uu for Programming Languages

Ulisses Ferreira

Jose.Ferreira-Jr@cs.tcd.ie
Trinity College Dublin, Department of Computer Science
Dublin 2, Ireland

ulisses@ufba.br
Universidade Federal da Bahia, DCC,
Av. Adhemar de Barros s/n, Ondina,
Salvador BA Brazil

Abstract

The Internet has motivated new programming languages features. I consider that programming for such a global environment requires the ability to deal with what is unknown because connections often fail or delay and programs should be robust.
In this article I present *uu*, a value that can represent lack of information in programming languages. A thesis is that this value is a good feature towards the unification of some programming paradigms. In this article I explore constructs as consequences of *uu* in such languages together with examples where this value helps in programming, and I explain the relationship between *uu* and another programming paradigm. The *uu* implementation is also briefly discussed.

1 Motivation

High school students usually solve systems of algebraic equations. They try to find values, either numeric or symbolic, to be assigned to variables. Once they find a value for a variable, this value remains constant. Variables in such a system have two general states: unknown and (subsequently) known. These states inspire a new programming paradigm, where variables in a program initially contain the *unknown* state, and later, might change their state to *known* as it receives a value in the problem domain. In imperative languages, if a variable is in the former state, it contains a special value that is called ***uu***. Otherwise, it contains a value in the problem domain.

As another example, a program presents a question to the user, but the user does not know or even does not want to answer the question. What state should we assign to the variable after the input operation?

One more example, in programming for a world-wide network such as the Internet, the languages designer should take into account that underlying connections often fail or delay. In such cases, a neutral state could be assigned to the variable that takes part in the request, in such a way that the program carries on running safely, that is, the variable could contain a value that would represent the lack of that piece of information, instead of being committed to some known value in the problem domain.

A question arises from the above situations: what are the programming languages concepts and constructs to provide a better abstraction for variables regardless of the programming paradigm? In this article, I partially provide an answer to this question. Here, I also refer to a non-***uu*** value as *known value*.

Section 2 addresses somewhat related work. Section 3 introduces the concept of the *unknown* value together with other related concepts, and section 4 explains how ***uu*** can replace specific constructs for exception handling. Section 5 explains the applicability of ***uu*** in OOP while section 6 contains a discussion on the use of the *unknown* value in the integration with other paradigms, in particular logic programming. Section 7 explains how ***uu*** can support lazy evaluation. Section 8 contains a comment on implementation, section 9 presents a synthesis and, finally, the appendix presents an operational semantics.

2 Related Work

The ability to represent and reason under lack of information is well known in the artificial intelligence and logics communities, and there has been much work on this subject. However, from the best of my knowledge, none of them is related to programming techniques: Extended Logic Programming, introduced by Gelfond and Lifschitz[3] is one exception, where the core idea is the negation itself, and *abstract negation*, which is an on-going research of ours to provide only one negation in the whole unifying language. Our approach here consists in using the ***uu*** value when a specific piece of information is unknown to the program. In this way a set of variables with their corresponding values can represent lack of information. Another somewhat related work is *boxed* representation[7], implemented for functional languages such as Haskell. Here, my main concern is in pragmatics of a programming language, although such technique can implement ***uu***. The present ideas do not apply to pure functional languages, in which variables do not contain values but instead denote them.

In the late 80's, most commercial Expert System Shells also made use of similar unknown value to represent lack of information over Boolean variables, although in a very restrictive way. I generalize the concept to the programming language level regardless of the application area.

Current programming languages, such as Java[4] and ML, provide the concept of *exception handling* and constructs for it. I present more general constructs that replace this concept at the language level.

The ANSI/IEEE Standard 754-1985 also supports the concept of "Not-a-Number" or NaN[6] instead of floating-point values to represent indeterminate quantities. Although it also propagates NaN, the approaches are not the same since that standard does not concern programming languages constructs. Moreover, our approach is not limited to some specific data type.

3 *uu*: the Unknown Value

In this section, I introduce ***uu*** for programming languages design. For every data type, the languages designer can add a special value, namely ***uu***, to represent the lack of information. For integers, we have uu_i; for real numbers, we have uu_r and so on. In this paper however, I simply write ***uu*** to mean that the type is irrelevant in the context. I apologize for such slight abuse of notation. Accordingly, I use the term *value* to refer to a value regardless of its type, and a *known* value to mean a value in the problem domain.

The *unknown* value, ***uu***, extends the semantics of the classical and intuitionistic connectives according to the Łukasiewicz[8] 3-valued logic. In arithmetic and relational expressions, the presence of ***uu*** as an operand implies that the expression results in ***uu*** without evaluating the other operand.

As a first example, I present a program to find the roots of an equation of the second degree:

```
float a, b, c;
float delta := b * b − 4 * a * c;
float x1 := (−b − sqrt(delta))/(2 * a);
float x2 := (−b + sqrt(delta))/(2 * a);
write x1, " ", x2;
```

I believe that the above syntax is intuitive or familiar for programmers.

In comparison to other programming languages, it is not syntactically much different and here, a bit like Prolog, the corresponding system tries to find values for variables when they are being requested in an evaluating expression. Initially, all variables contain or denote ***uu***. The programmer, then, uses the variables, $x1$ and $x2$, in the expressions of the **write** command and then their known values are found according to the formulas in the program. Although *delta* is used by both $x1$ and $x2$, it is desirable that its value be calculated only once. Thus, the program execution asks for the values of b, a, c, in this order, before giving the answer.

In terms of Domain Theory, ***uu*** is a result and there is no order relation between ***uu*** and other values in the domain. Thus, ***uu*** is a value in the object language, and not the bottom ($\perp$). However, although ***uu*** is not "undefined", it can be used where a function is undefined to transform a partial function into a total one. Thus, the factorial function can be written as follows:

```
int $fact$(int $x$) :=
  if $x == 0$, 1
  ifnot
    (if $x > 0$, $x * fact(x - 1)$
    ifnot uu)
  otherwise uu;
```

In the above example, if the value of the parameter x is ***uu*** or negative, the result is ***uu***. Notice that the conditional was extended to accommodate a 3-valued logic. I address conditionals as well as lazy evaluation later in this article.

Variables either contain ***uu*** or a known value. Most imperative programming languages adopt a default value as initial variable contents. Here, since I am introducing ***uu*** in this context, I adopt this value as initial for each variable according to its type. The programmer might want to initialize some of the variables according to the application.

3.1 Evaluators and Reactors

In imperative programming, the present idea is to allow the programmer to write a piece of code to "discover" the value in the problem domain whenever a variable that contains ***uu*** is being used in some evaluating expression. I call this piece of code an *evaluator*. Additionally, the programmer can write a piece of code, called *reactor*, to be triggered instead of letting values be stored in the variables. For every variable in a program, one can attach *handlers*. They can be one *evaluator* and/or one *reactor*, independently. Among other purposes, such handlers protect the variable. One can write handlers in the following way, intuitive for Pascal or C++ programmers:

```
int x, y;

when x do { // this is an evaluator
  x := 2 * y;
}

when x := do { // and this is a reactor
  x := $;
}
```

In the above example, two handlers were defined for x. At the first time that the value of the variable x is being requested in an expression, the above evaluator is triggered, which in turn computes the double of the value of the variable y assigning it to x. From the second time on, that computed value $2 * y$ is already available and the evaluator is not triggered.

The concept of evaluator can contain the **return** statement (similar to C) instead of assigning a value to the requested variable. In the case of the **return** statement, the evaluator is always triggered when that variable is used, unless a known value has been assigned to that variable outside the evaluator. Such an assignment can be statically allowed and dynamically allowed or forbidden, or simply statically forbidden.

On the other hand, whenever a value is to be stored in x, the control is jumped to the corresponding reactor. In the above example, the value is accepted: notice that the $ symbol is used in reactors to represent the value that, in other languages, would be stored unconditionally.

Two predicates are used to check whether a variable contains ***uu***, namely, **known** and **unknown**. In these cases, the value is accessed directly and the *Boolean* result from the condition is provided by the interpreter without evaluating the ***uu***-valued variable handler.

3.2 Comparing Handers with Methods and Functions

A pragmatic comparison between the use of handlers and the use of functions and methods can be done: after implementing an application system, handlers can always be added and updated without changing the system elsewhere. However, unlike functions and methods that are called elsewhere, handler definitions can always be removed as the system provides a default semantics for the absence of a handler.

The semantics of using a variable containing a known value is exactly the same as for other languages. However, the semantics of using a variable containing ***uu*** is not: if there is an evaluator, it is executed. Otherwise, i.e. when there is no evaluator, ***uu*** is used. On the other hand, the semantics of the use of variables is not the same as the semantics of function calls either, because the latter are always executed. Therefore, in some sense, ***uu*** combined with handlers are semantically somewhere between use of variable and function.

Before using ***uu*** as an operand in the expression evaluation, PLAIN[2], a proposed language for the Internet, optionally tries to get the value from other sources, e.g. asking the end-user, in such a way that in the first run programmers can be reminded that they forgot to initialize some variable.

Besides using some languages constructs to hide variables ($x2$, $y2$, uux, and uuy, below), the equivalent semantics is then achieved without a built-in ***uu*** in the following way:

```
int x2, y2;
logic uux := true, uuy := true;

int x(void) {
  if uux, { // the evaluator for x
    x2 := 2 * y(); // instead of x_eq(2 * y());
    uux := false;
  }
  ifnot return x2;
}

int x_eq(int dollar) { // the reactor for x
  x2 := dollar;
  uux := false;
}
```

Notice that the *Boolean* type, built-in in some languages, has been replaced by (3-valued) **logic**. Also, the parentheses that surround conditions in the **if** statement (and **while** statement in other contexts), which might contain commas in C, C++ and Java, have been replaced by the comma to mean **then**. Here, **ifnot** is being presented instead of the reserved word *else*, adopted in other imperative and functional languages.

In an object-oriented context, although it is possible to write a class, say $C1$, with the above code, and then to define many instances of that class and its subclasses, the use of classes does not encourage the use of these ideas as much as a programming language does. Moreover, the syntax for assignment and for handlers are more suggestive of the programmers intention, and the occurrence of variables in expressions should not need to have the same syntax as function calls. More importantly, handlers contain different code, and this diversity makes classes and subclasses definitions much more complicated and less readable. Another important difference: in object-oriented languages, programmers should take care when they are defining a public field because the field can be used outside the class and programmers can no longer change the field definition, e.g. to be private to that class, without considering elsewhere. Here, although it is not possible to change the field definition either, it is still possible to insert code related to that variable in a handler without changing the rest of the system. The use of public fields in the current scheme is surprisingly harmless. If we think in terms of classes as being downloaded and linked dynamically on a public network and of mobile agents that deal with resources that cannot move, this difference itself might become decisive in the language design. Using a variable in an evaluating expression might cause its value to be read from disk or requested from a remote process, provided that its current value is ***uu***. Thus, a variable may be a kind of cache, because in the subsequent uses of that variable its value is already locally available and the handler is not triggered. Conversely, assigning a value to a variable might cause its value to be stored on disk or sent to a remote host. Storing values of variables on disk and restoring the values by using the same variables implements persistent programming.

4 ***uu*** in Exception Handling

uu, as being a more primitive and general concept, when combined with handlers, replaces exception handling, which simplifies the language. The prefix operator **code** gets the exception code for a variable in a context where the reason for its value be unknown is required. Suppose that I want to access a value for a variable at some place on the WWW, given the address www.somewhere.on.the.earth.

```
string addr = "www.somewhere.on.the.earth";
int x = getint(addr) timeout 10;
if unknown x then {
  int c = code(x);
  if c == exc_to then {
    write "Time-out in the attempt to access ",addr,nl;
  }
```

```
    ifnot {
      write "Exception ",c, "in the access of x at",addr,nl;
    }
  }
```

Thus, if the requested integer value is not retrieved by 10 seconds, the program execution continues normally. In the above example, the case is treated as an exception, but the lack of information in the problem domain that was expected to be in x allows the computation to continue normally, with or without the above treatment, and without x being committed to any value in the problem domain, which guarantees safety and robustness.

5 Object-Oriented Programming with ***uu***

The ***uu*** value can be added to any programming language no matter its paradigm. In this section I give some examples in object oriented programming. Although a hybrid paradigm language can adopt only one construct for both frames and classes, I discuss them separately here.

5.1 ***uu*** and Frames

A programming language can adopt the following semantics for the use of a variable in an evaluating expression:

1. If the value is known, use this value.
2. otherwise, if the evaluator for that variable is defined, execute it. Then
   (a) if its value is now known, or the evaluator **return**s a value, use this value;
   (b) otherwise, use ***uu***.

   Success.
3. otherwise, if the variable is a field, look for the value of the corresponding field in the object class (and recursively, in its superclass), using the steps 1-3 of this algorithm, and then,
   (a) if a value in the problem domain is found, use this value in the expression instead of assigning to the requested variable;
   (b) otherwise, use ***uu***;

Success.

4. At this point, the variable is not a field, contains ***uu*** and there is no handler for the variable. If "possible", ask the application user for the value of that variable, and then,

   (a) If the value is entered, the value is assigned to the variable and the evaluation of the expression continues.

   (b) On the other hand, if the user does not want to answer, ***uu*** is used in the expression and its calculation carries on. The user will no longer be asked for a value of the same variable, unless ***uu*** is assigned to that variable.

5. If "not possible", e.g. the application is not interactive, use ***uu***.

Note. Steps 4-5, if implemented, require that the compiler generates the symbolic names of the variables. PLAIN has done so and recently, in spite of the size of the byte-code, it was realized that these names are also useful for symbolic communication between agents on an open system. Thus, because PLAIN was designed for knowledge representation, it has been relatively easy to adapt it to support mobile agents.

Now, looking back to the first example, of the equation of second degree, its coefficients a, b and c, do not have evaluators and hence their values can be asked at the terminal at the first time that they are requested. Assigning the result from an expression to a variable in its definition is a syntax simplification of writing an evaluator for that variable containing only that expression. Thus, initializations might be dynamic and lazy in a sense.

Handlers are very useful for *testing* and *debugging*, by inspecting what is being used, and this can also be done in mobile agents. For these programs, there can be handlers for other purposes that are outside the scope of this discussion, e.g. to be implicitly executed before departures and after arrivals.

5.2 Classes with *uu*

Using the Internet as an example, if a programmer wants to build a class to represent some specific measure, they should take into account the fact that some countries use certain measuring systems while others use different systems. Consider the temperature representation, either in Celsius or Fahrenheit:

```
class temperature {
  public float Fahr, Celsius;
```

```
    when Fahr do {
      Fahr := 9.0/5.0 * Celsius + 32;
    }
    when Fahr := do {
      Fahr := $; // it accepts the assignment
      Celsius := 5 * (Fahr − 32)/9;
    }
    when Celsius do {
      Celsius := 5 * (Fahr − 32)/9;
    }
    when Celsius := do {
      Celsius := $; // accepts the assignment
      Fahr := 9.0/5.0 * Celsius + 32;
    }
  }
```

The above example is a form of constraint programming. It is represented here, almost declaratively, the relationship between two variables concerning measurement of the same concept, temperature, and the handlers keep the variables always consistent. Regardless of the syntax, this is somewhat similar to method invocation. However, methods are always executed.

As another example, I consider that classes and frames are concepts that can be integrated. But while classes come from the set theory, frames come from the prototype theory. While a class is a shape and is conceived for re-usability and other programming concerns, a frame represents concepts of the real world. Because both class frames and instance frames are defined, I can integrate them easily:

```
  class Human {
    public logic dies := true;
    public int class cardinality := 6000000000; // not exact
    public string handed := "Right";
  }
  Human Socrates; // Therefore, Socrates.dies
```

However, in a frame system, classes are also treated as instances, e.g. it makes sense to compute *Human.cardinality*++; when someone is being born. The reserved word **class** is being used as a modifier in *cardinality* to remove the attribute from the instances of that class, although its subclasses can inherit the attribute and even change its value to represent exceptions. In the example, *Socrates.cardinality* might be meaningless. According to the step 3 of the algorithm presented in the previous section, the ***uu*** contribution here is that the field values of the instance *Socrates* is obtained from the class *Human* (or possible superclasses) dynamically when they currently contain ***uu***, i.e. if a definition is changed dynamically inside a frame, its instances will inherit the new value on demand.

According to the algorithm, ***uu*** does not necessarily depend on handlers, and it can be used as any other value, e.g. it can be assigned to a variable. *Handlers* are somewhat similar to ”ties” in Perl, “triggers” in SQL and ”tag methods” in Lua[5], which are languages that do not provide ***uu***.

Handlers and ***uu*** can be used to implement multiple inheritance in an object-oriented language that provides single inheritance and late binding: the programmer defines secondary conceptual super-classes as fields in the defining class as in the example below:

```
class c1 {
  public int i;
  public float f;

  public void m() { }

  when f do { f := 3.14; }
  when i do { }
}

class c2 {
  public int f;
  public float i;

  public void m() { write ”Hello”, nl; }

  when i do { }
  when f do { f := 15; }
}

class c3: c1 {
  private c2 sec; // c2 is a secondary super-class

  // here, the conflicts are solved by interception:
  public void m() { sec.m(); } // ... from c2
  when i do return sec.f; // ... from c2
  when i := do { sec.f := $; } // ... to c2
}
```

Thus, the language shifts responsibility to the programmer to define the interpretation of the conflicts among attributes from different conceptual superclasses. The exceptions are treated by overriding methods and handlers, which can be used to rename conflicting attributes when they are needed in the defining class. In $c3$ in the above example, by default, the attributes are inherited from $c1$ and handlers and methods are written to implement inheritance from $c2$. Because multiple inheritance is not very often needed, single inheritance in this context seems to be an interesting solution, in particular, together with late binding.

Classes do not necessarily need the **new** operator to create objects as they might be created on demand: when a variable of any class is used in an evaluating expression and it contains ***uu***, the object referred to is created.

The program below exemplifies the use of ***uu*** when the programmer wants some default value to be assumed. For example, we normally assume that English ought to be generally used on the Internet when we want to communicate with the public. Spanish ought to be used in Latin-America mailing lists, and so on. With ***uu***, value inheritance can be a dynamic relationship:

```
class General {
  public string you := “world”;
  public string hello := “Hello,”;
  public string sayhello;
  when sayhello do
    return hello + “ ” + you+ “!”;
}

class MailingList: General {
  initial {
    hello := “Ciao, ”;
    you := “Italia”;
  }
}

//...
MailingList b;
b.you := “caros brasileiros”;
write b.sayhello, nl;
b.you := uu;
```

write *b.sayhello*, *nl*;
MailingList.hello := *uu*;
write *b.sayhello*, *nl*;

In the above example, after the definition of b, the value of the field *b.you* is customized as “caros brasileiros”. In the following line, the field *b.sayhello* is requested and evaluated as “Ciao, caros brasileiros!”, and this content is written followed by the newline: the constant **nl**. Then, ***uu*** is stored in *b.you*. When *b.sayhello* is evaluated in the subsequent line, it is evaluated as “Ciao, Italia!”, that is, because *b.you* now contains ***uu***, its known value is picked up from its class (and so on, upwards, if it is also ***uu*** there). Then, this value is written followed by the newline. The next line assigns ***uu*** to the field *hello* of the class *MailingList*. Finally, the string “Hello, Italia” is written followed by the newline command. In this way, I represent default values. In this case, *sayhallo* is always evaluated.

6 Imperative and Logic-Based Features

uu can help combine imperative constructs in a hybrid language with other programming paradigms, such as Logic Programming. A Logic Programming system, besides answering a query with either *true* or *false*, can provide values for free variables. Some of these variables remain free after the computation from a query, which means that they represent “any answer”. Thus, besides the ability to answer a query with ***uu*** to mean “unknown”, ***uu*** can be used to implement free variables.

On the other hand, if a variable of the imperative paradigm is passed to a query of the logic paradigm and its value is ***uu***, the algorithm of the called paradigm will understand that the query includes a request for the value of that variable. After receiving the answer, the calling program interprets variables containing ***uu*** as an appropriate answer, and continues normally. If the answer list is empty, that means “no answer”, while ***uu*** is normally interpreted as “unknown answer”. As above, depending on the contract, ***uu*** can be interpreted as “any answer”.

7 ***uu*** in Lazy Evaluation (Call by Need)

uu can support lazy evaluation in almost all sequential operations in a language, that is, when the first operand results in ***uu***, the second operand is not evaluated. Another submitted article to this journal describes in detail the application of ***uu*** and lazy evaluation in programming for internets and mobile agents. Thus, in the rest of this section, I mention only call by need.

Some functional languages are eager (e.g. ML) and some are lazy (e.g. Haskell), but I can think of lazy and eager evaluations as concepts related to functions in the following way:

```
logic lazy $f$(int $x$, int $y$) :=
    if $x + x > y$, $true$
    ifnot $false$;

//...
logic $y := f(1 + 2 + 3, 2 + 2)$;
```

Here, although the parameter x is used more than once in the function *f*, x is evaluated only in its first occurrence in the expression. The **lazy** modifier postpones the evaluations of the parameters, e.g. $1 + 2 + 3$ and $2 + 2$. Regarding the implementation of this mechanism, it is not difficult with ***uu***: The compiler generates code to skip the actual parameter list. It also creates a pointer to every expression in the actual parameter list, $1+2+3$ and $2+2$ in this case, including references to the activation record[1], transforming every actual parameter into a local subroutine, and then passes the expression pointer to the corresponding formal parameter handler, x or y. As already explained, for the first time that a ***uu***-ed parameter is being evaluated, the corresponding actual parameter, either $1+2+3$ or $2+2$, is evaluated. From the second time on, the value is known and, because of this, it is not evaluated. However, if the formal parameter is not used, the corresponding actual parameter is not even evaluated.

It seems to be bizarre to provide lazy evaluation and imperative features because of side-effects. However, a hybrid paradigm programming language can provide the concept of "pure function" as its compiler forbids assignments and global objects inside its code.

8 Implementation

Although efficiency is not the main issue in the present paper, inefficiency might be the only negative point of the present ideas, in comparison with imperative languages that do not provide object-oriented constructs, because the interpreter has to check the presence of ***uu*** whenever a variable is being used in an evaluating expression. References to handlers for a variable also increase the size of the object code. This detail is not really a languages feature, but instead it depends on the decision of implementation which might require some analysis on the source code in terms of frequency of use of handlers, which in turn depends on use and experiments. Therefore, the implementation of ***uu*** is a challenge. The comparison is relative to application because method invocation, for example, requires pattern matching and search algorithm. Moreover, like a mobile-code language, a ***uu***-based language entails some form of code interpretation, which is becoming a normal conduct in programming languages as hardware is getting larger and faster.

9 Conclusion

As a consequence of the adoption of ***uu***, expressions in a programming language have to be able to consider this special value. Assuming that *not uu* results in ***uu***, statements such as **if-then-else** and **while**, as well as their semantics are adapted to deal with three logical values. The conditional statement or expression in their full forms, for example, becomes **if-then-ifnot-otherwise**.

uu and handlers can simplify a programming language by replacing common constructs such as multiple inheritance.

Handlers and ***uu*** have been experimented within PLAIN for a number of years successfully. Their application, along with other features, to programming for mobile agents and the Internet has been investigated.

The idea of a hybrid paradigm for programming allows programmers feel free to choose their own way of working. Some definitions are better written in some particular paradigm while others are better written in other paradigms.

At a more refined level, there can be two kinds of unknown states: the first represents the initial lack of information with potential for later discovery. The second kind also represents the lack of information after having attempted to discover its value. PLAIN distinguishes one kind from the other, to allow the inference machine to recognize variables whose value was already asked.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley series in computer science. Addison-Wesley Publishing Company, Reading (Mass.). - London, 1986. Originally published as: Principles of compiler design.

[2] U. Ferreira. The plain www page. *URL http://www.ufba.br/˜plain and http://www.cs.tcd.ie/˜ferreirj/plain.html*, 1996–2003.

[3] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing. Ohmsha Ltd and Spring-Verlag*, pages 365–385, 1991.

[4] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley Publishing Company, 1996.

[5] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. Lua - an extensible extension language. *Software: Practice and Experience*, 26(6), 1996.

[6] Institute of Electrical and Electronics Engineers. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*, 1985.

[7] S. P. Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International Series in Computer Science. Prentice-Hall, Inc., 1987.

[8] S. C. Kleene. *Introduction of Metamathematics.* D. Van Nostrand, Princeton, 1952.