Inheritance of Rewrite Rule Structures Applied to Symbolic Computation

Jason F. Harris
Department of Physics & Astronomy
University of Canterbury
Christchurch, New Zealand
j.harris@phys.canterbury.ac.nz

Abstract

This paper defines and presents a method of inheritance for structures that are defined by rewrite rules. This method is natural in the sense that it can be easily and cleanly implemented in rewrite rules themselves. This framework of inheritance is not that of classical Object-Oriented Programming. It is shown that this inheritance has particular application to structures implemented in rewrite rules and, more generally, to symbolic computation. The treatment is practical, and examples are presented in *Mathematica* for concreteness.

1 Introduction

An *algebraic specification* is a way of describing an abstract data type in an implementation independent way. Throughout this paper, an implementation of an algebraic specification *or* an abstract structure in terms of a system of rewrite rules will be called a *symbolic specification*. So a symbolic specification is a set of rewrite rules describing how to manipulate symbols in keeping with some algebraic specification *or* abstract structure.

It is worth noting that not all algebraic specifications can be translated into a system of compatible rewrite rules Huet & Oppen [5]. This means it is not always possible to find a system of rewrite rules that will reduce equivalent expressions to the same normal form. However, it is generally possible to implement abstract data types and algebraic specifications, at least elementary ones, cleanly and easily in systems of rewrite rules.

It would be highly desirable to have an inheritance mechanism usable as a tool/aid for creating new extended symbolic specifications. For instance, a commutative group can be loosely modelled as $group + commutativity$. It would then be natural to express the specification of a commutative group as

$$\begin{array}{c}\textbf{InheritedSpecification}(group)+\\ \textbf{Specification}(commutativity)\end{array} \tag{1}$$

i.e. it should be possible to have the group specification inherited rather than having to explicitly enter it afresh. Another example is that of a ring, which could be modelled as

$$\begin{array}{c}\textbf{InheritedSpecification}(group)+\\ \textbf{InheritedSpecification}(semigroup)+\\ \textbf{Specification}(distributivity\ properties)\end{array} \tag{2}$$

As illustrated above, the basic idea is that given one or more symbolic specifications, we want to be able to use them *via inheritance* to form more complex specifications. We also want the ability to override the ensuing rules, in analogy with classical object-oriented programming (OOP) (see §7). This paper defines and presents such a method of inheritance for rewrite rules (see §4). This method has a clean syntax and is intuitively nice. It is also easily implemented in rewrite rules themselves. It will satisfy the desired requirements of inheritance for symbolic specifications, and we shall see that it is extremely useful for symbolic computation systems.

The use of this inheritance mechanism is actually much more natural in symbolic computation paradigms than in typical procedural object-oriented paradigms. In typical OOP languages we usually consider passing messages to single objects. In symbolic computation we are interested in the interaction of different symbols, and so we are not really interested in passing messages. For instance where is the object in $a + 3$? What is the message that we are sending? What we really want to know is how two or more symbolic terms, such as a and 3, interact together over $+$ and/or $*$ etc. This will be discussed in §3.2.

Inheritance of structures is not a new concept. There are many contributions to this field, and a treatment of inheritance in the theory of algebraic specifications has been given by many authors [1, 11]. This list is far from exhaustive and for a general survey of algebraic specifications see Wirsing[12]. For a treatment of "classical" OOP(§3) in *Mathematica* see Maeder [9].

The approach of the current paper focusses mainly on rewrite rule inheritance, in particular on giving a practical working view of rewrite rule inheritance in a popular rule based symbolic computation system. I have therefore presented the examples, and the fundamentals of the implementation of this inheritance method, in *Mathematica* (see appendix:A for a brief overview of *Mathematica* syntax used in this paper). This presentation of rewrite rule inheritance is, to my knowledge, unique.

Limongelli [6] comes close to the topics covered here, but her paper is very different. It is mostly theoretical and she treats objects classically. Another nice paper is [2] by Grivas & Maeder. Their paper also uses a system of inheritance but they do not describe how it is implemented. In their system it appears that it would be difficult to perform some of the examples in the sections below. Nonetheless, it is very interesting.

In [8] Maeder describes how to implement in *Mathematica* abstract data types and algebraic specifications in terms of systems of rewrite rules. However here he does discuss inheritance. Of course inheritance of rewrite rules can be used on *any* structure defined by rewrite rules, not just those motivated by an algebraic specification or an abstract structure.

In summary, the goal of this paper is to show that inheritance is a desirable feature in symbolic specifications, and to present in detail a simple, natural and practical method of inheritance using rewrite rules.

2 Symbolic Specifications

In this section we present a simple example of a symbolic specification, one that will be referred to throughout the paper. Consider an abstract group $\langle \mathcal{G}, * \rangle$. It has the properties that

$$\forall a, b, c \in \mathcal{G} \qquad a * (b * c) = (a * b) * c \tag{3}$$
$$\exists e \in \mathcal{G} \text{ s.t. } \forall a \in \mathcal{G} \qquad a * e = e * a = a \tag{4}$$
$$\forall a \in \mathcal{G}\ \exists a^{-1} \in \mathcal{G} \text{ s.t.} \qquad a * a^{-1} = a^{-1} * a = e \tag{5}$$

This can easily be translated into a symbolic specification, i.e. a set of rewrite rules. In *Mathematica* this might look like

$$*_{\mathcal{G}}\texttt{[a\_] := a} \tag{6}$$
$$*_{\mathcal{G}}\texttt{[a\_\_\_, }*_{\mathcal{G}}\texttt{[b\_\_\_], c\_\_\_] := }*_{\mathcal{G}}\texttt{[a, b, c]} \tag{7}$$
$$*_{\mathcal{G}}\texttt{[a\_\_\_, e, c\_\_\_] := }*_{\mathcal{G}}\texttt{[a, c]} \tag{8}$$
$$*_{\mathcal{G}}\texttt{[a\_\_\_, b\_, (b\_)\^{}(-1), c\_\_\_] := }*_{\mathcal{G}}\texttt{[a, e, c]} \tag{9}$$
$$*_{\mathcal{G}}\texttt{[a\_\_\_, (b\_)\^{}(-1), b\_, c\_\_\_] := }*_{\mathcal{G}}\texttt{[a, e, c]} \tag{10}$$

For brevity we have used the symbol $*_{\mathcal{G}}$ in place of a more *Mathematica* like name such as `groupOperation`[1]. Rules (6) and (7) are our representation of an associative binary operator. Rule (8) represents the identity property (4), while (9) and (10) represent the inverse property (5).

In giving these rewrite rules we have made some assumptions in our choice of representations. For instance we have represented an associative binary operation as a indefinite-arity operation[2]. We have also represented the inverse via `(_)^(-1)`.

Rules (6)–(10) are sufficient to describe the behaviour of a generic group[3]. However it will become apparent (see §3.2 and §4.1) that we need to somehow include in the rules the fact that $*_{\mathcal{G}}$ will only work on group elements, i.e. things of type $\mathcal{G}$ (which will denote `groupElement`). To accomplish this we will use the convention that the head of an expression is its type. For instance $\mathcal{G}$`[a]` is of type $\mathcal{G}$(i.e. a group element) and is the symbol `a`. This practice is common in *Mathematica* and we will follow it throughout the majority of this paper. With this convention rules (6)–(10) should be changed to

$$*_{\mathcal{G}}\texttt{[a\_}\mathcal{G}\texttt{] \^{}:= }\mathcal{G}\texttt{[a]} \tag{11}$$
$$\mathcal{G}\texttt{/: }*_{\mathcal{G}}\texttt{[a\_\_\_}\mathcal{G}\texttt{, }*_{\mathcal{G}}\texttt{[b\_\_\_}\mathcal{G}\texttt{], c\_\_\_}\mathcal{G}\texttt{] := }*_{\mathcal{G}}\texttt{[a, b, c]} \tag{12}$$
$$*_{\mathcal{G}}\texttt{[a\_\_\_}\mathcal{G}\texttt{, }\mathcal{G}\texttt{[e], c\_\_\_}\mathcal{G}\texttt{] \^{}:= }*_{\mathcal{G}}\texttt{[a, c]} \tag{13}$$
$$*_{\mathcal{G}}\texttt{[a\_\_\_}\mathcal{G}\texttt{, }\mathcal{G}\texttt{[b\_], }\mathcal{G}\texttt{[(b\_)\^{}(-1)], c\_\_\_}\mathcal{G}\texttt{] \^{}:= }*_{\mathcal{G}}\texttt{[a, }\mathcal{G}\texttt{[e], c]} \tag{14}$$
$$*_{\mathcal{G}}\texttt{[a\_\_\_}\mathcal{G}\texttt{, }\mathcal{G}\texttt{[(b\_)\^{}(-1)], }\mathcal{G}\texttt{[b\_], c\_\_\_}\mathcal{G}\texttt{] \^{}:= }*_{\mathcal{G}}\texttt{[a, }\mathcal{G}\texttt{[e], c]} \tag{15}$$

Using $\mathcal{G}$ instead of `groupElement` and $*_{\mathcal{G}}$ instead of `groupOperation` in (11)–(15) should cause no confusion since this would be perfectly acceptable input in *Mathematica* if the symbols $*_{\mathcal{G}}$ and $\mathcal{G}$ were part of the allowed language (see appendix:B).

3 Comparison of Inheritance Paradigms

3.1 The Classical Object-Oriented Paradigm

Object oriented design is defined by Meyers[10] as

> ...the construction of software systems as structured collections of abstract data type implementations.

This is quite a general concept. A more common or classical view of object-oriented programming might be represented as in Limongelli & Temperini [6] by

Object + Class + Inheritance

Objects are created from classes. An object's methods and variables are defined by the class. Messages sent to the object are handled by the methods of the object. Finally there must exist a mechanism whereby one creates a new class of objects based on a previous class. Collectively this is "classical" inheritance.

We shall see that our inheritance of rules will fit into the former conceptual framework but not into the latter.

3.2 Inheritance in Symbolic Computation

As was indicated in the introduction, inheritance is highly desirable in symbolic specification systems. One example given was that of a commutative group, viewed as a group with the additional property of commutativity. These generic structures are related: one is a sub-class/sub-structure/sub-specification of the other, so some form of inheritance should be used.

Unfortunately such inheritance does not fit quite so easily into the "classical" mold of object-oriented programming. Consider an abstract ring $\langle \mathcal{R}, +, * \rangle$. Two of the many properties/axioms of a ring are

$$\forall a, b, c \in \mathcal{R} \qquad a * (b + c) = a * b + a * c \tag{16}$$
$$\exists 0 \in \mathcal{R} \text{ s.t. } \forall a \in \mathcal{R} \qquad a + 0 = 0 + a = a \tag{17}$$

What is the object here? Is the object $+$ or $*$ or maybe even the abstract ring $\mathcal{R}$?

It is relatively easy to see that the axioms should not be associated with any of these. For example, consider the case when we try to associate the axioms with $*$ in the classical sense. If we now try to define the notion of a commutative ring in keeping with

[1] See appendix:B

[2] This is typical in other languages. For example in Lisp '+' denotes an indefinite-arity operation, e.g. (+ 2 3 5) or (+ 2 3 5 6 7). Also note that (+ 2) is just 2.

[3] Rules (6)–(10) actually form a weakly typed structure since $\mathcal{G}$ is not present in them (see §6).

InheritedSpecification(*ring*) +
Specification(*commutativity of* $*$)

then axioms not involving $*$, e.g. axioms such as (17) associated with the group aspect of a ring, will not be inherited, leading to semantic nonsense. So *all* the axioms/rules have to be inherited when we create inherited structures, not just the ones to do with a particular symbol such as $*$ or $+$.

We can see in this type of specification that there are no "classical" objects; messages are not sent to objects to be handled by their methods. One might argue that the type is the object, i.e. that $\mathcal{R}$ (which is `ringElement`) is the object here. That viewpoint is marginally true but such an "object" will not be instantiated and does not pass or receive messages, and as such will not be a "classical" object.

A much better viewpoint in the context of symbolic computation is that of the interaction of symbolic terms, rather than the passing of messages. For example it is easy to describe how ∂_x, $\cos(x^2 y)$ and $\int_x$ interact with each other.

In summary, we need a new form of inheritance, applicable to collections of rules.

4 Inheritance of Rules

Rule inheritance will just transfer the rules from one or more parent symbols to a target symbol (possibly with substitutions). It says nothing about guaranteeing correctness or truth, or anything. It is just a syntactic inheritance. In this sense it is completely independent of any notions of specifications.

The basic idea of rule inheritance can be demonstrated by the following example. Consider the following rules for `f`

```
f/:  f[x_] + g[y_] := h[x, y]     (18)
f[t_, r] := h[t]                  (19)
```

The symbol `w` can inherit the rules for `f` by the command

```
InheritRules[w,{f}]               (20)
```

This results in `w` possessing the rules

```
w/:  w[x_] + g[y_] := h[x, y]     (21)
w[t_, r] := h[t]                  (22)
```

The command `InheritRules` is the main command in the *Mathematica* package **InheritRules**, which was written by the author [3]. This command is documented in the package but its functionality will become evident in the following. In essence (20) has just replaced `f` by `w` in all the rules. In more intuitive terms we performed the transformation

$$\textit{Rules}[\mathtt{w}] = \textit{Rules}[\mathtt{f}] \ \textit{substituting} \ \mathtt{f} \to \mathtt{w} \qquad (23)$$

But how could we actually accomplish this? In *Mathematica* it is simple: just use

```
DownValues[w]=DownValues[f] /.  f→w     (24)
UpValues[w]=UpValues[f] /.  f→w         (25)
```

It is now obvious that rule inheritance is extremely easy to perform and very elegant in *Mathematica*. The implementation of **InheritRules** is basically rules (24) and (25), with a few simple extensions. This is an *amazingly* easy implementation. Operations (24) and (25) may even seem trivial, indeed they are trivial to perform, but the following sections show that rule inheritance is extremely useful. The implementation in any other rule rewriting language should be equally simple.

It is useful to slightly generalise the above and consider inheriting the rules from one or more parent symbols. It will also be desirable to perform other replacements besides that of the parent symbol(s) for the target symbol. We shall see usages of these properties in the following sections.

4.1 Example: Group to a Commutative Group

We will now apply rule inheritance to the example indicated in (1). Again for brevity we will use some abbreviations[4], namely

$\mathcal{G}$	$\longleftrightarrow$	`elementOfGroup`
$*_{\mathcal{G}}$	$\longleftrightarrow$	`groupOperation`
$\mathcal{G}_C$	$\longleftrightarrow$	`elementOfCommutativeGroup`
$*_{\mathcal{G}_C}$	$\longleftrightarrow$	`commutativeGroupOperation`

The symbolic specification of a commutative group can be given as in (1)

InheritedSpecification(*group*)+
Specification(*commutativity*)

To realise this we first want to inherit the rules for $\mathcal{G}_C$ from those of $\mathcal{G}$ and then add the rule for commutativity. Inheriting the rules from $\mathcal{G}$ is done via:

$$\textit{Rules}[\mathcal{G}_C] = \textit{Rules}[\mathcal{G}] \ \textit{substituting} \ \{\mathcal{G} \to \mathcal{G}_C \,,\ *_{\mathcal{G}} \to *_{\mathcal{G}_C}\} \qquad (26)$$

As a result $\mathcal{G}_C$ will possess all of the following rules:

$*_{\mathcal{G}_C}$`[a_`$\mathcal{G}_C$`] ^:= `$\mathcal{G}_C$`[a]` (27)

$\mathcal{G}_C$`/:  `$*_{\mathcal{G}_C}$`[a___`$\mathcal{G}_C$`, `$*_{\mathcal{G}_C}$`[b___`$\mathcal{G}_C$`], c___`$\mathcal{G}_C$`] := `$*_{\mathcal{G}_C}$`[a, b, c]` (28)

$*_{\mathcal{G}_C}$`[a___`$\mathcal{G}_C$`, `$\mathcal{G}_C$`[e], c___`$\mathcal{G}_C$`] ^:= `$*_{\mathcal{G}_C}$`[a, c]` (29)

$*_{\mathcal{G}_C}$`[a___`$\mathcal{G}_C$`, `$\mathcal{G}_C$`[b_], `$\mathcal{G}_C$`[(b_)^(-1)], c___`$\mathcal{G}_C$`] ^:= `$*_{\mathcal{G}_C}$`[a,`$\mathcal{G}_C$`[e], c]` (30)

$*_{\mathcal{G}_C}$`[a___`$\mathcal{G}_C$`, `$\mathcal{G}_C$`[(b_)^(-1)], `$\mathcal{G}_C$`[b_], c___`$\mathcal{G}_C$`] ^:= `$*_{\mathcal{G}_C}$`[a,`$\mathcal{G}_C$`[e], c]` (31)

We must now add the rule for commutativity of $*_{\mathcal{G}_C}$, which is

$*_{\mathcal{G}_C}$`[a___`$\mathcal{G}_C$`, b_`$\mathcal{G}_C$`, c_`$\mathcal{G}_C$`, d___`$\mathcal{G}_C$`] ^:= `$*_{\mathcal{G}_C}$`[a, c, b, d] /; !OrderedQ[{b, c}]` (32)

This now completes the symbolic specification of a commutative group. Notice that we only had to add the single rule (32) to the inherited rules for $\mathcal{G}_C$. If we didn't inherit rules (27)–(31) we would have had to enter them by hand as was done in the past. This of course was prone to error, very tedious, and did not lead to modular construction or reuse of existing structures. The whole point of inheritance is to circumvent such problems.

It is very important to note that we could not have simply added a rule like (32) (with $\mathcal{G}_C \to \mathcal{G}$) to the rules for $\mathcal{G}$. This would have made the group specification commutative rather than making a whole new specification that represents a commutative group. It is essential to have a whole new set of rules for the symbol $\mathcal{G}_C$ even though most of these rules represent the same properties as the rules of $\mathcal{G}$.

[4] Again see appendix:B

One can now perform computations. For instance

$$*_{\mathcal{G}_C}\texttt{[}\mathcal{G}_C\texttt{[y^(-1)],}\mathcal{G}_C\texttt{[x],}\mathcal{G}_C\texttt{[y],}\mathcal{G}_C\texttt{[e],}\mathcal{G}_C\texttt{[a]]} \tag{33}$$

evaluates or reduces to

$$*_{\mathcal{G}_C}\texttt{[}\mathcal{G}_C\texttt{[a], }\mathcal{G}_C\texttt{[x]]} \tag{34}$$

5 Building a Hierarchy of Structures

For the remainder of this article we will use the *Mathematica* notation `/.` instead of *substituting* (see appendix:A). These notations can be thought of as completely equivalent and we have used `/.` solely because of formatting constraints.

We are now in a position to present more complex examples. First we will define a structure for generic associative binary operators. As before we will use some abbreviations. In this case $\mathcal{A}$ will be a generic associative binary operation, and $*_{\mathcal{A}}$ will be its corresponding operator. The representation can be specified by

$$*_{\mathcal{A}}\texttt{[a\_}\mathcal{A}\texttt{] ^:= }\mathcal{A}\texttt{[a]} \tag{35}$$

$$\mathcal{A}\texttt{/: } *_{\mathcal{A}}\texttt{[a\_\_\_}\mathcal{A}\texttt{, }*_{\mathcal{A}}\texttt{[b\_\_\_}\mathcal{A}\texttt{], c\_\_\_}\mathcal{A}\texttt{] := } *_{\mathcal{A}}\texttt{[a, b, c]} \tag{36}$$

If we further define a structure for the identity property

$$*_{\mathcal{I}}\texttt{[a\_\_\_}\mathcal{I}\texttt{, }\mathcal{I}\texttt{[e], c\_\_\_}\mathcal{I}\texttt{] ^:= }*_{\mathcal{I}}\texttt{[a, c]}$$

then we can create an associative binary structure with identity by

$$\begin{aligned} &Rules\texttt{[}\mathcal{A}_{\mathcal{I}}\texttt{]} \ = \\ &\quad Rules\texttt{[}\mathcal{A}\texttt{] /.} \{\mathcal{A} \to \mathcal{A}_{\mathcal{I}},\ *_{\mathcal{A}} \to *_{\mathcal{A}_{\mathcal{I}}}\} \\ &\quad +Rules\texttt{[}\mathcal{I}\texttt{] /.} \{\mathcal{I} \to \mathcal{A}_{\mathcal{I}},\ *_{\mathcal{I}} \to *_{\mathcal{A}_{\mathcal{I}}}\} \end{aligned} \tag{37}$$

Using these we could have defined the group symbolic specification, $\mathcal{G}$, by first performing

$$Rules\texttt{[}\mathcal{G}\texttt{]} = Rules\texttt{[}\mathcal{A}_{\mathcal{I}}\texttt{] /.} \{\mathcal{A}_{\mathcal{I}} \to \mathcal{G},\ *_{\mathcal{A}_{\mathcal{I}}} \to *_{\mathcal{G}}\} \tag{38}$$

and then just adding rules (14) and (15) by hand. Alternatively we could have made another structure for these inverse properties and then inherited that along with $\mathcal{A}_{\mathcal{I}}$. Both constructions would result in the same specification for $\mathcal{G}$.

5.1 Example: Generic Rings

The specification of a generic ring via inheritance (as in (2) can now be simply done by first letting

$$\begin{aligned} &Rules\texttt{[}\mathcal{R}\texttt{]} \ = \\ &\quad Rules\texttt{[}\mathcal{G}_C\texttt{] /.} \\ &\qquad \{\mathcal{G}_C \to \mathcal{R},\ *_{\mathcal{G}_C} \to +_{\mathcal{R}},\ \texttt{e} \to 0,\ (\_)^{-1} \to -(\_)\} \\ &\quad +Rules\texttt{[}\mathcal{A}\texttt{] /.} \{\mathcal{A} \to \mathcal{R},\ *_{\mathcal{A}} \to *_{\mathcal{R}}\} \end{aligned} \tag{39}$$

and then just entering the following rule (whose syntax is somewhat technical) to obtain the distributive properties:

$$\mathcal{R}\texttt{/: } *_{\mathcal{R}}\texttt{[a\_\_}\mathcal{R}\texttt{, }+_{\mathcal{R}}\texttt{[b\_\_\_}\mathcal{R}\texttt{], c\_\_\_}\mathcal{R}\texttt{] := } +_{\mathcal{R}}\texttt{ @@ Map[}*_{\mathcal{R}}\texttt{[a, \#, c]\&, \{b\}]} \tag{40}$$

Thus we can easily and intuitively obtain this specification through inheritance.

Finally we can specify a generic ring with identity by

$$\begin{aligned} &Rules\texttt{[}\mathcal{R}_{\mathcal{I}}\texttt{]} \ = \\ &\quad Rules\texttt{[}\mathcal{R}\texttt{] /.} \{\mathcal{R} \to \mathcal{R}_{\mathcal{I}},\ +_{\mathcal{R}} \to +_{\mathcal{R}_{\mathcal{I}}},\ *_{\mathcal{R}} \to *_{\mathcal{R}_{\mathcal{I}}}\} \\ &\quad +Rules\texttt{[}\mathcal{I}\texttt{] /.} \{\mathcal{I} \to \mathcal{R}_{\mathcal{I}},\ *_{\mathcal{I}} \to *_{\mathcal{R}_{\mathcal{I}}},\ \texttt{e} \to 1\} \end{aligned} \tag{41}$$

We could go on but it is fairly obvious that this type of problem is now easily handled.

6 A Weakly Typed Example

Up to this point the structures considered have been strongly typed or best described as strongly typed, which means that the type of every expression is known and specified. In contrast the following example, which is the same as that in Grivas & Maeder [2], presents a simple example of inheritance of a weakly typed specification of `Log`.

First our abstract specification of the properties of `Log` can be given by

$$\forall a, b \qquad Log(a\, b) = Log(a) + Log(b) \tag{42}$$

$$\forall a, b \qquad Log(b^a) = a Log(b) \tag{43}$$

We can symbolically specify this in *Mathematica* by the following rules

$$\texttt{Log[a+b] := Log[a\_] + Log[b\_]} \tag{44}$$

$$\texttt{Log[b\_^a\_] := a Log[b]} \tag{45}$$

Notice that the specification here has been given $\forall a, b$. We want these properties to be true no matter if we are dealing with elements of $\mathbb{R}$ or $\mathbb{C}$ or $\mathbb{Q}[a]$ or maybe even something exotic like Lie generators. Therefore this structure is best described as weakly typed or untyped.

The natural logarithm (`ln`) has the properties (42) and (43) as well as the extra property $ln(e) = 1$. Therefore we want to inherit the rules of `ln` from the rules of `Log` and add the new rule `ln[e]:=1`. So again we would intuitively perform

$$Rules\texttt{[ln]} = Rules\texttt{[Log]}\ substituting\ \texttt{Log}{\to}\texttt{ln} \tag{46}$$

which in *Mathematica* is done by `InheritRules[ln,{Log}]`. After adding the rule `ln[e]:=1`, `ln` will possess the rules

$$\texttt{ln[a+b] := ln[a\_] + ln[b\_]} \tag{47}$$

$$\texttt{ln[b\_^a\_]:= a ln[b]} \tag{48}$$

$$\texttt{ln[e] := 1} \tag{49}$$

As one can see, this form of inheritance of rewrite rules in no way depends on how strong or how weak the typing is. It is up to the user to dictate how strong or how weak the typing is, and to do this directly through his/her symbolic specification and the structure he/she wants to create. The method of inheritance advocated in this paper is natural with either form of typing.

7 A Non-Standard Example

In this next example we present the rewrite rules for generic differentiation. We inherit these rules, and finally override one of them. Because we are only interested in symbolics we do not need to think of the underlying mathematics of limits, differentiability, etc. All we have to do is symbolically model differentiation, even though the axiomatic definitions may not be well defined. The specification we choose is of a mixed typed i.e. partly strongly typed and partly weakly typed. We give this specification in *Mathematica* (where we have written operators like $*_{\mathcal{T}}$, $+_{\mathcal{M}}$ and $*_{\mathcal{M}}$ in infix form for readability) as follows:

$$\mathcal{T}\texttt{/: } \mathcal{D}_{\mathcal{T}}\texttt{[a\_}\mathcal{T}\ *_{\mathcal{T}}\ \texttt{b\_}\mathcal{T}\texttt{,z\_}\mathcal{T}\texttt{]:= (}\mathcal{D}_{\mathcal{T}}\texttt{[a,z] }*_{\mathcal{T}}\texttt{ b) + (a }*_{\mathcal{T}}\ \mathcal{D}_{\mathcal{T}}\texttt{[b,z])} \tag{50}$$

$$\mathcal{T}\texttt{/: } \mathcal{D}_{\mathcal{T}}\texttt{[a\_}\mathcal{T}\texttt{+b\_}\mathcal{T}\texttt{,z\_}\mathcal{T}\texttt{] := }\mathcal{D}_{\mathcal{T}}\texttt{[a,z]+}\mathcal{D}_{\mathcal{T}}\texttt{[b,z]} \tag{51}$$

$$\mathcal{T}\texttt{/: } \mathcal{D}_{\mathcal{T}}\texttt{[}\mathcal{T}\texttt{[c\_?ConstQ a\_],z\_}\mathcal{T}\texttt{]:= c }\mathcal{D}_{\mathcal{T}}\texttt{[}\mathcal{T}\texttt{[a],z]} \tag{52}$$

where $\mathcal{T}$ is the type generic and $\mathcal{D}$ is differentiation. We can then inherit these rules for generic differentiation to say differentiation of matrices. Let $\mathcal{M}$ be the type elementOfMatrix (the other new symbols should then be obvious but for reference are given in appendix:B). We perform the inheritance by

$$\mathit{Rules}[\mathcal{M}] = \mathit{Rules}[\mathcal{T}]\ /.\ \{\mathcal{T} \to \mathcal{M},\ + \to +_{\mathcal{M}},\ *_{\mathcal{T}} \to *_{\mathcal{M}},\ \mathcal{D}_{\mathcal{T}} \to \mathcal{D}_{\mathcal{M}}\}$$

This results in $\mathcal{M}$ having the following rules:

$$\mathcal{M}/:\ \ \mathcal{D}_{\mathcal{M}}[\mathtt{a\_}\mathcal{M}\ *_{\mathcal{M}}\ \mathtt{b\_}\mathcal{M},\mathtt{z\_}\mathcal{M}]\ :=\ (\mathcal{D}_{\mathcal{M}}[\mathtt{a},\mathtt{z}]\ *_{\mathcal{M}}\ \mathtt{b})\ +_{\mathcal{M}}\ (\mathtt{a}\ *_{\mathcal{M}}\ \mathcal{D}_{\mathcal{M}}[\mathtt{b},\mathtt{z}]) \quad (53)$$

$$\mathcal{M}/:\ \ \mathcal{D}_{\mathcal{M}}[\mathtt{a\_}\mathcal{M}\ +_{\mathcal{M}}\ \mathtt{b\_}\mathcal{M},\mathtt{z\_}\mathcal{M}]\ :=\ \mathcal{D}_{\mathcal{M}}[\mathtt{a},\mathtt{z}]\ +_{\mathcal{M}}\ \mathcal{D}_{\mathcal{M}}[\mathtt{b},\mathtt{z}] \quad (54)$$

$$\mathcal{M}/:\ \ \mathcal{D}_{\mathcal{M}}[\mathcal{M}[\mathtt{c\_?ConstQ\ a\_}],\mathtt{z\_}\mathcal{M}]\ :=\ \mathtt{c}\ \mathcal{D}_{\mathcal{M}}[\mathcal{M}[\mathtt{a}],\mathtt{z}] \quad (55)$$

Notice that we needed to type $*$ in (50) so we could perform $*_{\mathcal{T}} \to *_{\mathcal{M}}$, but we only needed $+ \to +_{\mathcal{M}}$ instead of $+_{\mathcal{T}} \to +_{\mathcal{M}}$. This is due to the fact that we implicitly had two different types of multiplication operators in (50) and (52), but only one type of addition.

To illustrate overriding assume for some reason that the inherited rule (55) is now no longer correct. To override this rule we simply remove the old rule via

$$\mathcal{M}/:\ \ \mathcal{D}_{\mathcal{M}}[\mathcal{M}[\mathtt{c\_?ConstQ\ a\_}],\ \mathtt{z\_}\mathcal{M}]\ =. \quad (56)$$

and then add the correct rule via

$$\mathcal{D}_{\mathcal{M}}[\mathcal{M}[\mathtt{a\_}\ *_{funny}\ (\mathtt{c\_})\mathtt{?ConstQ}],\ \mathtt{z\_}\mathcal{M}]\ \mathtt{\hat{}:=}\ \mathtt{c}\ *_{strange}\ \mathcal{D}_{\mathcal{M}}[\mathcal{M}[\mathtt{a}],\ \mathtt{z}] \quad (57)$$

Thus we can see that it is relatively easy to override rules, although it probably isn't all that common in mathematical specifications if one chooses the initial structures correctly.

8 Conclusions and Applications

In §4 we presented a method for the inheritance of rewrite rules. This inheritance was shown to be *very* simple and extremely intuitive. Many examples of using this type of inheritance were given throughout the paper (see §4.1, §5, §5.1, §6 and §7). This inheritance is very easy to implement in *Mathematica*. It was described in §3 why the conventional viewpoint of OOP is not really applicable to symbolic computation. A much better viewpoint is that of interaction of symbolic terms (see §3.2).

Almost all of the material given in this paper should be easily generalisable to any rule rewriting language. This is, however, conditional on the language being able to somehow perform rewriting on its own rules.

We can directly apply the techniques of rule inheritance to practical topics in symbolic computation, for instance the description of a hierarchy of quantum mechanical operators in physics (see Harris [4]). Previously calculations in this area were generally piecemeal, because the user had to describe how each particular symbol interacts with other symbols rather than how the types interact with each other. (There are some nice dedicated systems but they tend to be unextendable.)

It should now be clearly evident that rule inheritance is very useful in symbolic computation. Hopefully this presentation of rule inheritance will change the conception of inheritance in the context of symbolic computation.

Acknowledgements

This research was supported by the NZVCC. My thanks go to Gerrard F. Liddell and John H. Harris for valuable discussions in reference to this paper. I would also like to thank an anonymous reviewer for the reference to Grivas & Maeder [2]

Appendices

A *Mathematica* Syntax Used

This is a very brief summary of the *Mathematica* commands used in the above article. The description of the commands given here might be slightly misleading since all that is intended is to give the reader the barest knowledge sufficient to understand the text of this article. For more detailed descriptions consult the *Mathematica* reference guide [13].

`/.` Usage Example `expr /. rules`
This basically means perform the substitutions specified by `rules` throughout `expr` wherever they occur.

`/:` Usage Example `a/: f[a[x_],y_]:= ...`
Associates a rule with a symbol other than the `Head`; e.g. associate the rule with the symbol `a` and not the `Head` of the rule which is the symbol `f`.

`^:=` Usage Example $*_{\mathcal{G}}[\mathtt{a\_}\mathcal{G}]\ \mathtt{\hat{}:=}\ \mathcal{G}[\mathtt{a}]$
Associates a rule with all symbols one level lower than the `Head`; e.g. associate the rule with the symbol $\mathcal{G}$ and not the `Head` of the rule which is the symbol $*_{\mathcal{G}}$. For contrast we can see in (12) that if we had used `^:=` we would have associated the rule with both $\mathcal{G}$ and $*_{\mathcal{G}}$, whereas all we wanted to associate the rule with was $*_{\mathcal{G}}$, hence we used `/:` in (12).

`_` Usage Example `a_`
This is the primary pattern matching mechanism of *Mathematica* and it will match any expression.

`___` Usage Example `a___`
This will match a sequence of zero or more expressions.

_*ahead* Usage Example $\mathtt{a\_}\mathcal{G}$
This will match any expression with `Head` *ahead*. e.g. for the example above it would be equivalent to matching to the pattern $\mathcal{G}$`[_]` and calling it `a`. We can think of `a_` as being a pattern of type $\mathcal{G}$.

Examples of pattern matching

1. Consider the pattern $*_{\mathcal{G}}$`[a_]` which occurs in (6). This will match $*_{\mathcal{G}}$`[Sin[x]]` since `a_` can match `Sin[x]`, i.e. $*_{\mathcal{G}}$`[Sin[x]]` would be rewritten by (6) to `Sin[x]`. However $*_{\mathcal{G}}$`[d,f]` would not match the above pattern since `a_` can only match a single expression and not multiple expressions like `d,f`

2. Consider another pattern $*_{\mathcal{G}}$`[a___,` $*_{\mathcal{G}}$`[b___], c___]` This will match $*_{\mathcal{G}}$`[x,y,` $*_{\mathcal{G}}$`[t,z]]` since `a___` can match `x,y` and `b___` can match `t,z` and `c___` can match *null*. This pattern will even match say $*_{\mathcal{G}}$`[x,y,` $*_{\mathcal{G}}$`[]]` since `b___` as well as `c___` can match *null*. However this pattern will not match $*_{\mathcal{G}}$`[x,y,` $*_{\mathcal{A}}$`[t,z]]` since the symbol $*_{\mathcal{A}}$ does not match $*_{\mathcal{G}}$.

3. Consider the pattern $*_{\mathcal{G}}$[a___$\mathcal{G}$, $\mathcal{G}$[e], c___$\mathcal{G}$]. This will match $*_{\mathcal{G}}$[$\mathcal{G}$[x],$\mathcal{G}$[y], $\mathcal{G}$[e]] since a___$\mathcal{G}$ can match $\mathcal{G}$[x],$\mathcal{G}$[y] and c___$\mathcal{G}$ can match *null*. However it will not match $*_{\mathcal{G}}$[$\mathcal{G}$[x],y, $\mathcal{G}$[e]] because a___$\mathcal{G}$ cannot match the expression $\mathcal{G}$[x],y since y does not have a Head of $\mathcal{G}$.

B *Mathematica* Abbreviations Used

Unfortunately at the time of this writing *Mathematica* does not have the ability to accept non-standard expression such as $\mathcal{G}_C$ as input. However for reasons of readability such symbols have been used throughout this paper. If one wanted to actually try the examples in *Mathematica* one would have to use symbol names acceptable to *Mathematica*. Listed below are the names the author used for the various symbols. (Note: there are many other possible naming conventions.)

Symbols for Groups

$\mathcal{G}$	$\longleftrightarrow$	`elementOfGroup`
$*_{\mathcal{G}}$	$\longleftrightarrow$	`groupOperation`
$\mathcal{G}_C$	$\longleftrightarrow$	`elementOfCommutativeGroup`
$*_{\mathcal{G}_C}$	$\longleftrightarrow$	`commutativeGroupOperation`

Symbols for Rings

$\mathcal{R}$	$\longleftrightarrow$	`elementOfRing`
$+_{\mathcal{R}}$	$\longleftrightarrow$	`ringAddition`
$*_{\mathcal{R}}$	$\longleftrightarrow$	`ringMultiplication`
$\mathcal{R}_{\mathcal{I}}$	$\longleftrightarrow$	`elementOfRingWithIdentity`
$+_{\mathcal{R}_{\mathcal{I}}}$	$\longleftrightarrow$	`ringWithIdentityAddition`
$*_{\mathcal{R}_{\mathcal{I}}}$	$\longleftrightarrow$	`ringWithIdentityMultiplication`

Symbols for Generic Structures

$\mathcal{A}$	$\longleftrightarrow$	`elementOfAssociativeBinary`
$*_{\mathcal{A}}$	$\longleftrightarrow$	`associativeBinaryOperation`
$\mathcal{A}_{\mathcal{I}}$	$\longleftrightarrow$	`elementOfAssociativeBinaryWithIdentity`
$*_{\mathcal{A}_{\mathcal{I}}}$	$\longleftrightarrow$	`associativeBinaryWithIdentityOperation`
$\mathcal{I}$	$\longleftrightarrow$	`elementOfGenericIdentityStructure`
$*_{\mathcal{I}}$	$\longleftrightarrow$	`genericIdentityStructureOperation`
$\mathcal{T}$	$\longleftrightarrow$	`elementOfGeneric`
$*_{\mathcal{T}}$	$\longleftrightarrow$	`genericOperation`

Symbols for Matrices Structures

$\mathcal{M}$	$\longleftrightarrow$	`elementOfMatrix`
$+_{\mathcal{M}}$	$\longleftrightarrow$	`matrixAdd`
$*_{\mathcal{M}}$	$\longleftrightarrow$	`matrixTimes`
$\mathcal{D}_{\mathcal{T}}$	$\longleftrightarrow$	`genericDifferentiation`
$\mathcal{D}_{\mathcal{M}}$	$\longleftrightarrow$	`matrixDifferentiation`

References

[1] R. Breu, *Algebraic specification techniques in object oriented programming environments*, Lecture Notes in Computer Science, vol. 562, Springer-Verlag, 1991.

[2] G. Grivas and R. E. Maeder, *Matching and unification for the object-oriented symbolic computation system algbench*, Proc. of Design and Implementation of Symbolic Computation Systems DISCO '93. Lecture Notes in Computer Science, 1993, pp. 164–176.

[3] J. F. Harris, *Package* `InheritRules.m`, avalible on the *MathSource* archives item : 0206-525 (1994), email mathsource@wri.com.

[4] ______, *Symbolic implementation of hierarchies of quantum mechanical operators*, To be submitted (1994).

[5] G. Huet and D. C. Oppen, *Equations and rewrite rules: a survey*, Formal Language Theory: Perspectives and Open Problems, Academic Press, New York, 1980.

[6] C. Limongelli and M. Temperini, *Abstract specification of structures and methods in symbolic mathematical computation*, Theoretical Computer Science **104** (1992), no. 1, 89–107.

[7] R. E. Maeder, *Programming in mathematica*, Addison-Wesley, Redwood City, Calif, 1990.

[8] ______, *Abstract data types*, The Mathematica Journal **2** (1992), no. 3, 47–52.

[9] ______, *Object-oriented programming*, The Mathematica Journal **3** (1993), no. 1.

[10] B. Meyer, *Object-oriented software construction*, Prentice-Hall, 1988.

[11] I. Van Horebeek and J. Lewi, *Algebraic specifications in software engineering*, Springer-Verlag, 1989.

[12] M. Wirsing, *Algebraic specification*, Formal Models and Semantics. Handbook of Theoretical Computer Science (J. Van Leeuwen, ed.), MIT Press, 1990, pp. 677–788.

[13] S. Wolfram, *Mathematica: A system for doing mathematics by computer*, 2nd ed., Addison-Wesley, 1991.