A Non-Standard Semantics for Program Slicing and Dependence Analysis

Sebastian Danicic [a] Mark Harman [b] John Howroyd [c]
Lahcen Ouarbya [a]

[a] *Department of Mathematical and Computing Sciences Goldsmiths College, University of London, New Cross, London SE14 6NW*

[b] *Department of Computer Science, King's College London, Strand, London, WC2R 2LS.*

[c] *@UK PLC, 5 Jupiter House, Calleva Park, Aldermaston, Berkshire, RG7 8NN.*

Abstract

We introduce a new non-strict semantics for a simple `while` language. We demonstrate the this semantics allows us to give a denotational definition of variable dependence, neededness, which is consistent with program slicing. Unlike other semantics used in variable dependence, our semantics is substitutive. We prove that our semantics is preserved by traditional slicing algorithms.

Key words: Program Slicing, Non-Standard Semantics

1 Introduction

Program slicing [1] produces simpler programs from complicated ones and so can be thought of as a form of program transformation. Traditional program slicing [1–4] is a technique for isolating the components of a program which are concerned with the computation of a single variable or a set of variables at some point in the program. Slices are constructed with respect to a slicing criterion, $\langle V, n\rangle$, for some set of variables V and a program point n. Weiser's definition of program slicing is based on statement deletion. A slice of a program P consists of any subset of statements of P preserving the behaviour of the original program with respect to slicing criterion in all states where the original program terminates. Program slicing has many applications including reverse engineering [5,6], program comprehension [7,8], software maintenance [9–12], debugging [13–15,3], testing [16–20], component re-use [21,22], program integration [23,24], and software metrics [25–27]. There are several surveys of slicing techniques, applications and variations [28–31].

```
x:=1;
while (x>0) {y:=y+1;}
x:=5;
```

Fig. 1. A simple program P

According to Weiser [4], a program and its (end) slice must agree with respect to the set of variables in the slicing criterion. In other words, if we run the original program and the slice, then, in all states where the original terminates, the slice must also terminate with the same final values for the variables in the slicing criterion. This is the correctness criterion that needs to be proved for any slicing algorithm. The behaviour of the slice in states where the original does not terminate is left undefined. In fact, traditional slicing algorithms sometimes introduce termination: the standard semantics of a program is thus, less defined than the semantics of some of its slices. For example, consider the program P in Figure 1. Clearly, P does not terminate, the final state after executing P is always $\perp$. However, the slice of P with respect to `x` is just `x:=5`, which terminates in all states.

The standard semantics loses all semantic information beyond infinite loops. Therefore, it loses all information about control and data dependencies, which is essential for program slicing. Because of this it is very hard to prove correctness of slicing algorithm without the use some intermediate graph representation to track variable dependencies. Weiser [1] in his thesis used the control flow graph as intermediate representation to prove correctness of his slicing algorithm. Horwitz et al. [32] have shown that program dependence graphs [33,34], captures both control and data dependencies which are essential for program slicing. They show that two programs with the same program dependence graph have the same semantics. Since then much research has been carried out to define a semantics of program dependence graphs [35]. Cartwright and Felleisen in [35] were first to observe and discuss that if a semantics is to be useful to investigate semantic properties of program slicing, it has to be preserved by slicing algorithms. In [35], they defined a non-strict semantics, called *lazy semantics*, for program dependence graphs of a simple *while* language, and claimed that their semantics is preserved by slicing algorithms. For example, using the lazy semantics of Cartwright and Felleisen [35] the lazy value of the variable `x` after executing the program in Figure 1 is 5. Other efforts give a construction definition of the program dependence graph by transforming the denotational semantics of imperative languages. The fact that these semantics are defined for some intermediate graphs representations instead of the programming language itself makes it difficult to model (prove correctness) program manipulation techniques such as slicing. Before we can start, a semantics of the intermediate structure is required as well as mappings between programs and these intermediate structures in both directions.

Giacobazzi and Mastroeni [36] argued that it is unnatural to try to prove correctness of slicing properties using the standard semantics. They also argued that if a semantics is to be useful for modelling kinds of program manipulation such as slicing it should be able to capture semantic information 'beyond infinite loops' and be compositional. They do not consider the standard definition of compositionality of the semantics, where the semantics of the program is defined in terms of the semantics of its sub-program. Their definition of compositionality is restricted to a sequence of statements only. We call this property *sequentiality* of the semantics instead of compositionality, to avoid any confusion with the standard definition of compositionality. They use transfinite states traces of programs [37] and show the existence of such semantics using domain equations. They introduce a non-standard semantics, called *transfinite semantics* using a metric structure on their value domains. *Transfinite semantics* of a program is defined in terms of the set of all *possibly transfinite computations*: computations whose length can be any ordinal, finite or infinite.

The aim of this paper is to investigate slicing without intermediate structures. We regard the intermediate structures as mere 'implementation details'. Everything, the slicing algorithm and the semantics of the program language and the 'correctness criteria' of slicing are now expressed denotationally. This allows the possibility of using the full power and elegance of denotational semantics in definitions and correctness proofs.

1.1 Variable Dependence — Neededness

Central to slicing is the concept of *variable dependence* (or *neededness* as we call it): the set of variables *needed* by a set of variables V in program P, noted as $\mathbb{N}(P, V)$. Intuitively, this is the set of variables whose initial value 'may affect' the final value of at least one variable v in V after executing P. Our aim is to make the phrase 'may affect' semantically precise.

Neededness should be *partial standard semantically discriminating (PSSD).* This corresponds to our intuitive understanding of neededness. i.e. if x and y are variables such that there exists two initial states σ_1 and σ_2, differing only on y, such that the meaning of P gives rise to final terminating states with different values of x. Then y should be *needed* by x with respect to P ($y \in \mathbb{N}(P, \{x\})$).

```
x:=1;
y:=1;
while (x>0) {y:=y + 1;}
if (y>0)  then  x:=x;
```

Program P_1.

```
x:=1;
y:=1;
while (x>0) {y:=y + 1;}
if (y>0)  then  skip;
```

Program P_2.

Fig. 2. P_1 and P_2 do not have to same lazy semantics of Cartwright and Felleisen w.r.t. x.

Finally we require neededness to be sub-sequential, in the sense that:

$$\mathbb{N}(P;Q,V) \subseteq \mathbb{N}(P,\mathbb{N}(Q,V)).$$

If this was not the case, then it would mean that there was a variable z which affects the value of x in $P;Q$ but for no variable, k, which affects the value of x in Q, does z affect the value of k in P.

Standard semantics loses precision in the presence of infinite loops. Due to issues regarding non–termination, it turns out to be hard, if it is not impossible, to define neededness in terms of the standard semantics. Much of our research was trying to find a semantics which allowed us to define neededness satisfactorily that was consistent with standard semantics.

1.2 Substitutivity

Definition 1 (Substitutivity) *A semantic analysis is described as substitutive if, a sub-program Q of a program P can be replaced with another semantically equivalent sub-program, Q', and guarantee that the resulting program P' is semantically equivalent to our original program P.*

Substitutivity of the semantics simplifies correctness proofs for the sorts of transformations described in this paper and others, such as those used in amorphous slicing [38–40] where the program has to preserve only the semantics but not necessarily the syntax.

Although the lazy semantics of Cartwright and Felleisen [35] is able to look beyond an infinite loop, it loses precision for all variables defined in the body of an *if* or *while* statement in states where their corresponding predicate is evaluated to $\perp$. This is due to the fact that the evaluation of any expression demands its controlled predicate to be evaluated first. As a result of this the lazy semantics of Cartwright and Felleisen [35] is not substitutive.

```
x:=1;
y:=1;
while (x>0) {y:=-y;}
if (y>0)  then  x:=x;
```

Program P_1.

```
x:=1;
y:=1;
while (x>0) {y:=-y;}
if (y>0)  then  skip;
```

Program P_2.

Fig. 3. P_1 and P_2 do not have the same transfinite semantics w.r.t. `x`.

For example, in the program P_1 defined in Figure 2, the value of the variable `y` "after" executing the infinite loop is undefined, and thus, so is the value of the *if* predicate. Therefore, the final value of the variable `x` demands the evaluation of an undefined predicate and hence, using the lazy semantics by Cartwright and Felleisen [35], the final value of the variable x after executing the program P_1 is $\perp$. The assignment, `x:=x;` and `skip` have the same lazy semantics, then if this semantics is substitutive, then, the program P_1 and P_2 should be equivalent with respect to it. This however, is not the case as the final value when executing the program P_2 is 1, which is different from $\perp$ in the case of P_1.

Giacobazzi and Mastroeni [36] have illustrated the importance of sequentiality [1] of semantics. Nothing is said about substitutivity. Unsurprisingly their transfinite semantics is not substitutive. If an assignment to a variable `x` is controlled by an undefined predicate [2], then the transfinite semantics will map `x` to $\perp$. This implies that the transfinite semantics of Giacobazzi and Mastroeni [36] is not substitutive [3] : we can not replace a part of program with an equivalent program and preserve the semantics of the original program.

Unlike the lazy semantics of Cartwright and Felleisen [35], the two programs P_1 and P_2 in Figure 2 have the same transfinite semantics as the value of the variable y after executing the first infinite loop is ω which is always greater than 0.

Now consider the two programs P_1 and P_2 in Figure 3. P_1 and P_2 do not have the same transfinite semantics as the assignment `x:=x` is controlled by an undefined predicate.

The main contribution of this paper is to give a denotational definition of a

[1] Compositional with regards to sequences only.
[2] The predicate is evaluated to $\perp$.
[3] Private communication

non-strict semantics, which is substitutive, preserved by slicing algorithms and which is consistent with the standard semantics for terminating programs. We first remind the reader of some results of the standard denotational semantics.

2 Standard Denotational Semantics

Denotational semantics [41], enables mathematical meaning to be given to programming languages. It combines mathematic rigour and notational elegance [42].

In denotational semantics [41], a state, $\sigma \in \Sigma$, is a mapping from program variables in Variables to values in a set V.

$$\Sigma_\perp = \Sigma \cup \perp = [Variables \mapsto V] \cup \perp.$$

For example, the function $\sigma = \{x \mapsto 1, y \mapsto 2, z \mapsto 3\}$ is the state where the value of x is 1, the value of y is 2 and the value of z is 3. The meaning of a program is given by a function from states to states:

$$\mathcal{M} : P \longrightarrow \Sigma_\perp \longmapsto \Sigma_\perp.$$

Where P is the set of all programs. $\mathcal{M}[\![p]\!]\sigma$ represent the final state after executing the program p in the initial state σ. If the program p does not terminate in initial state σ, then $\mathcal{M}[\![p]\!]\sigma$ has the special value $\perp$, known as bottom. In standard semantics, in the bottom state all variables are deemed to have the value $\perp$. The bottom state is the state that maps every variable name to $\perp$. The final value of variable x after executing p in initial state σ is thus written $(\mathcal{M}[\![p]\!]\sigma)x$.

2.0.1 Ordering on States

In the standard denotational semantics [41], the ordering on states is such that two distinct non-terminating states are incomparable and $\perp$ is less than every state. The reason an ordering is required is that the meaning of loop is defined in terms of the least fixed point.

2.0.2 Evaluating Expressions

The meaning of an expression e is given by the function $\mathcal{E}$. It evaluates an expression in state to give a value.

$$\mathcal{E} \;:\; E \to \Sigma_\perp \longmapsto V_\perp$$

where

$$V_\perp = V \cup \{\perp\}.$$

2.0.3 Strictness of $\mathcal{E}$ in standard semantics

A function is strict if it gives $\perp$ when applied to $\perp$. In standard semantics $\mathcal{E}$ is strict. In other words, evaluating every expression in the $\perp$ state will give the $\perp$ value. Figure 4 shows the meaning of each construct of the language considered using the standard semantics.

Using Kleene's Theorem [43] the least fixed point of a monotonic functional is the limit of an ascending chain of functions. From this it follows that:

Lemma 2 $\qquad \mathcal{M}[\![\mathit{while}\ (B)\ S]\!] = \bigsqcup_{i=o}^{\infty} F_i$

Where: $F_0 = \lambda\sigma.\ \perp, \qquad F_{i+1} = \lambda\sigma.\mathcal{E}[\![B]\!]_\sigma \to F_i(\mathcal{M}[\![S]\!]\sigma),\ \sigma.$
and, $F_i \sqsubseteq F_{i+1} \quad \forall i \geq 0.$

2.1 Unfolding of while Loops

In [44,45], unfoldings of a *while* loop were defined. The n^{th} unfolding of the loop , $\mathcal{W}_n$, 'agreed' with the loop in all states where the loop terminates in n or fewer iterations. The meaning of the n^{th} unfolding when applied to any other state is $\perp$. The $(n+1)^{th}$ unfolding is defined recursively in terms of the n^{th} unfolding below:

Definition 3 (Unfoldings)

$\mathcal{W}_0(B, S) = \mathit{abort}$
$\mathcal{W}_{n+1}(B, S) = \mathit{if}\ (B)\ \mathit{then}\ S; \mathcal{W}_n(B, S)\ \mathit{else}\ \texttt{skip}$

for example the unfoldings of $\mathit{while}(x > 0)\ x{:=}x + y;$ statement are defined as follows:

$\mathcal{W}_0 = \mathit{abort},$

$\mathcal{W}_1 = \mathit{if}\ (x > 0)\ \mathit{then}\ x{:=}x + y; \mathit{abort}\ \mathit{else}\ \texttt{skip}$

$\mathcal{W}_2 = \mathit{if}\ (x > 0)\ \mathit{then}\ x{:=}x + y; \mathcal{W}_1\ \mathit{else}\ \texttt{skip}$

$\vdots$

$\mathcal{W}_n = \mathit{if}\ (x > 0)\ \mathit{then}\ x{:=}x + y; \mathcal{W}_{n-1}\ \mathit{else}\ \texttt{skip}$

We now show that the meaning of the *while* loop is the limit of the meanings of its unfoldings.

$\mathcal{M} : P \to \Sigma_\perp \to \Sigma_\perp$

skip statement

$$\mathcal{M}[\![\texttt{skip}]\!] = \lambda\sigma \cdot \sigma$$

abort statement

$$\mathcal{M}[\![abort]\!] = \lambda\sigma \cdot \perp$$

Assignment statements

$$\mathcal{M}[\![\texttt{x:=e}]\!] = \lambda\sigma \ \cdot \ \sigma[x \ \leftarrow \ \mathcal{E}[\![e]\!]\sigma]$$

(where the notation $\sigma[x \ \leftarrow e]$ represents the function which is the same as σ except that x gets mapped to e.)

Sequences of statements

$$\mathcal{M}[\![S_1; S_2]\!] = \lambda\sigma \ \cdot \ \mathcal{M}[\![S_2]\!](\mathcal{M}[\![S_1]\!]\sigma)$$

if statements

$$\mathcal{M}[\![if(B) \ then \ S_1 \ else \ S_2]\!] = \lambda\sigma \ \cdot \ \mathcal{E}[\![B]\!]\sigma \to \mathcal{M}[\![S_1]\!]\sigma, \mathcal{M}[\![S_2]\!]\sigma$$

(where the notation $e \to b, c$ is the expression which yields b if e is True and c if e is False).

while loops

$$\mathcal{M}[\![\ while(B) \ S]\!] = \ fix\ (\lambda f \cdot \lambda\sigma \cdot \mathcal{E}[\![B]\!]\sigma \to f(\mathcal{M}[\![S]\!]\sigma), \sigma))$$

Fig. 4. Standard denotational semantics.

Lemma 4 $\mathcal{M}[\![while\ (B)\ S]\!] \ = \ \bigsqcup_{i=0}^{\infty} \mathcal{M}[\![\mathcal{W}_i(B, S)]\!]$

Proof.
From Lemma 2, we only need to prove that the lazy meaning of the *ith* unfolding is the same as F_i. We prove this lemma by induction on i. The base case is trivial as $\mathcal{M}[\![\mathcal{W}_0(B, S)]\!]\sigma = \mathcal{M}[\![abort]\!]_\sigma = \perp\sigma = F_0(\sigma)$ for all $\sigma \in \Sigma$. Induction hypothesis: Assume for all $\sigma \in \Sigma$, $\mathcal{M}[\![\mathcal{W}_i(B, S)]\!]\sigma \ = \ F_i(\sigma)$ and

show that $\mathcal{M}[\![\mathcal{W}_{i+1}(B,S)]\!]_\sigma = F_{i+1}(\sigma)$, for all $\sigma \in \Sigma$.

$$
\begin{aligned}
&\mathcal{M}[\![\mathcal{W}_{i+1}(B,S)]\!]\sigma \\
&\quad = \mathcal{M}[\![\mathit{if}\ (B)\ S; \mathcal{W}_i(B,S)\ \mathit{else}\ \texttt{skip}]\!]\sigma && \text{(By definition)}\\
&\quad = \mathcal{E}[\![B]\!]\sigma \rightarrow \mathcal{M}[\![S; \mathcal{W}_i(B,S)]\!]\sigma,\ \sigma && \text{(By \textit{if} statement rule)}\\
&\quad = \mathcal{E}[\![B]\!]\sigma \rightarrow \mathcal{M}[\![\mathcal{W}_i(B,S)]\!](\mathcal{M}[\![S]\!]\sigma),\ \sigma && \text{(By sequence rule)}\\
&\quad = \mathcal{E}[\![B]\!]\sigma \rightarrow F_i(\mathcal{M}[\![S]\!]\sigma),\ \sigma && \text{(By Induction hypothesis)}\\
&\quad = F_{i+1}(\sigma).
\end{aligned}
$$

Hence, for all $i \geq 0$, $\mathcal{M}[\![\mathcal{W}_i(B,S)]\!] = F_i$. From which it follows immediately that the meaning of a while loop is the limit of the meanings of its unfoldings.

3 A Lazy Denotational Semantics

In our semantics, same as in lazy semantics [35], variables are allowed to have a $\perp$ value, i.e. some variables are mapped to $\perp$ and others to well defined values. Therefore we can have partially defined states. The set of such states is denoted as $\Sigma^\perp$.

$$\Sigma^\perp : Variables \rightarrow V_\perp.$$

Where

$$V_\perp = V \cup \perp.$$

$V_\perp$ is the union of the set of defined values, V, and the bottom value, $\perp$.

The ordering on $\Sigma^\perp$ is now a richer ordering than on $\Sigma_\perp$ as used in the standard semantics where all non $\perp$ states were incomparable. For these partially defined states,

$$\sigma_1 \sqsubseteq \sigma_2 \iff \sigma_2(\mathtt{x}) = \perp \implies \sigma_1(\mathtt{x}) = \perp \quad \forall \mathtt{x} \in Variables.$$

Since variables can be mapped to $\perp$ we now have the possibility that evaluating an expression in a partially defined state can yield $\perp$. A variable x referenced by an expression e does not necessarily mean it contributes to the evaluation of e, for example, the value of the expression $x - x$ is independent of the value of x. We define a function, det, which takes an expression e and returns the set of variables referenced by e which contribute to the evaluation of e.

Definition 5 (det) *The function* $\mathrm{det} : E \rightarrow P(Variables)$ *is defined to reflect the variables determining the value of expressions. Given an expression e, we* *say that a variable x is in* $\det(e)$ *if and only if there exists two states,* σ_1 *and* σ_2 *in* $\Sigma_\perp$*, differing only on the value of* x *with* $\mathcal{E}_{lazy}[\![e]\!]\sigma_1 \neq \mathcal{E}_{lazy}[\![e]\!]\sigma_2$*, where* $\mathcal{E}_{lazy}[\![e]\!]\sigma$ *is the lazy value of the expression* e *in a state* σ *defined below.*

If $\det(e)$ contains a variable which has $\perp$ as a value in σ, then the whole expression is evaluated to $\perp$ in σ. Otherwise the lazy value, $\mathcal{E}_{lazy}$, of an expression is the same as its strict value, $\mathcal{E}$. The meaning of an expression in our lazy semantics is, thus, the function:

$$\mathcal{E}_{lazy} : E \to \Sigma^\perp \longmapsto V_\perp.$$

$$\text{given by } \mathcal{E}_{lazy}[\![e]\!]\sigma = \begin{cases} \perp & \text{if } \exists v \in \det(e) \text{ with } \sigma v = \perp. \\ \mathcal{E}[\![e]\!]\sigma & \text{otherwise.} \end{cases}$$

In Figure 5 we show the difference between $\det(e)$ and the set of variables referenced by some expression, $\mathrm{Ref}(e)$. Clearly $\det(e) \subseteq \mathrm{Ref}(e)$.

`Expression`	`Ref`(e)	$\det(e)$
`e:=x-x+y-z`	$\{x, y, z\}$	$\{y, z\}$
`e:=x+x+y-z`	$\{x, y, z\}$	$\{x, y, z\}$

Fig. 5. Ref and det of an expression.

The lazy meaning of a program is given by the function $\mathcal{M}_{lazy}$, which, as in the case of standard semantics, is a state to state function:

$$\mathcal{M}_{lazy} \ : \ P \longrightarrow \Sigma^\perp \longrightarrow \Sigma^\perp.$$

The lazy semantics of each construct of our simple *while* language defined in Figure 6. The lazy semantics of *while* loops and if statements is given in terms of $\prod$ operator, called meet operator. The meet operator is defined as follows:

Definition 6 (Meet - $\prod$) *Let $\sigma_1, \sigma_2, ...\sigma_n$, be n states in $\Sigma^\perp$. Then the meet of these states is defined as follows:*

$$\prod_{i=1}^{n} \sigma_i = \begin{cases} \lambda v \cdot \sigma_1(v) & \textit{if } \ \sigma_1(v) = \sigma_i(v) \ \ \forall\ 1 \le i \le n \\ \perp & \textit{otherwise.} \end{cases}$$

As it is shown in Figure 6, the lazy meaning of `skip` statement is the identity function on states (the same as in standard semantics). The lazy meaning of the *abort* statement is the same as lazy meaning of the `skip` statement. This is a fundamental difference between lazy and standard semantics. Because of

$\mathcal{M}_{lazy} : P \to \Sigma^{\perp} \to \Sigma^{\perp}$

Lazy semantics of the `skip` statement

$$\mathcal{M}_{lazy}[\![\texttt{skip}]\!] = \lambda\sigma \cdot \sigma$$

Lazy semantics of the *abort* statement

$$\mathcal{M}_{lazy}[\![abort]\!] = \lambda\sigma \cdot \sigma$$

Lazy semantics of assignment statements

$$\mathcal{M}_{lazy}[\![\texttt{x:=e}]\!] = \lambda\sigma \cdot \ \sigma[\texttt{x} \ \leftarrow \ \mathcal{E}_{lazy}[\![\texttt{e}]\!]\sigma]$$

(where the notation $\sigma[x \ \leftarrow e]$ represents the function which is the same as σ except that x gets mapped to e.)

Lazy semantics of the sequences of statements

$$\mathcal{M}_{lazy}[\![S_1\,;S_2]\!] = \mathcal{M}_{lazy}[\![S_2]\!] \circ \mathcal{M}_{lazy}[\![S_1]\!]$$

Lazy semantics of *if* statements

$\mathcal{M}_{lazy}[\![\mathit{if}\ (B)\ \mathit{then}\ S_1\ \mathit{else}\ S_2]\!]$

$$= \lambda\sigma \cdot \mathcal{E}_{lazy}[\![B]\!]\sigma \to \mathcal{M}_{lazy}[\![S_1]\!]\sigma, \mathcal{M}_{lazy}[\![S_2]\!]\sigma, \mathcal{M}_{lazy}[\![S_1]\!]\sigma \sqcap \mathcal{M}_{lazy}[\![S_2]\!]\sigma$$

(where the notation $e \to a, b, c$ is the expression which yields a if e is True, b if e is False and c if e is $\perp$).

Lazy semantics of *while* loops

$$\mathcal{M}_{lazy}[\![\mathit{while}\ (B)\ S]\!] = \lambda\sigma \cdot \bigsqcup_{i=0}^{\infty}(G_i\sigma)$$

where $G_i\sigma = \prod_{n=i}^{\infty} \mathcal{M}_{lazy}[\![\mathcal{W}_n(B,S)]\!]\sigma$.

G_i is just the meet of the lazy meaning of the n^{th} unfoldings, $\mathcal{W}_n$, for all $n \geq i$. The unfolding is given in Definition 3, where the meet is defined in Definition 6.

Fig. 6. Lazy denotational semantics.

this, successive unfoldings of loops may not be monotonic. As in standard semantics, the meaning of an assignment is obtained by updating the state with the new value of the variable assigned to it. In the case of lazy semantics, this value is the lazy value of the corresponding expression. Since in lazy semantics, there are states which map some variables to proper values and other variables to $\perp$, the assignment rule implies that a variable can 'recover' from being undefined as shown in Figure 7, where after the loop `x` has the value $\perp$ but it recovers to 5 after the assignment `x:=5`.

```
x:=1;
while (x>0)x:=x+1;
x:=5;
```

Fig. 7. Recovering the value of `x`.

For a sequence of statements, the lazy meaning is simply the composition of the meanings of the individual statements. For *if* statements, if the predicate of an *if* statement is evaluated to *True* or *False* the lazy meaning rules are the same as those of the standard semantics. The only difference is when the guard evaluates to bottom. In this case if a variable `x` is assigned different values in the *then* and *else* parts its value is $\perp$. On the other hand, the value of `x` is the same in the *then* and *else* parts then this should be its final value even if the guard is $\perp$,

```
if (z>0)
    then {x:=1; y:=2;}
```

Fig. 8. With initial state$\{x \mapsto 1, y \mapsto 1, z \mapsto \perp\}$, `x` has 1 as its final value, whereas y has $\perp$.

For example given an initial state $\sigma\{x \mapsto 1, y \mapsto 1, z \mapsto \perp\}$ in $\Sigma^{\perp}$, the value of the *if* predicate in the program in Figure 8 in σ is equal to $\perp$. However, the value of the variable `x` after executing the *then* branch is the same as when executing the *else* branch and is equal to 1. In this case the lazy value of the variable `x` after executing the program in Figure 8 in state σ is equal to 1. Unlike the variable `x`, the value of the variable `y` is different when executing the *then* branch from when executing *else* branch, and hence, the final value of the variable `y` is $\perp$.

For *while* loops, Given a state σ and a variable x the *final lazy value* of x after executing a while loop starting in state σ is the limit of all the values of x after executing each of the unfoldings. If the limit does not exist, then we define the final lazy value to be $\perp$. Here we mean the limit with respect to a discrete metric i.e. for the limit to exist, there must exist an $N \in \mathbb{N}$ such that all unfoldings greater than N give the same value for x in σ. If this is the case we say the value of x *stabilises* after N unfoldings. The lazy meaning of *while* loop is thus the limit of the meet of the lazy meaning of all its corresponding unfoldings:

Although the $\mathcal{M}_{lazy}[\![\mathcal{W}_n(B,S)]\!]$ is not monotonic, i.e. $\mathcal{M}_{lazy}[\![\mathcal{W}_n(B,S)]\!]$ is not necessarily less defined than $\mathcal{M}_{lazy}[\![\mathcal{W}_{n+1}(B,S)]\!]$, clearly G_i is less defined than G_{i+1} ($G_i \sqsubseteq G_{i+1}$), hence the least upper bound of the G_i exists.

In states where the *while* loop does not terminate, if the value of the variable stabilises after i unfoldings for some $i \geq 0$, then its meaning will be the stabilised value. Otherwise, its value is just $\perp$. For example, given the infinite loop in the program in Figure 9 the value of the variable `x` stabilises to 1 after the first unfolding whereas the value of the variable `y` never stabilises. In this case, the lazy values of `x` and `y` are 1 and $\perp$ respectively.

```
while (True) { x:=1; y:=y+1;}
```

Fig. 9. The lazy value of x is 1 and of y is $\perp$.

In states where the predicate of the *while* loop evaluates to $\perp$, if the value of a variable is the same and equal to `v` for all the unfoldings, after executing zero or more unfoldings, then its value is just `v`. And if otherwise the variable is evaluated to $\perp$.

For example, after executing the first infinite loop in the program in Figure 10, the value of the variable `y` is undefined and therefore, the condition of the

```
z:=1;
x:=1;
while (True)
{  if(y>0) then y:=-1;
             else  y:=1;
}
while (y>0) { x:=1; z:=2;}
```

Fig. 10. x is evaluated to 1 where the final value of z is $\perp$.

second *while* loop is undefined. However, the value of the variable `x` does not change and is equal to 1 after executing the body of the loop zero, a finite or infinite number of times. In this case the value of the variable `x` after executing the second loop is equal to 1. Unlike, the variable `x`, the variable `z` has a different value when the body of the *while* is not executed at all, which is 1, from its value when the body is executed, which is equal to 2. In this case the final value of the variable `z` is equal to $\perp$.

The example in Figure 10 illustrates the difference of our semantics with both the lazy semantics of Cartwright and Felleisen [35] and the transfinite semantics by Giacobazzi and Mastroeni [36]. The final value of the variable x, in both these semantics, when executing the program in Figure 10 is $\perp$.

An important property of our lazy semantics is that for terminating programs, it agrees with the standard semantics.

Theorem 7 *Let P be a program and σ be a state in Σ, then,*

$$\mathcal{M}[\![P]\!]\sigma \;\neq\; \perp \;\Longrightarrow\; \mathcal{M}_{lazy}[\![P]\!]\sigma \;=\; \mathcal{M}[\![P]\!]\sigma.$$

Proof.
This is proved by structural induction over the language being considered, as follows.
<u>`skip` Statement</u>
Trivial as $\mathcal{M}_{lazy}[\![\texttt{skip}]\!]\sigma = \sigma = \mathcal{M}[\![skip]\!]\sigma$ for all σ in Σ.

<u>*abort* Statement</u>
The result is vacuously true as, $\mathcal{M}[\![abort]\!]\sigma = \perp$ for all σ in Σ.

<u>Assignment Statements</u>
Trivial, as $\mathcal{E}_{lazy}[\![e]\!]\sigma = \mathcal{E}[\![e]\!]\sigma$, for all $\sigma \in \Sigma$.

<u>Conditional Statements</u>
Induction hypothesis: Assume that the result holds for two given programs S_1 and S_2.
Let B be a boolean expression, we need to show that the result holds for *if* (B) *then* S_1 *else* S_2.
Let σ be a state in Σ with $\mathcal{M}[\![if\,(B)\ then\ S_1\ else\ S_2]\!]\sigma \neq \perp$. As σ is a state in Σ,then $\mathcal{E}_{lazy}[\![B]\!]\sigma = \mathcal{E}[\![B]\!]\sigma \neq \perp$.
If $\mathcal{E}[\![B]\!]\sigma = True$, then $\mathcal{M}[\![if\,(B)\ then\ S_1\ else\ S_2]\!]\sigma$ is reduced to just $\mathcal{M}[\![S_1]\!]\sigma$ and $\mathcal{M}_{lazy}[\![if\,(B)\ then\ S_1\ else\ S_2]\!]\sigma$ is reduced to just $\mathcal{M}_{lazy}[\![S_1]\!]\sigma$. Thus, the result follows immediately from the induction hypothesis. Similarly, if $\mathcal{E}[\![B]\!]\sigma = False$ as $\mathcal{M}[\![if\,(B)\ then\ S_1\ else\ S_2]\!]\sigma$ is reduced to just $\mathcal{M}[\![S_2]\!]\sigma$ and $\mathcal{M}_{lazy}[\![if\,(B)\ then\ S_1\ else\ S_2]\!]\sigma$ is reduced to just $\mathcal{M}_{lazy}[\![S_2]\!]\sigma$.

Sequences

Induction hypothesis: Assume that the result holds for two given programs S_1 and S_2.

We must show that the result holds for $S_1; S_2$ We must show that:

$$\mathcal{M}[\![S_1; S_2]\!]\sigma \neq \perp \implies \mathcal{M}_{lazy}[\![S_1; S_2]\!]\sigma = \mathcal{M}[\![S_1; S_2]\!]\sigma.$$

holds for all σ in Σ.

Let σ be a state in Σ with $\mathcal{M}[\![S_1; S_2]\!]\sigma \neq \perp$. Hence, $\mathcal{M}[\![S_2]\!](\mathcal{M}[\![S_1]\!]\sigma) \neq \perp$ and $\mathcal{M}[\![S_1]\!]\sigma \neq \perp$. The result follows immediately by application of the semantics rule for sequences and the induction hypothesis:

$$\begin{aligned}
\mathcal{M}_{lazy}[\![S_1; S_2]\!]\sigma &= \mathcal{M}_{lazy}[\![S_2]\!](\mathcal{M}_{lazy}[\![S_1]\!]\sigma) && \text{(by definition)}\\
&= \mathcal{M}[\![S_2]\!](\mathcal{M}[\![S_1]\!]\sigma) && \text{(induction hypothesis)}\\
&= \mathcal{M}[\![S_1; S_2]\!]\sigma.
\end{aligned}$$

while Loops

Induction hypothesis: Let S be a program and assume that for all σ in Σ, if $\mathcal{M}[\![S]\!]\sigma \neq \perp$ then $\mathcal{M}_{lazy}[\![S]\!]\sigma = \mathcal{M}[\![S]\!]\sigma$.

Show that:

$$\mathcal{M}[\![\mathit{while}\ (B)\ S]\!]\sigma \neq \perp \implies \mathcal{M}_{lazy}[\![\mathit{while}\ (B)\ S]\!]\sigma = \mathcal{M}[\![\mathit{while}\ (B)\ S]\!]\sigma$$

holds for all σ in Σ.

Let σ be a state in Σ, such that *while* (B) S terminates on σ. Let's say it terminates after n iterations, then $\mathcal{M}[\![\mathit{while}\ (B)\ S]\!]\sigma = \mathcal{M}[\![\mathcal{W}_i(B,S)]\!]\sigma$ for all $i \geq n$. Thus using the definition of the lazy meaning of *while* loops, it suffices to show that for all $i \geq 0$,

$$\mathcal{M}[\![\mathcal{W}_i(B,S)]\!]\ \sigma \neq \perp \implies \mathcal{M}_{lazy}[\![\mathcal{W}_i(B,S)]\!]\ \sigma \ = \ \mathcal{M}[\![\mathcal{W}_i(B,S)]\!]\ \sigma.$$

We show this by induction on i. $\mathcal{M}[\![\mathcal{W}_0(B,S)]\!]\sigma = \perp$, then the base case is vacuously true. We now assume that the result holds for i^{th} unfolding: for all $\sigma \in \Sigma$, if $\mathcal{M}[\![\mathcal{W}_i(B,S)]\!]\sigma \neq \perp$ then $\mathcal{M}_{lazy}[\![\mathcal{W}_i(B,S)]\!]\sigma = \mathcal{M}[\![\mathcal{W}_i(B,S)]\!]\sigma$. Let σ be a state in Σ, with $\mathcal{M}[\![\mathcal{W}_{i+1}(B,S)]\!]\sigma \neq \perp$. We must show that $\mathcal{M}_{lazy}[\![\mathcal{W}_{i+1}(B,S)]\!]\sigma = \mathcal{M}[\![\mathcal{W}_{i+1}(B,S)]\!]\sigma$.

If $\mathcal{E}[\![B]\!]\sigma = \mathit{False}$, then $\mathcal{M}_{lazy}[\![\mathcal{W}_{i+1}(B,S)]\!]\sigma = \mathcal{M}[\![\mathcal{W}_{i+1}(B,S)]\!]\sigma = \sigma$.

If otherwise, $\mathcal{E}[\![B]\!]\sigma = \mathit{True}$, then $\mathcal{M}[\![\mathcal{W}_{i+1}(B,S)]\!]\sigma$ is reduced to just the standard meaning of $S; \mathcal{W}_i(B,S)$, $\mathcal{M}[\![\mathcal{W}_i(B,S)]\!](\mathcal{M}[\![S]\!]\sigma)$, and in the same way $\mathcal{M}_{lazy}[\![\mathcal{W}_{i+1}(B,S)]\!]\sigma$ is reduced to just $\mathcal{M}_{lazy}[\![\mathcal{W}_i(B,S)]\!](\mathcal{M}_{lazy}[\![S]\!]\sigma)$. And the result follows immediately by application of the induction hypothesis on S and i. Thus completing the proof.

Program transformation is a form of program analysis or manipulation. Program transformation alters the syntax of a program while preserving its semantics: We can substitute some parts of a program by their corresponding equivalent and still preserve the semantics of the original program. Therefore, substitutivity (see definition 1) is an important property a semantics should have if it is to be useful to prove correctness of program transformation algorithms slicing [39,38,40]. Unlike the semantics of Cartwright and Felleisen [35] and the transfinite semantics of Giacobazzi and Mastroeni [36], the following theorem shows that our lazy semantics is substitutive.

Theorem 8 (Our lazy semantics is substitutive)
Let P be a program and P' be a program obtained by replacing a sub-program, Q, of P by a Q'. Then

$$\mathcal{M}_{lazy}[\![Q]\!] = \mathcal{M}_{lazy}[\![Q']\!] \implies \mathcal{M}_{lazy}[\![P]\!] = \mathcal{M}_{lazy}[\![P']\!].$$

Proof.
This is proved by structural induction over the language being considered. The result for base case is trivial as *abort* , `skip` and assignment statements are atomic statements.

<u>Conditional Statements</u>
Induction hypothesis: Assume that the result holds for two given programs S_1 and S_2.
Let S_1' be a program obtained by replacing a sub-program, Q_1, of S_1 by an equivalent program Q_1' and let S_2' be a program obtained by replacing a sub-program, Q_2, of S_2 by an equivalent program Q_2'. We need to Show that:

$$\mathcal{M}_{lazy}[\![\mathit{if}\,(B)\ \mathit{then}\ S_1\ \mathit{else}\ S_2]\!] = \mathcal{M}_{lazy}[\![\mathit{if}\,(B)\ \mathit{then}\ S_1'\ \mathit{else}\ S_2']\!].$$

Let σ be a state in $\Sigma^{\perp}$, then

$\mathcal{M}_{lazy}[\![\mathit{if}\,(B)\ \mathit{then}\ S_1'\ \mathit{else}\ S_2']\!]\sigma$

$= \lambda\sigma \cdot \mathcal{E}_{lazy}[\![B]\!]\sigma \rightarrow \mathcal{M}_{lazy}[\![S_1']\!]\sigma, \mathcal{M}_{lazy}[\![S_2']\!]\sigma, \mathcal{M}_{lazy}[\![S_1']\!]\sigma \sqcap \mathcal{M}_{lazy}[\![S_2']\!]\sigma$

(by definition)

$= \lambda\sigma \cdot \mathcal{E}_{lazy}[\![B]\!]\sigma \rightarrow \mathcal{M}_{lazy}[\![S_1]\!]\sigma, \mathcal{M}_{lazy}[\![S_2]\!]\sigma, \mathcal{M}_{lazy}[\![S_1]\!]\sigma \sqcap \mathcal{M}_{lazy}[\![S_2]\!]\sigma$

(Induction hypothesis)

$= \mathcal{M}_{lazy}[\![\mathit{if}\,(B)\ \mathit{then}\ S_1\ \mathit{else}\ S_2]\!]\sigma.$

<u>Sequences</u>

Induction hypothesis: Assume that the result holds for two given programs S_1 and S_2.
Let S_1' be a program obtained by replacing a sub-program, Q_1, of S_1 by an equivalent program Q_1' and let S_2' be a program obtained by replacing a sub-program, Q_2, of S_2 by an equivalent program Q_2'. We need to Show that: $\mathcal{M}_{lazy}[\![S_1'\,;S_2']\!] = \mathcal{M}_{lazy}[\![S_1\,;S_2]\!]$.
Let σ be a state in $\Sigma^{\perp}$, then

$$\begin{aligned}\mathcal{M}_{lazy}[\![S_1';S_2']\!]\sigma &= \mathcal{M}_{lazy}[\![S_2']\!](\mathcal{M}_{lazy}[\![S_1']\!]\sigma) \text{ (by definition)}\\ &= \mathcal{M}_{lazy}[\![S_2]\!](\mathcal{M}_{lazy}[\![S_1]\!]\sigma) \text{ (by induction hypothesis)}\\ &= \mathcal{M}_{lazy}[\![S_1;S_2]\!]\sigma.\end{aligned}$$

while Loops
Induction hypothesis: Assume that the result holds for a program S.
Let S' be a program obtained by replacing a sub-program, Q, of S by an equivalent program Q'. We need show that:

$$\mathcal{M}_{lazy}[\![\textit{while}\ (B)\ S']\!] = \mathcal{M}_{lazy}[\![\textit{while}\ (B)\ S]\!].$$

by applying the induction hypothesis and the result for *if* statements, we have

$$\mathcal{M}_{lazy}[\![\mathcal{W}_i(B,S')]\!] = \mathcal{M}_{lazy}[\![\mathcal{W}_i(B,S)]\!]\ \ \forall\ \ i \geq 0.$$

Hence, by applying the definition of the lazy semantics of *while* loops, the result for *while* loops follows immediately. Thus completes the proof.

In the following section we will define the semantics of slicing using lazy semantics.

4 Defining Slices using Lazy Semantics

We define P and Q to be *V-lazy* equivalent if and only if they have the same lazy semantics with respect to the set of variables V.

Definition 9 (*V-lazy* equivalence)
Let V be a set of variables and P and Q be two programs. We say P and Q are V-lazy *equivalent if and only if for all σ in $\Sigma^{\perp}$ and for all $\boldsymbol{x}$ in V, $\mathcal{M}_{lazy}[\![P]\!]\sigma\boldsymbol{x} = \mathcal{M}_{lazy}[\![Q]\!]\sigma\boldsymbol{x}$. This is clearly an equivalence relation.*

We write $P \overset{V}{\sim} Q$ to denote that P and Q are V–lazy equivalent.

Figure 11 shows two programs, P_1 and P_2, with the same lazy semantics with respect to x. i.e. $P_1 \overset{\{x\}}{\sim} P_2$.

```
x:=1;
y:=0;
while (y ≥ 0) {x:=0;}
```

Program P_1.

```
x:=0;
```

Program P_2.

Fig. 11. P_1 and P_2 have the same lazy semantics w.r.t. the variable x.

```
x:=1;
y:=0;
while (y=0) {y:=y+1;}
x:=0;
```

A program P.

```
x:=0;
```

End-slice of P w.r.t. x.

Fig. 12. P and its end-slice have the same lazy semantics w.r.t.x.

Definition 10 (Weiser's semantic definition of a slice)
Let V be a set of variables and P and Q be two programs. We say that Q is a slice of P with respect to V if and only if, for all σ in Σ,

$$\mathcal{M}[\![P]\!]\sigma \neq \perp \implies \mathcal{M}[\![P]\!]\sigma x \;=\; \mathcal{M}[\![Q]\!]\sigma x \quad \forall \; x \;\in\; V.$$

The program P in Figure 12 does not terminate. Using Weiser's semantic definition, the slice of the program P can be anything. This is a result of the fact that Weiser's definition does not take into account non-termination.

Unlike Weiser's standard semantic definition of a slice, Definition 10, we require a slice to preserve the semantics of the original in all states, not just for ones where the program terminates. We do not wish to allow arbitrary semantics for slices of non-terminating programs since Weisers's algorithm does not behave in an arbitrary way in these cases. We now give a new semantic definition of a slice, called a *lazy V-slice*, which we prove to be consistent with Weiser's one.

Definition 11 (*Lazy V-Slice*)
Let P and P_1 be two programs, and V be a set of variables. We say P_1 is a lazy V-slice of a program P if and only if

$$P \overset{V}{\sim} P_1, \; and \;\; (\mathcal{M}[\![P]\!]\sigma \neq \perp \implies \mathcal{M}[\![P_1]\!]\sigma \neq \perp \quad \forall \sigma \in \Sigma).$$

Using our new lazy semantic definition of a slice, it is clear that x:=0; is a semantically valid slice of the program P in Figure 12 on page 18.

Our definition of a *lazy V-slice* is one that preserves termination and lazy semantics of the original program projected onto all variables in V. Weiser's semantics definition of a slice is one that preserves termination and the projected standard semantics onto V only for terminating programs. Our lazy semantics agrees with the standard semantics in all states where the program terminates, (see Theorem 7, page 14). From this it follows immediately that any slice which satisfies our new definition also satisfies Weiser's semantic definition of slicing.

5 Lazy Neededness

Program slicing can introduce termination [4]. Because of this it is hard to give a denotational definition of variable dependence which is consistent with program slicing in terms of the standard semantics. In this section we define neededness in terms of our lazy semantics, called *lazy neededness*, and show that it satisfies the neededness criterion; i.e. it is *standard semantically discriminating* (*PSSD*) and it is sub-sequential (for *PSSD* and sub-sequentiality see Section 1.1, Page 3).

Definition 12 (Lazy Needed: $\mathbb{N}_{lazy}$)

$$\mathbb{N}_{lazy} : \mathbf{S} \times \mathcal{P}(Variables) \longrightarrow \mathcal{P}(Variables)$$

where $\mathbf{S}$ *is the set of programs and* $Variables$ *is the set of program variables.*

A variable y is in $\mathbb{N}_{lazy}(P, V)$ if and only if there is a variable $\boldsymbol{x}$ in V, and two states, σ_1 and σ_2 in $\Sigma^{\perp}$, differing only on the value of the variable $\boldsymbol{y}$, such that:

$$\mathcal{M}_{lazy}[\![p]\!]\sigma_1\boldsymbol{x} \neq \mathcal{M}_{lazy}[\![p]\!]\sigma_2\boldsymbol{x}.$$

From the definition of lazy neededness and Theorem 7 which states that the standard semantics and lazy semantics agree for terminating programs, it follows that lazy neededness is *standard semantically discriminating (PSSD).*

We now show that lazy neededness, $\mathbb{N}_{lazy}$, defined in Definition 12 is both sub-sequential Some intermediate results, exploring some properties of lazy neededness, are now given.

We begin by showing that lazy neededness of a program with respect to a set of variables V is just the union of its lazy neededness with respect to each one of the variables in V.

Lemma 13 *Given a set of variables V and program P then,*

$$\mathbb{N}_{lazy}(P, V) \;=\; \bigcup_{\boldsymbol{x} \in V} \mathbb{N}_{lazy}(P, \{\boldsymbol{x}\}).$$

Proof.
This follows immediately from Definition 12.

By definition, the lazy needed set of variables of a program P with respect to a set of variables V is the set of variables for which the initial value might affect the lazy final value of some of the variables in V after executing P. Therefore, if two states, σ_1 and σ_2 in $\Sigma^{\perp}$ agree on all elements in $\mathbb{N}_{lazy}(P, V)$ then the meaning of P in σ_1 and in σ_2 agree on all elements in V. The following lemma shows that our lazy semantics satisfies this property.

Lemma 14 *Given a set of variables V, a program P and two states, σ_1 and σ_2, differing only on a set V_0 of elements not in $\mathbb{N}_{lazy}(P, V)$, then*

$$\mathcal{M}_{lazy}[\![P]\!]\sigma_1 x = \mathcal{M}_{lazy}[\![P]\!]\sigma_2 x, \quad \forall\, x \in V.$$

Proof.
We show this by contradiction. By Lemma 13, it suffices to show the result for $V = \{x\}$. Suppose there exists two states, σ_1 and σ_2, differing only on elements in V_0, with $\mathcal{M}_{lazy}[\![P]\!]\sigma_1 x \neq \mathcal{M}_{lazy}[\![P]\!]\sigma_2 x$. And choose σ_1 and σ_2 to be the states differing on a minimal set, $W \subseteq V_0$, with $\mathcal{M}_{lazy}[\![P]\!]\sigma_1 x \neq \mathcal{M}_{lazy}[\![P]\!]\sigma_2 x$. Clearly $W \neq \emptyset$, so choose $y \in W$ and let $\sigma_1' = \sigma_1[y \leftarrow \sigma_2(y)]$. By the minimality follows $\mathcal{M}_{lazy}[\![P]\!]\sigma_1' x = \mathcal{M}_{lazy}[\![P]\!]\sigma_2 x$. Thus, $\mathcal{M}_{lazy}[\![P]\!]\sigma_1' x \neq \mathcal{M}_{lazy}[\![P]\!]\sigma_1 x$. This contradicts the minimality of W unless $W = \{y\}$. In this case, by definition, $y \in \mathbb{N}_{lazy}(P, \{x\})$ which contradicts the hypothesis.

The example in Figure 13 illustrates this property. $\mathbb{N}_{lazy}(P, \{x\}) = \{y\}$. The value of the variable x after executing P is always equal to 0 in all states where the value of y is 0. In all other states the value of x is equal to 1.

Given an expression e, if a variable x is not in $\det(e)$ then the initial value of x does not affect the value of the expression e. Therefore, the value of e in all states which agree in all elements in $\det(e)$ is the same. The following lemma illustrates this.

Lemma 15 *Let e be an expression, and σ_1, σ_2 be two states in $\Sigma^{\perp}$, differing*

```
if(y=0)
   then x:=0;
   else  x:=1;
```

Fig. 13. A simple program P.

only on a set V of variables not in $\det(e)$*, then* $\mathcal{E}_{lazy}[\![e]\!]\sigma_1 = \mathcal{E}_{lazy}[\![e]\!]\sigma_2$*.*

Proof.
This is entirely similar to the proof of Lemma 14.

5.1 Lazy Neededness is Sub-sequential

In the introduction we have explained our intuitive reasons why neededness has to satisfy the *sub-sequentiality* property. The objective of this section is to show that our definition of neededness, lazy neededness, satisfies this property.

Theorem 16 *Let P_1 and P_2 be two programs and V a set of variables, then*

$$\mathbb{N}_{lazy}(P_1; P_2, V) \ \subseteq \ \mathbb{N}_{lazy}(P_1, \mathbb{N}_{lazy}(P_2, V)).$$

Proof.
By Lemma 13, it suffices to show the theorem holds when $V = \{x\}$. Let σ_1 and σ_2 be two states differing only on the value of y, which is not in $\mathbb{N}_{lazy}(P_1, \mathbb{N}_{lazy}(P_2, \{x\}))$. Hence, by Lemma 14, it follows that $\mathcal{M}_{lazy}[\![P_1]\!]\sigma_1$ and $\mathcal{M}_{lazy}[\![P_1]\!]\sigma_1$ agree in all elements in $\mathbb{N}_{lazy}(P_2, \{x\})$. Thus, by Lemma 14, if follows that

$$\mathcal{M}_{lazy}[\![P_2]\!](\mathcal{M}_{lazy}[\![P_1]\!]\sigma_1)x = \mathcal{M}_{lazy}[\![P_2]\!](\mathcal{M}_{lazy}[\![P_1]\!]\sigma_2)x$$

$$\mathcal{M}_{lazy}[\![P_1; P_2]\!]\sigma_1 x = \mathcal{M}_{lazy}[\![P_1; P_2]\!]\sigma_2 x.$$

Hence, the variable y is not in $\mathbb{N}_{lazy}(P_1; P_2, \{x\})$ Thus, the result follows.

Theorem 16 shows that lazy neededness satisfies the sub-sequentiality property.

6 Related Work

A slice of a program can halt even if the original program does not terminate. Hence, the standard semantics is not preserved by program slicing algorithms. The semantics of different intermediate graph representation, such as the control flow graph [1] or the program dependence graph [33,34] has been used to investigate the semantics properties of program slicing [32,46,35]. Horwitz et al. [32], have shown that behaviour of a program is captured by its corresponding program dependence graph. They show that if the program dependence graphs of two programs are isomorphic, then the programs have the same semantics.

Cartwright and Felleisen [35] observed and discussed that if a semantic is to be useful to investigate semantics properties of program slicing, it has to be preserved by slicing algorithms with respect to slicing criterion. In [35], they defined a non-strict semantics for program dependence graphs of a simple *while* language and claimed that their semantics is preserved by slicing algorithms. Venkatesh [47] used Cartwright and Felleisen's lazy semantics to give a semantic justification of program slicing.

Giacobazzi and Mastroeni [36] argued that it is unnatural to try to prove correctness of slicing properties using the standard semantics. They also argued that if a semantics is to be useful for modelling program slicing manipulation techniques it should be able to capture semantic information 'beyond infinite loops' and be *sequential*. As it has been shown in Section 1, neither the semantics of Cartwright and Felleisen given in [35] nor the one of Giacobazzi and Mastroeni given in [36] is *s*ubstitutive. In this paper a new denotational semantics which is substitutive and preserved by slicing is given.

7 Conclusion and Future Work

The standard semantics is not preserved by slicing algorithms. As a result of this, it is very hard to prove correctness of program transformation such as program slicing using the standard denotational semantics without the use of some intermediate graph representations. Different intermediate graph representations such as program dependence graphs [32,35] have been used to investigate the semantic properties of program slicing algorithms. Proving correctness of slicing algorithms using a semantics which is not preserved by slicing, proved to be very hard.

In this paper we introduced a new non-strict semantics for a simple *while* language. Our new semantics allows us to give a denotational definition of variable dependence, neededness, which is consistent with program slicing. Finally, our semantics is proved to be preserved by slicing algorithms, which makes it very useful to prove correctness of slicing algorithms. Furthermore, in Theorem 8, page 16, we have shown that our new semantics is $s$ubstitutive. This property is very useful to prove correctness of the kind of transformations discussed in this paper.

A future direction of our research is to attempt to prove correctness of existing intraprocedural slicing algorithms such as Hausler's slicing algorithm [48], using our new semantics. We also intend to extend our semantics to handle real program features, such as procedures and recursion.

References

[1] M. Weiser, Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method, PhD thesis, University of Michigan, Ann Arbor, MI (1979).

[2] M. Weiser, Program slicing, in: 5^{th} International Conference on Software Engineering, San Diego, CA, 1981, pp. 439–449.

[3] M. Weiser, Programmers use slicing when debugging, Communications of the ACM 25 (7) (1982) 446–452.

[4] M. Weiser, Program slicing, IEEE Transactions on Software Engineering 10 (4) (1984) 352–357.

[5] G. Canfora, A. Cimitile, M. Munro, RE^2: Reverse engineering and reuse re-engineering, Journal of Software Maintenance : Research and Practice 6 (2) (1994) 53–72.

[6] D. Simpson, S. H. Valentine, R. Mitchell, L. Liu, R. Ellis, Recoup – Maintaining Fortran, ACM Fortran forum 12 (3) (1993) 26–32.

[7] A. De Lucia, A. R. Fasolino, M. Munro, Understanding function behaviours through program slicing, in: 4^{th} IEEE Workshop on Program Comprehension, IEEE Computer Society Press, Los Alamitos, California, USA, Berlin, Germany, 1996, pp. 9–18.

[8] M. Harman, R. M. Hierons, S. Danicic, J. Howroyd, C. Fox, Pre/post conditioned slicing, in: IEEE International Conference on Software Maintenance (ICSM'01), IEEE Computer Society Press, Los Alamitos, California, USA, Florence, Italy, 2001, pp. 138–147.

[9] G. Canfora, A. Cimitile, A. De Lucia, G. A. D. Lucca, Software salvaging based on conditions, in: International Conference on Software Maintenance (ICSM'96), IEEE Computer Society Press, Los Alamitos, California, USA, Victoria, Canada, 1994, pp. 424–433.

[10] A. Cimitile, A. De Lucia, M. Munro, A specification driven slicing process for identifying reusable functions, Software maintenance: Research and Practice 8 (1996) 145–178.

[11] K. B. Gallagher, Evaluating the surgeon's assistant: Results of a pilot study, in: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, California, USA, 1992, pp. 236–244.

[12] K. B. Gallagher, J. R. Lyle, Using program slicing in software maintenance, IEEE Transactions on Software Engineering 17 (8) (1991) 751–761.

[13] H. Agrawal, R. A. DeMillo, E. H. Spafford, Debugging with dynamic slicing and backtracking, Software Practice and Experience 23 (6) (1993) 589–616.

[14] M. Kamkar, Interprocedural dynamic slicing with applications to debugging and testing, PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, available as Linköping Studies in Science and Technology, Dissertations, Number 297 (1993).

[15] J. R. Lyle, M. Weiser, Automatic program bug location by program slicing, in: 2^{nd} International Conference on Computers and Applications, IEEE Computer Society Press, Los Alamitos, California, USA, Peking, 1987, pp. 877–882.

[16] D. W. Binkley, The application of program slicing to regression testing, in: M. Harman, K. Gallagher (Eds.), Information and Software Technology Special Issue on Program Slicing, Vol. 40, Elsevier, 1998, pp. 583–594.

[17] R. Gupta, M. J. Harrold, M. L. Soffa, An approach to regression testing using slicing, in: Proceedings of the IEEE Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos, California, USA, Orlando, Florida, USA, 1992, pp. 299–308.

[18] M. Harman, S. Danicic, Using program slicing to simplify testing, Software Testing, Verification and Reliability 5 (3) (1995) 143–162.

[19] R. M. Hierons, M. Harman, S. Danicic, Using program slicing to assist in the detection of equivalent mutants, Software Testing, Verification and Reliability 9 (4) (1999) 233–262.

[20] R. M. Hierons, M. Harman, C. Fox, L. Ouarbya, M. Daoudi, Conditioned slicing supports partition testing, Software Testing, Verification and Reliability 12 (2002) 23–28.

[21] J. Beck, D. Eichmann, Program and interface slicing for reverse engineering, in: IEEE/ACM 15^{th} Conference on Software Engineering (ICSE'93), IEEE Computer Society Press, Los Alamitos, California, USA, 1993, pp. 509–518.

[22] A. Cimitile, A. De Lucia, M. Munro, Identifying reusable functions using specification driven program slicing: a case study, in: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'95), IEEE Computer Society Press, Los Alamitos, California, USA, Nice, France, 1995, pp. 124–133.

[23] D. W. Binkley, S. Horwitz, T. Reps, Program integration for languages with procedure calls, ACM Transactions on Software Engineering and Methodology 4 (1) (1995) 3–35.

[24] S. Horwitz, J. Prins, T. Reps, Integrating non–interfering versions of programs, ACM Transactions on Programming Languages and Systems 11 (3) (1989) 345–387.

[25] J. M. Bieman, L. M. Ott, Measuring functional cohesion, IEEE Transactions on Software Engineering 20 (8) (1994) 644–657.

[26] A. Lakhotia, Rule–based approach to computing module cohesion, in: Proceedings of the 15^{th} Conference on Software Engineering (ICSE-15), 1993, pp. 34–44.

[27] L. M. Ott, J. J. Thuss, Slice based metrics for estimating cohesion, in: Proceedings of the IEEE-CS International Metrics Symposium, IEEE Computer Society Press, Los Alamitos, California, USA, Baltimore, Maryland, USA, 1993, pp. 71–81.

[28] D. W. Binkley, K. B. Gallagher, Program slicing, in: M. Zelkowitz (Ed.), Advances in Computing, Volume 43, Academic Press, 1996, pp. 1–50.

[29] A. De Lucia, Program slicing: Methods and applications, in: 1^{st} IEEE International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society Press, Los Alamitos, California, USA, Florence, Italy, 2001, pp. 142–149.

[30] M. Harman, R. M. Hierons, An overview of program slicing, Software Focus 2 (3) (2001) 85–92.

[31] F. Tip, A survey of program slicing techniques, Journal of Programming Languages 3 (3) (1995) 121–189.

[32] S. Horwitz, J. Prins, T. Reps, On the adequacy of program dependence graphs for representing programs, in: ACM (Ed.), POPL '88. Proceedings of the conference on Principles of programming languages, January 13–15, 1988, San Diego, CA, ACM Press, New York, NY, USA, 1988, pp. 146–157.

[33] D. Kuck, R. Kuhn, D. Padua, B. Leasure, M. Wolfe, Dependence graphs and compiler optimizations, In Conference Record of the 8^{th} ACM Symposium on Principles of Programing Languages (1981) 207–218.

[34] J. Ferrante, K. J. Ottenstein, J. D. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems 9 (3) (1987) 319–349.

[35] R. Cartwright, M. Felleisen, The semantics of program dependence, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, 1989, pp. 13–27.

[36] R. Giacobazzi, I. Mastroeni, Non-standard semantics for program slicing, Higher-Order and Symbolic Computation(HOSC) 16 (4) (2003) 297–339.

[37] J. Kennaway, J. Klop, M. Sleep, F. Vries, Transfinite reduction in orthogonal term rewriting systems., Information and computation 119 (1) (1995) 18–38.

[38] M. Harman, S. Danicic, Amorphous program slicing, in: 5^{th} IEEE International Workshop on Program Comprenhesion (IWPC'97), IEEE Computer Society Press, Los Alamitos, California, USA, Dearborn, Michigan, USA, 1997, pp. 70–79.

[39] D. W. Binkley, Computing amorphous program slices using dependence graphs and a data-flow model, in: ACM Symposium on Applied Computing, ACM Press, New York, NY, USA, The Menger, San Antonio, Texas, U.S.A., 1999, pp. 519–525.

[40] M. Harman, L. Hu, X. Zhang, M. Munro, GUSTT: An amorphous slicing system which combines slicing and transformation, in: 1^{st} Workshop on Analysis, Slicing, and Transformation (AST 2001), IEEE Computer Society Press, Los Alamitos, California, USA, Stuttgart, 2001, pp. 271–280.

[41] J. E. Stoy, Denotational semantics: The Scott–Strachey approach to programming language theory, MIT Press, 1985, third edition.

[42] D. A. Schmidt, Denotational semantics: A Methodology for Language Development, Allyn and Bacon, 1986.

[43] Z. Manna, Mathematical Theory of Computation, McGraw–Hill, 1974.

[44] S. Danicic, Dataflow minimal slicing, PhD thesis, University of North London, UK, School of Informatics (Apr. 1999).

[45] K. R. Apt, E.-R. Olderog, Verification of sequential and concurrent programs (2nd ed.), Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[46] T. W. Reps, W. Yang, The semantics of program slicing and program integration, in: TAPSOFT '89: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2, Springer-Verlag, London, UK, 1989, pp. 360–374.

[47] G. A. Venkatesh, The semantic approach to program slicing, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, Toronto, Canada, 1991, pp. 26–28, proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

[48] P. A. Hausler, Denotational program slicing, in: 22^{nd}, Annual Hawaii International Conference on System Sciences, Volume II, 1989, pp. 486–495.