# A Definition of Abstraction

Martin Ward
Computer Science Dept
Science Labs
South Rd
Durham DH1 3LE

January 17, 2003

**Abstract**

What does it mean to say that one program is "more abstract" then another? What is "abstract" about an abstract data type? What is the difference between a "high-level" program and a "low-level" program? In this paper we attempt to answer these questions by formally defining an abstraction relation between programs which matches our intuitive ideas about abstraction. The relation is based on examining the operational semantics of the programs, expressed as a set of traces (sequences of states) from a given initial state to a possible final state.

KEY WORDS:

Abstraction, Software Maintenance, Transformations, Refinement, Transformational Programming

## 1 Introduction

In discussing software development, refinement of specifications into programs, reverse engineering from programs into specifications, and other related areas, concepts such as "high-level program" verses "low-level program", "crossing levels of abstraction", "abstract data types", and so on are bandied about without always being given a clear definition. The concept of "refinement" has been formally defined: for example in [1,2,3,7]; but as we shall see below, the informal concept of abstraction would appear to be much sharper than the concept of refinement, since many programs which we would (informally) regard as very different in their degree of abstraction, are (formally) equivalent in the sense that each is a formal refinement of the other.

Some of the intuitive ideas about abstraction we would like to capture are listed below. These are the requirements which we would expect any abstraction relation to satisfy:

1. Abstract specifications say *what* a program does without necessarily saying *how* it does it.
2. Abstraction is a process of generalisation, removing restrictions, eliminating detail, removing inessential information (such as the algorithmic details).
3. Abstract specifications have "more potential implementations", moving to a lower level means restricting the number of potential implementations.

## 2 Examples

Some examples will help to fix our intuitive ideas about the different forms of abstraction:

1. Compare:
   (a) Calculate the product of $a$ and $b$ and store the result in $c$;
   (b) Calculate the product of $a$ and $b$ using only addition and store the result in $c$.

2. (a) A specification which assigns any value to $x$ which is larger than the value of $y$:

$$\langle x\rangle/\langle\rangle.(x > y)$$

   (b) A refinement of this is: $x := y + 1$.

3. (a) A recursive function (this form of recursion occurs for example in the solution of the famous "Towers of Hanoi" problem):

```
funct $F(n, x) \equiv$ if $n > 0$ then $F(n-1, \phi(n, F(n-1, x)))$
                                 else $x$ fi.
```

   (b) An equivalent iterative form is:

```
funct $F(n, x) \equiv$
    ⌈for $c := 2^n - 1$ step $-1$ to $1$ do
        $x := \phi(ntz(c) + 1, x)$ od;
     $x$⌋.
funct $ntz(c) \equiv$ "the number of trailing zeros in the binary representation of $c$".
```

4. (a) Some sorting examples: The first is a specification of a program which sorts the segment $a\,..\,b$ of array $A$:

$$\mathsf{SORT}(a, b) \;=_{\mathrm{DF}}\; A[a\,..\,b] := A'[a\,..\,b].(\mathsf{sorted}(A[a\,..\,b]) \wedge \mathsf{perm}(A[a\,..\,b], A'[a\,..\,b]))$$

   Where

$$\mathsf{sorted}(A[a\,..\,b] \;=_{\mathrm{DF}}\; \forall i, a \leqslant i < b.\, A[i] \leqslant A[i+1]$$

   and

$$\mathsf{perm}(A[a\,..\,b], A'[a\,..\,b]) \;=_{\mathrm{DF}}\; \exists \pi\colon a\,..\,b \rightarrowtail\!\!\!\!\twoheadrightarrow a\,..\,b.\, \forall i, a \leqslant i \leqslant b.\, A[i] = A'[\pi[i]]$$

   where $\pi\colon a\,..\,b \rightarrowtail\!\!\!\!\twoheadrightarrow a\,..\,b$ means *pi* is a bijection (a 1–1 and onto function) from the set $\{a, a+1, \ldots, b\}$ to itself, i.e. $\pi$ is a permutation of $a\,..\,b$;

   (b) The second is a specification of a quicksort program:

```
$\mathsf{QSORT}_1(a, b) =$
begin $p :\langle A[a\,..\,b], p\rangle := \langle A'[a\,..\,b], p'\rangle.$
  $(A'[a\,..\,p'-1] \leqslant A'[p'] \leqslant A'[p'+1\,..\,b] \wedge \mathsf{perm}(A[a\,..\,b], A'[a\,..\,b]));$
  $\mathsf{SORT}(a, p-1);$ $\mathsf{SORT}(p+1, b)$ end
```

   (c) The third ($\mathsf{QSORT}_2$) is a full implementation of the quicksort algorithm (for example using "median of three" partitioning, see [4,6]). [7] formally proves the equivalence of this algorithm to the specification $\mathsf{SORT}(a, b)$.

Each of these examples illustrates a different aspect of "abstraction", I would argue that in each case the first version is the most abstract, with later versions becoming more concrete. However, with the exception of case (2), all the examples are cases of formal equivalence.

Clearly a proper refinement of a specification (i.e. a refinement which is not equivalent) ought to be considered as "more concrete" than the specification, not least because some implementation freedom has been lost (see requirement 3). For similar reasons it is important to restrict the abstraction relation to programs and specifications which are already related by refinement or equivalence. However, as already noted, refinement by itself is not a sufficient test for abstraction.

A cursory examination of the examples reveals one obvious common feature: the more abstract versions are all shorter than the concrete versions. This leads to the following (rather naïve) definition of abstraction:

**Definition 1** *If* $\mathbf{S}_1$ *and* $\mathbf{S}_2$ *are statements such that* $\mathbf{S}_2$ *refines* $\mathbf{S}_1$ *then we say* $\mathbf{S}_1$ *is more abstract than* $\mathbf{S}_2$ *if and only if* $\mathbf{S}_1$ *is shorter than* $\mathbf{S}_2$*.*

This definition is unsatisfactory for several reasons. First we feel that abstraction is more of a semantic issue than can be captured in a crude syntactic test: for example, adding a long sequence of **skip** statements to an abstract specification does not turn it into a concrete implementation! This particular failing can be rectified by insisting on the application of a small set of "simplifying" transformations (such as **skip** deletion) to the programs before their sizes are compared. A more substantive counterexample is a program which carries out a fairly complex task with a few short lines of code. Here the high-level description of "what the program does" could turn out to be considerably longer than the program itself. For example consider the following graph-marking algorithm:

```
begin mark(root) where
  proc mark($x$) ≡ if $m[x] = 0$
                     then $m[x] := 1$; mark($l[x]$); mark($r[x]$) fi. end
```

This program marks all the nodes $x$ reachable from the root node root via unmarked nodes. For simplicity we assume that any unused pointers point to a special node which is always marked. The abstract specification involves defining when a node is reachable:

$$\begin{aligned}\mathsf{MARK} \;\equiv\; m := m'.\forall x.\,\big(\; & (x \in \mathsf{reachable}(root, m) \Rightarrow m'[x] = 1)\\ & \wedge\; (x \notin \mathsf{reachable}(root, m) \Rightarrow m'[x] = m[x])\big)\end{aligned}$$

where:

$$\begin{aligned}\mathsf{reachable}(\mathsf{root}, m) \;&=_{\mathrm{DF}}\; \bigcup_{n<\omega} \mathsf{reachable}_n(\mathsf{root}, m)\\ \mathsf{reachable}_0(\mathsf{root}, m) \;&=_{\mathrm{DF}}\; \{\mathsf{root}\}\\ \mathsf{reachable}_{n+1}(\mathsf{root}, m) \;&=_{\mathrm{DF}}\; \mathsf{reachable}_n(\mathsf{root}, m)\\ &\quad \cup \{\, y \mid \exists x \in \mathsf{reachable}_n(\mathsf{root}, m).\,(y = l[x] \;\vee\; y = r[x]) \;\wedge\; m[y] = 0\,\}\end{aligned}$$

i.e. $\mathsf{reachable}_n(\mathsf{root}, m)$ is the set of nodes reachable from root in $n$ or fewer steps through a sequence of nodes which are unmarked in $m$.

An alternative definition of reachable which may correspond more closely to the intuitive idea, is to define a reachable node to be an unmarked node for which there is a path of unmarked nodes reaching from the root to that node:

$$\begin{aligned}\mathsf{reachable}(\mathsf{root}, m) \;&=_{\mathrm{DF}}\; \{\, x \mid \exists p \in \mathsf{paths}(\mathsf{root}, m).\, p[\ell(p)] = x\,\}\\ \mathsf{paths}(\mathsf{root}, m) \;&=_{\mathrm{DF}}\; \bigcup_{n<\omega} \{\langle x_1, \dots, x_n\rangle \mid x_1 = \mathsf{root} \;\wedge\; \forall i, 1 \leqslant i \leqslant n.\, m[x_i] = 0\\ &\qquad\qquad \wedge\; \forall i, 1 \leqslant i < n.\,(x_{i+1} = l[x_i] \;\vee\; x_{i+1} = r[x_i])\}\end{aligned}$$

Either of these definitions results in an abstract program which is considerably longer than the recursive implementation.

We are looking for a *semantic* definition of abstraction: as discussed above, denotational semantics alone are insufficient to express the relation so we will examine operational semantics.

In [8] and [7] we introduced a wide-spectrum programming and specification language (called WSL) with its formal syntax and denotational semantics. A *proper state* $s$ consists of a finite non-empty set $V$ of variables, each of which is assigned a value taken from the universal set of values, $\mathcal{H}$. The special state $\perp$ is used to denote nontermination or error. $V_{\mathcal{H}}$ denotes the set of all state on $V$ and $\mathcal{H}$ (including $\perp$). A WSL program $\mathbf{S}$, executing from an initial state $s \in V_{\mathcal{H}}$, may either run forever without terminating (in which case the "final state" is $\perp$), or must terminate in some state $t \in W_{\mathcal{H}}$ for some set of final variables $W$. (The set $W$ is deducible from $V$ and $\mathbf{S}$). Since WSL programs may be nondeterministic, there may be a set of possible final states for each initial state. So the denotational semantics of a WSL program can be given by a *state transformation* $f$, a function from $V_{\mathcal{H}}$ to $\wp(W_{\mathcal{H}})$, which maps each initial state $s$ to the set $f(s)$ of possible final states.

## 3 Operational Semantics

State transformations are sufficient to express the denotational semantics of programs and specifications. However, to define our abstraction relation we need a more "detailed" semantics, namely operational semantics. The operational semantics of a program gives for each initial and final state the set of traces (sequences of intermediate states) which the program passes through.

**Definition 2** *Traces:* A *trace* from finite non-empty sets of variables $V$ to $W$ on a set $\mathcal{H}$ of values is a finite sequence of states of length $\geqslant 2$ whose first element is in $V_{\mathcal{H}}$ and final element in $W_{\mathcal{H}}$. The length of a trace $\sigma$ is $\ell(\sigma)$, the first element is $\sigma[1]$ and the last element is $\sigma[\ell(\sigma)]$. A subsequence of $\sigma$ may be denoted $\sigma[a \mathinner{..} b]$. The concatenation of two traces $\rho$ and $\sigma$ is denoted $\rho \mathbin{+\!\!+} \sigma$

**Definition 3** *State trace:* A *state trace* $T$ from $V$ to $W$ is a set of traces from $V$ to $W$ on $\mathcal{H}$ with each trace $\sigma \in T$ having its first element in $V_{\mathcal{H}}$ and last element in $W_{\mathcal{H}}$. If the trace $\sigma \in T$ includes $\perp$ as its $n$th element (for $n > 1$) then $T$ must also include all possible ways of extending $\sigma$ from the $(n-1)$th element onwards. Let $T_{\mathcal{H}}(V, W)$ denote the set of all state traces from $V$ to $W$ on $\mathcal{H}$ and let $\Gamma_{VW\mathcal{H}}$ be the set of all traces from $V$ to $W$ on $\mathcal{H}$. Then:

$$\begin{aligned} T \in T_{\mathcal{H}}(V, W) \iff \langle \perp, \perp \rangle \in T \;\wedge\; & \forall \sigma \in T.\, \big(\sigma[1] \in V_{\mathcal{H}} \;\wedge\; \sigma[\ell(\sigma)] \in W_{\mathcal{H}} \\ & \wedge\; \forall i, 2 \leqslant i \leqslant \ell(\sigma).\, (\sigma[i] = \perp \Rightarrow \forall \rho \in \Gamma_{VW\mathcal{H}}.\, \sigma[1 \mathinner{..} i-1] \mathbin{+\!\!+} \rho \in T)\big) \end{aligned}$$

For each state trace $T$ there corresponds a state transformation, $f_T$ formed by taking $f_T(s)$ to be the set of final elements of the traces in $T$ whose initial element is $s$, i.e.

$$f_T(s) \;=_{\text{DF}}\; \{\, t \in W_{\mathcal{H}} \mid \exists \sigma \in T.\, (\sigma[1] = s \;\wedge\; \sigma[\ell(\sigma)] = t) \,\}$$

In [5,7] the semantics of state transformations are further developed and used to prove various refinements and transformations of programs.

If we examine the operational semantics of the various examples we note that the more concrete versions are either proper refinements of the abstract cases, or have more states in their traces (compare $\mathsf{QSORT}_1$ which contains a specification statement where $\mathsf{QSORT}_2$ has a loop), or have more (local) variables in the inner states in their traces (the iterative version of example (3a) uses the local variable $c$). The third case is expressed in this definition of abstraction on states:

**Definition 4** *Abstraction on states:* If $s \in V_{\mathcal{H}}$ and $s' \in V'_{\mathcal{H}}$ are states where $V \subseteq V'$ and $\forall x \in V.\, s(x) = s'(x)$ (i.e. $s$ and $s'$ have the same values on variables in $V$) then we say $s$ is more abstract than $s'$ (or $s'$ is more concrete than $s$) and write $s \sqsubseteq s'$

We use this relation to define abstraction between sequences of states, where the more concrete sequence may "fill in" gaps in the abstract sequence:

**Definition 5** *Abstraction on state sequences:* If $\rho = \langle s_1, \dots, s_n \rangle$ and $\rho' = \langle s'_1, \dots, s'_m \rangle$ are sequences of states with $s_1, s'_1 \in V_{\mathcal{H}}$ and $s_n, s'_m \in W_{\mathcal{H}}$ and $s_1 = s'_1$ and $s_n = s'_m$ and $n, m > 1$ and there is a 1-1 increasing function $\pi$ from $\{2, \dots, n-1\}$ to $\{2, \dots, m-1\}$ such that $\forall i, 1 < i < n.\, s_i \sqsubseteq s'_{\pi(i)}$ then we say that $\rho$ and is more abstract than $\rho'$ and write $\rho \sqsubseteq \rho'$.

Finally, this extends to a definition of abstraction on state traces:

**Definition 6** *Abstraction on state traces:* If $T$ and $T'$ are state traces in $T_{\mathcal{H}}(V, W)$ and if $\forall \rho \in T. \exists \rho' \in T'. (\rho \sqsubseteq \rho')$ then we say that $T$ is more abstract than $T'$ and write $T \sqsubseteq T'$.

This definition satisfies the Lemma:

**Lemma 1** *For any state traces $T, T' \in T_{\mathcal{H}}(V, W)$ with corresponding state transformation $P, P' \in F_{\mathcal{H}}(V, W)$:*

$$\text{If} \quad T \sqsubseteq T' \quad \text{then} \quad P \leqslant P'$$

In other words, a concrete version of an abstract program is always a refinement of it. The converse does not hold in general: the sorting programs are all equivalent but clearly at different levels of abstraction.

The most abstract possible program is also the least refined, namely **abort**. This fits with our intuition of abstraction as the removal of information: in some sense **abort** contains no information at all and does not restrict the implementor in any way.

## 3.1 The Replacement Theorem

An important property for any notion of refinement is the replacement property: if any component of a statement is replaced by any refinement then the resulting statement is a refinement of the original one. This is easily proved by an induction, on a lexical order of: (i) The depth of recursion nesting; (ii) The length of the program text.

We have a corresponding theorem for the abstraction relation: if we replace any component of a program by a more abstract (more concrete) component then the whole program becomes more abstract (concrete).

## 4 Non-Semantic Specifications

We have yet to consider in detail the first example which considers the following specifications:

1. Calculate the product of $a$ and $b$ and store the result in $c$.
2. Calculate the product of $a$ and $b$ using only addition and store the result in $c$.

The first of these specifications may be expressed as the single atomic specification $\langle c \rangle / \langle \rangle . (c = a.b)$, which assigns some value to $c$ such that the condition $c = a.b$ is satisfied. The second specification says something about the kind of steps allowed in the computation, it cannot therefore be expressed simply as a specification statement. A specification statement only defines the denotational semantics, but this specification puts a restriction on the operational semantics: in this case the value of each variable in each state must be either a known constant or a sum or difference of values of variables in the previous state. One way of expressing this restriction is as follows:

```
begin $a_0 := a$; $b_0 := b$; $n :=$ "some positive integer" :
  for $i := 1$ step 1 to $n$ do
      Carry out some addition or subtraction
      or assign a constant value to a variable
      ... od;
  $[c = a_0.b_0]$ end
```

Here the **if** statement in the loop picks a random addition/subtraction operation between any two variables and the final guard ensures that the outcome of these operations results in $c$ having the value $a_0.b_0$. (See [7] for a definition of the *guard statement*). We claim that this correctly expresses the specification in the sense that this program meets the specification and is more abstract than any other program which meets the specification. Note however that the following program could be argued as meeting our specification although it is clearly against the spirit of the informal specification:

```
begin $a_0 := a$; $b_0 := b$; $n := a.b$ :
  $c := 0$;
  for $i := 1$ step $1$ to $n$ do
      $c := c + 1$ od
```

## 5 Conclusion

In this paper we have sought to provide a formal definition of an "abstraction" relation which corresponds more closely to the intuitive ideas of abstract and concrete programs, and high-level verses low-level programs. A simple syntactic definition (size) is shown to be inadequate, and any definition of abstraction which is based only on the denotational semantics of a pair of programs is also shown to be inadequate. Our definition is therefore based on the operational semantics of programs: a program $\mathbf{S}_1$ is an abstraction of another program $\mathbf{S}_2$ if each of the possible execution sequences for $\mathbf{S}_1$ consists of a subsequence of a possible execution sequence for $\mathbf{S}_2$.

## 6 References

[1] R. J. R. Back, *Correctness Preserving Program Refinements*, Mathematical Centre Tracts #131, Mathematisch Centrum, Amsterdam, 1980.

[2] R. J. R. Back & J. von Wright, "Refinement Concepts Formalised in Higher-Order Logic," *Formal Aspects of Computing* 2 (1990), 247–272.

[3] C. C. Morgan, *Programming from Specifications*, Prentice-Hall, Englewood Cliffs, NJ, 1994, Second Edition.

[4] R. Sedgewick, *Algorithms*, Addison Wesley, Reading, MA, 1988.

[5] M. Ward, "Proving Program Refinements and Transformations," Oxford University, DPhil Thesis, 1989.

[6] M. Ward, "Derivation of a Sorting Algorithm," Durham University, Technical Report, 1990, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/sorting-t.ps.gz⟩.

[7] M. Ward, "Foundations for a Practical Theory of Program Refinement and Transformation," Durham University, Technical Report, 1994, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/foundation2-t.ps.gz⟩.

[8] M. Ward, "Abstracting a Specification from Code," *J. Software Maintenance: Research and Practice* 5 (June, 1993), 101–122, ⟨http://www.dur.ac.uk/~dcs0mpw/martin/papers/prog-spec.ps.gz⟩.