# Expressing and Implementing the Computational Content Implicit in Smullyan's Account of Boolean Valuations*

Stuart Allen Robert Constable Matthew Fluet
Department of Computer Science,
Cornell University,
Ithaca, NY 14853

March 29, 2004

**Abstract**

In Smullyan's classic book, *First-Order Logic* [21], the notion of a *Boolean valuation* is central in motivating his analytical tableau proof system. Smullyan shows that these valuations are unique if they exist, and then he sketches an existence proof. In addition he suggests a possible computational procedure for finding a Boolean valuation, but it is not related to to the existence proof.

A computer scientist would like to see the obvious explicit recursive algorithm for evaluating propositional formulas and a demonstration that the algorithm has the properties of a Boolean valuation. Ideally, the algorithm would be derived from the existence proof. It turns out to be unexpectedly difficult to find a natural existence proof from which the algorithm can be extracted, and it turns out that the implicit computational content of Smullyan's argument is not found where one might expect it.

We show that using the notion of a very dependent function type, it is possible to specify the Boolean valuation and prove its existence constructively so that the natural recursive algorithm is extracted and is known to have the mathematically required properties by virtue of its construction. We illustrate all of these points using the Nuprl proof development system [9].

# 1 Introduction

## 1.1 Computational Aspects of Logic

Propositional logic can be developed in a purely noncomputational way, stressing the completeness theorem and various properties of truth sets, satisfiability, and compactness. Often these topics are cast in a way that motivates them for the corresponding treatment of predicate logic where they cannot be treated computationally. Raymond M. Smullyan adopts this purely mathematical, noncomputational approach [21]. He stresses mathematical elegance and connections to natural proof styles.

Propositional logic can also be presented in a completely computational way in which the formulas are given by a recursive data type, evaluation is a recursive function over this type, and the principle theorem is that validity and satisfiability of propositional formulas is decidable [6]. Indeed, the Satisfiability problem (SAT) in computing theory deals with deep computational questions about the algorithms arising in this approach.

We contrast these approaches as "pure" and "computational." We are interested in exploring how the two approaches can be simultaneously presented, thus providing the best of each. We show that this can be done, but not as straightforwardly as one might expect.

*This material is based upon work supported in part by the National Science Foundation under Grant No. 9812360.

### 1.2 Algorithm Development

Computer scientists are not only interested in developing fast algorithms for a problem, they also want to know the properties of these algorithms, including knowing what problems they solve.

One interesting way to develop an algorithm and its properties together is to synthesize the algorithm from an existence proof [8, 2, 18, 19, 10, 12, 16, 17]. This is a common practice in computational mathematics, and computer scientists have automated the practice for a large class of formal theories. There are systems such as Coq [1], Nuprl [9], and MetaPRL [15, 13], which automatically synthesize algorithms from proofs. The process is called the *extraction* of an algorithm [2]. Programs developed this way are known to be *correct-by-construction* [12, 20].

The practice of algorithm synthesis from constructive proofs is derived from a more general idea that proofs can be viewed as programs [8, 3]; that is, they have computational content which can be systematically extracted.

The method of programming with proofs has achieved considerable success as a means of reliable programming [5]. Several sophisticated algorithms have been developed in this way [7, 11], including protocols for distributed systems [4].

Experience has shown that proofs have unexpected computational content, and it has shown that sometimes the expected content is deeply hidden in the proof. This work will show both features.

## 2 Preliminaries

We cast our account of formulas and Boolean valuations in the Nuprl proof development environment [9]. All of the definitions (`ABS`), theorem statements (`STM`), proofs (`PRF`), and rules (`RULE`) are taken directly from the system.

### 2.1 Formulas

Our account of formulas is most closely related to Smullyan's final scheme ("of a radically different sort") for defining formulas [21, p. 7], in that formulas are not strings to be parsed, but are recursively structured from their immediate subformulas:

`ABS: form`

```
form() ==
  rec(form.$pvar:$v:ℕ +
           $pnot:$U:form +
           $pand:$U:form × $V:form +
           $por:$U:form × $V:form +
           $pimp:$U:form × $V:form)
```

The notation `$l:t` tags a token parameter with a term; for example, `pnot` labels the term `$U:form` and `U` labels the term `form`. These tokens are used by abstract syntax tree tactics to label case analyses, invoke appropriate folding/unfolding of definitions, and to generate names for introduced variables.

A particularly useful definition is `form_cases` which simultaneously discriminates and destructs an element of the `form()` type:

`ABS: form_cases`

```
form_cases(z;
           v.pvar_f[v];
           U.pnot_f[U];
           U,V.pand_f[U; V];
           U,V.por_f[U; V];
           U,V.pimp_f[U; V]) ==
  case z
```

```
  of inl(z) => pvar_f[z]
   | inr(z) => case z
               of inl(z) => pnot_f[z]
                | inr(z) => case z
                            of inl(z) => pand_f[z.1; z.2]
                             | inr(z) => case z
                                         of inl(z) => por_f[z.1; z.2]
                                          | inr(z) => pimp_f[z.1; z.2]
```

Definitions for injections into the `form()` type, tactics for induction (structural, size, and depth), and decidability theorems for equality and membership in the `form()` type are all generated automatically from a succinct description of the desired abstract syntax.

## 2.2 Subformulas

With this definition of the type of formulas in place, we are able to follow Smullyan fairly closely in his definition of *immediate subformula* and *subformula*, which is:

> The notion of *immediate subformula* is given explicitly by the conditions:
>
> $I_0$**:** Propositional variables have no immediate subformulas.
>
> $I_1$**:** $\neg X$ has $X$ as an immediate subformula and no others.
>
> $I_2 - I_4$**:** The formulas $X \wedge Y$, $X \vee Y$, $X \Rightarrow Y$ have $X, Y$ as immediate subformulas and no others.
>
> The notion of *subformula* is implicitly defined by the rules:
>
> $S_1$**:** If $X$ is an immediate subformual of $Y$, or if $X$ is identical with $Y$, then $X$ is a subformula of $Y$.
>
> $S_2$**:** If $X$ is a subformula of $Y$ and $Y$ is a subformula of $Z$, then $X$ is a subformula of $Z$.
>
> The above implicit definition can be made explicit as follows: $Y$ is a subformla of $Z$ iff (i.e., if and only if) there exists a finite sequence starting with $Z$ and ending with $Y$ such that each term of the sequence except the first is an immediate subformla of the preceding term.[21, p. 8]

We formulate the immediate subformula concept as a relation between two formulas, satisfied when the first is an immediate subformula of the second:

```
ABS: IsImmedSubForm
```

```
IsImmedSubForm(X;Z) ==
  form_cases(Z;
             v.False;
             U.X = U ∈ form();
             U,V.(X = U ∈ form()) ∨ (X = V ∈ form());
             U,V.(X = U ∈ form()) ∨ (X = V ∈ form());
             U,V.(X = U ∈ form()) ∨ (X = V ∈ form()))
```

This slightly simplifies the use of the immediate subformula concept, avoiuding an unnecessary detour into membership in a list of immediate subformulas.

The implicit definition of *subformula* is difficult to analyze, as $S_2$ can be applied an arbitrary number of times by applying $S_1$ with equal formulas. Hence, we choose a definition that enforces a strict chain of immediate subformulas from `X` to `Z`:

ABS: IsSubForm_ml

```
IsSubForm(X;Z) ==r
  (X = Z ∈ form()) ∨
  ((¬(X = Z ∈ form())) ∧ (∃Y:form(). (IsImmedSubForm(Y;Z) c∧ IsSubForm(X;Y))))
```

We also define set types (i.e., comprehension types) corresponding to the sets of immediate sub- and sub-formulas of a given formula:

ABS: ImmedSubForm, SubForm

```
ImmedSubForm(X) ==  {A:form()| IsImmedSubForm(A;X)}
SubForm(X) ==  {A:form()| IsSubForm(A;X)}
```

## 2.3 Valuations, Extensions and Boolean Valuations

Smullyan defines *valuations* and *extensions* as follows:

> Now we consider, in addition to the formulas of propositional logic, a set $\{t, f\}$ of two *distinct* elements, $t$, $f$ as *truth-values*. For any set $S$ of formulas, by a *valuation* of $S$, we mean a function $v$ from $S$ into the set $\{t, f\}$ – i.e., a mapping which assigns to every element $X$ of $S$ one of the two values $t$, $f$.
>
> If $S_1$ is a subset of $S_2$ and if $v_1$, $v_2$ are respective valuations of $S_1$, $S_2$, then we say that $v_2$ is an *extension* of $v_1$ if $v_2$, $v_1$ agree on the smaller set $S_1$.[21, pp. 9–10]

For convenience, we identify Smullyan's set $\{t, f\}$ with the Nuprl type of Booleans, $\mathbb{B}$. There is little advantage to defining a specific abstraction for *valuations*; Nuprl's function type suffices. We therefore only define the `ValuationExtension` abstraction and prove that it is well-formed:

ABS: ValuationExtension

```
ValuationExtension(S1;f1;S2;f2) ==  ∀X:S1. f1 X = f2 X ∈ ℬ
```

STM: ValutionExtension_wf

```
∀S1:𝕌. ∀f1:S1 ⟶ ℬ. ∀S2:𝕌. ∀f2:S2 ⟶ ℬ.
  ((S1 ⊆r S2) ⇒ (ValuationExtension(S1;f1;S2;f2) ∈ ℙ))
```

Note that the subset relation is placed in the well-formedness theorem, rather than in the abstraction. This seems more in the spirit of Smullyan's definition, although the alternative could be used with little complication.

The definition of a Boolean valuation is Smullyan's first "formal" definition:

> Now we wish to consider valuations of the set $E$ of all formulas of propositional logic. We are not really interested in *all* valuations of $E$, but only in those which are "faithful" to the usual "truth-table" rules for the logical connectives. This idea we make precise in the following defintion.
>
> **Definition 1.** A valuation $v$ of $E$ is called a *Boolean* valuation if for every $X$, $Y$ in $E$, the following conditions hold:
>
> $B_1$**:** The formula $\neg X$ receives the gvalue $t$ if $X$ receives the value $f$ and $f$ if $X$ receives the value $t$.
>
> $B_2$**:** The formula $X \wedge Y$ receives the value $t$ if $X$, $Y$ both receive the value $t$, otherwise $X \wedge Y$ receives the value $f$.

$B_3$: The formula $X \vee Y$ receives the value $t$ if at least one of $X$, $Y$ receives the value $t$, otherwise $X \vee Y$ receives the value $f$.

$B_2$: The formula $X \Rightarrow Y$ receives the value $f$ if $X$, $Y$ receive the respective values $t$, $f$, otherwise $X \Rightarrow Y$ receives the value $t$.

This concludes our definition of a Boolean valuation. [21, p. 10]

(Note that the set $E$ corresponds to the `form()` type.) Rather than simply using the concept of a Boolean valuation for the set of all formulas, we relativize the concept by defining *partial Boolean valuation* as a valuation that satisfies $B_1$ – $B_4$ on all elements of a given subset of formulas:

ABS: PartialBooleanValuation

```
PartialBooleanValuation(S;f) ==
  ∀X:S.
    form_cases(X;
               v.True;
               Y.f X = ¬b(f Y) ∈ 𝔹;
               Y,Z.f X = (f Y) ∧b (f Z) ∈ 𝔹;
               Y,Z.f X = (f Y) ∨b (f Z) ∈ 𝔹;
               Y,Z.f X = f Y ⇒b (f Z) ∈ 𝔹)
```

Note that this definition does not make sense when a compound formula $X$ is a member of the set $S$ but it's immediate subformulas are not. Therefore, we define the predicate `DownClosed` on subsets of formulas, which is true exactly when all subformulas of a formula $X$ are members of the set $S$ whenever $X$ is a member of the set $S$:

ABS: DownClosed

```
DownClosedForm(S) ==  ∀X:S. (SubForm(X) ⊆r S)
```

The predicate is well-formed (i.e., makes sense) whenever $S$ is a type corresponding to a set formulas:

STM: DownClosed_wf

```
∀S:{S:𝕌| S ⊆r form()}. (DownClosedForm(S) ∈ ℙ)
```

With this definition in hand, we can state and prove the following well-formedness theorem for the `PartialBooleanValuation` abstraction:

STM: PartialBooleanValuation_wf

```
∀S:{S:𝕌| S ⊆r form()}.
  (DownClosedForm(S) ⇒ (∀f:S ⟶ 𝔹. (PartialBooleanValuation(S;f) ∈ ℙ)))
```

## 2.4 Existence and Uniqueness of Boolean Valuations

There are a number of ways of approaching the proof of the existence and uniqueness of a Boolean valuation on a single formula $Z$ given an interpretation $v_0$ of the variables of $Z$. However, our final goal will always be:

STM: Valuation Theorem

```
∀Z:form(). ∀v0:Var(SubForm(Z)) ⟶ 𝔹.∃!f:SubForm(Z) ⟶ 𝔹.
  ValuationExtension(Var(SubForm(Z));v0;SubForm(Z);f) ∧
  PartialBooleanValuation(SubForm(Z);f)
```

That is, we wish to show that for any formula $Z$ and interpretation $v_0$ of the variables of $Z$, there exists one and only one function $f$ assigning truth values to the subformulas of $Z$ such that $f$ is an extension of $v_0$ and $f$ is a partial Boolean valuation on subformulas of $Z$.

# 3 Four Approaches

As described above, we can consider both "pure" and "computational" approaches to the presentation of propositional logic. While the preliminaries given in the previous section are aimed towards a computation approach, they do not exclude the pure approach. In this section, we sketch four approaches to the proof of `Valuation Theorem` and consider their advantages and disadvantages. Principally, we are concerned with gauging the approaches along a number of different axes. Foremost, we would like an elegant proof, one that can be presented both formally and informally. Second, we would like to extract the computational content of the proof as an algorithm for evaluation propositional formulas, one that is recognizable as the obvious recursive algorithm. Third, we would like to minimize the effort required to complete the proof, measured loosely in terms of the number of inductive arguments and the complexity of the proofs.

We present these four approaches as the Mathematician, who embodies the "pure" approach, the Programmer, who embodies the computational approach without using extracts, the Formalist, who embodies the computational approach using extracts, and the Computer Scientist, who embodies the computational approach using very dependent function types.

## 3.1 The Mathematician

Smullyan adopts a purely mathematical, noncomputational approach to the proof of `Valuation Theorem`. In his text, he writes:

> Consider a single formula $X$ and an interpretation $v_0$ of $X$ – or for that matter any assignment $v_0$ of truth values to a set of propositional variables which includes at least all variables of $X$ (and possibly others). It is easily verified by induction on the degree of $X$ that there exists one and only one way of assigning truth values to all *subformulas* of $X$ such that the *atomic* subformulas of $X$ (which are propositional variables) are assigned the same truth values as under $v_0$, and such that the truth value of each *compound* subformula $Y$ of $X$ is determined from the truth values of the immediate subformulas of $Y$ by the truth-table rules $B_1 - B_4$. [We might think of the situation as first constructing a formation tree for $X$, then assigning truth values to the end points in accordance with the interpretation $v_0$, and then working our way up the tree, successively assigning truth values to the junction and simple points, in terms of truth values already assigned to their successors, in accordance with the truth-table rules]. In particular, $X$ being a subformula of itself receives a truth value under this assignment; if this value is *true* then we say that $X$ is true *under the interpretation* $v_0$, otherwise *false* under $v_0$. Thus we have now defined what it means for a formula $X$ to be true under an *interpretation*.[21, pp. 10–11]

While Smullyan's parenthetical remark ("[We might think ...]") appears to describe the obvious recursive algorithm for computing the evaluation of a propositional formula, it is far from clear that this procedure corresponds in any meaningful way to the main proof. In particular, Smullyan suggests proving `Valuation Theorem` by induction on $X$; in a formal proof, we proceed as follows:

```
PRF: BySmullyan_UniqueExistence
┌──────────────────────────────────────────────────────────────────────────────┐
DIR BySmullyan_UniqueExistence
P BySmullyan_UniqueExistence
               #
               ∀Z:form(). ∀v0:Var(SubForm(Z)) ⟶ 𝔹.
                 {!f:SubForm(Z) ⟶ 𝔹 |
                   ValuationExtension(Var(SubForm(Z));v0;SubForm(Z);f)
                   ∧ PartialBooleanValuation(SubForm(Z);f)}
               BY
               UnivCD THENW SubFormClean THEN
               FORMInd 1 THEN PreserveL (
               UnivCD THENW SubFormClean)·
               # 3
               .....pand case.....
```

```
1. Q : form() ⟶ ℙ
2. ∀Z:{Z:form()| Q Z}. ∀v0:Var(SubForm(Z)) ⟶ 𝔹.
     {!f:SubForm(Z) ⟶ 𝔹 |
       ValuationExtension(Var(SubForm(Z));v0;SubForm(Z);f)
       ∧ PartialBooleanValuation(SubForm(Z);f)}
3. U : {Z:form()| Q Z}
4. V : {Z:form()| Q Z}
5. v0 : Var(SubForm(pand(U;V))) ⟶ 𝔹
⊢ {!f:SubForm(pand(U;V)) ⟶ 𝔹 |
     ValuationExtension(Var(SubForm(pand(U;V)));v0;SubForm(pand(U;V));f)
     ∧ PartialBooleanValuation(SubForm(pand(U;V));f)}
BY
```

At this point, our only option is to instantiate the induction hypothesis (2) with the immediate subformulas (U and V) of the formula under consideration. This yields a *function* fU (resp. fV) for computing Boolean valuations on subformulas of U (resp. V). At this point, we can construct a witness term of the following form:

PRF: BySmullyan_UniqueExistence

```
λZ.if form_equalF <Z, pand(U;V)> then (fU U) ∧b (fV V)
   if IsSubFormF <Z, U> then fU Z
   if IsSubFormF <Z, V> then fV Z
   else tt
   fi
```

where form_equalF and sfaIsSubFormF are functions extracted from proofs of the decidability of the corresponding propositions. In addition to being unweildy in the proofs of the resulting subgoals, these terms yield a grossly inefficient extracted algorithm, because explicit formula equality and subformula checks are used at every step in the recursion.

## 3.2 Programmer

In an alternative proof of Valuation Theorem, we can instantiate the witness for f with a recursively defined valuation function:

CODE: Value_ml

```
Value(v0;X)
==r form_cases(X;
               v.v0 pvar(v);
               A.¬bValue(v0;A);
               A,B.Value(v0;A) ∧b Value(v0;B);
               A,B.Value(v0;A) ∨b Value(v0;B);
               A,B.Value(v0;A) ⇒b Value(v0;B))
```

However, we never explicitly prove that this definition is well-formed for any general choices of v0 and X. In the proof of Valuation Theorem, we instantiate v0 and X with specific choices and, as a consequence of completing the proof, implicitly show that this instance of the recursive function is well-formed (and furthermore is a Boolean valuation).

Formally, we prove:

STM: ByExplicit_UniqueExistence

```
∀Z:form(). ∀v0:Var(SubForm(Z)) ⟶ 𝔹.
  {!f:SubForm(Z) ⟶ 𝔹 | ValuationExtension(Var(SubForm(Z));v0;SubForm(Z);f)
```

```
                    ∧ PartialBooleanValuation(SubForm(Z);f)}
```

where we replace ∃!... with {!...}, a set type that suppresses extraneous computational content in the extraction. Unsurprisingly, viewing the extract yields exactly the recursive proceduce `Value`:

```
PRF: ByExplicit_UniqueExistence_view
```

```
DIR ByExplicit_UniqueExistence_view
P ByExplicit_UniqueExistence_view
                #
                TERMOF{ByExplicit_UniqueExistence:o, \v:l}
                BY RepeatFor 1 (RW (HigherC AnyExtractC) 0) THEN Reduce 0⋅
                # 1
                λZ,v0,X.Value(v0;X)
                BY
END ByExplicit_UniqueExistence_view
```

While this method transparently yields the obvious recursive algorithm for the evaluation of a propositional formula, there is an implicit duplication of effort. First, we must discover the algorithm and, then, we must prove that it has the desired properties. This results in a proof being dictated by the algorithm, rather than the algorithm being extracted from the proof. To remedy this situation, we turn to the next approach.

### 3.3 The Formalist

In our third proof of `Valuation Theorem`, we wish to extract an algorithm from the existence proof. However, this turns out to be rather difficult. Surprisingly, we find a formulation in terms of a theorem tantamount to the uniqueness of a Boolean valuation rather than the existence of a valuation, the existence to be proved afterwards.

In particular, we prove the following theorem:

```
STM: ByExtract_Uniqueness
```

```
∀Z:form(). ∀v0:Var(SubForm(Z)) ⟶ 𝔹.
  {y:𝔹|
   ∀f:SubForm(Z) ⟶ 𝔹.
     ((ValuationExtension(Var(SubForm(Z));v0;SubForm(Z);f) ∧
       PartialBooleanValuation(SubForm(Z);f))
     ⇒ y = f Z ∈ 𝔹)}
```

Intuitively, this theorem says that for any formula $Z$ and interpretation $v_0$ of the variables of $Z$, there is a unique Boolean value $y$ which is the value assigned to $Z$ by any Boolean valuation $f$ of $Z$ under $v_0$. We will prove this theorem by induction on $Z$. Moreover, the implicitly defined computable function from $Z$ to $y$ is thus a Boolean valuation that applies to any formula $Z$ and any interpretation of the set of variables of $Z$. We will extract this function automatically from the proof of `ByExtract_Uniqueness`.

First, here is a sketch of the proof of `ByExtract_Uniqueness`.

> In the base case of the induction $Z$ is a variable, say $v$. We take $y = v_0\ v$. Any Boolean valuation of $Z$ under $v_0$ must equal $v_0\ v$ by the definition of `ValuationExtension`.
>
> Suppose for the induction case that this result is true for all immediate subformulas of $Z$. Consider the structure of any $Z$ which is not a variable.
>
> > If $Z$ is a negation, say $\neg U$, then $U$ is an immediate subformula of $Z$ and hence there is a unique $y_U \in \mathbb{B}$ which is the value assigned to $U$ by any Boolean valuation. Let $y = \mathsf{not}(y_U)$. If $f$ is a Boolean valuation of $Z$, then $f(Z) = \mathsf{not}(f(Z))$ by the definition of `PartialBooleanValuation`, and by the induction hypothesis, $y_U = f(U)$; hence, $y = f(Z)$.

If $Z$ is $U$ *op* $V$ for any binary connective, then since $U$ and $V$ are immediate subformulas of $Z$, by the induction hypothesis, there are unique $y_U$ and $y_V$ in $\mathbb{B}$. Let $y = \mathsf{bop}(y_U, y_B)$ where $\mathsf{bop}$ is the boolean operation corresponding to *op*.

By the definition of `PartialBooleanValuation`, any boolean valuation $f$ satisfies

$$f(U\ op\ V) = \mathsf{bop}(f(U), f(V)).$$

By the induction hypothesis, $y_U = f(U)$, $y_V = f(V)$. Thus, $f(Z) = \mathsf{bop}(f(U), f(V)) = y$ as required.

Completing this proof in Nuprl is relatively straightforward. We can automatically extract the computation content (i.e., the evaluation algorithm) of this proof and view it:

```
PRF: ByExtract_Uniqueness_view
```

```
DIR ByExtract_Uniqueness_view
P ByExtract_Uniqueness_view
                #
                TERMOF{ByExtract_Uniqueness:o, \v:l}
                BY RepeatFor 2 (RW (HigherC AnyExtractC) 0) THEN Reduce 0·
                # 1
                λZ.rec_ind(Z;f,x.form_cases(x;
                                            v.λv0.(v0 pvar(v));
                                            X.λv0.(¬b(f X v0));
                                            X,Y.λv0.((f X v0) ∧b (f Y v0));
                                            X,Y.λv0.((f X v0) ∨b (f Y v0));
                                            X,Y.λv0.(f X v0 ⇒b (f Y v0))))
                BY
END ByExtract_Uniqueness_view
```

This is recognizable as the recursive algorithm for the evaluation of a propositional formula, although this extract needlessly passes the interpretation `v0` at each recursive call. (Note that the `rec_ind` term describes a recursive procedure over the formula `Z`, where the recursive procedure is bound to the variable `f` in the second subterm.)

In the proof of `Valuation Theorem`, we instantiate the witness with the extracted algorithm: `TERMOF{ByExtract_Uniqueness:o, \v:l}`. The disadvantage of this proof is that must explicitly manipulate the extraction term.

## 3.4 The Computer Scientist

In our final proof of `Valuation Theorem`, we advocate using Jason Hickey's very-dependent function type [14, 15] to combine the specification and the witness. In particular, we form the following very-dependent function type:

```
ABS: RFunctionValue
```

```
RFunctionValue(X;v0) ==
  {f | A:SubForm(X)
       ⟶ {form_cases(A;
                     v.v0 pvar(v);
                     Y.¬b(f Y);
                     Y,Z.(f Y) ∧b (f Z);
                     Y,Z.(f Y) ∨b (f Z);
                     Y,Z.f Y ⇒b (f Z)):𝔹}}
```

We interpret {f | x:A ⟶ B} as the type of functions with domain A and range type B[f;x] on argument x, where f is the function itself. That is, this is a type of very dependent functions whose range type depends on calls to the function itself. Hence, RFunctionValue(X;v0) is the type of functions from subformulas of X to a Boolean value equal to the evaluation of the function on immediate subformulas and combined with the appropriate Boolean operator. We use a singleton type to lift value of the Boolean valution to a type. Note that RFunctionValue(X;v0) is a *type*, not a *term*; that is, RFunctionValue(X;v0) cannot be presented as the witness function in the proof of Valuation Theorem. Rather, any inhabitant of the type RFunctionValue(X;v0) suffices as a witness in the proof of Valuation Theorem. Hence, we proceed in the development of Valuation Theorem as follows.

First, we must prove that RFunctionValue is a type: We expect this proof to be straightforward, but there are a few subtle difficulties. The proof begins in a standard manner, giving a well-founded ordering for the very-dependent function domain (i.e., formulas under the IsImmedSubForm relation):

STM: IsImmediateSubForm_well_founded2

```
∀Z:form(). WellFnd{i}(SubForm(Z);x,X.IsImmedSubForm(x;X))
```

However, we quickly arrive at this point:

PRF: RFunctionValue_wf

```
1. X : form()
2. v0 : Var(SubForm(X)) ⟶ 𝔹
3. A : SubForm(X)
4. f : {f | A:{z:SubForm(X)| IsImmedSubForm(z;A)}
           ⟶ {form_cases(A;
                         v.v0 pvar(v);
                         Y.¬b(f Y);
                         Y,Z.(f Y) ∧b (f Z);
                         Y,Z.(f Y) ∨b (f Z);
                         Y,Z.f Y ⇒b (f Z)):𝔹}}
⊢ {form_cases(A;
              v.v0 pvar(v);
              Y.¬b(f Y);
              Y,Z.(f Y) ∧b (f Z);
              Y,Z.(f Y) ∨b (f Z);
              Y,Z.f Y ⇒b (f Z)):𝔹} ∈ 𝕌
```

The difficulty here is that in order to prove that the singleton is indeed a type, we must prove that the singleton value is indeed an element of the Boolean type. However, to prove that the form_cases(A;...) term is a member of the Boolean type, we must in turn prove that f B is a member of the Boolean type whenever B is an immediate subformula of A. Thus, we attempt to cut in a proof of ∀B:ImmedSubForm(A). (f B ∈ 𝔹). Attempting to use very-dependent function elimination yields that f B is a member of the singleton type {form_cases(B;...):𝔹}. Unfortunately, the rule for dependent set membership elimination is the following:

RULE: dependent_set_memberEquality

```
H  ⊢ a1 = a2 ∈ {x:A| B}
  BY dependent_set_memberEquality !parameter{i:l} y ()
  H  ⊢ a1 = a2 ∈ A
  H  ⊢ !subst(B; x.a1)
  H y:A ⊢ !subst(B; x.y) = !subst(B; x.y) ∈ 𝕌
```

Thus, trying to apply this rule to the hypothesis f B ∈ {form_cases(B;...):𝔹}, yields as the second sub-goal, the requirement to prove that the form_cases(B;...) term is a member of the Boolean type.

As one can see, we have not made any real progress, as this will immediately require cutting in a proof of ∀C:ImmedSubForm(B). (f C ∈ 𝔹).

Various (complicated) solutions suggest themselves. We may be able to construe some inductive proof that simultaneously proves RFunctionValue(X;v0) ∈𝕌 and ∀f:RFunctionValue(X;v0). ∀A:SubForm(X). (f A ∈ 𝔹). Yet, this is unsatisfactory for two reasons. First, the very-dependent function type should incoroprate the inductive argument. Second, the essential difficult is not with the very-dependent function type, it is with the subset type. In the cut proof of ∀B:ImmedSubForm(A). (f B ∈ 𝔹), we arrive at this point:

```
PRF: RFunctionValue_wf
1. X : form()
2. v0 : Var(SubForm(X)) ⟶ 𝔹
3. A : SubForm(X)
4. f : {f | A:{z:SubForm(X)| IsImmedSubForm(z;A)}
           ⟶ {form_cases(A;
                         v.v0 pvar(v);
                         Y.¬b(f Y);
                         Y,Z.(f Y) ∧b (f Z);
                         Y,Z.(f Y) ∨b (f Z);
                         Y,Z.f Y ⇒b (f Z)):𝔹}}
5. B : ImmedSubForm(A)
6. f B ∈ {form_cases(B;
                     v.v0 pvar(v);
                     Y.¬b(f Y);
                     Y,Z.(f Y) ∧b (f Z);
                     Y,Z.(f Y) ∨b (f Z);
                     Y,Z.f Y ⇒b (f Z)):𝔹}
⊢ f B ∈ 𝔹
```

Arguing from semantics at this point suffices: given that f B is a member of a subset of the Boolean type, then f B must be a member of the Boolean type. However, no primitive rule allows us to make this inference. To complete our proof, we add the following rule:

```
RULE: dependent_set_memberEqualityInv
H  ⊢ a1 = a2 ∈ A
  BY dependent_set_memberEqualityInv x B ()
  H  ⊢ a1 = a2 ∈ {x:A| B}
```

Next, we show that RFunctionValue(X;v0) is inhabitted:

```
STM: RFunctionValue_inhab
∀Z:form(). ∀v0:Var(SubForm(Z)) ⟶ 𝔹.  RFunctionValue(Z;v0)
```

The proof of RFunctionValue_inhab is very similar to the proof of RFunctionValue_wf, and again requires the use of the dependent_set_memberEqualityInv rule.

Finally, we show that an inhabitant of RFunctionValue(X;v0) is a Boolean valuation on the subformulas of X:

```
STM: RFunctionValue_Existence
∀Z:form(). ∀v0:Var(SubForm(Z)) ⟶ 𝔹.
  {f:SubForm(Z) ⟶ 𝔹|
   ValuationExtension(Var(SubForm(Z));v0;SubForm(Z);f) ∧
   PartialBooleanValuation(SubForm(Z);f)}
```

Once again, we expect this to be straightforward, but there are a few subtle difficulties. The proof begins in a standard manner: appeal to `RFunctionValue_inhab` to choose an arbitrary inhabitant of `RFunctionValue(X;v0)` and supply this element as the witness term. This yields two sub-goals:

```
PRF: RFunctionValue
1. Z : form()
2. v0 : Var(SubForm(Z)) ⟶ 𝔹
3. f : RFunctionValue(Z;v0)
⊢ f ∈ SubForm(Z) ⟶ 𝔹
```

and

```
PRF: RFunctionValue
1. Z : form()
2. v0 : Var(SubForm(Z)) ⟶ 𝔹
3. f : RFunctionValue(Z;v0)
⊢ ValuationExtension(Var(SubForm(Z));v0;SubForm(Z);f)
  ∧ PartialBooleanValuation(SubForm(Z);f)
```

The proof of the second subgoal is remarkably simple, as the very-dependent function type is almost exactly the specification of `PartialBooleanValuation`. As we have seen before, well-formedness goals asking to show `f A` $\in\mathbb{B}$ for subformulas of `Z` occasionally arise, which are handled by the `dependent_set_memberEqualityInv` rule.

The first subgoal, however, is troublesome. We would like to apply the `functionExtensionality` rule; however, the rule is stated as follows:

```
RULE: functionExtensionality
H  ⊢ f = g ∈ (x:A ⟶ B) ext t
  BY functionExtensionality !parameter{i:l} (y:C ⟶ D) (z:E ⟶ F) u ()
  H u:A ⊢ (f u) = (g u) ∈ !subst(B; x.u) ext t
  H  ⊢ A = A ∈ 𝕌
  H  ⊢ f = f ∈ (y:C ⟶ D)
  H  ⊢ g = g ∈ (z:E ⟶ F)
```

It is impossible to apply this rule to the first subgoal, because the "function type" we would like to supply as the type of `f` is a very-dependent function type, not a normal function type. Hence, it doesn't match the argument slots `(y:C ⟶ D)` and `(z:E ⟶ F)`.

Two potential solutions suggest themselves. The first is the introduction of a libralized function extensionality rule:

```
RULE: functionExtensionalityLiberal
H  ⊢ f = g ∈ (x:A ⟶ B) ext t
  BY functionExtensionalityX !parameter{i:l} u ()
  H u:A ⊢ (f u) = (g u) ∈ !subst(B; x.u) ext t
  H  ⊢ A = A ∈ 𝕌
```

This change is advocated by Stuart Allen (see `http://www.cs.cornell.edu/Info/People/sfa/Nuprl/NuprlPrimitives/Xycomb_typing2_doc.html`), and corresponds to defining the function type simply in terms of application, not syntactic form of the function term.

A second solution is to preserve the definition of the function type, but assert that every very-dependent function type is also a "normal" function type. One possible rule is the following:

```
RULE: rfunctionFunction
H f:{g | x:A ⟶ B}, J ⊢ f = f ∈ (x:A ⟶ !subst(B; g.f))
  BY rfunctionFunction #$i
  No Subgoals
```

This rule says that given a term `f` of very-dependent function type, `f` is also a term of the function type corresponding to the "unrolling" of the very-dependent function type. Various other rules also suggest themselves, but the above was sufficient for our purposes. We can justify this rule by examining the `rfunction_lambdaFormation` rule:

```
RULE: rfunction_lambdaFormation
H  ⊢ {f | x:A ⟶ B} ext λy.(Y (λg.b))
  BY rfunction_lambdaFormation !parameter{i:l} u v R g y z ()
  H  ⊢ A = A ∈ 𝕌
  H  ⊢ R = R ∈ (A ⟶ A ⟶ 𝕌)
  H  ⊢ ↓WellFnd{i}(A;u,v.R u v)
  H y:A, g:{f | x:{z:A| z R y}  ⟶ B} ⊢ !subst(B; x.y; f.g) ext b
```

which shows that every term of very-dependent function type is in fact a lambda term. (A variation of this approach is taken in the MetaPRL system, where the dependent function type is defined in terms of the very dependent function type (ignoring the function argument in the range type), just as the independent function type is defined in terms of the dependent function type (ignoring the domain argument in the range type).)

Finally, we can view the computational content of the proof using very dependent function types:

```
PRF: RFunctionValue_Existence_view
DIR RFunctionValue_Existence_view
P RFunctionValue_Existence_view
               #
               TERMOF{RFunctionValue_Existence:o, \v:l}
               BY RepeatFor 2 (RW (SweepUpC AnyExtractC) 0) THEN Reduce 0·
               # 1
               λZ,v0,A.
                (Y
                (λf.form_cases(A;
                               v.v0 pvar(v);
                               Y.¬b(f Y);
                               Y,Z.(f Y) ∧b (f Z);
                               Y,Z.(f Y) ∨b (f Z);
                               Y,Z.f Y ⇒b (f Z))))
               BY
END RFunctionValue_Existence_view
```

Once again, this is recognizable as the recursive algorithm for the evaluation of a propositional formula, where `Y` is the recursive Y-combinator. Note that in this proof, we merely replicated the specification in the definition of `RFunctionValue` using the very dependent function type.

# 4 Conclusion

We have given a formal account of an important first step in the development of propositional logic, namely the existence and uniqueness of Boolean valuations. While the result is far from novel, we have cast the

problem in a subtly different light than most textbook presentations. Many textbooks, and Smullyan's in particular, stress a "pure" approach; however, such a presentation is disadvantageous to programmers, who are equally concerned with the implementation and efficiency of the underlying evaluation algorithm.

Therefore, we sought a presentation of the proof that would be satisfactory to all parties interested in the result: mathematicians, who are more comfortable manipulating "pure" functions than algorithm descriptions; programmers, who want to verify the correctness of an efficient evalution algorithm; and formalists, who subscribe to the "correct-by-construction" principle and wish to extract the evaluation algorithm from the appopriate existence proof. We feel that the proof using the very-dependent function type provides the best balance among these various criteria. To be certain, the proof is susceptible to some criticism from each party, but is more mutually acceptable than any of the other proofs. For the mathematician, no "exotic" algorithmic terms need to be manipulated; instead (accepting the isomorphism between a very-dependent function type and the function type corresponding to it's unrolling), we need only manipulate functions. For the programmer and the formalist, we extract the expected evaluation algorithm directly from the existence proof.

While there were some difficulties with the proof using very-dependent functions, we feel that they are symptomatic of the relative infancy of the very-dependent function type (e.g., lack of tactic support), rather than pointing to fundamental issues with the type constructor. On the contrary, we feel that the very-dependent function type will prove a natural method for proving the existence (and hence, admitting algorithm extraction) of functions satisfying a specification given by a recursive definition. We hope to report on these results in future work.

# References


[1] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard-Mohring, Amokrane Saïbi, and Benjamin Werner. *The Coq Proof Assistant Reference Manual*. INRIA-Rocquencourt, CNRS and ENS Lyon, 1996.

[2] J. L. Bates. *A Logic for Correct Program Development*. PhD thesis, Cornell University, 1979.

[3] J. L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):53–71, 1985.

[4] Mark Bickford and Robert L. Constable. A logic of events. Technical Report TR2003-1893, Cornell University, 2003.

[5] James Caldwell. Moving proofs-as-programs into practice. In *Proceedings of the 12$^{th}$ IEEE International Conference on Automated Software Engineering*. IEEE Computer Society, 1997.

[6] James Caldwell. Classical propositional decidability via Nuprl proof extraction. In Jim Grundy and Malcolm Newey, editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume 1479 of *Lecture Notes in Computer Science*, pages 105–122, Canberra, Australia, September-October 1998. Springer.

[7] James Caldwell, Ian Gent, and Judith Underwood. Search algorithms in type theory. *Theoretical Computer Science*, 232(1–2):55–90, February 2000.

[8] Robert L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of the IFIP Congress*, pages 229–233. North-Holland, 1971.

[9] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[10] Thierry Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[11] H. Geuvers, F. Wiedijk, and J. Zwanenburg. A constructive proof of the fundamental theorem of algebra without using the rationals. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs, Proceedings of the International Workshop TYPES 2000*, volume 2277 of *Lecture Notes in Computer Science*, pages 96–111. Springer, 2001.

[12] J-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*, volume 7 of *Cambridge Tracts in Computer Science*. Cambridge University Press, 1989.

[13] Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu. MetaPRL — A modular logical environment. In David Basin and Burkhart Wolff, editors, *Proceedings of the 16$^{th}$ International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 287–303. Springer-Verlag, 2003.

[14] Jason J. Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages 3*, 1996. Available electronically through the FOOL 3 home page.

[15] Jason J. Hickey. *The MetaPRL Logical Programming Environment*. PhD thesis, Cornell University, Ithaca, NY, January 2001.

[16] Christoph Kreitz. Towards a formal theory of program construction. *Revue d'intelligence artificielle*, 4(3):53–79, November 1990.

[17] Christoph Kreitz. Program synthesis. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume III, chapter III.2.5, pages 105–134. Kluwer, 1998.

[18] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, Amsterdam, 1973.

[19] Per Martin-Löf. Constructive mathematics and computer programming. In *Proceedings of the Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[20] Douglas R. Smith and Cordell Green. Toward practical applications of software synthesis. In *FMSP'96, The First Workshop on Formal Methods in Software Practice*, pages 31–39., 1996.

[21] Raymond M. Smullyan. *Diagonalization and self-reference*. Number 27 in Oxford Logic Guides. Clarendon Press, Oxford, 1994.