Mathematically Rigorous Software Design *with pointers*

Wilson E. Alvarez[1], Dr. Marko Schütz[2*]
wilson.alvarez@upr.edu[1], marko.schutz@upr.edu[2]

December 19, 2012

Abstract

Small software bugs, such as the case of Therac-25[1] and the rocket Ariane 5 [4][5], have cost at least three lives and about 7 billion dollars of loss. An introduction of memory locations and two new operators to an error-free software design work [6] is given. An analysis shows that [6] still holds true in data environments where memory locations are added to its program variables. Only a simple modification to portions of [6] is required.

1 Introduction

Between 1985 and 1987, a medical radiation therapy machine, called Therac-25 and produced by Atomic Energy of Canada Limited, killed three patients by treating them with a dose of radiation of about a hundred times the intended dose. The cause: a software bug in the machine[1]. On September 21, 1997, a crew member of the USS Yorktown (DDG-48/CG-48) entered data in a computer network which caused a buffer overrun "crashing the entire network and causing the ship to lose control of its propulsion system", leaving the passengers in the middle of the ocean[2]. On August 14, 2003, a power outage on the northeast of the United States, and part of Canada, which left about fifty five million people without power was caused by a software bug that prevented the system administrators from receiving important error messages[3]. On the 4th of June 1996 the launch of the rocket Ariane 5 ended in a failure when a software error occurred just about 40 seconds after take off. About 7 billion dollars were wasted in the project[4][5]. These

[*]Project supervisor

four cases of software design mistakes, which are but a handful of what has happened in history, motivate the study of error-free software design.

On September 9, 2002, Robert L. Baber released a book titled "Mathematically Rigorous Software Design" where he talks about the requirements for developing error-free software [6]. From touching very general aspects to some other particular ones in software development, Baber was able to prove mathematically the veracity of his propositions, but [6], however, does not include pointers in his analysis, and thus does not include two operators found in today's software design: the *addressOf* operator and the *dereference* operator.

The goal of this study is to expand [6] by adding the mentioned program variable (i.e. the pointer), the *address of* operator, and the *dereference* operator. In the following three sections you will find a brief summary of the elements that make the very foundation of [6] with their corresponding definitions, according to [6], and some practical meanings and examples. After, in the following sections, we will explain explicitly what needs to be changed in [6] so that it continues to work even with the addition of memory locations and the mentioned operators and a few particular examples will be considered.

2 Program Variables and Data Environments

Just as in physics one might make mathematical models by "stopping time" conveniently to analyze or calculate momentum, force, work, etc, at some point or interval of time, Baber's mathematical models of programs can be thought of as currently running programs that are being stopped right before the execution of a statement. At that point in time, the modeled program may have some data allocated in memory for which the statements have access to. This inspires the introduction of Baber's first definition:

Definition 2.1 *Program Variable*
A program variable is an ordered triple that contains:

1. the name of the variable,
2. the set of possible values the variable can hold, and
3. the value the variable is currently holding.

For example, $(x, \mathbb{Z}, 3)$ and $(n, \mathbb{Q}, 5)$ are program variables to which some statement has access to, whereas $(r, \mathbb{R}, 2i)$ and $(z, \mathbb{N}, -2)$ are not because their values are not within the variables' corresponding set. Note that since the value of the program variable must correspond to a set given in the

ordered triple, it is not allowed to have a program variable x such that its set is $\varnothing$. That is, the triple $(x, \varnothing, p)$ is not a program variable since $p \notin \varnothing$.

Now, going back to that point in time before where a statement is about to be executed, it is clearly possible that the statement may have access to one or more program variables. This inspires the following definition:

Definition 2.2 *Data Environments*
A data environment is a sequence of program variables.

Using the previous example, it is clear that the sequence $d = \{(x, \mathbb{Z}, 3)\}$ is a data environment, whereas $d' = \{(x, \mathbb{Z}, 3), (n, \mathbb{Q}, 5), (z, \mathbb{N}, -2)\}$ is not because z is not a program variable since -2 is not in $\mathbb{N}$.

The evaluation of a variable name in a statement usually results in the substitution of their names for their values in a given a data environment. For example, let $d_0 = \{(x, \mathbb{Z}, 3), (z, \mathbb{Q}, 5)\}$ and $d_1 = \{(x, \mathbb{R}, 3.14), (z, \mathbb{Q}, 42)\}$. Clearly, d_0 and d_1 are data environments, but note that $x(d_0) = 3$ and that $x(d_1) = 3.14$. The expression $(x + z).d_0 = x.d_0 + z.d_0 = 3 + 5 = 8$, whereas $(x + z).d_1 = 45.14$. That is, the values that program variables are assigned change with the data environments that they are in. Mathematically, given a program variable x that can hold any value from the set K, x is defined to be a function from $\mathbb{D}|_x \to K$ where $\mathbb{D}|_x$ is the set of all data environments that contain the program variable x. The assumption of the existence of the set $\mathbb{D}$ is not troublesome in practice[6], yet it falls into Russell's paradox. For more information on this topic see page 8 of [6].

3 Statements

One of the main purposes of [6] was to "familiarize the student and reader with the basic concepts underlying correctness proofs"[1] which includes proof rules for software development that helps make the verification of the veracity of propositions faster. Informally, these rules help ensure that a statement is defined and will yield the desired results right before it is executed. This study focuses on that area too, but through the expansion of [6] such that it appeals more to other programming languages that make use of pointers. Therefore, we will begin this section with the following definition:

Definition 3.1 *Statement*
A statement is a function that changes a data environment into another data environment.

[1] see page 6 of [6]

More precisely, we can see the statements S as functions from $\mathbb{D} \to \mathbb{D}$. That is, given a a data environment d_1, applying the statement S to d_1 will yield d_2, and applying the statement S to d_2 will yield d_3, and so on. Note that the all these data environments may actually be equal because the statement, by definition, does not necessarily change a data environment into a *different* data environment. This inspires the following statement definition:

Definition 3.2 *Null Statement*
The null statement is an function that takes a data environment as its argument but does nothing to it.

The null statement can be thought of as the identity function. That is, for a data environment d, $\texttt{null}(d) = d$. Now, since statements work on data environments and data environments themselves are made of program variables, when the need of storing new information in memory arises, the ability to add program variables to a data environment surely comes in handy. This gives purpose for the following definition:

Definition 3.3 *Declare Statement*
The declare statement is an function that adds a program variable through the *left* of the given data environment.

That is, the declare statement can be seen as a function from the set of all data environments $\mathbb{D}$ to the same set, given a name for the variable. For example, given the data environment $d = \{(p, \mathbb{Q}^*, \sqrt{2})\}$, the declare statement: $(\texttt{declare}\ (l, \mathbb{N}, 1)).d$ will result in a new data environment, say d', equal to $\{(l, \mathbb{N}, 1)), (p, \mathbb{Q}^*, \sqrt{2})\}$. Note that the definition of the declare statement does not exclude the declaration of program variables with the same name. In the case that there are two or more variables with the same name in a data environment, the evaluation of the variable name will result in the value of the first variable found from left to right in the data environment. For example, let $d = \{(z, \texttt{Strings}, \text{`}\texttt{abc}\text{'})), (z, \mathbb{N}, 42)\}$, then, $z(d) = \text{`}\texttt{abc}\text{'}$. Also, given an expression E defined in a data environment d, the evaluation of an array variable, say x_E, occurs as follow: first the expression E is evaluated and then the array variable name with the resulting expression value is evaluated in the data environment. Symbolically, $x_E.d = (x_{E.d}).d$. For example, let $d = \{(x_{10},\ \mathbb{R}, 23), (x_9, \mathbb{N}, 3), (x_2, \mathbb{Z}, -7)\}$. Then, the expression $x_{x_9 - x_2}$ in d is the same as $(x_{x_9 - x_2}).d = (x_{x_9.d - x_2.d}).d = x_{(3-(-7))} = x_{10} = 23$.

In cases where there is a limited amount of memory or memory management is needed, releasing or deleting program variables is certainly necessary.

This gives purpose to the following definition:

Definition 3.4 *Release Statement*
A release statement is an function that takes a variable name and a data environment as arguments and returns a data environments in which the first program variable found, from left to right, with that name in the data environment has been eliminated.

In the case where there is more than one program variable with the same name in a data environment, the release statement will release the first program variable found from left to right.

For example, if $d = \{(z, \texttt{Strings}, \text{`}\texttt{abc}\text{'}), (z, \mathbb{N}, 42)\}$, executing the statement $(\texttt{release } z).d$ will result in a new data environment, say d', equal to $\{(z, \mathbb{N}, 42)\}$. In the case that there is no variable with the given name, the release statement will be the same as the null statement.

When the need for changing the value of a program variable arises, [6] gives the following definition:

Definition 3.5 *Assignment Statement*
The assignment statement is an function that takes a program variable in a data environment and changes its value.

For example, executing $x := 5$ will change the value of the variable x to 5 if two things are given: first, x is in the data environment, and second the value that is assigned to the variable, in this case 5, is an element of the set of possible values for x. That is, we can execute $(x := 5).d$ if and only if the program variable $(x, \{5\} \cup \mathbb{A}, q)$ is in d, where $\mathbb{A}$ can be any set.

Now, in many real-life situations there might be times when there are at least two options upon which one can choose a possible course of action given an environment or desire that enforces our decision. In the most simple case we would have only two courses of action and a desire that, if satisfied, will help choose one course of action, but otherwise will choose the other course of action. This inspires the following definition:

Definition 3.6 *If-Else statement*
Let d be a data environment and let B be a boolean expression defined in d. Then the statement $\texttt{If } B \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ endif}$ will result in:

- $S_1.d$ if $B.d$ is true, or
- $S_2.d$ if $B.d$ is false

For example, let $d = \{(x, \mathbb{Z}, 10)\}$. The statement $\texttt{If } x = 1 \texttt{ then } x := 2 \texttt{ else } x := 3 \texttt{ endif}$ in d will result in $(x := 3).d$, which results in the data environment $d' = \{(x, \mathbb{Z}, 3)\}$.

Nowadays, many home appliances such as microwave ovens and washing machines repeat a single process until the user chooses not to do so, or until a specific goal has met. In the case of the microwave ovens, the user usually commands the microwave oven to spin the plate and to send microwaves until the end of the given time has been met. This action of doing something until a certain condition has been met is what inspires the following definition:

Definition 3.7 *While Loop*
Let d be a data environment and let B be a boolean expression defined in d. Suppose that there is an element of the boolean expression B such that each time the statement S is executed, that element changes which might, or might not, change the value of the boolean expression B when evaluated in the data environment d. Then, the statement `while` B `do` S `endwhile`, abbreviated $\texttt{W}(B, S)$, will result in:

- $\texttt{W}(B, S).S(d)$ if B is true.or
- $\texttt{null}.d$ if B is false

For example, let $d = \{(i, \mathbb{N}, 1)\}$. Executing the statement `while` $i < 2$ `do` $i := i + 1$ `endwhile` in d, which is the same as $(\texttt{W}(i < 2, i := i + 1)).d$ will result in $\texttt{W}(i.d < 2, i := i.d + 1) = \texttt{W}(1 < 2, i := 2)$ Since $1 < 2$ is true, i becomes 2 and $\texttt{W}(i < 2, i := i + 1)$ is executed again with the new i in, say, d'. Therefore, $\texttt{W}(i < 2, i := i + 1).d' = \texttt{W}(2 < 2, i := 3).d'$. Since $2 < 2$ is false, then $\texttt{W}(i < 2, i := i + 1).d' = \texttt{null}.d' = d'$, where the resulting data environment $d' = \{(z_1, \mathbb{Z}, 2), (i, \mathbb{N}, 2)\}$.

4 Preconditions and Postconditions

As we mentioned earlier, one of the main purposes of [6] was to familiarize the reader with rules of software development such that the definedness of a statement, that is about to be executed, would be ensured with respect to a subset of the range of the statement. In other words, if we look at the statements as functions from data environments to data environments, his goal was to show the reader the available ways of describing precisely and exactly the preimage of a statement and its range. In [6], the preconditions and postconditions are generalized ideas of the domain and range of a function. The former two, additionally, might also hold important information about the platform upon which the code is being developed for, a property of the statements such as the loop invariant[2], or any other information expressed as a boolean expression.

[2] see page 66 of [6], for more information about the loop invariant

Definition 4.1 *Ordinary Precondition*
Let V and P be subsets of $\mathbb{D}$. Then, V is a precondition of a given postcondition P with respect to a statement S if and only if for every $d \in V \cap S^{-1}(\mathbb{D})$, $S(d)$ is in P.

Note that the definition of the Ordinary Precondition does not state if the precondition V is a subset of the preimage of S, that is $S^{-1}(\mathbb{D})$. Therefore, an ordinary precondition does not ensure us that the statement S will be defined since $V \cap S^{-1}(\mathbb{D})$ might actually be empty. This inspires the definition of a new type of precondition:

Definition 4.2 *Strict Precondition*
Let V and P be subsets of $\mathbb{D}$. Then V is a strict precondition of a given postcondition P with respect to a statement S if and only if V is an ordinary precondition and $V \subseteq S^{-1}(\mathbb{D})$.

Note that since $V \subseteq S^{-1}(\mathbb{D})$, strict preconditions ensure us that the statement S will always be defined for any data environment d in V, however, since the precondition V is simply a subset of $S^{-1}(\mathbb{D})$, it does not ensure us that V contains all the data environments for which S maps to P. In other words, strict preconditions do not ensure us that $S^{-1}(P) \subseteq V$. This inspires the following definition:

Definition 4.3 *Complete Precondition*
Let V and P be subsets of $\mathbb{D}$. Then V is a complete precondition of a given postcondition P with respect to a statement S if and only if V is an ordinary precondition and $S^{-1}(P) \subseteq V$.

Now, note that the strict precondition also tells us that for any $d \in V$, $S(d) \in P$. That is, $V \subseteq S^{-1}(P)$ due to its definition[3]. And also note that the complete precondition tells us that $S^{-1}(P) \subseteq V$ in its definition. It is, therefore, clear that a strict and complete precondition V will be equal to $S^{-1}(P)$.[4]

5 Analysis of Program Variables and Operators

As we mentioned earlier, the purpose of this study is to integrate the pointer variable, the dereference and addressOf operators into [6] such that

[3] see "Lemma for a strict precondition" on page 21 of [6] for a formal proof

[4] see "Lemma for a strict and complete precondition" on page 22 of [6] for a formal proof

it would appeal more to programming languages that make use of these. We, therefore, begin with the definition of the new program variables:

Definition 5.1 *New Program Variables*
A new program variable is an ordered *quadruple* containing:

1. the name of the variable,
2. the set of possible values that the variable can hold,
3. the value that the variable is currently holding, and
4. a unique memory address.

Clearly, the only difference between this definition and Definition 2.1 is the addition of the *unique* memory address to the program variable. This uniqueness can be formally stated as follow: given any program variable in a data environment, there does not exist a different program variable in the same data environment with the same memory address. This inspires the redefinition of the data environments:

Definition 5.2 *New Data Environments*
A sequence is a new data environment if and only if:

1. its elements are new program variables
2. for every program variable in the data environment there does not exist a different program variable in the same data environment with the same memory address (i.e. their memory addresses are unique)

This uniqueness can be stated symbolically as follows: $\forall x \in d \; \neg\exists y \in d$ such that $y \neq x$ and the memory address of x is the same as the memory address of y. Which can also be restated as: $\forall x, y \in d$, if $y \neq x$, then the memory address of x is different from the memory address of y.

As a convention, from here on we are going to use the terms program variable and data environment to refer to the new program variable and the new data environment respectively. To refer to the program variables and data environments from the previous sections we will write old program variables and old data environments.

Now, note that due to the uniqueness of the memory addresses, sequence operations on data environments need to be considered carefully. For example, let $d_0 = \{(n, \mathbb{Z}, 3, addr1)\}$ and $d_1 = \{(z, \mathbb{R}, .33, addr1)\}$. The concatenation of these two sets does not yield a data environment. That is, $d_0 \| d_1 = \{(n, \mathbb{Z}, 3, addr1)\} \| \{(z, \mathbb{R}, .33, addr1)\} = \{(n, \mathbb{Z}, 3, addr1), (z, \mathbb{R}, .33, addr1)\}$, but n and z have the same memory addresses which makes $d_0 \| d_1$ differ from a data environment. This can be fixed by releasing and declaring the variables in either d_0 or d_1 such that their memory addresses differ.

Since [6] takes into consideration the concatenation of data environments, to address this issue we formulated a necessary and sufficient condition such that for the concatenation of any number of data environments, the result will always yield a data environment. We now to proceed to state this result formally, with $\mathbb{D}^*$ being the set of all *new* data environments.

Result 1. *Sufficient and necessary condition for conc. of new data env.*[5,6] Let $\mathbb{K}$ be a subset of $\mathbb{D}^*$. Then, the concatenation of arbitrary data environments, namely $\|\mathbb{K}$, results in a data environment if and only if for each program variable x in each data environment d_0 of $\mathbb{K}$ there does not exist another program variable y in any data environment d_1 of $\mathbb{K}$ such that $y \neq x$ and the memory address of x is the same memory address of y

Symbolically, the above statement can be restated as follows: Let $\mathbb{K}$ be a subset of $\mathbb{D}^*$. $\|\mathbb{K}$ is a data environment $\Leftrightarrow \forall x \in d_0 \in \mathbb{K}\ \neg\exists y \in d_1 \in \mathbb{K}$ such that $y \neq x$ and the memory address of x is the same as the memory address of y. Which is the same as: $\|\mathbb{K}$ is a data environment $\Leftrightarrow \forall x \in d_0 \in \mathbb{K}$ $\forall y \in d_1 \in \mathbb{K}$, $y = x$ or the memory address of x differs from memory address of y but not both. We proceed to prove the latter.

Proof
($\Rightarrow$) Let $\mathbb{K}$ be a subset of $\mathbb{D}^*$. Suppose the concatenation of arbitrary data environments in $\mathbb{K}$ results in a data environment. Let d_0 be a data environment in $\mathbb{K}$. Let x be a program variable in d_0. Let $d_1 \in \mathbb{K}$ and let y be a program variable in d_1. We want to prove that either $y = x$ or the memory address of x differs from the memory address of y.

First, note that d_0 and d_1 are subsets of $\|\mathbb{K}$, but $\|\mathbb{K}$ is a data environment. Therefore, either $y = x$ or the memory address of x differs from the memory address of y by the definition of data environments.

($\Leftarrow$) Let $\mathbb{K}$ be a subset of $\mathbb{D}^*$. Suppose that $\forall x \in d_0 \in \mathbb{K}\ \forall y \in d_1 \in \mathbb{K}$, either $y = x$ or the memory address of x differs from the memory address of y. Consider the concatenation of all the data environments in $\mathbb{K}$. Our goal is to prove that $\|\mathbb{K}$ is a data environment, but note that $\|\mathbb{K}$ is already made of program variables, therefore we need only to prove the uniqueness of their memory addresses. That is, we need to prove that $\forall x, y \in \|\mathbb{K}$, $y = x$ or the memory address of x differs from the memory address of y, but not both.

First, note that since d_0 and d_1 are arbitrary and are subsets of $\|\mathbb{K}$, the hypothesis can be restated as follows: $\forall x \in \|\mathbb{K}\ \forall y \in \|\mathbb{K}$, either $y = x$ or

[5] conc. = concatenations
[6] env. = environments

the memory address of x differs from the memory address of y. Which is, in fact, what we were looking for. ■

The last two statements of our proof can also be seen this way. Say we pick the very first variable of $\|\mathbb{K}$. Due to the property stated by the hypothesis, the address of this variable is different from any other. That is, at the very least it is certain that the first two variables of $\|\mathbb{K}$ are different. Suppose we now pick the second program variable in $\|\mathbb{K}$. We already know that the first two program variables are different, but since the second variable is also different from any other, we have that at least the first three program variables in $\|\mathbb{K}$ are different, and so on.

This result suggests that any statement in [6] that makes use of concatenations of data environments, or simply a single data environment, needs to be "rewritten" in order to add a memory uniqueness statement.

We now present our next result, which perhaps is somewhat trivial but still needs to be considered:

Result 2. *Any subsequence of a data environment is a data environment*
Formally, let d be a data environment. If $A \subseteq d$, then A is a data environment.

Proof
First note that the elements of A are program variables. Therefore, we only need to prove that the memory addresses of every program variable in A are unique. Let x be a program variable in A. Since $A \subseteq d$, x is in d. Therefore, the memory address of x is unique. ■

Now that we have laid the foundations of what changes in [6] with the introduction of unique memory addresses, it is the perfect time to introduce two new operations to the field:

Definition 5.3 *AddressOf Operator*
The addressOf operator is a function that takes a variable name in a data environment as an argument and returns its memory address.

That is, the addressOf operator is a function that goes from a data environment to the set of all memory addresses. For example, let $d = \{(x, \mathbb{C}, 2 + 4i, addr0\}$. Executing $(\texttt{addressOf}(x)).d$ will result in $addr0$. Formally, the domain of the addressOf operator is the set of all data environments in which the the given variable name is an element of, and its range is the set of all memory addresses.

One of the purposes of adding unique memory addresses to the program variables is to be able to have access to the "hidden" variables in [6] that ap-

pear when two or more variables in a data environment have the same name. For example, in the data environment $d = \{(x, \mathbb{Z}, 4, addr0), (x, \mathbb{Q}, \frac{4}{10}, addr1)\}$, only the value 4 can be accesses through the variable name x, and the second program variable $(x, \mathbb{Q}, \frac{4}{10}, addr1)$ is hidden from the statements. Given now that we have unique memory addresses in each program variable, we will be able to access the values of the hidden program variables through the following operator:

Definition 5.4 *Dereference Operator*
The dereference operator is a function that takes an address of a program variable in a data environment and gives access to the program variable whose address was given.

For example, let d be the same data environment as before. Executing $(\texttt{dereference}\, addr1).d$ will result in $\frac{4}{10}$. But executing $(\texttt{dereference}\ addr1 := 10)$ will change the value of the second variable x in d to 10.

This operator opens a new path for a new type of program variable, namely the pointer.

Definition 5.5 *Pointer*
A pointer is a program variable that holds as its value a memory address that points to a program variable that has a specific set of possible values.

Formally, the domain of the dereference operator that is operating on a pointer is the set of all data environments such that: 1. the name of the pointer is in the data environment, 2. the address the pointer is currently holding is in the data environment, and 3. the set of possible values that the pointer can point to is the same as the set of possible values of the program variable at the address the pointer is currently holding as its value.

For example let $\mathbb{Z}^*$ represent the set of all memory addresses that point to an integer variable. Suppose $d = \{(x, \mathbb{Z}, 42, addr0), (x, \mathbb{R}, 42, addr1)\}$. The statement $(\texttt{declare}\ \ (p, \mathbb{Z}^*, addr0, addr3)).d$ creates a pointer p that when dereferenced *can only give access to* program variables whose set is $\mathbb{Z}$. That is, the dereference operator that is operating on p can access the program variable at the address p is currently holding io if and only if $Set(\texttt{dereference}\ (p)).d = Set(z).d$, where $Set(\texttt{dereference}\ (p))$ refers to the set of possible values the pointer can point to.

In this case, executing $(\texttt{dereference}\ p).d$ will result in 42, but executing $(\texttt{dereference}\ p := 2).d$ will change the variable of the first x in d to 2. Now, note that releasing x in d will result in the data environment $d' = \{(p, \mathbb{Z}^*, addr0, addr3), (x, \mathbb{R}, 42, addr1)\}$. Here, p is now holding an address in memory that is not in d'. Also, assuming p is holding the address

$addr1$, dereferencing p is undefined as it was design to point to an integer variable and not a real variable. In other words, it is undefined because $Set(\texttt{dereference}\ p) = \mathbb{Z} \neq Set(x) = R$.

Whenever any of these two cases occur, we say that p *is defined*, but *dereferencing* p *is undefined*. That is, p is defined whenever its value is a memory address regardless of which variable it is pointing to, and dereferencing p is undefined either when its address points to a program variable of a different set than what it was designed for or its address is not in the data environment.

Also, note that the definition of pointer can be iterated upon the pointer itself. That is, suppose that there is a pointer p whose set of possible values is the set of all the memory addresses that point to an integer program variable, namely $\mathbb{Z}^*$. Then we can also have a pointer, say p', whose possible values are the memory addresses of pointers that point to integer variables, namely the set $\mathbb{Z}^{**}$. In this case suppose p' is holding the memory address of p. Since p' holds the memory address of p, dereferencing p' will be the same as accessing the value of p. But since the value of p is actually the memory address of the integer variable, dereferencing p' will be the same as evaluating the name p, which is also the same accessing the address of the integer program variable. Dereferencing p' twice, will therefore be the same as dereferencing p once which access the value of the integer variable.

6 Analysis of Statements and Operators

Our goal in this section will be to find complete and strict preconditions for the given statements and postconditions that make use of pointers. That is, given a particular statement S and a postcondition P, we will find a complete and strict precondition that such that $V = S^{-1}(P)$. We begin our analysis by considering the following case:

Case 6.1 $\{?\}\ x := 1\ \{(\texttt{dereference}\ p) > 1 \wedge x = 1\}$
In general, the key to solving this kind of problem where one needs to find a precondition for a given statement and postcondition, is to know what the domain of the statement is and to assume that the statement is defined, from there we check if in the postcondition any change arise once the statement has been executed and from there we proceed to find the correct precondition.

Assuming the statement is defined, but not executing it yet, we can unfold the postcondition of the above case which will give us the following:

$\{(\texttt{dereference}\ \ p) > 1 \wedge x = 1 \wedge (p = \texttt{addressOf}(x) \vee p \neq \texttt{addressOf}(x))\}$

which expands to

$$\left\{\begin{array}{c}(\texttt{dereference}\ p) > 1 \wedge x = 1 \wedge p = \texttt{addressOf}(x) \\ \vee(\texttt{dereference}\ p) > 1 \wedge x = 1 \wedge p \neq \texttt{addressOf}(x)\end{array}\right\}$$

which is equivalent to

$$\left\{\begin{array}{c}x > 1 \wedge x = 1 \wedge p = \texttt{addressOf}(x) \\ \vee(\texttt{dereference}\ \ p) > 1 \wedge x = 1 \wedge p \neq \texttt{addressOf}(x))\end{array}\right\}$$

Now, executing the statement, which we assumed was defined, will yield:

$$\left\{\begin{array}{c}1 > 1\ \wedge\ 1 = 1\ \wedge\ p = \texttt{addressOf}(x) \\ \vee\ (\texttt{dereference}\ p) > 1\ \wedge\ 1 = 1\ \wedge\ p \neq \texttt{addressOf}(x)\end{array}\right\}$$

which simplifies to

$$\left\{(\texttt{dereference}\ p) > 1\ \wedge\ p \neq \texttt{addressOf}(x)\right\}$$

Now, from Definition 3.5 we know that the domain of the assignment statement alone in this case is the set of all data environments which contain the numerical variable x whose set of possible values contains, at the very least, the number 1. Since the resulting postcondition 6 is, in fact, a set that contains the required properties of the pointer p and also contains the domain of the assignment statement in this case, we can intersect the two to get the resulting correct precondition to the case above:

$$\left\{x \text{ is numerical } \wedge (\texttt{dereference}\ p) > 1\ \wedge\ p \neq \texttt{addressOf}(x)\right\}$$

□

Case 6.2 $\{?\}\ p := \texttt{addressOf}(g)\ \{\texttt{dereference}\ \ p > 1\}$
As before, lets assume that the assignment statement above is defined, and lets unfold the postcondition.

Note that the boolean expression $\texttt{dereference}\ \ p > 1$ implies, due to the definition of the pointer p, that the $Set(\texttt{dereference}\ p)$ must be a set that contains numbers so that the boolean expression itself is defined. Thus, the expanded postcondition would be:

$$\left\{\texttt{dereference}\ \ p > 1\ \wedge\ Set(\texttt{dereference}\ p) \text{ contains numbers}\right\}$$

Now, executing the assignment statement above, we have that the post-condition actually is:

$$\left\{\begin{array}{c}(\texttt{dereference addressOf}(g)) > 1 \\ \wedge\ \mathit{Set}(\texttt{dereference addressOf}(g)) \text{ contains numbers}\end{array}\right\}$$

which simplifies to

$$\big\{(\texttt{dereference addressOf}(g)) > 1\big\}$$

which in this case is the correct precondition we were looking for. □

7 Results

The proofs located in the fifth section (i.e. Result 1, and Result 2) show that [6] still hold true with a simple modification of some portions of [6]. That is, whenever a data environment appears in [6], that portion needs to be rewritten to simply include that the address of each program variable in the data environment is unique, but whenever the concatenation of an arbitrary number of data environments appear in [6], the following statement needs to be added since it ensures that the result of the operation will yield a data environment: "$\forall x \in d_0 \in \mathbb{K}\ \forall y \in d_1 \in \mathbb{K}$, $y = x$ or the memory address of x differs from memory address of y but not both", where $\mathbb{K}$ contains all the data environments entangled in the operation.

References

[1] Leveson, Nany. Turner, Clark S. *An investigation of the Therac-25 Accidents.*
`http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html`
Visited on November 17, 2012

[2] *Sunk by Windows NT.* Wired.com.
`http://www.wired.com/science/discoveries/news/1998/07/13987`
Visited on November 18, 2012

[3] Poulsen, Kevin. *Software Bug Contributed to Blackout.* SecuritiFocus.
`http://www.securityfocus.com/news/8016`
Visited on November 8, 2012

[4] Lions, J. L. et al. *ARIANE 5 Flight 501 Failure.*
`http://www.di.unito.it/~damiani/ariane5rep.html`
Visited on November 5, 2012

[5] Gleick, James. *A bug and a crash: Sometimes a bug is more than nuisance.* `http://www.around.com/ariane.html`
Visited on November 7, 2012

[6] Baber, Robert L. *Mathematically Rigorous Software Design.*
`http://www.cas.mcmaster.ca/~baber/Courses/46L03/`
Visited on November 23, 2012