System Modelling and Design

Generalised Substitutions and Proof Obligationss

Revision: 1.2, March 27, 2002

Ken Robinson

22nd March 2005

©Ken Robinson 2005

mailto::k.robinson@unsw.edu.au

Contents

1 Assigning Meanings to Programs **2**

1.1 Weakest Precondition and Strongest Postcondition 2

1.2 The Simple Substitution 3

1.3 The Machine State 3

1.4 The Semantics of the Simple Substitution 3

1.5 An example 3

1.6 Generalised Substitutions 4

1.7 Basic Substitutions 4

1.8 . . . Basic Substitutions 4

1.9 Skip 4

1.10 Multiple substitution 5

1.11 Parallel composition 5

1.12 Sequential composition 5

1.13 Preconditioned substitution 5

1.14 Guarded substitution 6

1.15 Alternative substitution 6

1.16 Non-Deterministic Substitutions 6

1.17 Unbounded choice 6

1.18 Choice from a set 7

1.19 Choice by predicate 7

2 Defining AMN constructs **7**

2.1 The IF-THEN-ELSE substitutions 7

2.2 The SELECT substitutions 8
2.3 Preconditioned substitutions 8
2.4 Unbounded choice 8
2.5 Distribution of Substitution through conjunction 8

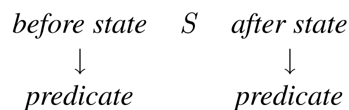
3 Computing proof obligations **8**
3.1 The Constraint Proof Obligations 9
3.2 The Context Proof Obligations 9
3.3 The Initialisation Proof Obligations 10
3.4 Operation Proof Obligations 10
3.5 Computing the Weakest Precondition 10

- To introduce the idea of a *substitutions* and in particular the *Generalised Substitution Language* (GSL), which is the basis of *B Method* (*B*).
- To introduce the formal semantics defining the meaning of each construct in the *GSL*.
- To show the relation between *GSL* and *AMN*.
- To show how *proof obligations* are computed.

1 Assigning Meanings to Programs

How do we assign a meaning to a program (or specification) construct?

If the construct acts on a state (set of variables) then we can consider the construct to be acting between a *before state* and an *after state* and we can represent the state by a *predicate* —a function from variables to Boolean:



1.1 Weakest Precondition and Strongest Postcondition

Now there are at least two questions that might be asked:

1. Given the before state and S, what is the *strongest* predicate on the possible after state?
2. Given an after state and S, what is the *weakest* predicate on the before state that will guarantee that S terminates in the after state?

We will generally refer to the before state as the *pre* state and the after state as the *post* state, then 1 represents the *strongest post condition* and 2 represents the *weakest precondition*.

1.2 The Simple Substitution

Substitutions are the constructs in B that specify *state changes*, or values to be bound to variables.

The simple substitution $x := E$ *sets the value of* x equal *to the value of the expression* E.

If x is free in E, then the *old* value of x is used in the evaluation of E.

If x is a state variable then this specifies a change of state.

1.3 The Machine State

An Abstract Machines has a set of variables.

We will often refer to *the state* of a machine.

The value of the state at any particular time is a *binding*, ie mapping, between variable *names* and variable *values*.

Thus, a state is a function from variables to values.

$state \in variables \rightarrow values$

where $variables$ is the actual set of variables for a particular machine.

1.4 The Semantics of the Simple Substitution

Suppose we wish the final state of a computation to satisfy some predicate R, and suppose we wish to achieve this my using the simple substitution $x := E$.

We would like to know *what constraint on the initial state,* before $x := E$*, will —is* sufficient *to— ensure this outcome*.

In particular, we would be interested in the *weakest —necessary* and *sufficient*— constraint on the initial state that would guarantee that we can achieve this outcome.

Let us write $[x := E]\ R$ to denote the *smallest* set of states from which the simple substitution $x := E$ will achieve a state that satisfies the predicate R.

The value of $[x := E]\ R$ is the predicate R with the value E substituted for every *free* instance of x in R.

Thus, the semantics of a construct known as a substitution is defined by a substitution in the algebraic sense.

1.5 An example

$[x := x + 1]\ x < y + 1$

$\equiv$

$x + 1 < y + 1$

$\equiv$

$x < y$

1.6 Generalised Substitutions

The semantics of the simple substitution is generalised to all basic constructs, which will be called *substitutions*.

If S is a general substitution, then

$[S]\ R$

denotes the set of states from which the substitution S will achieve a state satisfying the predicate R.

Note that $[S]$ is called a *predicate transformer*: a function from predicates to predicates.

1.7 Basic Substitutions ...

Description	Substitution	Definition
Simple	$xx := E$	$[xx := E]\ R \mathrel{\widehat=} (\lambda\, xx \cdot R)E$
Skip	*skip*	$[skip]\ R \mathrel{\widehat=} R$
Choice from set	$xx :\in S$	$[xx :\in S]\ R \mathrel{\widehat=}$
		$[@xx' \cdot xx' \in S \Longrightarrow xx := xx']\ R$
Choice by predicate	$xx : P$	$[xx : P]\ R \mathrel{\widehat=}$
		$[@xx' \cdot [xx := xx']P \Longrightarrow xx := xx']\ R$
Mulitple	$x, yy := E, F$	$[x, yy := E, F]\ R \mathrel{\widehat=} (\lambda(x, yy) \cdot R)(E, F)$

xx, yy are variables; x is a variable or list of variable identifiers; E, F are expressions; S is a set; P, R are predicates

Note: in B variable identifiers must have at least two characters.

1.8 ... Basic Substitutions

Description	Substitution	Definition
Parallel	$G \parallel H$	$[G \parallel H]\ R$ is $[G]\ R$ and $[H]\ R$ concurrently
Sequential	$G\ ;\ H$	$[G\ ;\ H]\ R \mathrel{\widehat=} [G]\ [H]\ R$
Preconditioned	$P \mid G$	$[P \mid G]\ R \mathrel{\widehat=} P \wedge [G]\ R$
Guarded	$P \Longrightarrow G$	$[P \Longrightarrow G]\ R \mathrel{\widehat=} P \Rightarrow [G]\ R$
Alternate	$G [] H$	$[G [] H]\ R \mathrel{\widehat=} [G]\ R \wedge [H]\ R$
Unbounded choice	$@z \cdot G$	$[@z \cdot G]\ R \mathrel{\widehat=} \forall z \cdot ([G]\ R)$

z is a list of variable identifiers; G, H are substitutions

Note: parallel composition is not given a substitution semantics. Instead parallel composition is handled by a set of rewrite rules.

1.9 Skip

$skip$

is the substitution that does not change the state.

Can be used to specify that the state *must not* change.

$[skip]\ R \mathrel{\widehat=} R$

If the state is required to satisfy R after $skip$ then it must satisfy R before the substitution.

1.10 Multiple substitution

$x, y := E, F$

concurrently substitutes multiple values into *distinct* multiple variables.

$[x, y := E, F]\ R$ is the concurrent substitution of E and F for all *free* instances of x and y in R, respectively.

$[x, y := y, x]\ x < y + 1 \equiv y < x + 1$

$[x, y := x + y, y - x]\ x > y \equiv x + y > y - x \equiv x > 0$

1.11 Parallel composition

$G \parallel H$

represents performing G and H concurrently.

$[G \parallel H]\ R$ is not given a formal definition in B.

Instead parallel substitutions are handled by rules, for example:

$xx := E \parallel yy := F == xx, yy := E, F$

where $==$ means *may be rewritten as*.

Parallel composition is the only form of composition available in B at the specification of top-level. It is also available in refinements, but not in implementations.

Parallel composition describes a single state change.

1.12 Sequential composition

$G\ ;\ H$

denotes performing substitution G followed *sequentially* by H

$[G\ ;\ H]\ R \mathrel{\widehat{=}} [G]\ ([H]\ R)$

Sequential composition is only available in B in refinements and implementations.

1.13 Preconditioned substitution

$P \mid G$ "*P preconditions G*"

substitution G on the assumption of the predicate P.

$[P \mid G]\ R \mathrel{\widehat{=}} P \wedge [G]\ R$

Note: the substitution $[G]\ R$ is strengthened by the precondition P.

In particular, when P is false, $[P \mid G]\ R$ is false.

The set of states satisfying false is the empty set of states; that is, *no state satisfies* false.

1.14 Guarded substitution

$P \Longrightarrow G$ "P *guards* G"

the substitution G guarded by the predicate P.

$[P \Longrightarrow G]\ R \mathrel{\widehat{=}} P \Rightarrow [G]\ R$

Note: when P is false then $[P \Longrightarrow G]\ R$ is true, independently of R!

The set of states satisfying true is *all states*; that is, *any* state satisfies true.

A single guarded substitution is a very strange beast that may not be implementable, since it appears that it is capable performing *miracles*!

Starting in any state, the substitution $\mathit{false} \Longrightarrow G$ will yield a state satisfying any predicate R, for any G!

So, start in any state; choose any G you like, skip will do; think of any state you would like to be in; run $\mathit{false} \Longrightarrow G$, and you'll be there!

1.15 Alternative substitution

$G \mathbin{[\!]} H$ "G alternative H

the substitution that chooses, nondeterministically, between substitutions G and H.

$[G \mathbin{[\!]} H]\ R \mathrel{\widehat{=}} [G]\ R \wedge [H]\ R$

Since $G \mathbin{[\!]} H$ might be either G or H, then $[G \mathbin{[\!]} H]\ R$ must be at least as strong as either

$[G]\ R$, *required if G is chosen*

or $[H]\ R$, *required if* H *is chosen*.

1.16 Non-Deterministic Substitutions

Some of the substitutions may be non-deterministic, that is they may specify state changes involving arbitrary choice.

B has a single unbounded choice substitution and two simple choice substitutions that are specialisations of the unbounded choice substitution.

Non-determinism is used frequently in specification, not because we wish to specify non-deterministic behaviour, but because there is often a choice and we don't care how the choice is resolved.

In many cases the choice will be resolved in the implementation. Non-determinism gives the implementor a choice.

1.17 Unbounded choice

$@z \cdot G$

Behave like the substitution G with the variables z chosen non-deterministically.

The semantics of $[@z \cdot G]\ R$ reflects the fact that $[G]\ R$ must be satisfied for all choices of z.

$[@z \cdot G]\ R \mathrel{\widehat{=}} \forall\, z \cdot ([G]\ R)$

There is an implication that in the general case the substitution G will behave miraculously in some parts of its domain of application.

1.18 Choice from a set

$xx :\in S$

Set xx to a value chosen arbitrarily from the set S.

The semantics is defined in terms of the unbounded choice substitution

$[xx :\in S]\ R \mathrel{\widehat{=}} [@xx' \cdot xx' \in S \Longrightarrow xx := xx']\ R$

Notice that the semantics of $xx :\in S$ involves a guard $xx' \in S$.

For all choices of xx' that are not elements of S, the guard will be *false*, and $[@xx' \cdot xx' \in S \Longrightarrow xx := xx']\ R$ will be trivially $true$.

Elsewhere, $[@xx' \cdot xx' \in S \Longrightarrow xx := xx']\ R$ reduces to $[xx := xx']\ R$.

If the set S is empty then a miracle is required.

1.19 Choice by predicate

$xx : P$

Set xx to any value that satisfies the predicate P.

Again the semantics is defined in terms of the unbounded choice substitution

$[xx : P]\ R \mathrel{\widehat{=}} [@xx' \cdot [xx := xx']\ P \Longrightarrow xx := xx']\ R$

Note: $[xx := xx']\ P$ is the substitution of the value xx' for xx in P.

Notice that for any value of xx' for which $[xx := xx']\ P$ is not $true$ the guarded substitution behaves like a miracle.

Elsewhere, the guarded substitution behaves like $xx := xx$, where $[xx := xx']\ P$ is $true$.

2 Defining AMN constructs

When writing machines, especially for use with a toolkit, we use the *Abstract Machine Notation (AMN)* to provide a *syntactically sugared* version of the substitutions that take on something of a programming notation appearance.

Notice that GSL can be used with the BToolkit, but the toolkit will translate the GSL into the equivalent AMN.

skip and simple and multiple substitutions have the same representation in both GSL and AMN.

2.1 The IF-THEN-ELSE substitutions

$$IF\ P\ THEN\ G\ ELSE\ H\ END \quad \widehat{=} \quad P \Longrightarrow G \mathbin{[\!]} \neg P \Longrightarrow H$$

Notice that the definition is an alternative of two guarded substitutions with mutually exclusive guards. This ensures that in any state where one of the guards is *false*, the other guard will be $true$. Thus, when

one of the guarded substitutions must behave like a miracle that other doesn't need to. This ensures that the construct has a real —non-miraculous— implementation.

$$\begin{array}{lcl} \mathit{IF}\ P\ \mathit{THEN}\ G\ \mathit{END} & \widehat{=} & \mathit{IF}\ P\ \mathit{THEN}\ G\ \mathit{ELSE}\ \mathit{skip}\ \mathit{END} \\ & \equiv & P \Longrightarrow G \mathbin{[\!]} \neg P \Longrightarrow \mathit{skip} \end{array}$$

Notice the mutually exclusive guards again, and the use of $skip$ to ensure no state change.

2.2 The SELECT substitutions

$$\begin{array}{l} \mathit{SELECT}\ P_1\ \mathit{THEN}\ G_1 \\ \mathit{WHEN}\ P_2\ \mathit{THEN}\ G_2 \\ \cdots \\ \mathit{END} \end{array} \quad \widehat{=} \quad P_1 \Longrightarrow G_1 \mathbin{[\!]} P_2 \Longrightarrow G_2 \mathbin{[\!]} \cdots$$

In any state where $P_1 \vee P_2 \ldots$ is not $true$, then all the guarded substitutions must behave as miracles.

$$\begin{array}{l} \mathit{SELECT}\ P_1\ \mathit{THEN}\ G_1 \\ \mathit{WHEN}\ P_2\ \mathit{THEN}\ G_2 \\ \cdots \\ \mathit{ELSE}\ G_n \\ \mathit{END} \end{array} \quad \widehat{=} \quad \begin{array}{l} P_1 \Longrightarrow G_1 \mathbin{[\!]} P_2 \Longrightarrow G_2 \mathbin{[\!]} \cdots \\ \mathbin{[\!]} \neg(P_1 \vee P_2 \ldots) \Longrightarrow G_n \end{array}$$

In all states, at least one guard is $true$.

2.3 Preconditioned substitutions

$\mathit{PRE}\ P\ \mathit{THEN}\ G\ \mathit{END} \mathrel{\widehat{=}} P \mid G$

Simply "syntactic sugar".

2.4 Unbounded choice

ANY z *WHERE* P *THEN* G *END* $\widehat{=}\ @\, z \cdot P \Longrightarrow G$

Notice that the predicate, P, in the *ANY* construct becomes a guard in the semantics, which has the consequence that

if the choice of values for the variables z *does not satisfy* P *then the construct will behave like a* miracle*!*

2.5 Distribution of Substitution through conjunction

It can be shown that

$[G]\,(P_1 \wedge P_2) \equiv [G]\,P_1 \wedge [G]\,P_2$

This allows us to give two smaller proof obligations rather than one larger proof obligation.

This property is important as state invariants and other constraints are frequently given as a conjunction.

3 Computing proof obligations

There are four sets of proof obligations computed for a machine:

Constraint an existence, or feasibility, proof that sets and constants given as machine parameters and satisfying the machine constraints clause exist.

Context an existence, or feasibility, proof that the sets and constants satisfying the properties clause exist.

Initialisation proof that the initialisation substitution establishes a state satisfying the invariant.

Operation for each operation, a proof that the operations maintains the state invariant. That is, given the operation is initiated in a state satisfying the invariant, the state will satisfy the invariant after the operation.

Given the distribution of substitution through conjunction, many of the above proof obligations will be broken into a number of simpler proof obligations.

3.1 The Constraint Proof Obligations

Consider a machine header:

$MACHINE\ A(X, n)$
$CONSTRAINTS\ C$
$\ldots$

where X is a set parameter, and n is a numeric parameter. There may be more than one of each type of parameter.

It is a constraint in B that X must be a non-empty set, and n must be a natural number.

The constraints clause C, in general, introduces extra constraints on X and n.

The constraints proof obligation introduces a check that the implicit and explicit constraints are consistent:

$\exists(X, n).(card(X) \in \mathbb{N}_1 \wedge n \in \mathbb{N} \wedge C)$

3.2 The Context Proof Obligations

Continuing the machine header to add sets, constants and properties:

$MACHINE\ A(X, n)$
$CONSTRAINTS\ C$
$SETS\ S$
$CONSTANTS\ K$
$PROPERTIES\ Q$
$\ldots$

Again there may be many (or no) sets and constants. Deferred sets in B are non-empty sets.

The context proof obligation is a consistency and existence check that there exists S and K satisfying Q, given X and n satisfying C:

$card(X) \in \mathbb{N}_1 \wedge n \in \mathbb{N} \wedge C \Rightarrow$
$\quad \exists(S, K).(card(S) \in \mathbb{N}_1 \wedge Q)$

3.3 The Initialisation Proof Obligations

We now extend the machine header to add:

$\ldots$
$VARIABLES\ V$
$INVARIANT\ I$
$INITIALISATION\ G$
$\ldots$

The initialisation substitution, G, must establish the machine invariant, given only the *constraints* and *properties*.

$card(X) \in \mathbb{N}_1 \land n \in \mathbb{N} \land C \land card(S) \in \mathbb{N}_1 \land Q \Rightarrow [G]\ I$

Note: in the above proof obligation and any other formula, all free variables are implicitly universally quantified.

3.4 Operation Proof Obligations

Maintaining the machine invariant

Consider an operation

$r \longleftarrow Op(args) \mathrel{\widehat{=}} PRE\ P\ THEN\ G\ END$

in a machine with an invariant I.

The invariant is $true$ before the operation and must be $true$ after the operation. Therefore, the proof obligation becomes

$constraints \land properties \land I \land P \Rightarrow [P \mid G]\ I$

$\equiv$

$constraints \land properties \land I \land P \Rightarrow P \land [G]\ I$

$\equiv$

$constraints \land properties \land I \land P \Rightarrow [G]\ I$

where $constraints \land properties \mathrel{\widehat{=}} card(X) \in \mathbb{N}_1 \land n \in \mathbb{N} \land C \land card(S) \in \mathbb{N}_1 \land Q$

3.5 Computing the Weakest Precondition

Consider an operation whose body is $PRE\ P\ THEN\ G\ END$ in a machine whose invariant is I. The proof obligation for the operation is:

$I \land P \Rightarrow [G]\ I$

$\equiv$

$P \land I \Rightarrow [G]\ I$

$\equiv$

$P \Rightarrow (I \Rightarrow [G]\ I)$

Now, the weakest solution for P in $P \Rightarrow Q$ is Q, so

so $I \Rightarrow [G]\ I$ is the weakest solution for P in $P \Rightarrow (I \Rightarrow [G]\ I)$

that is, $I \Rightarrow [G]\ I$ is the weakest value of P (precondition) that will ensure that the operation restores the invariant, I

This gives a justification to the use of proof obligations to compute preconditions.