From dy/dx to `[]`P : a matter of notation

Stuart F. Allen
Cornell University

Abstract

An analysis is given of the conventional $\frac{dy}{dx}$ notation for derivatives that explains it as a notational abbreviation for expressions using the simpler binding structure standard in modern formalizations. The Nuprl display system was used to implement examples of such notation.

It turns out that the same methods can be used to explain conventional modal logic notations. We construe necessity as a first-order quantifier, in a well known way, then explain standard modal notation as a way simply to display these formulas of a non-modal logic.

We contrast the method with the interpretation of necessity as a sentential operator, and also with higher-order interpretations that have been used to interpret temporal logic in HOL. The methods are then applied to a simple first-order temporal logic. The intention is that the user can work in this notation interactively, not just produce it for printing.

The methods to be discussed here for formalizing a few mathematical and logical concepts are *already* well known, or are small variations on well known methods, and are *not* the true subject of this paper. This paper is about notational enhancements for exploiting those methods, and may also serve as an explanation for some notations that are conventional, but do not obviously conform to the simpler syntax and semantics of current-day computerized formal mathematics.

We apply a particular combination of notational devices to a few examples, revealing their notational similarity. We start with Leibniz's notation for derivatives, $\frac{dy}{dx}$, and end with first-order temporal logic for programs. These notational methods have been made precise, and implemented in the Nuprl proof development system,[1] where they are meant for use as working notation. These examples were developed within it, although almost none of the mathematics for which these notations were implemented has been carried out in Nuprl.

The basic idea: How $\frac{dy}{dx}$ works.

Suppose `Deriv(x. e(x) ; a)` is a binding operator used to stand for the derivative, at a, of the function denoted by $e(x)$ in variable x.[2] So, for example,

[1]The author was the principal designer of the current (since 1991) Nuprl display system and editing primitives; Richard Eaton implemented them and provided supplementary design. The Nuprl project[6, 8] is based at Cornell, and is directed by Robert Constable. See www.cs.cornell.edu/Info/Projects/NuPrl .

[2]Dummett presents such a notation as the form adhering to Frege's notational demand that the sign for a function occur only in application to its arguments.[5]
Of course, another useful way of denoting the derivative is with a non-binding one-place operator

```
    Deriv(f) == λa.Deriv(x. f(x) ; a)
```

or if we take `Deriv(f)` as more basic we may define

```
    Deriv(x. e(x) ; a) == Deriv(λx.e(x))(a)
```

Each is a composition of the other with function application and lambda. These are perfectly compatible ways of denoting the derivative, each circumstance determining which is simpler to use. See the section below (p6) on "lifting".

```
∀b:ℝ. Deriv(x. x•x+b•x ; 3) = 6+b
```

or more generally,

```
∀b,a:ℝ. Deriv(x. x•x+b•x ; a) = 2•a+b
```

or equivalently, and this is the *key move*, by changing bound variable "a" to "x",

```
∀b,x:ℝ. Deriv(x. x•x+b•x ; x) = 2•x+b
```

At this point, we notice that the standard Leibniz style notation for derivatives acts a lot like the last notation; familiarly,

```
∀b,x:ℝ. d(x•x+b•x)/dx = 2•x+b
```

Construing `d(e(x))/dx` as a mere notational abbreviation for `Deriv(x. e(x) ; x)` makes plain both the dependency of the whole expression on "x", and the use of "x" simultaneously to indicate the argument place in the expression for the function being differentiated.

Although there are other more complicated usages of the "d/dx" notation, nevertheless, this one is fairly common, and explanation along these lines may alleviate confusion about how the notation works semantically. We take the key characteristic of this usage to be simply that the binding variable is also used as the other (non-function) argument.

Another familiar kind of construction from mathematical vernacular, which one might explain similarly, is exemplified by:

x is the unique integer such that $P(x)$

which we could take as simply a way of displaying a term:

u is the unique integer x such that $P(x)$

in the special case that u is x, i.e., the binding variable is also the free variable.

The Nuprl display system allows such display methods to be stipulated for terms of a general purpose syntax of (possibly binding) operators. Below we shall apply this notational method more elaborately to modal and first-order temporal logic, but first let's digress a bit and give some background on the system in which the notation is implemented.

Nuprl Term Structure

All the expressions in this paper using tt font were produced by Nuprl's display system. This is the notation that the Nuprl user sees while editing the underlying terms with a structure editor which does essentially no parsing; the forms of display are not inherent in these terms and the display forms may be changed at any time.

The terms of Nuprl are iterated operators of various arities; one specifies for each subterm which variables become bound. The concepts of free and bound variable, and capture-avoiding substitution are then the usual ones. A Term is essentially a 5-tuple $< op, n, t, k, x >$ where, treating the class Op of "operator names" and the class Var of variables abstractly,

- $op \in$ Op
- $n \in \mathbf{N}$, indicating the number of places for immediate subterms
- $t \in \{1..n\} \rightarrow$ Term, indicating the immediate subterms
- $k \in \{1..n\} \rightarrow \mathbf{N}$, with k_i indicating the number of binding variables that may become bound in the i-th subterm.
- $x \in \Pi i : \{1..n\}.\ \{1..k_i\} \rightarrow$ Var, with $x_{i,j}$ indicating the j-th binding variable for the i-th subterm.

For our purposes, we can assume that Op is a sequence of one or more strings, numbers, etc. Usually the Op of a term is just a single identifier.

There are no further restrictions on term structure. How terms are displayed in Nuprl is not inherent in either the structure of terms or the definitions of constants and operators; display is specified separately. Operations for term editing act directly on these structures[3], modulated by the display forms in force at the time.

When we can't think of a better way to display a term, we usually just write the Op followed by the subterms, if any, and prefix each subterm by the binding variables for that place, if any. So, $op(t_1; u.t_2)$ would be the Term $< op, 2, t, k, x >$, where $k_1 = 0$, $k_2 = 1$, and $x_{2,1} = u$.

For example, `all(A; x.B(x))` is used in the standard Nuprl libraries to represent instances of the universal quantifier, where `A` is the domain of quantification, and `B(x)` is the formula being quantified; the standard display is `∀x:A. B(x)` .

Sometimes other values are included in the operator as a way of getting those values into the term structure as literals. When we don't have a better way of displaying them, these extra values are usually just written directly after the first identifier, in braces. For example, the basic numeric literals used in Nuprl are exemplified by "`natural_number{2}` ", which has no immediate subterms, and is normally displayed simply as `2`.

Normally, such a discussion of term structure would lead to a description of operator definitions, and indeed we'll see some examples below, but our concern here is really with how such terms are displayed.

Displaying Terms

At any point during a Nuprl session, there is a set of named objects of various kinds, mostly loaded from library files. In addition to proofs, operator definitions, inference rules, program code, and documentation objects, Nuprl libraries contain objects that specify how to display terms.

A specification includes a term such as `deriv(<x>.<e>; <a>)` , called the "display model," which may contain schematic variables such as `<e>`, in place of various parts. The display spec is applied by matching for these schematic variables, then instantiating into a "formatting command" that is also part of the specification; formatting commands specify what characters to display, as well as break/margin control similar to Oppen's pretty printer methods[14].

Here are the display specifications used above to display the derivative. The main things to observe are the display models; the reader need not really understand the rest of the specification, but we show it simply to demonstrate that it is a fairly simple schematic method. Here is the display specification that generates the non-d/dx display form used above:

```
Model: deriv(<x>.<e>; <a>)
  Deriv(<x:var>. <e:real>
                ←MARGIN ;[ ]
        <a:real>)
        ←MARGIN←SOFT
```

Given a particular term, there may be several ways to display it – there may be several specifications having display models which match the term. In addition to this display spec, we have added another one for our special case:

[3]Structure editing of terms, via grammars for those terms, was pioneered by Teitelbaum[16, 15]

```
Model: deriv(<x>.<e> ; <x>)
Attrs: *Open form*; =apply.standard
  d<e:real:(<self), AddIparms(<x>)>
   ⟵MARGIN[]
  /d<x:var>
  ⟵MARGIN⟵SOFT
```

Notice how the display model indicates the special circumstance of applicability, namely, that the same variable name must be used both as the binding variable, and as the second argument to the operator. (During substitution, Nuprl usually attempts to retain variable names, as well as identity and difference between variables bound by the same operator occurrence.) The "AddIparms" element will become significant for our discussion, and will be addressed below.

When a term is displayed, it runs through the display forms in a specific order trying to find one that may be applied to the term in question. As a result, the term `d(x·x+b·x)/dx` is normally displayed as such rather than as `Deriv(x. x·x+b·x ; x)` , although the user may temporarily disqualify the d/dx form for whatever motive, such as finding the notation mysterious or ambiguous. The term `Deriv(x. x·x+b·x ; a)` , however, is simply ineligible for the d/dx form, and so the long form is used. Indeed, if "a" is substituted for free "x" in `d(x·x+b·x)/dx` , say in the course of a proof, then the d/dx form will be automatically abandoned in favor of `Deriv(x. x·x+b·x ; a)` . Or, utilizing a simple substitution operator defined by `e(x)|x=a == e(a)` ,

`d(x·x+b·x)/dx|x=a` rewrites by definition to `Deriv(x. x·x+b·x ; a)` .

Let us return to the "AddIparm" element in the display spec above, which has been attached to the formatting command for a subterm. It is rather common in informal practice to elide certain variables from expressions which nevertheless depend upon them, such as when the same variable is used repeatedly throughout a long argument or other discourse. Nuprl terms must include any variables they depend on, so these *implicit parameters* must be elided merely as as matter of display.

The recursive descent display algorithm has as one argument a set of variables considered to be implicit parameters. The display form in question stipulates that whatever variable of the instance matches the schematic variable `<x>` will be added to the implicit parameter set when the subterm is displayed.

It *is possible* to stipulate that a given display form is usable only if certain variables are in the implicit parameter set. To continue with our d/dx example, suppose we wish to work with functions that will normally depend on the variable x. We may define a special function-application form that is intended for use mainly with x as its argument, and whose display elides x when it's an implicit parameter, but shows it otherwise. Here's an example of using this apply form with function y.

```
∀b:ℝ,y:ℝ⟶ℝ. (∀x:ℝ. y(x) = x·x+b·x) ⇒ ∀x:ℝ. dy/dx = 2·x+b
```

(Note the abbreviation of the iterated quantifier. The display system has several iteration-related capabilities.) If we alpha-convert `∀x:ℝ. dy/dx = 2·x+b` , changing all binding `x`'s to `v`, say, we get `∀v:ℝ. dy(v)/dv = 2·v+b` , in which the argument to `y` now stands revealed because it is not `x`. There is further discussion of calculus notation in [13].

To summarize, Nuprl has been used to explain certain d/dx notations, *not* by extending the basic term structure and altering concepts of binding, but rather by construing them simply as notational abbreviations for certain forms of notation having the more commonly understood binding conventions, and explicitly containing the variables upon which they depend. In search of other applications for this device, let us turn our attention to the notations of Modal Logic.

Modal Logic: first-order necessity

Rather than proceeding directly to first-order temporal logic for programs, we begin more simply with modal logic under a possible-worlds semantics.
Nonstandard Nomenclature Warning: Our consideration of modal language will compare three ways that modality may be expressed, depending on whether necessity is taken as a propositional operator, a first-order quantifier, or an operation on proposition-valued functions. We assume that any modal logics we are interested in might have ordinary quantifiers, and perhaps higher-order functions; so we shall appropriate the adjectives *propositional*, *first-order*, and *second-order*, when applied to modal concepts, to indicate which of these three treatments of the modal operators is pertinent. We shall use the adverb "modally," to modify these three adjectives.

Let us use the term *modally propositional formula* in reference to the standard syntax of modal logic, in which the modal operators are sentential operators, and with the semantics defined with respect to possible worlds in the manner of Kripke[9]. Below, we assume the type `PW` is the type of all possible worlds, and (`W Pwrt W'`) means `W` is possible with respect to `W'`.

It is well known that (what we call here) modally propositional formulas, can be translated into a non-modal language.[3, 2, 1] The modal sentential operators are eliminated in favor of explicit quantification over possible worlds, and various properties and relations are extended to take a possible world as an extra parameter. The same goes for temporal logics and logics of tense. Thus, the definition

```
Nec(W. P(W) ; W') == ∀W:PW. (W Pwrt W') ⇒ P(W)
```

having as its definiens the result of the standard translation, may be regarded as a definition of the necessity operator. Let's call this operator *first-order* necessity, since it is works like a first-order quantifier.

Now, applying the method used above to explain d/dx, we stipulate that terms of the form `Nec(theWorld. <P> ; theWorld)` are to be displayed as `[]<P>` . Note that we are further restricting this method of display so that it applies *only* when the variable is exactly `theWorld`. Again, we shall make `theWorld` an implicit parameter in the display of the subterm `<P>`, in order to allow its suppression.

We can now characterize the *modally first-order formulas* of our language as those in which `theWorld` is the only variable, free or bound, that ranges over `PW` , and further, that `theWorld` is bound only by modal quantifiers, such as necessity. These formulas are very nearly those that result from the standard translations of modally propositional formulas into non-modal formulas. The differences are two: we use the first-order necessity operator instead of what it expands to by definition, and we completely limit the choice of possible world variable. So the actual standard translations are all the alpha-variants of our modally first-order formulas after eliminating the first-order necessity operator by expanding it according to its definition.

Next, let us say an operator is *first-order world-relative* when it has an argument place for expressions of type `PW` . When adding first-order modal operators such as necessity, it is not necessary to make all other operators world-relative as well; we just need to ascertain whenever a new operator's meaning depends on possible worlds, and make *it* world-relative.

For example, let `Person(W)` comprise the persons existing in world `W`, and when `theWorld` is implicit, display `Person(theWorld)` simply as `Person` . Further, for `W` ∈ `PW` and for all *possible* persons `x`, let "`x invented bifocals in W` " mean what it looks like. Then

```
¬[](∃ x:Person. x invented bifocals)
```

is simply

```
¬Nec(theWorld. ∃ x:Person(theWorld).
               x invented bifocals in theWorld ;
     theWorld)
```

but is more effectively displayed. More generally, if for every world-relative operator, the user has specified a display form which elides `theWorld` when it is implicit, then modally first-order formulas will be displayed with ordinary modal notation.

It should be emphasized here, that *this* is not a translation from modally propositional to modally first-order formulas – there is only the one non-modal language here, and we are simply providing an alternative explanation of the standard modal *notation* as a combination of notational devices for the non-modal language. When one constructs proofs in Nuprl, the display forms are not visible to the inference engine, so the usual tactics apply to these formulas.

If in the course of using modally first-order formulas, the user should generate a formula that is not, this becomes evident from the disappearance of the standard modal notation. For example, even rewriting a first-order necessity operator by its definition results in a term that is not modally first-order, since avoiding capture of `theWorld` forces a change of bound variable.

```
[][](∃ x:Person. x invented bifocals)
```

becomes

```
∀W:PW. (W Pwrt theWorld) ⇒
Nec(theWorld. ∃ x:Person(theWorld). x invented bifocals in theWorld ; W)
```

Even though the second occurrence of the necessity operator remains, it is no longer a modally first-order formula because it is no longer necessity with respect to `theWorld`; and observe that no alpha conversion of the whole can restore it. Roughly put, denoting a relation between two possible worlds, gets you kicked out of the modal notation. Of course, you can still proceed with the non-modal formulas, and maybe you'll recover modal formulas down the road. Either way, it will be easy to discern in the display of the terms.
Before moving on to temporal logic, we shall review how to construe modal notation in non-modal higher-order logic.

Lifting: second-order necessity

Here we examine an already-existing method for embedding modality in a non-modal higher-order language. The HOL system[7] has been host to embeddings of Lamport's Temporal Logic of Actions (TLA)[12]. Exposition may be found in [11, 17, 4]. Let us examine how their methods apply to the simpler modal logic we have been using.

As it was with the derivative,[2] we may also define

```
NEC(F) == λW'.Nec(W. F(W) ; W')
```

which may even be used in concert with the first-order necessity operator we have been discussing.[4] These two operators denote the same function, in a sense, by different methods. Their use differs by how one applies them in order to form propositions. Let us call this operator *second-order* necessity, since it is most likely to be useful in a higher-order language.

In the previous section, modal notation was explained by adopting notational conventions for eliding `theWorld` from the display of world-relative operators. Any other constants and operators were simply left intact with the usual meanings and notations. Observe, for example, that in
`¬[](1 = 1 ∨ (∃ x:Person. x invented bifocals))` , the operators for negation, equality, existence, disjunction, and 1 are the ordinary operators, with no unusual interpretation.

In contrast, interpreting necessity as the second-order necessity operator `NEC(F)` entails interpreting the propositional formulas generally as *functions* on possible worlds. Adding an operator for bifocal inventing would go like this. Who there is, and bifocal inventing, are world dependent, so `PERSON ∈ PW⟶Type` is a simple function-valued constant, having no built-in argument places. The operator "`x INVENTED BIFOCALS` " has only the one built-in argument

[4]Again, we could have defined these operators in the other order: `Nec(W. Q(W) ; W') == NEC(λW.Q(W))(W')`

place, and for any *possible* person `x`, `(x INVENTED BIFOCALS)` $\in$ `PW`$\longrightarrow$`Prop` so the possible world must be supplied through function application. Call such operators *second-order* world-relative.

Let us say a *modally second-order* formula is an expression of type `PW`$\longrightarrow$`Prop` , having *no* variables ranging over `PW`. So, just as expanding the definition of first-order necessity within a modally first-order formula destroys its status as such by forcing the use of two different `PW` variables, expanding the definition of second-order necessity within a modally second-order formula destroys is status as such by forcing the use of a binding `PW` variable.

Now we come to what may be the main drawback of this perfectly legitimate higher-order method of interpreting modality. In order to build complex formulas to which we shall apply modal necessity, we must *lift* all the non-world-relative operators of the language, which means to define variations of them that operate on functions from `PW`. For example, we must define a lifted version of negation, perhaps by,

```
NOT(F) == λW.¬F(W)
```

Then, by stipulating that the second-order modal operators, the lifted operators, and the remaining second-order world-relative operators are displayed the same as their modally propositional counterparts, we achieve yet another alternative explanation of standard modal notation. Then, `NEC(NOT(x INVENTED BIFOCALS))` would display as `[]¬(x invented bifocals)` . Again, this is essentially the method of [11, 17, 4].

Of course, in a higher-order language, both first-order and second-order necessity can coexist. If all the first-order world-relative operators have been supplied with second-order relative counterparts, and all the non-world-relative operators have lifted counterparts, then one can translate between the modally first- and second-order formulas. To translate a modally second-order formula, form its application to `theWorld`, then do the beta-conversions. Reverse the process to translate back.

So, if you want to use a first-order logic, or otherwise avoid lifting all your operators, then use first-order necessity.

Temporal Logic: addresses

Now we move to temporal logic, which is a modal logic where the possible worlds are times. The interesting additional feature is a class of "variables" whose values are taken relative to a state that varies with time. Let us use the terms *temporal* and *temporally* instead of *modal* and *modally*. Our purpose here will be to give an alternative explanation of temporal *notation* as a display form for *temporally first-order formulas.*

Let's take the first-order temporal logic given by Kröger[10] as our standard. Its temporal operators such as *nexttime*, *henceforth*, and *atnext* are applied to propositions, so we may say this is a *temporally propositional* logic. Time is taken to be the natural numbers, with successor being the next time.

Each "variable" is classified as "local" or "global." The globals are ordinary logical variables, but the locals cannot become bound, and their values are time-dependent. This is effected by a parameter to the semantics which is a function from time and locals to values. We shall avoid this nomenclature, reserving the term "variable" for logical variables susceptible of quantification-as-usual, and instead of "local variable" we shall say "address" in order to emphasize the connection to program state.

Let's begin. The type of addresses will be `Addr`, which we may assume here to be some class of identifiers. Address literals will be distinguished by some characteristic form such as `addr{XYZ}`, which we'll simply display as `XYZ` in the same way as done above for numeric literals. Here are our first-order temporal operators:

```
NextTime(t. P(t) ; r) == P(r+1)
HenceForth(t. P(t) ; r) == ∀n:{r...}. P(n)
AtNext(t. Q(t), P(t) ; r)
```

```
== ∀n:{r+1...}. (∀j:{r+1...}. Q(j) ⇒ n≤j) ⇒ Q(n) ⇒ P(n)
```

We define `{i...} == {j:ℤ| i≤j}` . The AtNext operator says that the next time, if any, that `Q` holds, then so does `P`. Also note that when the same variable binds in two subterms, we don't need to display the binding occurrence twice. Now we apply the display device for first-order modal operators to these first-order temporal operators, using `theTime` as the distinguished variable, getting these three standard temporal notations:

```
o<P> ; []<P> ; <P> atnext <Q>
```

Now for the time-dependent values at addresses. The type of state sequences, or *behaviors*, is `ℕ⟶Addr⟶T` for some type `T` of values. We define an operator for the value of an address `a`, at a time `t`, for a behavior `s` as `a{s@t} == s(t,a)` and we stipulate that the display of `<a>{theBeh@theTime}` when `theBeh` and `theTime` are implicit, shall just be `<a>`. (Of course, the user will need help from the editor to effectively use such "invisible" operators, to make their detection and manipulation easy.) Thus, the formula

```
∀x:T. x = a ⇒ o(x = a)
```

where `a` is an address literal, is just an informative display of

```
∀x:T. x = a{theBeh@theTime} ⇒
NextTime(theTime. x = a{theBeh@theTime} ;
          theTime)
```

This temporal formula is used by Kröger as part of a demonstration that you are not free to perform all-elimination on arbitrary expressions. In particular you must make sure that no address expressions are introduced into the scope of a temporal operator. One must not, say for address literal `b`, infer `b = a ⇒ o(b = a)` from `∀x:T. x = a ⇒ o(x = a)` .

The corresponding phenomenon in the temporally first-order interpretation, where one may of course do the substitution without such a restriction, is this:
we are actually substituting `b{theBeh@theTime}` for free `x`, which forces a change of bound variable in the first-order NextTime operator. So, the substitution works, but we have been kicked out of the temporal notation for trying to relate two Times, `t` and `theTime`, in the scope of the temporal operator, ending up with

```
b = a ⇒ NextTime(t. b = a{theBeh@t} ; theTime)
```

(Remember, `theTime` is implicitly attached to `b` here.)

Let us conclude with a remark about Lamport's TLA[12]. He separates it into the *simple* TLA and the *full* TLA. In [17] there is a straightforward treatment of the simple TLA in HOL using the higher-order method; there is quite a lot of operator lifting, and several kinds of it. With the first-order methods we have been describing, the simple TLA is quite tractable, and there is no lifting (although there is one coercion from "actions" to "formulas" of TLA). There is no space to give it here, but with a copy of Lamport's Figure 4 of [12] in hand, you may well find that our methods apply fairly directly, though non-trivially. Suggestions if you want to try it: Use three special variables, two for states and one for behaviors, say theState, theNextState, and theBeh. Use the TLA "prime" notation f' simply as a display form for "let theState = theNextState in f", where f may contain theState free. Let the notation be your guide.

The difficulty with extending our temporally first-order explanation of simple TLA to full TLA is that we treat state-dependent "variables" as addresses, and this doesn't help us to explain Lamport's use of these "variables" (which he calls *flexible variables*) for existential quantification, which is key to the full TLA.

Thanks

Bob Constable and Jim Caldwell discussed these ideas with me, and commented on drafts. I appreciate their generous efforts and the resulting improvements in my presentation. My thanks also to the referees, who provided useful criticisms.

References

[1] Martín Abadi and Zohar Manna. Nonclausal deduction in first-order temporal logic. *Journal of the ACM*, 37(2):279–317, 1990.

[2] J. F. A. K. van Benthem. *The logic of time : a model-theoretic investigation into the varieties of temporal ontology and temporal discourse*, volume 156 of *Synthese library*. D. Reidel Pub. Co., Dordrecht, Holland, 1983.

[3] Brian F. Chellas. *Modal logic : an introduction.* Cambridge University Press, 1980.

[4] Ching-Tsun Chou. Predicates, temporal logic, and simulations. In *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop, HUG'93*, volume 780 of *Lecture Notes in Computer Science*, pages 310–323. Springer-Verlag, 1994.

[5] Michael Dummett. *Frege: philosophy of language, Second Edition.* Harvard University Press, 1981.

[6] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Development System.* Prentice-Hall, NJ, 1986.

[7] Michael J. C. Gordon and Tom F. Melham. *Introduction to HOL.* University Press, Cambridge, 1993.

[8] Paul Jackson. *The Nuprl Proof Development System, Version 4.2 Reference Manual and User's Guide.* Cornell University, Ithaca, NY, January 1996.

[9] Saul Kripke. A completeness theorem in modal logic. *Journal of Symbolic Logic*, 24:1–14, 1959.

[10] Fred Kröger. *Temporal Logic of Programs.* Springer-Verlag, 1987.

[11] Thomas Långbacka. A HOL formalisation of the temporal logic of actions. In *Higher Order Logic Theorem Proving and its Applications. 7th International Workshop*, volume 859 of *Lecture Notes in Computer Science.* Springer-Verlag, 1994.

[12] Leslie Lamport. The temporal logic of actions. Systems Research Center 79, Digital Equipment Corp., Palo Alto, CA, December 1991.

[13] Conal Mannion and Stuart Allen. A notation for computer aided mathematics. Technical report, Cornell University, Ithaca, NY, June 1994.

[14] Derek C. Oppen. Prettyprinting. *ACM Trans. on Prog. Lang. and Systems*, 2(4):465–483, October 1980.

[15] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual.* Springer-Verlag, New York, third edition, 1988.

[16] Tim Teitelbaum and Thomas W. Reps. The Cornell Program Synthesizer: a syntax–directed programming env ironment. *Comm. Assoc. Comput. Mach.*, 24(9):563–73, 1981.

[17] Joakim von Wright. Mechanising the temporal logic of actions in HOL. In *1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 155–159. IEEE Computer Society Press, 1992, 1992.