Formalism and the Variable (work in regress)

Eric C.R. Hehner
1999 July 21

Department of Computer Science, University of Toronto,
Toronto ON M5S 3G4, Canada
hehner@cs.toronto.edu

Abstract. We consider what natural language explanations are required to present a formalism. We desire to be as complete as possible, leaving nothing implicit, and yet to keep the natural language to a minimum, moving as quickly as possible away from informal explanations and into the formalism. We also try to keep the informal explanation as independent as possible of the formalism being presented. We pay particular attention to scope-ignoring and scope-respecting substitution. Variable renaming turns out to be a consequence of scope and extension. We speculate on the possibility of recursive variable introduction.

1 Introduction

A mathematical formalism is a language for describing and reasoning about the world with more precision and conciseness than is possible using natural language. How can we explain, or present, a mathematical formalism? A precise and concise explanation is desirable, and so formalisms have been developed for the purpose of presenting formalisms. There are grammatical formalisms to say how to write the expressions of a formalism, and there are formal meta-notations to present the rules for their use (often called "proof rules"). These formalisms then need to be explained, so the explanation problem is not solved. And there are some aspects of mathematical formalism, such as the rules for substitution, that are awkward to express formally.

We have no choice but to begin informally when we explain a formalism. My goal is to be as brief and clear as possible, and to move as quickly as possible away from informal explanations and into the formalism.

A formalism consists of expressions, which are used to express values. I consider that values are in the application domain; for example, the values may be amounts of water, or voltage, or frequency of vibration, or guilt and innocence. Other people may consider that values are abstract, mathematical objects, although it is unnecessary and unhelpful to postulate such abstract objects. For this essay, either viewpoint is acceptable; all I need is that the expressions of a formalism express values.

As an economy of speech, we say " 2+3 has value 5 " to mean "expression 2+3 expresses the same value as expression 5 expresses". In other words, we speak as though simple expressions are values.

With that meagre beginning, we can already make some key definitions.

Consistency:	no expression expresses more than one value
Completeness:	every expression expresses a value
Expressiveness:	every value is expressed by an expression
Uniqueness:	no value is expressed by more than one expression

Consistency is essential; completeness is not. Expressiveness is desirable; uniqueness is not. These definitions precede the choice of formalism and the choice of application domain; we have not yet said whether we will have boolean expressions, and we have not yet said whether we are expressing truth values.

As simple as these definitions are, they already raise questions of definedness and partiality that I would prefer to avoid. Whether 0/0 has no value, or has a value but we cannot say what it is, is a sterile argument. It is of no interest whether an expression expresses a value if we cannot determine the value, so I propose to reword the definitions as follows.

Consistency:	at most one value can be determined for each expression
Completeness:	at least one value can be determined for each expression
Expressiveness:	at least one expression can be determined for each value
Uniqueness:	at most one expression can be determined for each value

2 Evaluation Rules

How do we determine the value of an expression? Even before saying what our expressions and values are, we can already show some examples of rules for determining the value of an expression.

> Direct Rule: An expression may be given a value by physical means, or by other means outside the formalism.

For example, by marking numbers along a stick we give them length values. We might say that we will use $\top$ to express truth and $\bot$ to express falsity. This is the way a formalism is applied.

> Table Rule: If the values of all subexpressions of an expression are known, then value tables can be used to determine its value.

If we decide to introduce boolean expressions, and to use them to represent truth values, we might like to use “truth tables” to say how to evaluate them. For example, one of the entries for $=$ will tell us that $\top=\top$ has value $\top$. Value tables can similarly be used for the expressions of ternary algebra. For larger algebras tables are too cumbersome, though they could be used for a portion of a large algebra.

> Completion Rule: If the values of some subexpressions of an expression are unknown, and all ways of assigning them values give it the same value, then it has that value.

For example, if we are presenting boolean algebra, this rule tells us that $x \vee \neg x$ has value $\top$, and that $x \wedge \neg x$ has value $\bot$. Should we say explicitly that when a subexpression occurs more than once, all occurrences must be assigned the same value? When there are infinitely many values, it is not always clear how to determine if all assignments give the same answer; thus we enter the classical/constructive debate. Some people like this rule, and some dislike it; I am not taking a side, but only using it as an example.

> Consistency Rule: If it would be inconsistent for an expression to have a particular value, then it has another value. More generally, if it would be inconsistent for several expressions to have a particular assignment of values, then they have another assignment of values.

Again using the example of boolean algebra, and assuming $\Rightarrow$ has been defined (perhaps by a value table), this rule gives us modus ponens: if x and $x \Rightarrow y$ have value $\top$, then y has value $\top$. Assuming $\neg$ has been defined, this rule similarly tells us that if $\neg x$ has value $\top$, then x has value $\bot$. Assuming $=$ has been defined, it also says if $x=y$ has value $\top$, then x and y have the same value.

> Transparency Rule: An expression does not change value when a subexpression is replaced by another with the same value.

For example, if x and y have the same value, then $x \wedge z$ and $y \wedge z$ have the same value. This rule is sometimes presented as "substitution of equals for equals"; the version here is more general because it does not assume we have an $=$ operator.

> Indirect Rule: An expression whose value cannot be determined by the other rules may be given a value by saying that it has the same value as another expression whose value is already known.

Axioms, or laws, are boolean expressions to which we give value $\top$; the Indirect Rule is a generalization of a rule that says it is permissible to use axioms. Indeed, it is more than permissible; this is the route by which we stop explaining in misinterpretable English, and start to use the formalism we are trying to present. The Indirect Rule, as it is worded here, says that it must not be used to introduce inconsistency.

Do we need all these rules, or can we just use the Indirect Rule? The Table Rule is just a special case of the Indirect Rule in which some expressions are given values by means of a table, so we don't need it; but it is a convenient way to get started, and it wasn't problematic. The Completion, Consistency, and Transparency Rules were problematic, but they cannot be eliminated in favor of the Indirect Rule. Axioms like

$$x \vee \neg x$$
$$x \wedge (x \Rightarrow y) \Rightarrow y$$
$$x=y \Rightarrow fx=fy$$

are special cases; they do not cover all that the evaluation rules cover.

3 Variables and Instantiation

For the rest of this essay, I assume that the formalism being presented includes variables. Not all formalisms do; combinatory logic [2] claims that it does not. The problems that I now address apply to those formalisms that include variables.

We could say that identifiers like x are used to stand for variables; identifiers are expressions, and variables are objects being expressed by them. Some programming languages insist on the dichotomy between identifiers and variables. I consider that a variable is a kind of expression; in other words, x is a variable. Expressions represent values, and a variable represents an arbitrary value. (There may be more than one type of value, but I do not consider typing in this essay.) A variable can be replaced by an arbitrary expression. Replacing a variable by another expression is called instantiation. Three points must be explained:

- We sometimes have to insert parentheses around expressions that are replacing variables in order to maintain the order of evaluation.

- When the same variable occurs more than once in an expression, it must be replaced by the same expression at each occurrence.

- Different variables may be replaced by the same or different expressions.

With instantiation, we might like to add one more evaluation rule that requires natural language.

> Instance Rule: If the value of an expression can be determined, then all its instances have that same value.

For example, if $x{=}x$ has value T, then $0/0 = 0/0$ has value T.

4 Scope

The final complication that requires some natural language is the introduction of local scope. The bracketing operator $\langle x|\ \ |x\rangle$ encloses an expression, and introduces local variable x within the brackets. For example, within

$\langle n|\ n{+}m\ |n\rangle$

n is local but m is nonlocal. As an abbreviation, we may omit the local variable and vertical bar whenever the local variable appears first inside the scope. For example, the preceding expression may be written

$\langle n{+}m\rangle$

Instantiation is now more complicated by the following two points:

- Replace nonlocal variables only.

- Do not place a nonlocal variable where it will appear to be local.

The preceding paragraph is misleading; specifically, the words “local within” and “nonlocal within” are ill-chosen. (There are dangers lurking in even the most innocent-looking sentences.) A variable may be local to one scope, but nonlocal to another inner nested scope, so it is misleading to say it is “local within” the outer scope. Similarly, a variable may be nonlocal to one scope, but local to another inner nested scope, so it is misleading to say it is “nonlocal within” the outer scope. We who know about locality and nested scopes tend to read our knowledge into the explanations; the challenge is to make the explanations clear to someone who doesn't already know.

Scope brackets were called an operator, applying to an operand. And they have an inverse operator. The axiom relating scoping and unscoping is called the

Scope Law $\langle x|\ E\ |x\rangle\ x \quad = \quad E$

The following, numbered (0) for later reference, has the same form as the Scope Law.

(0) $\langle n|\ n+m\ |n\rangle\ n \quad = \quad n+m$

Notice that n is used both locally and nonlocally; it is a kind of pun. The reason for the pun is that the Scope Law explains what it means to cross a scope boundary. Now we instantiate (0) by replacing n with $a+1$, but remember: instantiation replaces only the nonlocal occurrences of n.

(1) $\langle n|\ n+m\ |n\rangle\ (a+1) \quad = \quad a+1+m$

From (0) we could replace n by m to get

(2) $\langle n|\ n+m\ |n\rangle\ m \quad = \quad m+m$

but from (0) we cannot replace m by n because we would have to place a nonlocal n where it would appear to be local.

The Scope Law provides us with a notation for substitution (or instantiation). We can read (1) as follows: replace n in $n+m$ by $a+1$ to get $a+1+m$. It would be pointless to introduce another notation for substitution and then equate it to the one we already have. We can write (0), (1), and (2) more briefly as follows.

(0) $\langle n+m\rangle\ n \quad = \quad n+m$

(1) $\langle n+m\rangle\ (a+1) \quad = \quad a+1+m$

(2) $\langle n+m\rangle\ m \quad = \quad m+m$

We went from (0) to (1) and from (0) to (2) by means of ordinary instantiations, obeying all the rules. But how did we get from the Scope Law to (0)? All occurrences of x, both local and nonlocal, were replaced by n, contrary to the rules. And when the first E was replaced by $n+m$, an n was placed in a context where it became local, again contrary to the rules.

There are two kinds of variables: syntactic and semantic. Or if you prefer, we can say equivalently that there are two kinds of substitution: syntactic and semantic. Syntactic substitution (substitution for a syntactic variable) ignores scope; semantic substitution (substitution for a semantic variable), also called instantiation, respects scope; that is the difference. If we do not have local scope, the two kinds are the same. If we do have local scope, we must have the scope-respecting substitution, for that is the reason we have local scope. The Scope Law expresses scope-respecting substitution, but to use the law we must use the scope-ignoring substitution. The scope-ignoring substitution is limited to this one use only. All variables are

semantic, all substitutions are scope-respecting, with the single exception of the law that gives us scope.

5 Extension and Renaming

The Extension Law looks very similar to the Scope Law.

Scope Law	$\langle x\mid E \mid x\rangle\, x$	=	E
Extension Law	$\langle x\mid Ex \mid x\rangle$	=	E

The scope brackets are really just an alternative to lambda notation. The Scope Law is a remarkable version of Church's β-rule; the Extension Law is the ordinary version of his η-rule; missing is his α-rule for renaming local variables. One reason we need renaming is so that instantiation will not place a nonlocal variable where it will appear to be local. Without any more awkward English, we have renaming as a consequence of the Scope and Extension Laws. Instantiate the Extension Law by replacing E by $\langle y\mid Ey \mid y\rangle$ to obtain

$$\langle x\mid \langle y\mid Ey \mid y\rangle x \mid x\rangle \quad = \quad \langle y\mid Ey \mid y\rangle$$

On the left side, the inner scope can be applied to its argument to obtain

$$\langle x\mid Ex \mid x\rangle \quad = \quad \langle y\mid Ey \mid y\rangle$$

This equates scopes with different local variables.

Other ways of introducing a local variable, such as quantifiers, can all be treated uniformly as operators on functions, so we need not consider them separately.

6 Function and Domain

We next present functions, but first we introduce two operators: colon and arrow. The colon (inclusion) is a reflexive, transitive, antisymmetric operator with a boolean result. For example,

$$3\colon 3 \qquad\qquad 3\colon nat$$

both have value $\top$. For details see [0] or [1]. The arrow (guarded expression) requires a boolean to its left; for example $x<5 \to 7$ (if x is less than five then seven). Its law $\top \to x \;=\; x$ says that if the guard is $\top$, we can drop it; otherwise we must carry it around. When simplifying the right operand of a guarded expression, we can assume the left operand has value $\top$ for the same reason that we can assume an antecedent has value $\top$ when simplifying the consequent of an implication. For details see [3].

A function is a scope whose operand is a guarded expression whose left operand is an inclusion whose left operand is the local variable. For example,

$$\langle n\mid n\colon nat \to n+m \mid n\rangle$$

or, more briefly,

$$\langle n\colon nat \to n+m\rangle$$

Starting with the Scope Law, we can syntactically (ignoring the scope rules) replace x with n and E with $n\colon nat \to n+m$ to obtain

$\langle n: nat \to n+m \rangle\ n \quad = \quad n: nat \to n+m$

Now we instantiate by replacing n with 3 (respecting the scope rules):

$\langle n: nat \to n+m \rangle\ 3 \quad = \quad 3: nat \to 3+m$

Then $3: nat$ is simplified to $\top$, and $\top \to 3+m$ is simplified to $3+m$, and the Scope Law has told us the result of applying $\langle n: nat \to n+m \rangle$ to 3:

$\langle n: nat \to n+m \rangle\ 3 \quad = \quad 3+m$

The domain of a function is obtained by the domain operator Δ. Here is the law:

Simple Domain Law $\quad \Delta\langle x: D \to Rx \rangle \quad = \quad D$

We can look at a function as a sort of binary tree data type, with constructor and destructors as follows:

Constructor	$\langle x: \quad \to \quad \rangle$			
Destructors	$\Delta\langle x: D \to Rx \rangle$	$=$	D	
	$\langle x: D \to R \rangle\ x$	$=$	R	(if $x: D$)

As simple as it is, the Simple Domain Law has a problem. Suppose we add the reasonable law

$a \to (b \to x) \ = \ a \wedge b \to x$

which gathers guards into a conjunction. Then we have an inconsistency. The Simple Domain Law does not look deeply enough into the function. So define Γx (the guard of x) as follows:

Γx if x is not guarded

$\Gamma(a \to x) \quad = \quad a \wedge \Gamma x$

The phrase "if x is not guarded" is the kind of informal mathematics that often passes without comment. In this essay, it must be noted that the phrase is hopelessly inadequate: no arrow appears in fx, yet its guard may not be $\top$. Perhaps Γ has to be defined by cases over the syntax of expressions. But pressing bravely onward, we can now define the domain of a function by means of the temporarily named

New Domain Law $\quad x: \Delta f \quad = \quad \Gamma(fx)$

If we have a guard operator, perhaps we don't want a domain operator; they are performing approximately the same job.

The Simple Domain Law has another limitation: it can only be instantiated by domains that do not mention x (otherwise we place a nonlocal variable where it appears to be local). When the function was defined, there was no prohibition against a domain that mentions the local variable. If the domain does mention the local variable, how do we make sense of it? Is this freedom useful? Is it harmful?

The function $\langle x: Dx \to Rx \rangle$ introduces a local variable x, and it also introduces a local law $x: Dx$ which can be used within the body Rx. A law is just a boolean expression that has been assigned the value $\top$; there is no reason why a law shouldn't mention x several times. For example,

$\langle x: x^2 \to x+1 \rangle$

introduces local variable x, and local law $x: x^2$, which says that x is included

among its squares. In other words, x might be 0 or 1, so that is the domain. The New Domain Law works for $\langle x{:}\ x^2 \to x{+}1\rangle$; it says the domain is the solutions of $x{:}\ x^2$.

If the law introduced by a function has no solutions, as in

$\langle x{:}\ x{+}1 \to x{+}1\rangle$

we have a function with an empty domain. Application of such a function results in a guarded expression with guard $\bot$, which cannot be eliminated. This guard protects us from using the result, so there is no harm. Far more dangerous are the functions whose laws have too many solutions; for example

$\langle x{:}\ x \to x{+}1\rangle$

If this freedom leads inevitably to Russell's paradox, we may have to impose the restriction that the domain must not mention the local variable, making the domains predicative, creating a Russell type hierarchy for functions. But there is some hope that we may not need to impose that restriction; we have successfully interpreted recursive data type definitions, and we no longer live in the Russell hierarchy for sets.

7 Conclusion

When we explain a formalism, the most basic rules of the game need to be explained in natural language, after which the rest can be presented formally as laws. If we are building a calculator (or prover), the natural language parts become hard-coded program, while the laws can be data in an easily changed table. It is desirable to be as complete as possible, leaving nothing implicit, and yet to keep the natural language to a minimum, and to keep it as independent as possible of the formalism being automated. Up to scopes, we managed to be independent of all types and operators — even booleans. (This independence was motivated by work on Unified Algebra [1], in which the booleans and numbers are unified.)

To explain scope, one might take a global view, or a local view. In the global view, scopes are considered to be a syntactic convenience. They are explained away by renaming all variables uniquely, then flattening. This makes the meaning of an expression depend on its context; we must first have an entire expression before we can work with any of its subexpressions. In this essay, I have taken the local view, in which the meaning of an expression is independent of its context. Scopes are explained by explaining how to cross a scope boundary, without any need to know what other scopes it is nested within.

Scope formation is by means of an operator with an inverse. The Scope Law provides a notation for ordinary, scope-respecting substitution. All substitutions are scope-respecting, with the sole exception of the instantiation of the Scope Law. Even variable renaming turns out to be a consequence of the Scope and Extension Laws.

The essay finishes with the highly speculative suggestion that a function is just a scope applied to a guarded expression, and that the domain expression in a function can mention the local variable. This suggestion may prove fruitless, or it may be as useful as dependent types; it is too soon to tell.

Acknowledgments

I thank Dimitrie Paun for discussions leading to this paper; the idea to couple syntactic substitution with the Scope Law was his [4]. I thank Michel Sintzoff, Jim Horning, and Alberto Petorossi for suggestions. I thank IFIP Working Groups 2.1 and 2.3 for being my research fora.

References

0. E.C.R.Hehner: *a Practical Theory of Programming*, Springer, 1993
1. E.C.R.Hehner: Unified Algebra, www.cs.toronto.edu/~hehner/UA.pdf, 1999
2. J.R.Hindley, J.P.Seldin: *Introduction to Combinators and λ-Calculus*, Cambridge University Press, Cambridge, 1986
3. R.Hoogerwoord: *the Design of Functional Programs: a Calculational Approach*, PhD thesis, Technische Universiteit Eindhoven, 1989
4. D.Paun: *Closure under Stuttering in Temporal Formulas*, MSc thesis, University of Toronto, 1999