Plain Text Specification Notation

H. Conrad Cunningham

22 April 2022

Contents

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla.

Plain Text Specification Notation

Chapter Introduction

This chapter informally defines the mathematical and logical notation we use to express the semantics of program units such as functions, procedures, methods, modules, and classes. We use it to specify abstract models, invariants,

preconditions, and postconditions.

We adopt a plain-text notation that can be typed as comments in program source code, not the notation that would be typeset in a mathematics or logic textbook. The logic notation follows that popularized by David Gries [9,10], Edsger Dijkstra [7,8], and others [3–5,12,14] in the formal methods community.

We assume that the readers of our specifications are familiar with basic mathematical concepts such as logical connectives, sets, relations, functions, etc. The primary purpose of this chaper is to introduce the plain-text notation.

Program States and Specification

The *state* of a program is a mapping of all the *variables* to their respective values. Here we use the term "variable" flexibly to enable this notation to be used for a variety of languages and circumstances. It includes all the program's named constants, variables, parameters, and types and perhaps other entities such as the program counter(s), input/output channels, and program metadata.

A *state space* for a program is the set of all possible states.

If `E` is an expression defined in some state space, `x` is a variable, and `v` is a possible value for that variable, then the notation `E(x := v)` denotes the expression `E` where *all* occurrences of the variable `x` are replaced by the value `v`. Similarly, the notation `E(x,y := u,v)` denotes the expression `E` where *all* occurrences of the variables `x` and `y` are simultaneously replaced by the values `u` and `v`, respectively.

To *evaluate an expression in a state* means to replace all the variables in the expression by their respective values in that state and then compute the resulting value.

In specifying computer programs, we use *predicates* (i.e., logical expressions) to assert that a condition must hold at all points in the program's state space.

Consider a function in a programming language. The functions precondition predicate asserts what must be true about the program's state and the values of the function's arguments at a program location where the function is called. Note that a precondition implicitly includes an assertion about value of the program counter. Similarly, the function's postcondition predicate asserts what must be true about the program's state and the function's return value at a program location where the function returns.

Predicate Logic

A predicate is an expression that evaluates to either the Boolean value *true* or the Boolean value *false* at a point in the state space in which predicate is well-defined. In the following, we informally describe the syntax and semantics of predicates.

- The primitive predicates include `true` and `false`.

Predicates `true` and `false` evaluate to *true* and *false*, respectively, in all points of the state space.

- The primitive predicates also include the usual arithmetic *relational expressions* from mathematics and the computer programming language.

  These expressions evaluate to *true* or *false* according to their usual meanings at all points of the state space in which they are well defined.

  Note: We use the operators `==` and `!=` for equality and inequality. Similarly, we use `<`, `<=`, `>`, and `>=` for the ordered relational operator. (However, in some cases it is better to use whatever operator symbols are used in the programming language.)

  Note: To specify computer programs, it is necessary and useful to include common operations from mathematics and the programming language. However, these must be pure expressions, without side effects.

- The primitive predicates can also include Boolean functions defined in the the programming language, its runtime library, or elsewhere in the program being specified.

  These expressions evaluate to *true* or *false* according to their usual meanings at all points of the state space in which they are well defined.

  Note: We can also use functions that return other types as components of relational expressions.

- If `P` is a predicate, then `NOT P` is a predicate.

  `NOT P` is the *logical negation* of `P`. The predicate `NOT P` evaluates to *true* in any state in which `P` evaluates to *false* and evaluates to *false* in any state in which `P` evaluates to *true*.

- If `P` and `Q` are predicates, then `P && Q`, `P OR Q`, `P <=> Q`, and `P => Q` are also predicates.

  - `P && Q` is *logical conjunction* (AND). The predicate `P && Q` is *true* at every point of the state space where both `P` and `Q` are *true*; it is *false* elsewhere.
  - `P OR Q` is *logical disjunction*. The predicate `P OR Q` is *false* at every point of the state space where both `P` and `Q` are *false*; it is *true* elsewhere.
  - `P <=> Q` is *logical equivalence*. The predicate `P <=> Q` is *true* at every point of the state space where the values of `P` and `Q` are equal; it is *false* elsewhere.
  - `P => Q` is *implication*. The predicate `P => Q` is *false* at every point of the state space where `P` is *true* and `Q` is *false*; it is *true* elsewhere.

- If `P`, `Q`, and `R` are predicates, then `if P then Q else R` is a predicate.

  The predicate `if P then Q else R` is equivalent to
  `(p => q) && (NOT p => r)`.

- If `R` and `T` are predicates and `x` is a variable, then `(ForAll x : R :: T)` is a predicate. In quantified expressions like this one:

  - `x` is a distinct new variable (called a *dummy variable*) whose scope is between the parentheses
  - `R` is the *range*
  - `T` is the *term*

  `(ForAll x : R :: T)` is a *universal quantification*; it is essentially a generalized variant of logical conjunction (AND). In more formal logical notation, we normally replace the plain-text symbol `ForAll` by ∀.

  The predicate `(ForAll x : R :: T)` is *true* at every point of the state space in which, for any value `v`, `R(x := v) => T(x := v)` is *true*; it is *false* elsewhere.

  We can extend this notation to quantify over several variables such as in `(ForAll x, y, z : R :: T)`. In this case, we consider
  `R(x,y,z := u,v,w) => T(x,y,z := u,v,w)`
  for all distinct tuples of values `(x,y,z)`.

  If `R` is `true`, we can omit the range from the expression and just write `(ForAll x,y,z :: T)`.

  Note: We can also extend the other quantification expressions below to allow multiple variables and omission of a `true` domain predicate `R`.

- If `R` and `T` are predicates and `x` is a variable, then `(Exists x : R :: T)` is a predicate.

  `(Exists x : R :: T)` is an *existential quantification*; it is essentially a generalized variant of logical disjunction (OR). In more formal logical notation, we normally replace the plain-text symbol `Exists` by ∃.

  The predicate `(Exists x : R :: T)` is *true* at every point of the state space in which, for some value `v`, `R(x := v) && T(x := v)` is *true*; it is *false* elsewhere.

Other Quantifications

In addition to the logical quantification above, we can also use quantified expressions that involve other operators and yield other types of values (e.g., integers).

If `OP` is a binary operator on some type that is *associative* and *commutative* (or symmetric) and has an *identity element*, then we can readily define a quantification `(OP x : R :: T)` [10].

To evaluate the quantification `(OP x : R :: T)`, we:

a. generate a "list" containing the values of term expression `T(x := v)` for each value `v` of the dummy variable `x` that satisfies the range predicate `R(x := v)`

b. insert the associative and commutative binary operator `op` between adjacent elements of the list

c. compute the value of the generated expression

If the range is empty (i.e., there are no values of `x` for which `R`is *true*), then the value of the quantification is the identity element for binary operator `OP`.

Thus, in addition to universal and existential quantification discussed in the previous section, we can define the following common arithmetic quantifications according to above general rules:

- *Summation.* `(+ x : R :: T)` yields, for all values `v` of the same type as `T` (e.g., integers), the sum of all values `T(x := v)` for which predicate `R(x := v)` is *true*. If the range is empty, then the value of the quantification is 0.

  Note: In mathematics, the symbol Σ is often used to denote summation.

- *Product.* `(* x : R :: T)` yields, for all values `v` of the compatible type (e.g., integers), the product of all values `T(x := v)` for which predicate `R(x := v)` is *true*. If the range is empty, then the value of the quantification is 1.

  Note: In mathematics, the symbol Π is often used to denote summation.

In addition, we can define *numerical quantification* expressions as follows:

- `(# x : R :: T)` yields the value `(+ x : R AND T :: 1)`, a count of all distinct values of `x` for which logical predicate `R AND T` is *true*. If the range is empty, then the value of the quantification is 0.

Consider the binary operators `min` and `max`. They are associative and commutative.

On the one hand, if we consider any set of values that is bounded above by a maximum value, then the maximum value is the identity element for the `min` operation on that set. Similarly, if we consider any set of values that is bounded below by a minimum value, then the minimum value is the identity element for the `max` operation on that set.

On the other hand, if we consider any set of values that is unbounded above (e.g., the positive integers), then `min` does not have an identity element within the set. Similarly, if we consider any set of values that is unbounded below (e.g., the negative integers), then `max` does not have an identity element within the set. We could, however, assign some special values such as $+\infty$ as the identity element for `min` and $-\infty$ as the identity element for `max`.

Thus, we can define two other useful quantifications:

- *Minimization.* `(Min x : R :: T)` yields, for all values `v` of the same type as `T` (e.g., integers), the minimum of all values `T(x := v)` for which predicate `R(x := v)` is *true*.

  If restrict the type of `T` to be the integers, then, for an empty range, the value of the quantification is $+\infty$, which we can write as `POS_INF`.

- *Maximization.* `(Max x : R :: T)` yields, for all values `v` of the same type as `T` (e.g., integers), the maximum of all values `T(x := v)` for which predicate `R(x := v)` is *true*.

  If restrict the type of `T` to be the integers, then, for an empty range, the value of the quantification is $-\infty$, which we can write as `NEG_INF`.

Note: Given that it may be impossible to represent `POS_INF` and `NEG_INF` in a program, in most circumstances it is best to avoid use of minimimization and maximization in specifying computer programs.

Sets

A *set* is an unordered collection of elements without duplicates.

- Operator `IN` denotes *set membership*. For any set `C` and value `v`, predicate `v IN C` is *true* if and only if `v` is an element in set `C`. Similarly, `v NOT_IN C` denotes `NOT (v IN C)`.
- A programming language *type* consists of a set of values and a set of operations. We sometimes say a value is `IN` a type to mean the value is an element of the set associated with the type.
- Operator `CARD` denotes *set cardinality*. For any set `C`, function `CARD(C)` yields the number of elements in the set `C`.
- Binary operator `SUBSET_OF` denotes the *subset* relationship between sets. For sets `C` and `D`, predicate `C SUBSET_OF D` if and only if all the elements in `C` are also elements in `D`.
- Binary operator `UNION` denotes *set union*. For any sets `C` and `D`, operation `C UNION D` yields the set consisting of all the elements in `C` and in `D`, including those in both `C` and `D`.
- Binary operator `INTERSECT` denotes *set intersection*. For any sets `C` and `D`, operation `C INTERSECT D` yields the set consisting only of the elements that are in both `C` and `D`.
- Binary operator `DIFF` denotes *set difference*. For any sets `C` and `D`, operation `C DIFF D` yields the set `C` with all elements of set `D` removed.
- A *set comprehension* has the form `{ x : R :: T }`, which has a syntax and semantics like the quantifications described above. `x` is a dummy

variable whose scope is inside the braces, `R` is a range predicate, and `T` is the term.

The set `{ x : R :: T }` consists of all elements `T(x:=v)` where `v` ranges over all values for which `R(x:=v)` is *true*.

Relations

- A *Cartesian product* of two sets `C` and `D` is the set of all *ordered pairs* `(x,y)` where `x IN C` and `y IN D`.

  We extend this concept to any finite number of sets. We also call these fixed-length ordered groupings *tuples*.

- A *relation* on sets `C` and `D` is a subset of the Cartesian product of `C` and `D`. That is, a set of tuples.

- A *function* on sets `C` and `D` is a special case of a relation on (domain) `C` and (range) `D` where each element in `C` occurs in at most one tuple in the relation.

  - A *total function* is a function defined for all elements in its domain.
  - A *partial function* is a function defined for a subset of its domain.

Bags

Mathematically, a *bag* (or *multiset*) is a function from some arbitrary set of elements (the domain) to the set of nonnegative integers (the range) [10:11.7,15,16]. We interpret the nonnegative integer as the number of occurrences of the element in the bag. Zero means the element does not occur. The number of occurrences of an element in a bag is also called its *multiplicity*.

From another perspective, a bag is an unordered collection of elements. Each element may occur one or more times in the bag. (It is like a set except values can occur multiple times.)

- The literal `{| |}` denotes an *empty bag*.
- The literal `{| 2, 3, 2, 1 |}` denotes a bag containing four integers including two 2's, one 3, and one 1. The order is not significant! There may be one or more occurrences of an element.
- Operator `IN` also denotes the *bag membership* operation. `x IN C` is true if and only if there are one or more occurrences of the value `x` in bag `C`. Similarly, `x NOT_IN C` denotes the negation of `x IN C`.
- Operator `OCCURRENCES` denotes *occurrence counting*. For any bag `C`, `OCCURENCES(x,C)` yields the number of times element `x` occurs in bag `C` (i.e., the multiplicity of `x`.)

```
OCCURRENCES(1,{| 1, 1, 2, 1 |}) =  3
OCCURRENCES(3,{| 1, 1, 2, 1 }}) =  0
```

- Operator `CARD` denotes *cardinality*. `CARD(C)` is the total number of occurrences of all elements in bag `C`.

```
CARD({| 1, 1, 2, 1 |}) =  4
CARD({| |) =  0
```

- `REPEAT(e,n)` denotes a bag containing exactly `n` occurrences of `e` or an empty bag if `n < 1`.
- Binary operator `UNION` also denotes *bag union*. For any bags `C` and `D`, `C UNION D` yields the bag consisting of all elements in `C` and in `D`, or in both; the number of occurrences of an element in the union is the number in `C` or in `D`, whichever is *greater* (i.e., the maximum multiplicity).

  ```
  {| 3, 1, 1, 3 |} UNION {| 1, 2 |} = {| 1,1,2,3,3 |}
  ```

- Binary operator `SUM` denotes *bag sum*. The sum of bags `C` and `D` yields the bag consisting of all elements in `C` and in `D`, or in both; the number of occurrences of an element is the sum of the occurrences in `C` and `D`.

  ```
  {| 3, 1, 1, 3 |} SUM {| 1, 2 |} = {| 1,1,1,2,3,3 |}
  ```

- Binary operator `INTERSECT` also denotes *bag intersection*. For any bags `C` and `D`, `C INTERSECT D` yields the bag consisting of all elements that occur both in `C` and in `D`; the number of occurrences of an element is the number in `C` or in `D`, whichever is *lesser* (i.e., the minimum multiplicity).

  ```
  {| 3, 1, 1, 3 |} INTERSECT {| 1, 2 |} = {| 1 |}
  ```

- Operator `DIFF` also denotes *bag difference*. For any bags `C` and `D`, `C DIFF D` yields the bag consisting of all the elements of `C` that occur in `D` fewer times; the number of occurrences of an element is the number of occurrences in `C` minus the number in `D`. (If there are more occurrence in `D` than `C`, then the result has no occurrences.)

  ```
  {| 1, 1, 2, 1 |} DIFF {| 2,1,1,3 |} = {| 1 |}
  ```

Sequences

TODO

What Next?

This document defines the plain-text notation we use to specify the semantics of programs. Source code comments and other documents can refer to this chapter rather than describe the notation used.

Acknowledgements

In the early 1990s, I wrote *A Programmer's Introduction to Predicate Logic* [4] as a supplement for my students in the CSci 550 (Program Semantics and Derivation) [5] and Engr 664 (Theory of Concurrent Programming) courses. For CSci 555, I used the textbooks by Gries [9] and Cohen [3] and gradually developed my own set of mostly compatible notes [4,5]. My use of predicate logic and equational reasoning was also influenced by Jan Tijmen Udding from whom I took my first course on program derivation in the 1980s, my dissertation advisor Gruia-Catalin Roman, Dijkstra [7,8], Gries's discrete mathematics textbook [10], and other sources [1,2,11,12,14].

In Fall 1996, I began teaching courses on object-oriented programming and software architecture. For those courses, I informally used approaches such as constructive semantics for abstract data types (e.g., as described in my notes on Data Abstraction [6]) and Meyer's design-by-contract [13]. I also developed example programs written in various languages (e.g., Java, Ruby, Scala, Lua, Elixir, and Python and annotated the programs with comments giving preconditions, postconditions, and invariants, usually without much explanation of the specification notation I was using. Over the years, I wrote a number of partial and often incompatible explanations of the notation.

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on possible textbooks based on the course materials I had developed during my three decades as a faculty member. In January 2022, I began refining the existing content, integrating separately developed materials together, reformatting the documents, constructing a unified bibliography (e.g., using citeproc), and improving my build workflow and use of Pandoc.

I wrote this document in 2022 to unify the description of the notation I have used in mt example programs (e.g., the CookieJar, CandyBowl, and Digraph ADTs) and other documents.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

References

[1] Richard Bird and Philip Wadler. 1988. *Introduction to functional programming* (First ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[2] K. Mani Chandy and Jayadev Misra. 1988. *Parallel program design: A foundation*. Addison-Wesley, Boston, Massachusetts, USA.

[3] Edward Cohen. 1990. *Programming in the 1990's: An introduction to the calculation of programs*. Springer, New York, New York, USA.

[4] H. Conrad Cunningham. 2006. *A programmer's introduction to predicate logic*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/PredicateLogicNotes/Programmers_Introduction_to_Predicate-Logic.pdf

[5] H. Conrad Cunningham. 2006. *Notes on program semantics and derivation*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/reports/umcis-1994-02.pdf

[6] H. Conrad Cunningham. 2019. *Notes on data abstraction*. University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/DataAbstraction/DataAbstraction.html

[7] Edsger W. Dijkstra and Wim H. J. Feijen. 1988. *A method of programming*. Addison-Wesley, Boston Massachusetts, USA.

[8] Edsger W. Dijkstra and Carel S. Scholten. 1990. *Predicate calculus and program semantics*. Springer, New York, New York, USA.

[9] David Gries. 1981. *Science of programming*. Springer, New York, New York, USA.

[10] David Gries and Fred B. Schneider. 1993. *A logical approach to discrete math*. Springer, New York, New York, USA.

[11] Robert R. Hoogerwoord. 1989. The design of functional programs: A calculational approach. PhD thesis. Eindhoven Technical University, Eindhoven, The Netherlands.

[12] Anne Kaldewaij. 1990. *Programming: The derivation of algorithms*. Prentice Hall, New York, New York, USA.

[13] Bertrand Meyer. 1997. *Object-oriented program construction* (Second ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[14] Antonetta J. M. van Gasteren. 1990. *On the shape of mathematical arguments*. Springer, Berlin, Germany.

[15] Wikpedia: The Free Encyclopedia. 2022. Multiset. Retrieved from https://en.wikipedia.org/wiki/Multiset

[16] Wolfram Research, Inc. 2022. Multiset. Retrieved from https://mathworld.wolfram.com/Multiset.html