \pi_{\sigma,n}(\cdot))$$
$$= \mathrm{id}_{S_1}) \in \mathcal{S}))$$

For example, in a an algebra of lists, `cons` is a constructor. The corresponding functions which decompose an element are `car` and `cdr`. The following equation holds.

`cons(car(x),cdr(x))` $=$ `x`

Notice that this equation is only well formed when `x` is not the empty list. For the empty list, which is constructed by the 0-ary function `nil()`, there are no decomposers, and the equation is as follows.

$$\texttt{nil}() = \texttt{x}$$

It is important to note that the part functions $\pi_{\sigma,i}$ are usually partial functions. For example `car` is not defined on the empty list.

Notice that we have not yet defined a way of differentiating between which constructors construct which element. We shall do this in the following section.

7.3 Constructibility

We now come to the most important concept of this thesis. The aim of this thesis is to provide a method for creating coercions constructively. We now supply the means do so.

As usual, we need to make some definitions first. We will state what we mean by a(n) (algorithmically) (re)constructible algebra. What we really mean is that an algebra is (algorithmically) (re)constructible if some subset of the elements of the most interesting sort-carrier of that algebra can be "built up" from a finite family of operators.

This finite family consists of "constructors" used to create increasingly "large" elements of the principal sort.

We shall also have equations in the theory linking the constructors to "part functions" which we shall use to split large elements into an n-tuple of smaller elements. The constructor applied to its associated part functions acting on an element will be equivalent to the identity function.

We shall be able to tell how an element is constructed by using a "query function". That is to say we must be able to know which constructor(s) may construct any particular element of the principal sort.

We require these definitions so that if an algebra is a model of some algorithmically reconstructible theory, and another algebra contains some of the constructors of the first algebra, then some subset of the elements of the most interesting sort-carrier can be "coerced" from the first type to the second. We shall show this to be a Ξ-homomorphism, where Ξ is a particular signature to which both algebras belong.

We demand our extensions to be protecting extensions so that performing operations on an algebra (or looking at equations of elements of that algebra) when viewed as a model of the original theory or the extension of the theory yield "the same result".

Definition 7.3.1 *We call a theory $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$* constructible *iff Σ is a protecting exten-*

sion of a finitely generated theory. We then call Σ a constructing *signature, and any model for the theory, a* constructible algebra*.*

Similarly,

Definition 7.3.2 *We call a theory $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$* reconstructible *iff Σ is a protecting extension of a decomposable theory. We then call Σ a* reconstructing *signature, and any model for the theory, a* reconstructible algebra*.*

Moreover,

Definition 7.3.3 *If $\langle\langle\Xi, X\rangle, \mathcal{X}\rangle$ is the decomposable theory extended by the reconstructible $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$, we define the* constructors *of $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$ to be*

$$\Sigma^C := \ddot{\Xi}$$

We define the constant constructors *to be the constructors of arity 0, denoted by Σ^0 and the* non-constant constructors *to be the constructors of all other arities. We denote this set by $\Sigma^{C-0} := \Sigma^C \setminus \Sigma^0$*

We also define the part functions *of $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$ associated with $\sigma \in \Sigma^{C-0}$ to be the $\pi_{\sigma,i}$. (Constant constructors have no associated part functions.)*

Finally we define the constructor equations *to be the following set and demand it to be a subset of $\mathcal{S}$ (and hence $\mathcal{X}$) given by,*

$$\{(\sigma(\pi_{\sigma,1}(\cdot), \ldots, \pi_{\sigma,n}(\cdot)) = \mathrm{id}_{S_1}) \mid \sigma \in \Sigma^{C-0}\}$$

We also need to ensure that certain other relationships between constructors which hold in our source algebra hold in our target algebra.

If an element of the source algebra may be constructed in more than one way, we require that reconstructing that element in the target algebra using any of those methods yields the same result.

Notice that since our theories are protecting extensions we are in good shape, since any equation that holds in the protecting extension which concerns only the sorts from the original (unextended) signature must hold in the original signature. Thus these equations will hold in the target algebra.

We call these equations that link different constructors together the secondary constructor equations.

Definition 7.3.4 *We call a reconstructible theory* $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$ algorithmically reconstructible *iff : S contains a boolean sort, B; Σ contains a set of symbols, called the* query functions*, denoted and defined by*

$$\Sigma^Q := \{\chi_\sigma \in \Sigma_{1,S_1,B} \mid \sigma \in \Sigma^C\};$$

and $\mathcal{S}$ contains the following equations, called the query equations

$$(\forall \sigma_{n,q,S_1} \in \Sigma^C)(\chi_{\sigma_{n,q,S_1}}(\sigma_{n,q,S_1}(a_1, \ldots, a_n)) = T)$$

We also demand that for each σ_{n,q,S_1} in Σ^C,

$$((\nexists a_1, \ldots, a_n)(\omega = \sigma_{n,q,S_1}(a_1, \ldots, a_n))) \Rightarrow (\chi_{\sigma_{n,q,S_1}}(\omega) = F)$$

Finally we demand that all secondary constructor equations are defined in the theory.

A secondary constructor equation *is any equation which has a constructor as the final symbol on both right and left sides.*

We say "final symbol" to mean this is operator applied last. In our notation (functions written on the left) a final symbol is written to the left of all the other elements of a formula.

For example, $\sigma()$ and $\sigma(e_1, \sigma'(e_2, e_3))$ both have σ as a final symbol.

In the algorithm we shall demand that the decomposable theory which both types model is the smallest theory which they both model. This is to ensure that the algorithm not only creates a homomorphism, but that it is a coercion (definition 5.5.2). The proof of this is given in corollary 7.6.8.

Example 7.3.5 We shall take the variety of *Lists* as our example, and we shall assume that `List` is a member of this variety. Then the constructors for the model are `nil` and `cons`, where `nil` is a constant constructor whereas `cons` is a non-constant constructor.

`cons`' associated part-functions are `car` and `cdr`. Thus we have the constructor equation:

$$\texttt{cons}(\texttt{car}(x), \texttt{cdr}(x)) = \mathrm{id}(x)$$

The query functions are `null` and `consp`, and our query equations are,

$$\texttt{null}(\texttt{nil}) = T$$

and for any list l and any element of the underlying type, i,

$$\texttt{consp}(\texttt{cons}(i, l)) = T$$

Notice also that $\texttt{consp}(\texttt{nil}) = F$ and also, for the same l and i, $\texttt{null}(\texttt{cons}(i, l)) = F$.

It is now worth discussing some of the finer points of homomorphism.

For ϕ to be a homomorphism we require that the constructor functions are preserved by homomorphism. We do not require that the part or query functions are homomorphically preserved; indeed we do not even require that they are in the signature of the algebra of the target of ϕ.

Thus we do not require that both the source and target of ϕ are models of the same algorithmically reconstructible theory, but that the source is model of an algorithmically reconstructible theory, $\mathcal{T}$, and that all the constructors of $\mathcal{T}$ are functions of the target of ϕ, inherited from the *same* signature (theory).

This is an important point. As an example, we shall consider polynomial rings. A polynomial is a function such as

$$5x^2y^3 + 9xy^{45} - 34x + 7y^2 - 12$$

That is a sum of products of an element of the underlying ring (which in the above example might be the integers) and variables raised to non-negative powers. A monomial is a polynomial which is a product of a non-zero element of the underlying ring and variables raised to non-negative powers.

A polynomial ring is ordered by an extension of the order on the variables. In particular, in $\mathbf{Z}[x, y]$ if $x > y$ then a monomial m_1 is greater than another m_2 if the exponent of x is greater in m_1 than in m_2. Should the exponent of x be equal in both, then the exponent of y is compared in the same manner.

The leading monomial of a polynomial is the largest monomial of a polynomial. If $x > y$ in the above polynomial then $5x^2y^3$ is the leading monomial. Should $y > x$ then the leading monomial would be $9xy^{45}$.

The reductum of a polynomial is the polynomial less its leading monomial.

Coercions between two polynomial rings need only use `leadingMonomial` as a part function (which is not preserved via homomorphism, since it depends on the ordering given to the variables) instead of some (unnatural) fixed (for all polynomial rings with variable from a fixed domain) "most-important-monomial" function which would choose the same monomial, regardless of variable ordering.

Also, this allows us to form the natural monomorphism from $\mathbf{Q}[x]$ to $\mathbf{Z}(x)$, which in Axiom is

`Polynomial Fraction Integer` to `Fraction Polynomial Integer`

which are, in Axiom's view (without clever hackery in the interpreter) two unrelated `Rings`. This may be constructed *without* having to force `+` to be a constructor of `Fraction Polynomial Integer`, or indeed `leadingMonomial` etc. to be available in `Fraction Polynomial Integer`.

7.4 The algorithm

We are now in a position to state the automated coercion algorithm.

The algorithm to create the coercion will be stated in English. It is too implementation dependent to state any finer.

The actual algorithm to coerce will be stated as Lisp pseudo-code. In Lisp, `(a b c)` means apply the function `a` to the arguments `(b,c)`. `cond` is the like the `switch` statement in C or Java.

`cond`s or `switch`es are equivalent to "if-then-else" statements — if a condition is true, then we evaluate and return the following statement (and leave the `cond` block), else go to the next condition and repeat.

For example, the line

$$\texttt{((}\alpha_{\chi_{\sigma_{0,\emptyset,S_1}}}\ \texttt{x) (}\beta_{\sigma_{0,\emptyset,S_1}}\texttt{))}$$

means

$$\texttt{if } \alpha_{\chi_{\sigma_{0,\emptyset,S_1}}}\texttt{(x) then } \beta_{\sigma_{0,\emptyset,S_1}}\texttt{()}$$

The final statement `t`, is the default statement, since `t` is always true. This line will only be reached when all the other conditions have not been satisfied, and shows that we have failed to build a total automated coercion function.

This could happen if one fails to list all the constructors (and their queries) for a type.

The lines of the form

$$\texttt{((}\alpha_{\chi_{\sigma_{0,\emptyset,S_1}}}\ \texttt{x) (}\beta_{\sigma_{0,\emptyset,S_1}}\texttt{))}$$

check for the constants of the type. For example, the constant polynomial 0 can not be constructed using a non-zero number of parts (using any normal construction methodology) thus coercing 0 from $\mathbf{Q}[x]$ to $\mathbf{Z}(x)$ we might say

```
if zero?(x)$Polynomial Fraction Integer
then zero()$Fraction Polynomial Integer
```

returning the appropriate 0 in $\mathbf{Z}(x)$.

The lines of the form

$$\texttt{((}\alpha_{\chi_{\sigma_{n,q,S_1}}}\ \texttt{x) (}\beta_{\sigma_{n,q,S_1}}\ \texttt{(}\psi_{q_1}\ \texttt{(}\alpha_{\pi_{\sigma_{n,q,S_1},1}}\ \texttt{x))}\ \ldots\texttt{(}\psi_{q_n}\ \texttt{(}\alpha_{\pi_{\sigma_{n,q,S_1},n}}\ \texttt{x))))}$$

are the constructing lines. For example, in a polynomial ring we might write

```
    if x is the sum of a monomial and a polynomial
then coerce(leadingMonomial(x)) + coerce(reductum(x))
```

where the addition function is that taken from the target domain.

In this case `leadingMonomial(x)` and `reductum(x)` are both polynomials. In general, the parts of the element need not be of the same type as the original element.

For example, in List algebras, coercing from `List(A)` to `List(B)` where there exists (or we can build) a coercion from `A` to `B` then we have (back in Lisp terminology)

```
((consp x) (cons (coerce (car x)) (coerce (cdr x))))
```

`(car x)` is the first element of the list and is of type `A` rather than `List(A)` — the type of `x`. So `(coerce (car (x)))` is of type `B` and thus may be `cons`ed on to the front of `(coerce (cdr x))` which is of type `List(B)`.

Algorithm 7.4.1 (The Automated Coercion Algorithm) *Let $\langle A, \alpha\rangle$ be a model for the algorithmically reconstructible theory $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$.*

Let $\langle B,\beta\rangle$ be a model of $\langle\langle\Delta,D\rangle,\mathcal{D}\rangle$ where $\langle\langle\Sigma,S\rangle,\mathcal{S}\rangle$ is a protecting extension of $\langle\langle\Delta,D\rangle,\mathcal{D}\rangle$ and some (or all) of the constructors of $\langle\langle\Sigma,S\rangle,\mathcal{S}\rangle$ are in fact in $\langle\langle\Delta,D\rangle,\mathcal{D}\rangle$.

Also we demand that there does not exist an extension of $\langle\langle\Delta,D\rangle,\mathcal{D}\rangle$ which both $\langle A,\alpha\rangle$ and $\langle B,\beta\rangle$ model.

Then the following is an algorithm to coerce from $\langle A,\alpha\rangle$ to $\langle B,\beta\rangle$.

The ψ_i for $i \neq 1$ are the (potentially automated) coercions from A_i to B_i from the abstract type of S_i.

The entire morphism created thus is called ψ and not only is it a homomorphism, it is a coercion (definition 5.5.2).

```
$\psi_1(x)$ := (cond
          (($\alpha_{\chi_{\sigma_{0,\emptyset,S_1}}}$ x) ($\beta_{\sigma_{0,\emptyset,S_1}}$))
            ⋮ Repeat for each $\sigma_{0,\emptyset,S_1}$ in $\Sigma^0$.
          (($\alpha_{\chi_{\sigma_{n,q,S_1}}}$ x) ($\beta_{\sigma_{n,q,S_1}}$ ($\psi_{q_1}$ ($\alpha_{\pi_{\sigma_{n,q,S_1},1}}$ x)) ...($\psi_{q_n}$ ($\alpha_{\pi_{\sigma_{n,q,S_1},n}}$ x))))
            ⋮ Repeat for each $\sigma_{n,q,S_1}$ in $\Sigma^{C-0}$.
          (t (error)))
```

The algorithm to create the coercion ψ_1 is

```
createCoerce($\langle A,\alpha\rangle$,$\langle B,\beta\rangle$) :=
          determine $\langle\langle\Delta,D\rangle,\mathcal{D}\rangle$ (error if doesn't exist)
          determine $\langle\langle\Sigma,S\rangle,\mathcal{S}\rangle$ (error if doesn't exist)
          determine $\Sigma^0$
          for $\sigma \in \Sigma^0$
            determine $\alpha_{\chi_\sigma}$
            determine $\beta_\sigma$
          determine $\Sigma^{C-0}$
          for $\sigma \in \Sigma^{C-0}$
            determine $\alpha_{\chi_\sigma}$
            determine $\beta_\sigma$
            determine $\alpha_{\pi_{\sigma,i}}$ (for all relevant i)
          construct and return $\psi_1$ as defined above
```

So the algorithm presented above allows us to algorithmically reconstruct elements of one type as elements of another.

7.5 Existence of the coercion

Before we prove that the automated coercion algorithm constructs a coercion when one exists we must prove that it does not create some unnatural function when no coercion exists.

We shall use the terminology of algorithm 7.4.1 in this section.

In the simplest case, some of the functions will not exist in the target type and the algorithm will error at an early stage. This will be because there is no $\langle\langle\Delta, D\rangle, \mathcal{D}\rangle$ which both $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ model.

If a homomorphism exists but not a coercion then this means that there exists an extension of $\langle\langle\Delta, D\rangle, \mathcal{D}\rangle$ which both $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ model. In this case the algorithm will error at an early stage.

If all the constructors of $\langle A, \alpha\rangle$ are available in $\langle B, \beta\rangle$ but no homomorphism (coercion) exists then this could be because of the (at least one of the) following causes.

One of the coercions to be used does not exist

At least one of types which is required for construction or recursively required for construction (and which is not the carrier of the principal sort) may not be coercible to its counterpart in the target. Provided the algorithm checks at construction time that every coercion used directly or indirectly by ψ_i exists (or is constructible) then we may report an error at an early stage.

Non-homomorphic constructors

The fact that $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$ is a protecting extension of $\langle\langle\Delta, D\rangle, \mathcal{D}\rangle$ and that all secondary constructor equations hold in $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$ guarantees that these equations hold in $\langle\langle\Delta, D\rangle, \mathcal{D}\rangle$ and hence in $\langle B, \beta\rangle$.

This not only ensures that the order of the lines in ϕ_1 is unimportant but also that if a certain relationship holds in $\langle A, \alpha\rangle$ it *must* hold in $\langle B, \beta\rangle$.

For example, if someone defines that all finite field algebras are constructed by `0` and `succ` (the successor function) the automated coercion algorithm will not attempt to create the "coercion" $\mathbf{Z}_5 \to \mathbf{Z}_3$. This is because there exists a secondary constructor equation in the theory of finite fields of size n

$$\mathtt{succ}^n(\mathtt{0}) = \mathtt{0}$$

So in this example $\langle A, \alpha\rangle = \mathbf{Z}_5$ and $\langle B, \beta\rangle = \mathbf{Z}_3$. The equation $\mathtt{succ}^5(\mathtt{0}) = \mathtt{0}$ holds in the theory of finite fields of size 5. $\mathbf{Z}_5$ is a obviously a model for this theory.

This equation is not true in any theory which the above theory extends and of which $\mathbf{Z}_3$ is a model. Otherwise the equation would hold in $\mathbf{Z}_3$ and that patently is not true.

Thus the automated coercion algorithm will error at an early stage from not being able to find $\langle\langle\Delta, D\rangle, \mathcal{D}\rangle$.

7.6 Proving homomorphicity and coerciveness

We now make some notational definitions so that we may prove the final result of the chapter. That is, we shall show that algorithm 7.4.1 constructs a homomorphism (theorem 7.6.7) which is a coercion (corollary 7.6.8).

Recall notation 4.2.10. Thus for any constructor symbol σ_{n,q,S_1} the associated constructor function in $\langle A, \alpha\rangle$ is $\alpha_{\sigma_{n,q,S_1}}$ the query function is $\alpha_{\chi_{\sigma_{n,q,S_1}}}$ and for $i \in \{1, \ldots, n\}$ the part functions are $\alpha_{\pi_{\sigma_{n,q,S_1},i}}$.

Definition 7.6.1 *For a term t in a term algebra $T_\Sigma(X)$ if*

1. $t \in X_s$ *then* $\mathrm{length}(t) = 1$
2. $t \in \Sigma_{0,(),s}$ *then* $\mathrm{length}(t) = 1$
3. $t = \sigma(t_1, \ldots, t_n)$ *then* $\mathrm{length}(t) = 1 + \mathrm{length}(t_1) + \cdots \mathrm{length}(t_n)$

This definition of length of an element will form the basis of our inductive proof of the automated coercion algorithm. However, in general, we shall not be dealing with term algebras but their homomorphic images.

Definition 7.6.2 *For x an element of a Σ-algebra, we define*

$$\mathrm{length}(x) := \min\{t \in T_\Sigma(X) | \theta^*(t) = x\}$$

where θ^* is the unique homomorphism given in the first universality theorem 4.4.7.

Assumption 7.6.3 *If one of the part functions $\alpha_{\pi_{\sigma_{n,q,S_1}},i}$ corresponds to $\alpha_{\sigma_{1,(S_1),S_1}}$, (thus $q_i = S_1$) then we demand that,*

$$(\forall t \in T_{\Xi}(X)_1)(\mathrm{length}(\alpha_{\pi_{\sigma_{n,q,S_1}},i}(\theta^*(t))) < \mathrm{length}(\theta^*(t))).$$

where θ^* is the unique homomorphism given in the first universality theorem 4.4.7.

A couple of technical definitions to make the next assumption easier to understand.

Definition 7.6.4 *Suppose $\langle A, \alpha\rangle$ is a constructed algebra, constructed by the S-sorted signature Σ. If for any $i \in \{2, \ldots, |S|\}$ we have that S_i appears in the arity of any of the constructors of Σ, we say that A_i is* required for construction *by A_1.*

So for example, in `List(Integer)` we have that `Integer` is required for construction since it is an argument of `cons`. Notice that `List(Integer)` is not required for construction itself since it is the carrier of S_1.

Definition 7.6.5 *Suppose $\langle A, \alpha\rangle$ is a constructed algebra, constructed by the S-sorted signature Σ. Let A_i be required for construction by A_1. Now consider A_i as the carrier of the principal sort of $\langle B, \beta\rangle$, a constructed algebra.*

If B_j is required for construction by $B_1 = A_i$ (hence $j \neq 1$) then we say that B_j is recursively required for construction *by A_1.*

Now suppose that the carrier D_1 of the principal sort of a constructed algebra $\langle D, \delta\rangle$ is recursively required for construction by A_1. If D_k is required for construction by D_1, then we also say that D_k is recursively required for construction *by A_1.*

So for example in `List(List(Integer))`, the only type which is required for construction is `List(Integer)` since this is the only argument of `cons` which is not the carrier of the principal sort, `List(List(Integer))`.

Thus the only types which are recursively required for construction are `List(Integer)` and the types which are recursively required for construction by `List(Integer)`.

Since `Integer` is the only type required for construction by `List(Integer)` it is the only type recursively required for construction by `List(Integer)`.

Thus the types which are recursively required for construction by `List(List(Integer))` are `List(Integer)` and `Integer`.

The following assumption is required so that we may prove that the automated coercion algorithm terminates.

Assumption 7.6.6 *We demand that A_1 is not recursively required for construction.*

Theorem 7.6.7 *Let $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$ be an algorithmically reconstructible theory which is a protecting extension of the theory $\langle\langle\Xi, X\rangle, \mathcal{X}\rangle$ such that $\Sigma^C \subseteq \ddot{\Xi}$.*

Let $\langle B, \beta\rangle$ be a model for $\langle\langle\Xi, X\rangle, \mathcal{X}\rangle$ and $\langle A, \alpha\rangle$ be a model for $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$.

Then the function ψ given in algorithm 7.4.1 is a Ξ-homomorphism $\langle A, \alpha\rangle \to \langle B, \beta\rangle$.

PROOF.

Let ϕ be the correct homomorphism which ψ is attempting to emulate.

Since ψ_1 contains finitely many cases, and covers all cases of constructors for Σ, we need only consider one line from ψ_1. Also by induction on com(t) we can assume that $(\forall i \in \{2, \ldots, |S|\})(\psi_i = \phi_i)$.

Since ψ_1 covers all cases of constructor for Σ we do not need to consider the error line since it will never be reached.

For ψ_1 we may induct on length, and we are in good shape by assumption 7.6.3 (on length) and 7.6.6 (on the interaction between length and com).

Here is one line from ψ_1.

$$\texttt{(if } (\alpha_{\sigma_q}\ \texttt{a})\ (\beta_{\sigma_c}\ (\psi_{c_1}\ (\alpha_{p_1}\ \texttt{a}))\ \ldots(\psi_{c_n}\ (\alpha_{p_n}\ \texttt{a}))))$$

Now, assuming $(\alpha_{\sigma_q}\ \texttt{a})$, we know that,

$$(\phi_1\ \texttt{a}) = (\phi_1\ (\alpha_{\sigma_c}\ (\alpha_{p_1}\ \texttt{a})\ \ldots(\alpha_{p_n}\ \texttt{a})))$$

then since ϕ is a Σ-homomorphism,

$$(\phi_1\ (\alpha_{\sigma_c}\ (\alpha_{p_1}\ \texttt{a})\ \ldots(\alpha_{p_n}\ \texttt{a}))) = (\beta_{\sigma_c}\ (\phi_{c_1}\ (\alpha_{p_1}\ \texttt{a}))\ \ldots(\phi_{c_n}\ (\alpha_{p_n}\ \texttt{a})))$$

Now, we know that, by our induction argument

$$(\beta_{\sigma_c}\ (\phi_{c_1}\ (\alpha_{p_1}\ \texttt{a}))\ \ldots(\phi_{c_n}\ (\alpha_{p_n}\ \texttt{a}))) = (\beta_{\sigma_c}\ (\psi_{c_1}\ (\alpha_{p_1}\ \texttt{a}))\ \ldots(\psi_{c_n}\ (\alpha_{p_n}\ \texttt{a})))$$

whence for this line, and therefore every line and the entire function $(\psi_1\ \texttt{a}) = (\phi_1\ \texttt{a})$. So by induction, $\psi = \phi$ and thus ψ is a homomorphism. □

Now by adding one extra condition, we may prove that the automated coercion algorithm generates coercions in the sense of definition 5.5.2.

Corollary 7.6.8 *Let $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$ be an algorithmically reconstructible theory which is a protecting extension of the theory $\langle\langle\Xi, X\rangle, \mathcal{X}\rangle$ such that $\Sigma^C \subseteq \ddot{\Xi}$.*

Let $\langle B, \beta\rangle$ be a model for $\langle\langle\Xi, X\rangle, \mathcal{X}\rangle$ and $\langle A, \alpha\rangle$ be a model for $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$ such that there does not exist any extension of $\langle\langle\Xi, X\rangle, \mathcal{X}\rangle$ which both $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ model.

Then the function ψ given in algorithm 7.4.1 is a coercion $\langle A, \alpha\rangle \to \langle B, \beta\rangle$.

Proof. No other theories extend $\langle\langle\Xi, X\rangle, \mathcal{X}\rangle$ which both $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ model so the only thing which could stop ψ being a coercion would be for there to exist a theory which both $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ were to model which was not extended by $\langle\langle\Xi, X\rangle, \mathcal{X}\rangle$.

So, suppose (for a contradiction) $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ were to model a theory Ω which $\langle\langle\Xi, X\rangle, \mathcal{X}\rangle$ does not extend, then we may manufacture a new theory containing all the sorts, operator symbols and equations of both theories.

We may have some duplication of sorts and operators, so this manufacturing process would need to be performed intelligently. Explicitly, Ω and $\langle\langle\Xi, X\rangle, \mathcal{X}\rangle$ may both be extensions of some theory Θ. We might (but not always[1]) only wish our manufactured theory to contain the sorts, operator symbols and equations of Θ once, not twice.

This new theory would clearly extend $\langle\langle\Xi, X\rangle, \mathcal{X}\rangle$ and $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ would both model this new theory. Hence we have a contradiction and such an Ω can not exist.

So $\langle\langle\Xi, X\rangle, \mathcal{X}\rangle$ must specify the unique "smallest" variety to which both $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ belong. We already know by theorem 7.6.7 that ψ is a Ξ-homomorphism; by definition 5.5.2 it must be a coercion. □

This is why algorithm 7.4.1 looks for $\langle\langle\Delta, D\rangle, \mathcal{D}\rangle$, since it must specify the smallest variety to which both $\langle A, \alpha\rangle$ and $\langle B, \beta\rangle$ belong.

7.7 Conclusion

In conclusion, we have shown that it is possible to create a homomorphism between two types from an important subset of types.

Moreover, this homomorphism may be constructed algorithmically and hence the general construction of homomorphisms is implementable on a computer.

Furthermore, if the type system adheres to all the assumptions made, and the homomorphism is from the theory which specifies the smallest variety to which both types belong, then it is the unique *coercion* between the two types.

[1]The theory of rings inherits from the theory of monoids twice.

Chapter 8

Implementation details

8.1 Introduction

In this chapter we shall discuss the implementation details of the automated coercion algorithm. Axiom (version 2.0a) was chosen as the computer algebra system for this work. The only other options would have been:

1. Magma;
2. writing an entirely new computer algebra system from scratch;
3. extending another computer algebra system (eg. Reduce).

These were the only options since a computer algebra system with a notion of types, and types of types, falling either into a category theory-like framework or a universal algebra set up was required.

Axiom and Magma are the only two systems which are currently based on these sound principles, and it was thought that either extending Reduce to be Axiom-esque or writing a new (order-sorted) algebra system were beyond the scope of the project. Since the source code for Axiom 2.0a was made available (by NAG) to the author, this was the soundest choice.

8.2 Boot and Axiom

Versions 1.0 to 2.0a of Axiom were based on Austin Kyoto Common Lisp (AKCL). Previous versions (of Scratchpad) were built on top of other Lisps, and since AKCL is now GNU Common Lisp (GCL) and covered by the GNU Public Licence (GPL), future versions of Axiom (including the current version, 2.1) are to be based on Codemist Common Lisp (CCL).

This has had a useful side effect. CCL is a far smaller, faster and more modern implementation of Common Lisp. This means that Axiom's footprint is now far smaller than it has been in the past.

Axiom has been said to have been "born big" — it initially only ran on large IBM mainframes. With the advent of CCL based Axiom it now runs on the humble PC. This is true whether the PC is running an operating system like Linux or whether it is running Windows 95 or NT.

This research was performed on Axiom 2.0a, which is AKCL-based. The hardware used was usually an IBM RS6000 (running AIX 3.2.5) and occasionally a Sparc Server 1000 (running Solaris 2.5.1).

Axiom is built in the following manner. Boot is a syntactically-sugared, interpreted (or compiled) form of Lisp. Boot retains all the functionality of AKCL, with a lot more thrown in. A lot of these extra features are there for historical reasons, for example, there are plenty of functions to allow code written in VM-Lisp to continue to run. Some are there to genuinely extend the usability of the language.

The Axiom interpreter is written in Boot, and hence, any alteration to the Axiom world must be written in Boot.

Compiled libraries of code (`Packages`, `Domains`, `Categories`) are either written in Spad or Aldor. These are the strict languages which define the types, algebras and theories of Axiom. They may be compiled and loaded into the Axiom interpreter.

The Axiom interpreter itself contains many features not available to the `Package`, etc. writer. In both Spad and Aldor, strong types are the order of the day. The interpreter is meant to be a bit more user-friendly. Hence, type inference and on-the-fly coercions are available to the user in the interpreter

So for this research, the code had to be written in Boot.

8.3 The top level

The basic design was as follows. To perform a coercion from a `Domain A` to a `Domain B`, one needs to be able to ascertain which constants, queries, constructors and part functions are available to both `A` and `B`. These functions should all come from the same algorithmically reconstructible theory (`Category`).

The automated coercion function should be created and then compiled and stored away ready for fast access in case of future use. Such items are called "cached lambdas" or "clams", for short, and are already part of the Axiom system. Thus this was implemented.

The automated coercion function should be integrated with the current coercion mechanism in Axiom. This was done by inserting the relevant line at the correct point of the Boot function, `coerceInteractive`.

8.4 Labelling operators

We had to decide on a mechanism to mark which operators were constructors, part functions, etc.

One method would have been to create sub-types of the Axiom `Domain`, `Mapping`, which would have been infeasible, requiring much rewriting of the Spad (and Aldor) compilers. This is also true of the method given in a later section.

A Spad `Domain` contains the following parts:

1. the name of the type constructor;
2. a (variable name, `Category`) pair[1] for each[2] of the parameters of the type constructor;
3. a `Category` or `Join` (intersection) of `Categories` to which the `Domain` belongs;
4. a list of additional operator symbols (functions) for the type;
5. a list of methods for (some of or all) the operator symbols of the type.

Every operator symbol definition in a `Domain` or `Category` (and hence available in a `Domain` of that `Category`) has a certain number of special comments called `++` comments.

These `++` comments (as opposed to ordinary comments in most languages and Axiom's other style of comments, `--` comments) are parsed by the compiler to produce the

[1]Occasionally a (variable name, `Domain`) pair. For example, `IntegerMod(p:PositiveInteger)`.

[2]This is not the same as all the sorts of the signature. One may declare extra unused sorts. Also we may declare a function where the source or the target may contain a ground type. For example, functions which return a boolean-like type are usually declared to return `Boolean` rather than a pair where the `Category` is the `Category` of boolean-like types.

documentation for Axiom's sophisticated on-line help system, HyperDoc. (HyperDoc was one of the first hypertext systems.)

This parsing of the `++` comments allowed us to enter special keywords for each special function which were read and stored at compile time for each `Domain`. We could then read them at run-time using the Axiom interpreter's built in `++` comments reader.

8.5 Getting information from domains

The `++` comments were gathered for a `Domain` by recursion up the `Category` inheritance lattice, using the `GETDATABASE` function. We asked the database for all the documentation for each `Category`, and then asking which `Categories` it extended. The documentation includes all the functions and their respective `++` comments. This allowed the automated searching for keywords, and hence, the special functions of a `Domain`.

Note that we only needed the comments for functions (operator symbols) declared in a `Category`. Functions declared in `Domains` are of no interest to the automated coercion algorithm. This is because these do not correspond to any of the special functions (constructors, part functions or queries) of a signature or theory.

The keywords were checked quite simply, being members of the following list:

```
list('"constant",'"constructor",'"part",'"query")
```

Part number checking was only slightly more difficult with each number having to be converted from a (Lisp) string to a (Lisp) integer. A restriction was placed on queries that they must be called the same as their associated constant (or constructor), but with a "?" appended. This restriction was not necessary and could have been worked around easily.

The special function lists from both `Domains` were compared and the required functions were extracted from both.

8.6 Checking information from domains

For each special function, a check was performed to see whether that function was really available in the `Domain`.

The envisaged problem was that Axiom's `Categories` can be conditional: that is to say, some function definitions only exist if certain relationships hold for the `Category`'s parameters.

`GETDATABASE` ignores the parameters to a `Category` and returns all the functions which may be exported by a `Domain` of that `Category`. Hence, the extra check was necessary to ensure that the automated coercion function did not try to include these unavailable functions.

Thus the homomorphism created by the automated coercion algorithm was always equivalent to definition of a conditional homomorphism given in section 5.3.

8.7 Flaws in the implementation

The implementation was originally envisaged as a fully working piece of code of production standard and hence shippable with a commercial release of Axiom. For various reasons outlined below, only a working prototype was implemented.

Deliberate restrictions placed by the author on the design were:

1. Query function names were restricted to be the name of their associated constant function with a "?" appended.
2. The number of (non 0-ary) constructors was limited to one.
3. Recursion through `coerceInteractive` was achieved using naïve means.
4. Neither the existence of $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$ nor $\langle\langle\Delta, D\rangle, \mathcal{D}\rangle$ were checked. (See algorithm 7.4.1 for the meaning of these).

Both the first two items could have easily been worked around but would have required a more complex `++` comment reader. This was considered to be a minor detail, which would have required too much implementation time.

The third item was more problematic. Attempts were made to remedy this situation. However, the depth of knowledge needed to implement this correctly would have required too much time to learn. This was the main factor in the downgrading of the implementation to mere prototype status.

It was a certain amount of naïvety on the author's part to assume that a production standard implementation of the automated coercion Boot package was within the scope of this project.

The method to work around the complexities of Axiom's evaluation loop was to add a (Spad) `Package` which exported a function which then called the Boot function. The code for this `Package` (called `NCoerce`) follows.

```
--% Coercion Package
)abbrev package NCOERCE NCoerce

NCoerce(Source : Type, Target : Type) : Exports == Implementation where

  Exports ==> with

    nCoerce: Source -> Target

  Implementation ==> add

    nCoerce(x : Source) : Target ==
      nCreateCoerce(x,Source,Target)$Lisp
```

To call a function from a `Package` is relatively easy in Boot, and uses the same syntax as calling a function from a `Domain`.

The function (`nCoerce` above) then uses the Axiom `$Lisp` syntax for calling a function defined in Boot. This "breaking out" of the Boot code to call a Spad function (which then in turn calls Boot code) is most undesirable.

However, should an expert in the Axiom interpreter be given the Boot code for the automated coercion algorithm, then it is merely a matter of changing one line. This one line currently calls a function which creates lines of the form

```
nCoerce(x)$NCoerce(Source,Target)
```

This somehow needs to be changed to something which performs

```
coerce(x)@Target
```

`@` is the Axiom syntax meaning "the previous function should return the following type". In this case, "`coerce x` to the type `Target`".

The automated coercion Boot package was only implemented to deal with Spad code and not Aldor code. This is because Boot handles code written in the two different languages in a different way. There are different interfaces to the two internal representations of a `Domain` (or a `Category` or `Package`).

This, again, could have been remedied, but would have required too much time compared with the prospective gain in functionality.

Occasionally, Axiom's modemap selection can get confused. Modemap selection is Axiom's method for selecting methods from an overloaded operator name. This causes the automated coercion package to fall over at a premature stage.

The fourth item is a bit more difficult to solve. The existence of $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$ is implicit if a `Domain` has special functions. It would also be possible to check if both source and target are both members of a common `Category` (using `GETDATABASE`). It would even be possible to check that all the constructors of the source are from this `Category`. Finally, it would be possible to ensure that this is the smallest such `Category`.

However, it would be most problematical to determine whether $\langle\langle\Sigma, S\rangle, \mathcal{S}\rangle$ is a protecting extension of $\langle\langle\Delta, D\rangle, \mathcal{D}\rangle$ or not.

This is a more general problem than one which just applies to the automated coercion algorithm. This is due to the fact that Axiom believes everything that you say. One may produce a `Category` and claim that it extends another. Yet if Axiom's `Categories` were really varieties, specified by theories there would need to be some way of checking whether the equations in the original `Category` still held in the new one. Similarly, `Domains` need not model the theory specifying the variety (`Category`) to which they have been declared to belong.

The current situation in automated theorem proving means that Axiom will believe any false assertions like those mentioned in the previous paragraph. This is not a design flaw of either Axiom or the automated coercion algorithm; neither Axiom's designers nor I had any choice — automated theorem proving has not yet advanced sufficiently for us to take advantage of it. Similarly (and moreover) Axiom will believe that any function (which one may implement) called `coerce` is a valid coercion.

This means that in this implementation of the automated coercion algorithm in the current release of Axiom, it would be possible to create new "coercions" between (potentially) unrelated types or non-homomorphically between similar types. Using the example from section 7.5 we could create a "coercion" $\mathbf{Z}_5 \to \mathbf{Z}_3$.

The present state of automated theorem proving technology with special reference to Axiom is detailed in [MS97]. This paper details recent research and near-future directions for the subject.

Other flaws which shall be dealt with in chapter 9 include:

1. Some Axiom `Categories` do not agree with our notion of varieties (specified by signatures or theories). (Sections 9.2, 9.4, 9.5.)

2. Axiom `Domains` have too much in common with their `Categories`. There is a lack of distinction between operator symbols and operators. Similarly between sorts and carriers. (Sections 9.3, 9.6.)

8.8 Conclusion

We conclude that the automated coercion algorithm is probably implementable in the Axiom interpreter. It is certainly possible to create a version which works. However, it would take someone with deeper knowledge of the code to force a complete integration of the current implementation with the Axiom interpreter.

What we do have is a proof of concept. If `Category` authors correctly label their operators, then the coercion function does the right thing. It is only the perfect integration into the Axiom's kernel that has not been demonstrated.

Chapter 9

Making Axiom algebraically correct

9.1 Introduction

In this chapter we shall discuss various details which need to be changed to make a computer algebra system (in this case, Axiom) more "algebraically correct". By this, we mean, so that Axiom's `Categories` behave more like our concept of varieties (specified by order-sorted theories) and that the `Domains` act more like our notion of order-sorted algebras more correctly.

9.2 Explicitly defined theories

In Axiom, the `Categories` were originally intended to be akin to varieties specified by signatures. The `Domains` were then meant to be like algebras of those signatures. This seems, in retrospect, to have been an over-ambitious aim.

The `Categories` do behave and look to the casual observer like they were specified by multi-sorted algebraic type theories, but there are some clear differences.

9.3 Operator symbols and names

In Axiom, if a category defines an operator symbol $\sigma_{n,q,s}$ then for all domains in that category, that operator name $\alpha_{\sigma_{n,q,s}}$ of $\sigma_{n,q,s}$ will be $\sigma_{n,q,s}$.

It may appear to be not that much of a disadvantage to have such a restriction on operator names, but an important case in mind is that of the monoid. A ring, has two binary operators over which it forms a monoid. The only way in Axiom that we can ensure this is to have two different monoidal categories. A distinct disadvantage.

If we allowed operator symbols to differ from operator names then another `Category` could extend `Monoid` in two different ways.

Syntactically this would be most difficult. We would need a way of declaring the two operator symbols in `Ring` (traditionally `+` and `*`) as being from different operator symbols in `Group` (which extends `Monoid`) and `Monoid`.

For example we could write something like the following

```
Ring : Category == with ( M : Monoid, G : Group )

  -- Operators from Monoid
  * : ( * : (%,%) -> % )$M
  1 : ( 1 : () -> % )$M

  -- Operators from Group
  + : ( * : (%,%) -> % )$G
  0 : ( 1 : () -> % )$G
  inv : ( inv : % -> % )$G
...
```

This would identify `*` and `1` with the binary operator and identity element, respectively, from `Monoid`. Whereas `+`, `0` and `inv` would be identified with the binary operator, identity element and inverse function from `Group`.

Notice that both the operator symbol (eg. `*`) and both the source arity (eg. `(%,%)`) and target (eg. `%`) sort are needed to uniquely identify the operator. These correspond to σ, q and s respectively. (We do not need to know n since this is deducible from q).

A function in Axiom (in either a `Domain` or `Category`) is currently represented by `(name (target source) comments)`. Function declarations in `Categories` are normally of the form[1]

$$(\sigma\ (s\ q)\ \texttt{comments})$$

[1] s and q are reversed in Axiom's internal representation of a function.

and once exported by a ground type (ie. genuine Axiom `Domain` with all parameters to the `Domain` fixed) it becomes

$$(\alpha_{n,q,s,\sigma}\ \ (A^s\ \ A^q)\ \ \texttt{comments})$$

where $\alpha_{n,q,s,\sigma} = \sigma$.

For the coercion algorithm, we inserted special words in the comments field to try and mimic an operator symbol.

Another approach might have been to declare other types eg. `Constructor(a,b)` which are sub-types of `Mapping(a,b)`.

A far better approach is to extend the information contained within the function construct. For initial definitions in `Categories` — that is new operators which have come from this `Category`, not ones which this `Category` may extend — then the current declaration method suffices.

However, as in the above `Ring` example for `Categories` which extend others then a better method may be,

$$(\sigma'\ \ (\sigma\ \ (s\ \ q)\ \ \Sigma)\ \ \texttt{comments})$$

where σ' is the new operator symbol corresponding to σ a member of $\Sigma_{n,q,s}$ where this signature extends Σ.

We have used variable names to avoid confusion. One may wish to define a `Category` called `DoubleMonoid` where `G` is a `Monoid` instead of a `Group`. (Obviously no `inv` operator would be available to such a `Category`.)

The same terminology could also be used to define specific algebras (`Domains`). For example, for all n in $\mathbf{N}$, $n\mathbf{Z}$ (whose elements is the set $\{nz \mid z \in \mathbf{Z}\}$) is an additive group, whereas $S(n)$ (the symmetric group on n elements) is usually considered to be a multiplicative group.

So, for $n\mathbf{Z}$ we might write:

```
IntegersTimes( n : PositiveInteger) : Group ==

  + : ( * : (%,%) -> % )
  0 : ( 1 : () -> % )
  - : ( inv : % -> % )
...
```

However for $S(n)$ we might write:

```
SymmetricGroup( n : PositiveInteger ) : Group ==

  * : ( * : (%,%) -> % )
  () : ( 1 : () -> % )
  inv : ( inv : % -> % )
...
```

In this example we can see the following proposed methodology for declaring operator names which correspond to operator symbols.

$$(\alpha_{n,q,s,\sigma}\ (\sigma\ (s\ q))\ \texttt{comments})$$

The signature Σ does not need to be mentioned in each function definition since the algebra is only declared to be a model of one signature.

This does create more "unnecessary" confusion for `Domain/Category` writers, but correctness should overrule ease-of-use. It is also conceivable that this will have a negative effect on the amount of time it takes to compile a `Domain`. However, this is true of all type-checking compilers.

More importantly, it should not have any effect on run-time speed of execution. This should definitely be the case if the internal representation is moved to something approximating Axiom's current order.

$$(\alpha_{n,q,s,\sigma}\ (s\ q)\ \sigma\ \texttt{comments})$$

This puts the target and source closer (and less operations away) from being discovered.

Clearly, in this proposed methodology the carriers of the source and target are not mentioned explicitly in the function representation.

To speed up function-type look-ups (modemap selection) each `Domain` could provide a hash of sorts and carriers. Indeed, then the internal representation could be

$$(\alpha_{n,q,s,\sigma}\ (A^s\ A^q)\ \sigma\ \texttt{comments})$$

and then modemap selection would be as fast as in the current version of Axiom, but the hash could be used (going in the other direction) to determine s and q for the automated coercion algorithm.

9.4 Moving certain operators

Another factor which stops Axiom acting totally homomorphically, is that certain operators appear in `Categories` too far "up" the inheritance lattice. This is best illustrated by an example.

In Axiom, one often would like to be able to convert `Lists` to `Sets`. (The `Domain` whose items are finite sets (or actually, sometimes classes), over some particular type is called `Set` in Axiom.)

`Sets` and `Lists` in Axiom are both certainly finite collections which can be built by adding in another element at a time. In `List` this may be achieved using either `cons` or `append` (though obviously, `cons` is far more efficient) whereas `Sets` can be built using the (sometimes non-effective) command `insert`[2].

Axiom knows that `Lists` are sorted whereas `Sets` are not. The problem, which is immediately obvious, is that adding in a new element to a `List` will always increase the length of the `List`. This is not true of (Axiom's finite) `Sets`. They both however get the same element-counting-operator from the same `Category`. This function, `#` comes from the the `Category`, `Aggregate`, which is the most general type of "collection" in Axiom.

`List(S)` (for some fixed `S`) is a model of `ListAggregate` and `Set(S)` is a model of `FiniteSetAggregate`. Both of these theories are extensions of the theory `Aggregate`.

Therefore if we had a homomorphism, ϕ from `Lists` to `Sets`, then the following equation should hold, yet it clearly does not,

$$\phi(\texttt{\#([1,1])}) = \texttt{\#}(\phi(\texttt{[1,1]}))$$

since the left hand side is,

$$\phi(2) = \texttt{2}$$

and the right hand side is,

$$\texttt{\#}(\{\texttt{1}\}) = \texttt{1}$$

and unless the carrier sort of collection length is not `NonNegativeInteger` but a different type, one in which all elements are equal, we will not be able to create any homomorphisms from `List` to `Set`.

[2]This operation inserts a (potentially) new element into a set. $\texttt{insert}(x, s) = s \cup \{x\} = s; x$

In section 7.3 we demanded that if the automated coercion algorithm was to be applicable (in this case) from `List(S)` to `Set(S)` then `ListAggregate` would need to be a protecting extension of `Aggregate`[3].

It is clear that in `ListAggregate` the following equation holds

$$\texttt{\#(cons(a,b))} = \texttt{1+\#(b)}$$

If this were a protecting extension then this equation would hold in `Aggregate`. However, as we have already observed

$$\texttt{\#(insert(a,b))} \neq \texttt{1+\#(b)}$$

when $a \in b$.

Thus `ListAggregate` is not a protecting extension of `Aggregate` and the automated coercion algorithm can not be applied.

There are three obvious solutions:

1. Disallow coercions from `List` to `Set`. Although we would still allow a non-homomorphic `convert` operator;
2. Move the element-counting-operation further down the `Category` inheritance lattice until it does not appear in any `Categories` to which *both* `List` and `Set` belong. Or if they do both belong to this `Category`, ensure that none of the part functions, constants, queries or constructors are from this `Category` or any of its ancestors.
3. Move the adding-new-element-operation further down the lattice, in a similar manner. Thus `cons` and `insert` would not know anything about each other. In this case the automated coercion algorithm would still not be applicable.

This is merely one example of many such operators which could require moving.

9.5 Retyping certain sorts

As in section 9.4, this is best illustrated by an example.

In Axiom, the `Category`, `Ring` exports a function,

[3] Or an extension of `Aggregate` which both `ListAggregate` and `FiniteSetAggregate` extends.

```
characteristic:  () -> NonNegativeInteger
```

(remembering that Axiom does not differentiate between sorts and carriers). So imagine the natural ring-epimorphism $\phi : \mathbf{Z}_6 \to \mathbf{Z}_2$. Then the following equation should hold,

$$\texttt{characteristic}(\phi()) = \phi(\texttt{characteristic}())$$

yet this is clearly is not the case, since ϕ must send the `Void` sort to `Void`, and thus the left hand side must equal the characteristic of $\mathbf{Z}_2$, which is 2. Whereas, for the right hand side, we have $\phi(6)$. This ϕ is the natural map from $\mathbf{N} \cup \{0\}$ to itself. This is, of course, the identity map, and hence the right hand side has the value 6.

There is clearly something very wrong here. There are two solutions, the first of which is highly unsatisfactory:

1. Stop `characteristic` being a function. One could persuade it to be an attribute of the `Ring`;
2. Alter the carrier of the return type of `characteristic` from `NonNegativeInteger` to being a type with the same elements, but a different idea of equality.

This is one of many such cases in Axiom.

9.6 Sorts and their order

We have been discussing Axiom's `Category` inheritance system as if it were a true attempt at modelling order-sorted algebra. There are, however, two areas which are distinctly missing from the Axiom model; these are, the sorts and their order.

There is a distinct confusion in Axiom between sorts and carriers. The author believes that all `Category` definitions (the signatures or theories which specify varieties) should not use genuine types at any point in the signatures, or any other point. These should only occur in the `Domains` of that `Category`.

For example, many `Categories` assume that the only boolean-like type is `Boolean`. Similarly the types `NonNegativeInteger`, `PositiveInteger` and `Integer` are often "assumed" and are not parameters of a type.

For the types `Boolean`, `NonNegativeInteger` and `PositiveInteger` this is not normally a complete disadvantage. There are not likely to be other algebras in our system which behave similarly enough to replace these types.

However, there are coercions between `NonNegativeInteger`, `PositiveInteger` and `Integer` (in the obvious directions). This then imposes some order on the sorts of the type which is not explicitly mentioned.

Some of the sorts of an algebra are defined. `%` is always the principal sort, and any parameters of the `Category` are there, too. Others, however, are merely mentioned in function prototypes (signatures). All should be listed in a sort-list (and/or sort-lattice, see below).

Neglected sorts normally include those which are carried by the four types mentioned above. However, other types are often neglected too. These include `List` which is often present so that elements of the type may be constructed. (The underlying representation of most types in a Lisp-based system are often lists). More seriously, particular polynomial representations are sometimes present.

There is no real mechanism built into Axiom to order the sorts of an signature (`Category`). The order is implicit in `Categorical` inheritances, such as `CoercibleTo` and `RetractibleTo`, which give information on how the principal sort relates to some other sorts.

I believe that a more sensible way would be to declare a lattice like arrangement with the declaration of a `Category`. Indeed, this would then make the sort-list clear, too, since this is never explicitly defined either.

For example, we could add the syntax, `SortOrder` to be used as follows,

```
ACategory(a:A,b:B,...) : Category ==

  with SortOrder{
    a < %;
    % < b;
    ...
  }

  exportedFunction : List A -> %;
...
```

or (better), `SortOrder` could be a `Category` which could be defined as (assuming this is compilable[4]),

[4]According to [Bro97] "One problem is that it may not be possible for the compiler to figure out exactly what this construct will export" but it might still be implementable.

```
SortOrder(ls : List Pair Type) : Category == {

   for pr in ls repeat{
     coerce : first pr -> second pr;
   }
}
```

This `for` loop would not build the entire sort lattice for any particular category. This would be better done by the compiler at compile time, on a per-`Category` or per-`Type` basis.

Thus, any `Category` which does not extend `SortOrder` would be a non-order-sorted signature.

9.7 Altering Axiom's databases

Axiom has many built in databases for looking up comments, functions, attributes, etc. from each `Category` or `Domain`. These are usually text files, with character number keys for faster cross-referencing.

Database and hash-table technology has certainly improved in vast amounts over recent years, and is in fact, quite an evolving area of computing. Magma utilises many different, hand-written (in C) databases for the large number of pre-computed tables of facts, needed by the modern discrete mathematician.

Each Magma database utilises the fastest look-up method available to the authors for that particular task. While Axiom's method is not excessively slow, it is also not particularly fast.

More importantly, from our point of view, Axiom could have some extra databases to aid the automated coercion algorithm. Specifically, there could be a database containing for each `Category`, a list (of lists) of special functions exported by that algebra, but not its ancestors. This could just be also be a compiled fact about each algebra, but the look-up overhead would be greater.

The restriction on only having functions from that `Category` and not its ancestors' special functions allows for the dynamic nature of Axiom's `Category` system. Without this restriction, altering a `Category` further up the lattice could cause the database information to become outdated for all of its descendants.

9.8 Conclusion

Axiom behaves very similarly to a language based on order sorted theories and the algebras that model them. However, there are some key areas in which Axiom differs to the mathematical notions.

We summarise these areas in the following table.

Mathematical Notion	**Axiom**
Operator names need not be the same as operator symbols	Operator names are always the same as operator symbols
Coercions are homomorphisms and hence act homomorphically on all operators	Coercions need not act like homomorphisms at all. (If it were possible to implement a universal equation-checker then Axiom would be alright)
A signature depends on its sort-list, and hence a sort-list is part of its definition	`Category` definitions do not explicitly list their sorts
No algebras are mentioned in a signature definition, only sorts	`Category` definitions do depend on specific `Domains`
The order on the sorts is part of the definition of a signature	`Category` definitions do not explicitly order their sorts

We have presented methods for addressing all of these differences.

Chapter 10

Conclusions

10.1 Introduction

In this chapter we shall summarise the work we have done and what problems we have managed to overcome. We shall also say what future work may be performed to extend the ideas presented here.

10.2 Summary of work done

In this section we shall summarise the work done in this thesis and what problems these ideas overcome. We shall split this down into the following sections.

1. Category theory and order sorted algebra bases for computer algebra systems. Section 10.2.1.
2. Representation and syntax of order sorted algebra. Section 10.2.2
3. Coherence of a type system. Section 10.2.3
4. The automated coercion algorithm. Section 10.2.4.

10.2.1 Category theory and order sorted algebras as the bases for sound strongly typed computer algebra systems

We have shown why both category theory and order sorted algebra both provide solid models for the types systems found in computer algebra.

In sections 3.3, 3.3, and 3.4 we have demonstrated the correlation between a computer algebra type system and category theory. (See also appendix B.5).

In sections 4.7 and 5.2 we have shown how a computer algebra type system correlates with order sorted algebra. In section 5.3 we extended the notion of order sorted signatures to cover conditional signatures which occur in Axiom.

We have mentioned some of the correspondence between category theory and order sorted algebra (section 5.4). We have usually used order sorted algebra as our model. This is because order sorted algebra more readily corresponds to the algebraic inheritance mechanism. This is mainly due to the order on the sorts which is not available in category theory. Also category theory has more difficulty expressing higher order polymorphism.

10.2.2 Representation and syntax issues of an order sorted algebra based type system

In chapter 9 we demonstrated various methodologies for extending Axiom's current type system so that it may more closely model order sorted signatures and algebra.

In section 9.3 we showed how Axiom's syntax may be extended so that a signature may extend another more than once[1]. We also showed that this syntax could be used to uniquely identify operator names with operator symbols. This then allows operator symbols to differ from operator names.

In section 9.6 we discussed how both the sort list and the order on the sorts could be introduced to Axiom's signatures.

Sections 9.4 and 9.5 commented on how the Axiom signature tree may be altered to allow coercions to act more homomorphically.

Finally, we discussed how Axiom signatures could be compiled to contain extra information which would enhance the speed of the automated coercion algorithm (section 9.7).

10.2.3 On coherence

In chapter 6 we first stated all the mathematics used by Weber to state and "prove" Weber's coherence conjecture 6.3.7 (sections 6.2 and 6.3).

In section 6.4 we then showed that this "proof" is not correct. We showed that altering a definition and adding a couple of new assumptions that let us prove the coercion

[1] Thus this solves an interesting class of multiple inheritance problem.

theorem 6.4.7.

In section 6.5 we then showed that it was possible to relax one of the assumptions which Weber made, and still have a coherent type system. (The extended coherence theorem 6.5.4.)

10.2.4 The automated coercion algorithm

In sections 7.2, 7.3 and 7.6 we provided enough mathematics to show that the automated coercion algorithm 7.4.1 (section 7.4) is an algorithm which returns a function which is not only a homomorphism, but a coercion which we defined in definition 5.5.2.

This homomorphism is unique in a coherent type system, which we can guarantee providing we can satisfy all the assumptions of the extended coercion theorem 6.5.4. The homomorphism created may be built from the four basic types of coercion given in the statement of that theorem.

In chapter 8 we showed that a demonstration implementation of this algorithm is possible in Axiom.

10.3 Future work and extensions

We have presented in this thesis the basis for a mathematically sound computer algebra system. We have also shown that it would be possible to implement the automated coercion algorithm in such a system.

At present there is no such system, though Axiom and Magma come close.

We have shown how Axiom's syntax could be extended to allow it to model an order sorted algebra system. the author would be most interested in implementing either a future release of Axiom or maybe a freeware Axiom-like system.

To implement a totally new Axiom-like system, it would be nice to take advantage of a language which already supports objects. Thus such a system could be written in C++ [Str97], youtoo [Kin96] (an implementation of EuLisp Level 1 [Pad95]) or Java [Gra97].

However, note that Java does not allow multiple inheritance amongst classes. It does allow multiple inheritance from interfaces though. Interfaces are always (effectively) abstract classes.

Thus a Java implementation of an Axiom-like computer algebra system would have all

`Domains` as classes and all `Categories` as interfaces.

Axiom's categories are not always abstract - that is they may contain implementations of functions[2]. This is a good thing as it enforces certain truths about a particular function.

Forcing any important facts by making a class which implements a particular interface would then ruin any chance of reattaining multiple inheritance. This effectively means that Java would not make a sensible implementation language.

A more serious omission of Java is that functions are not first class objects. This means that the automated coercion algorithm would be unimplementable in an algebra system with Java classes for objects.

C++ is a tempting choice, but the time for one person to implement the entire system in such a low level language might make it infeasible. (The source code for the Axiom interpreter alone takes up over 100,000 lines of Boot and Lisp code).

Youtoo is an implementation of EuLisp Level 1 which does allow some forms of multiple inheritance. So while the temptation of using a fashionable language (Java) with potentially many more users is there, to implement the new algebra system *correctly* we see that Lisp is still the sensible choice.

Access to OBJ and the Magma source code would also prove interesting. The author would like to see if the automated coercion algorithm is easily implementable for any system other than Axiom, which provided all the original inspiration.

Sticking with the current release of Axiom, it would be nice to spend a few hours with one of the few experts who truly understand the Axiom interpreter. This would undoubtedly lead to a successful integration of the automated coercion algorithm with Axiom.

However, this would be only available in the interpreter. An implementation inside the Aldor compiler would be far more difficult, and hence make a very interesting research project.

Future research by Martin [MS97] and research students of Davenport should ensure that Axiom's theory lattice is *guaranteed* to be mathematically correct. Eventually, we may even be able to prove that one theory is a protecting extension of another (though this may be some years from now).

Until such research is completed, the accuracy of the automated coercion algorithm depends on the ability of `Category` and `Domain` authors to adhere to to the assumptions

[2]The "not equals" function is defined to be "not(equals( ))".

made in this thesis.

Appendix A

Extra category theory

A.1 Introduction

This appendix merely lists some extra category theory needed by appendix B.

A.2 Extra definitions

Definition A.2.1 *The* constant functor Δc *sends all objects of J to c and all arrows to 1_c.*

Definition A.2.2 *The* diagonal functor $\Delta : C \to C^J$ *sends each object $c \mapsto \Delta c$ the constant functor all arrows $f : c \to c'$ of C to the natural transformation $\Delta f : \Delta c \to \Delta c'$. A limit for $F : J \to C$ is a universal arrow $\langle r, u \rangle$ from Δ to F.*

Definition A.2.3 *A category is* complete *if given any small category J, and any functor $F : J \to C$, then F has a limit.*

Appendix B

Set theory

B.1 Introduction

The work in this section and sections B.3, B.4 and B.5 is not strictly necessary for the rest of this thesis, however, the author believes that it is necessary if one wishes to have a strict foundation to one's computer algebra system.

I also believe this work to be necessary if one wishes to form categories of categories in Axiom. (See section B.5.)

B.2 Basics

We need not go into a precise axiomatisation of set theory (or indeed class theory) in this thesis, but merely point the reader towards some suitable references: [Dev79] for Zermelo-Fraenkel set theory; [Ber91] for von Neumann-Bernays; and [BHFL73] for a deep exposition on many formulations of set theory.

Mac Lane [ML71] avoids this by letting categories only form over small and large sets (also known as sets and classes in von Neumann-Bernays set theory). Briefly, to formulate set theory in this fashion, one takes naïve set theory and appends the existence of a fixed set U, called the *Universe*. The universe has the following properties,

1. $x \in u \in U \Rightarrow x \in U$
2. $u, v \in U \Rightarrow \{u, v\}, \langle u, v \rangle, u \times v \in U$
3. $x \in U \Rightarrow \mathcal{P}(x), \cup_{y \in x} y \in U$

4. the set of all finite ordinals is in U

5. if $f : a \to b$ is a surjective function with $a \in U$ and $b \subset U$, then $b \in U$

One then defines a set to be a *small set* iff it is a member of U.

Thus he refers to the "category" with all categories as objects, and all functors as arrows, as the *meta-category of all categories*. Meta-categories are (in his book) are what would be called categories, if only their objects and arrows formed (small or large) sets.

Mac Lane's book being the definitive volume on category theory also refers to other formulations of set theory, and even mentions (in passing) Lawvere's paper [Law64]. In that paper[1], Lawvere formulates eight axioms for categories and proves that if one has a complete category (definition A.2.3) satisfying those axioms, then it is equivalent to the category of sets and mappings and moreover, is *unique*.

Lawvere continues further down this abstract path in his 1965 paper [Law65] which attempts to axiomatise all of mathematics without recourse to any form of set theory.

Herrlich and Strecker [HS73] use von Neumann-Bernays to overcome the difficulties of having categories and meta-categories. They consider (as we have) that a category may be defined over any "collection" of objects and arrows.

von Neumann-Bernays set theory introduces the notion of *class* in an attempt to overcome the problems associated with trying to form the set of all sets. Any collection of sets is a class, but if A is a *proper class* (that is a class which is not representable as a set) then there exists no class which has A as a member.

It is for this reason that the authors of [HS73] introduce the notion of a *conglomerate*. They loosely define conglomerates to be "collections" of classes of conglomerates[2] and say that they require that:

1. Every class is a conglomerate;

2. Conglomerates are closed under formation of ordered pairs, unions, intersections, complements, disjoint unions and Cartesian products.

However, they also state that there is no conglomerate of all conglomerates, and thus they introduce the term *cartel* for collections of conglomerates.

[1]It should be noted that Lawvere's paper is extremely abstract in content, and that it does not even formulate a traditional form of the relationship "$\in$". Indeed, $A \subset B$ is defined to mean that there exists a monomorphism A with codomain B, and $x \in A$ is defined via $(\exists! B \supset A)(x : 1 \to B)(\exists \bar{x})(A(\bar{x}) = x)$

[2]I believe that they may only mean collections of classes, (or conglomerates representable by classes) and will take this meaning from here on.

Indeed, we see that if we define the author's concept of "inductive classes" as follows, we will be in good shape.

Definition B.2.1 (Inductive classes)

- *An entity, X is called a* class$_0$ *iff X is a set.*
- *An entity, X is called a* class$_\omega$ *(where ω is any ordinal number) iff* $(\exists\xi$ *an ordinal number*$)(\xi < \omega)($*all members of X are* class$_\xi$*s*$)$.
- $(\forall\xi, \omega$ *ordinal numbers*$)((\xi < \omega) \Rightarrow ((X$ *is a* class$_\xi) \Rightarrow (X$ *is a* class$_\omega)))$

There is no reason for the following not to hold, but the author can not prove it from the definitions alone. Thus we state it here as an axiom.

Axiom B.2.2 *A finite collection of* class$_\omega$*s is a* class$_\omega$.

So if we now define $\mathbf{Cat}_\omega$ as follows, we will always be able to find a category in which we can work, rather than a meta-category. Unless, of course, one still wishes to form a category of *all* categories, or a category with itself as an object. We are just saying that we can always find a category "large enough" in which we can work.

Many theoretical computer scientists do not believe that there really exists a problem at all. Davenport states in [Dav92],

> "many mathematical sets are not represented in computing languages quite as we would like. For example the set **Z**, [...and m]ore seriously, no computer can really represent **R**"

yet we often pretend that they can. For example, in most computer algebra systems allow the existence, calculation and manipulation of "bignums" (integers larger than the largest representable within one machine word). In a lazily evaluated system, one might even allow the creation of integers larger than the number of molecules in the universe, let alone the usable workspace of any current computer. Indeed, this is already commonplace.

So just because one's computer cannot represent every element of **Z** does not mean we cannot have a type of *all* integers.

Davenport also states in [Dav93],

> "One might think of [the universe] U as being the set of all possible data objects, or bit-patterns"

Hence, since everything with which we work is finite, we would not be required to do any more work at this point. The author disagrees, simply because one's machine contains only finitely may states does not mean that we may assume that all of our collections are (small) sets.

B.3 Inductive class approximation

In Axiom,

```
S : Set Any := { } ; S := insert(S,S)
```

defines `S` to be a finite collection, with itself as its sole element; however, it most certainly is not a (small) set. In fact, there does not exist an ordinal number[3] ω such that `S` is a class$_\omega$.

So what use is the idea of inductive classes in this situation? The method of *inductive class approximation* allows us to identify the `S`'s on each side of a defining equation with different "approximations" to the value of `S`.

To demonstrate the method of inductive class approximation, let us fix C to be a class$_\omega$. Now alter C, by letting

$$C := C \cup \{C\}$$

The operation on the right hand side is the traditional ";" function and should be familiar to set theorists from one of the first three axioms of general set theory [Ber91]. It is also used to define successors in ordinal theory.

As before, there does not exist an ordinal η such that C is now a class$_\eta$. What we may do is identify the Cs on the right hand side with a copy of the original C, and thus the C on left is now a class$_{\omega+1}$. For those (myself included) who do not like the idea of the same symbol having different values in the same equation, then here is a different version,

$$C_{\omega+1} := C_\omega \cup \{C_\omega\}$$

We now have an $\omega + 1$ approximation to C. Given any ordinal $\eta > \omega$ such that C_η is an η approximation to C, then we define $C_{\eta+1}$ the $(\eta + 1)$th approximation via,

$$C_{\eta+1} := C_\eta \cup \{C_\eta\}$$

[3]An ordinal is an ordinal number if it is a set [Men87].

Thus we can always find an approximation to C, containing a "large enough" (but different) approximation to C.

In traditional notation,

$$\mathrm{Od}(\eta) \Rightarrow C_{\eta;\eta} = C_\eta ; C_\eta$$

If we could continue inductively approximating *ad infinitum* then we would reach a limit ordinal $\mathrm{Lim}(c)$. However, $C_{\mathrm{Lim}(c)}$ may still not be equal to C.

At first glance, this appears to be neither a usable nor implementable idea, however, for dealing with categories on a computer, it is in fact a useful concept as we shall see below.

B.4 Inductive classes and category theory

The following mathematics, presented here for the first time, defines a way for us to always have a category of "all the categories in which we currently wish to work". It allows for Axiom's `Category` to be a category, and indeed (by lazy representation) to be a member of `Category`.

Definition B.4.1 *For any ordinal number, ω, $\mathbf{Cat}_\omega$ is the category which has as objects, any category C, for which both $\mathrm{Obj}(C)$ and $\mathrm{Arr}(C)$ are representable as class_ωs. The arrows of $\mathbf{Cat}_\omega$ are the functors between these categories.*

Proposition B.4.2 *$\mathbf{Cat}_\omega$ is an object of $\mathbf{Cat}_{\omega+1}$.*

Proof. $(\forall \xi, \omega \text{ ordinal numbers})((\xi < \omega) \Rightarrow ((X \text{ is a } \mathrm{class}_\xi) \Rightarrow (X \text{ is a } \mathrm{class}_\omega)))$. $\omega < \omega + 1$. Hence, trivial. □

So, for example, $\mathbf{0}$ is an object of $\mathbf{Cat}_0$, and $\mathbf{Set}$ is an object of $\mathbf{Cat}_1$.

B.5 Inductive classes, categories and Axiom

Suppose we wish to define the identity functor for the category of all `Categories` in Axiom. We shall call this functor `CopyCat`.

If we assume that Axiom's `Category` is normally a $\mathbf{Cat}_\omega$, and we wished for a category which was parameterised by a `Category`, then the mathematical way of representing,

```
CopyCat(C : Category) :  Category == C
```

is that `CopyCat` takes `C` a $\mathbf{Cat}_\omega$, but the `CopyCat()` category is a $\mathbf{Cat}_{\omega+1}$.

Thus we now have `Category` as both a $\mathbf{Cat}_\omega$ and a $\mathbf{Cat}_{\omega+1}$, depending on whether we have any instantiations of `CopyCat()` or not. We may carry on inductively, changing the approximation of `Category` (to the category of all `Categories`), should we have `CopyCat(CopyCat(CopyCat(...)))` etc. *ad infinitum* (and beyond, but we shouldn't need that on a computer).

Obviously, this is only a trivial example. This mathematics is a "neat trick" and would allow category theorists to perform their research in Axiom.

However, `Join` is a functor from

$$\prod_{n \in \mathbf{N} \cup 0} \texttt{Category} \to \texttt{Category}$$

in Axiom, so `CopyCat` (defined above) is equivalent to `Join` acting on one argument. Axiom does not consider `Join` to be a functor since it is only defined in the compiler(s) and acts solely as syntax in the construction of new `Categories`.

Bibliography

[AKC97] GNU. `http://www.gnu.org`, 1997. GNU (the copyright owners of AKCL/GCL) homepage on WWW.

[BCM94] W. Bosma, J. Cannon, and G. Matthews. Programming with algebraic structures: Design of the Magma language. In *Proceedings of the ISSAC-94 Conference*, pages 52–57. ACM, 1994.

[Ber91] P. Bernays. *Axiomatic Set Theory (with a historical introduction by A.A. Fraenkel).* Dover, 1991.

[BHFL73] Y. Bar-Hillel, A.A. Fraenkel, and Y. Levy. *Foundations of set theory.* Elsevier Science Publishing BV., 1973.

[BHPS86] R.J. Bradford, A.C. Hearn, J.A. Padget, and E. Schrüfer. Enlarging the REDUCE domain of computation. In *Proceedings of the 1986 Symposium on Symbolic and Algebraic Computation*, pages 100–106. ACM, 1986.

[Bra92] R.J. Bradford. C78: Computer algebra, 1992. Lecture Notes for final year undergraduate and master's course in computer algebra, given at the University of Bath.

[Bro88] M. Broy. Equational specification of partial higher-order algebras. *Theoretical Computer Science*, (57):pages 3–45, 1988.

[Bro97] P.A. Broadberry. Private communication, August 1997.

[CGG$^+$92] B.W. Char, K.O. Geddes, G.H. Gonnet, B.L. Leong, M.B. Monagan, and S.M. Watt. *Maple V Language Reference Manual.* Springer-Verlag, 1992.

[Cro93] R.L. Crole. *Categories for types.* Cambridge University Press, 1993.

[Dav92] J.H. Davenport. C31: Computing foundations, 1992. Lecture Notes for a second year undergraduate course in computing foundations given at the University of Bath.

[Dav93] J.H. Davenport. C9: Universal algebra, 1993. Lecture Notes for second year undergraduate and master's course in universal algebra, given at the University of Bath.

[Der97] Soft Warehouse Inc. `http://www.derive.com`, 1995-97. Derive's parent company's homepage on WWW.

[Dev79] K.J. Devlin. *Fundamentals of Contemporary Set Theory.* Springer-Verlag, 1979.

[DF94] J.H. Davenport and C.R. Faure. The "unknown" in computer algebra. *Programmirovanie*, pages pages 4–10, January 1994.

[DST92] J.H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra, Systems and Algorithms for Algebraic Computation.* Academic Press, second edition, 1992.

[DT90] J.H. Davenport and B.M. Trager. Scratchpad's view of algebra I: Basic commutative algebra. Technical Report 90-31, Computing Group, School of Mathematical Sciences, University of Bath, Bath, England., January 1990.

[Eif97] Eiffel. `http://www.eiffel.com`, 1997. Eiffel homepage on WWW.

[ffi97] J.P. ffitch. Codemist Ltd. `http://www.codemist.tc`, 1997. Codemist Ltd. homepage on WWW.

[ffi98] J.P. ffitch. Codemist REDUCE distribution information. `http://www.codemist.tc/reduce`, January 1998. Codemist Reduce 3.6 homepage on WWW.

[Fod83] J.K. Foderaro. *The Design of a Language for Algebraic Computation Systems.* PhD thesis, University of California at Berkeley, 1983.

[GAP97] The GAP Team, Lehrstuhl D für Mathematik, RWTH Aachen, Germany and School of Mathematical and Computational Sciences, U. St. Andrews, Scotland. *GAP – Groups, Algorithms, and Programming, Version 4*, 1997.

[GJ71] J.H. Griesmer and R.D. Jenks. SCRATCHPAD/1: An interactive facility for symbolic mathematics. In S.R. Petrick, editor, *Proceedings of the Second Symposium for Symbolic and Algebraic Manipulation.* ACM, 1971.

[GM82] J.A. Goguen and J. Meseguer. Completeness of many-sorted equational logic. *ACM Sigplan Notices*, 17(1):pages 9–17, January 1982.

[GM92] J.A. Goguen and J. Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, (105):pages 217–273, 1992.

[Gra97] M. Grand. *Java Language Reference.* O'Reilly and Associates, second edition, 1997.

[GWM$^+$93] Joseph A. Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ, October 1993.

[Har94] J.F. Harris. Inheritance of rewrite rule structures applied to symbolic computation. In *Proceedings of the ISSAC-94 Conference*, pages 318–323. ACM, 1994.

[Hea71] A.C Hearn. Reduce 2, a system and language for algebraic manipulation. In S.R. Petrick, editor, *Proceedings of the Second Symposium for Symbolic and Algebraic Manipulation.* ACM, 1971.

[Hea91] A.C. Hearn. *Reduce User's Manual Version 3.4.* Number CP78. RAND, rev. 7/91 edition, 1991.

[HS73] H. Herrlich and E. Strecker. *Category Theory.* Allyn and Bacon Inc., 1973.

[HS95] A.C. Hearn and E. Schrüfer. A computer algebra system based on order-sorted algebra. *Journal of Symbolic Computation*, 19(1-3):pages 65–77, January/February/March 1995. (Article received September 1993).

[JS92] R.D. Jenks and J. Sutor. *Axiom: the scientific computation system.* Springer-Verlag, 1992.

[JT94] R.D. Jenks and B.M. Trager. How to make Axiom into a Scratchpad. In *Proceedings of the ISSAC-94 Conference*, pages 32–40. ACM, 1994.

[Kin96] A. Kind. youtoo home page. `http://www.maths.bath.ac.uk/~ak1/youtoo`, 1996. youtoo homepage on WWW.

[KR88] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, second edition, 1988.

[KR91] E. Kounalis and M. Rusinowitch. A proof system for conditional algebraic specifications. In S. Kaplan and M. Okada, editors, *Conditional and Typed Rewriting Systems (2nd International CTRS Workshop Montreal, Canada, June 1990 Proceedings)*, number 516 in LNCS, pages 51–63. Springer-Verlag, 1991.

[Law64] F.W. Lawvere. An elementary theory of the category of sets. In *Proceedings of the National Academy of Sciences, USA*, volume 52, pages 1506–1511, 1964.

[Law65] F.W. Lawvere. The category of categories as a foundation of mathematics. In *Proceedings of the conference on categorical algebra*, pages 1–20, 1965.

[Mac97] Macsyma Inc. `http://www.macsyma.com`, 1997. Macsyma homepage on WWW.

[Mae92] R.E. Maeder. Abstract data types. *The Mathematica Journal*, 2(no. 3):pages 47–52, 1992.

[Mar97] U. Martin. Private communication, December 1997.

[Men87] E. Mendelson. *Introduction to mathematical logic.* Wadsworth and Brooks/Cole, third edition edition, 1987.

[ML71] S. Mac Lane. *Categories for the Working Mathematician.* Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971.

[MS97] U. Martin and D. Shand. Investigating some embedded verification techniques for computer algebra systems. In *Proceedings of THEOREMA Workshop*, volume 97-20 of *RISC-Linz technical report 97-20*, 1997. Also available at `http://www-theory.dcs.st-and.ac.uk/~ddshand/Papers/pub.html`.

[Neu96] W. Neun. REDUCE information. `http://www.zib.de/Symbolik/reduce`, October 1996. ZIB Reduce 3.6 homepage on WWW.

[Pad80] P. Padawitz. New results on completeness and consistency of abstract data types. In *Mathematical Foundations of Computer Science 1980 (Proceedings of the 9th Symposium Held in Rydzyna, Poland)*, number 88 in LNCS, pages 460–473. Springer-Verlag, 1980.

[Pad95] J.A. Padget. EuLisp FAQ. `http://www.bath.ac.uk/~masjap/EuLisp/eulisp.html`, 1995. EuLisp homepage on WWW.

[Pla96] C. Playoust. The Magma system for algebra, number theory and geometry. `http://www.maths.usyd.edu.au:8000/comp/magma/Overview.html`, 1996. Magma homepage on WWW.

[Ric97] D.S. Richardson. Private communication, December 1997.

[Str95] A. Strotmann. REDUCE Home Page. `http://www.rrz.uni-koeln.de/REDUCE`, September 1995. Reduce 3.6 homepage on WWW.

[Str97] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, third edition, 1997.

[WBD$^+$94a] S.M. Watt, P.A. Broadberry, S.S. Dooley, P. Iglio, S.C. Morrison, J.M. Steinbach, and R.S. Sutor. *Axiom Library Compiler User Guide.* Number NP2777. NAG, 1st ed. edition, November 1994.

[WBD$^+$94b] S.M. Watt, P.A. Broadberry, S.S. Dooley, P. Iglio, S.C. Morrison, J.M. Steinbach, and R.S. Sutor. A first report on the A♯ compiler. In *Proceedings of the ISSAC-94 Conference*, pages 25–31. ACM, 1994.

[WCS96] L. Wall, T. Christiansen, and R.L. Schwartz. *Programming Perl.* O'Reilly and Associates, second edition, 1996.

[Web93a] A. Weber. On coherence in computer algebra. In *DISCO*, number 722 in LNCS, 1993. Also available at: `http://www-ca.informatik.uni-tuebingen.de/PEOPLE/weber/ENTRIES/papers.html`.

[Web93b] A. Weber. *Type Systems for Computer Algebra.* PhD thesis, Tübingen, 1993. Also available at: `http://www-ca.informatik.uni-tuebingen.de/PEOPLE/weber/ENTRIES/papers.html`.

[Web95] A. Weber. On coherence in computer algebra. *Journal of Symbolic Computation*, 19, January/February/March 1995. Also available at: `http://www-ca.informatik.uni-tuebingen.de/PEOPLE/weber/ENTRIES/papers.html`.

[WG91] T. Weibel and G. Gonnet. An algebra of properties. In *Proceedings of the ISSAC-91 conference*, pages 352–359, July 1991. Bonn.

[WG92] T. Weibel and G. Gonnet. An assume facility for CAS with a sample implementation for Maple. In *DISCO '92 Conference Proceedings*, April 1992. Bath.

[Wol92] S. Wolfram. *Mathematica: A System for doing Mathematics By Computer*. Addison-Wesley, 2nd edition, 1992.