A Lazy Semantics for Program Slicing

Sebastian Danicic[1], Mark Harman[2], John Howroyd[1] and Lahcen Ouarbya[1]

[1]Department of Computing, Goldsmiths College, University of London, New Cross, London SE14 6NW
[2]Department of Information Systems and Computing, Brunel University, Uxbridge, Middlesex, UB8 3PH.

Abstract. This paper demonstrates that if a slicing algorithm is expressed denotationally, without intermediate structures, then the power of denotational semantics can be used to prove correctness.
The semantics preserved by slicing algorithms, however, is non-standard. We introduce a new lazy semantics which we prove is preserved by slicing algorithms.
It is demonstrated how other concepts in program dependence, difficult or impossible to express using standard semantics, for example variable dependence, can be expressed naturally using our new lazy semantics.

Traditionally, slicing algorithms and program analysis algorithms in general are defined in terms of a variety of intermediate graph representations: control flow graphs [1], data dependence graphs [2] and program dependence graphs [3]. There have been many efforts to give a formal semantics of these intermediate program representations [9, 10]. Horwitz et al. [10] have shown that two programs with the same program dependence graph have the same semantics. Cartwright and Felleisen [9] defined a non-strict semantics for program dependence graphs of a simple procedural language. Other efforts give a constructive semantics of the program dependence graph by transforming the denotational semantics of imperative languages. The fact that these semantics are defined for some intermediate graph representations instead of the programming language itself makes it difficult to prove correctness of program transformation techniques such as slicing.

This paper demonstrates that if the slicing algorithm and the semantics of the program language and the 'correctness criteria' of slicing should are all expressed denotationally then the full power of denotational semantics can be used in correctness proofs.

In 1989, Hausler [11] introduced a denotational slicing algorithm. His algorithm has not been formally proved correct. This was our goal. To do this, a satisfactory definition of correctness had to be given. According to Weiser [12], 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 the program is thus, less defined than the semantics of some of its slices. Because of this it is unnatural to try to prove correctness of slicing properties using *Standard Semantics* [13].

In an attempt to solve this problem, Cartwright and Felleisen [9] introduce a non-strict lazy semantics of program dependence graphs. Unfortunately, their lazy semantics is not substitutive i.e. we cannot always replace a subprogram Q of a program P with another semantically equivalent subprogram, Q', and guarantee that the resulting program P' is semantically equivalent to P.

Giacobazzi and Mastreoni [13] argued that if a semantics is to be useful for modelling kinds of program manipulation such as slicing it should be compositional, just as standard semantics, and be able to capture semantic information 'beyond infinite loops'. They use semantics represented by transfinite states traces of programs [14] and showed 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.

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. 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.

Intuitively, neededness should be *semantically descriminating*. That is, if x and y are variables such that there exists two initial states σ_1 and σ_2, differing only on x, such that the meaning of P gives rise to final states with different values of y. Then y should be *needed* by x with respect to P.

A definition of neededness, clearly will also have to be consistent with Weiser's algorithm. That is, all variables that are 'semantically needed' must also be 'Weiser Needed'. Otherwise, Weiser's algorithm would be deemed in some cases not to produce valid slices.

Due to issues regarding non–termination, it turns out to be hard, if not impossible, to define neededness in terms of standards semantics. This leads to the *Lazy Semantics* which is at the heart of our work. Our semantics is, unsurprisingly, closely related to the semantics of Cartwright and Felleisen [9], as that was a semantics for program dependence graphs. Unlike theirs, however, our lazy semantics is substitutive. In terms of our lazy semantics the definition of neededness turns out to be straightforward.

In *Lazy Semantics*, variables are allowed to have a $\perp$ value and hence partially defined states, where some variables are mapped to $\perp$ and others to well defined values. The set of such states is denoted as $\Sigma^{\perp}$.

$$\Sigma^{\perp} : Variables \to V_{\perp}.$$

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, $\sigma_1 \sqsubseteq \sigma_2$ means $\sigma_2(x) = \perp \implies \sigma_1(x) = \perp$.

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 for argument and returns the set of variables referenced by e which contribute to the evaluation of e. The function $\mathrm{det}(e)$ is defined later on. If $\mathrm{det}(e)$ contains a variable which has $\perp$ as a value in σ, then the whole expression is evaluated to $\perp$ in σ. The meaning of an expression in our lazy semantics is given by the function $\mathcal{E}_{lazy}$.

$$\mathcal{E}_{lazy} \;:\; E \rightarrow \Sigma^{\perp} \longmapsto V_{\perp}$$

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

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:

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

For `skip`, assignment, and statement sequences, the rules are as in standard semantics: $\mathcal{M}_{lazy}[\![\texttt{skip}]\!]\sigma = \sigma$, $\mathcal{M}_{lazy}[\![x{=}e]\!]\sigma = \sigma[x \leftarrow \mathcal{E}_{lazy}[\![e]\!]\sigma]$ and $\mathcal{M}_{lazy}[\![S_1; S_2]\!] = \mathcal{M}_{lazy}[\![S_2]\!] \circ \mathcal{M}_{lazy}[\![S_1]\!]$. The first departure from standard semantics appears in the way we handle conditional statements:

$$\mathcal{M}_{lazy}[\![\mathit{if}\ (B)\ \mathit{then}\ S_1\ \ \mathit{else}\ S_2]\!]\sigma = \begin{cases} \mathcal{M}_{lazy}[\![S_1]\!]\sigma \text{ if } \mathcal{E}_{lazy}[\![b]\!]\sigma = \mathit{True}. \\ \mathcal{M}_{lazy}[\![S_2]\!]\sigma \text{ if } \mathcal{E}_{lazy}[\![b]\!]\sigma = \mathit{False}. \\ \mathcal{M}_{lazy}[\![S_1]\!]\sigma \sqcap \mathcal{M}_{lazy}[\![S_2]\!]\sigma \text{ if } \mathcal{E}_{lazy}[\![b]\!]\sigma = \perp. \end{cases}$$

where $\sigma_1 \sqcap \sigma_2$ the *meet* of σ_1 and σ_2 is defined as $\lambda i \cdot \sigma_1(i) = \sigma_2(i) \rightarrow \sigma_1(i), \perp$. The only difference from standard semantics is when the guard evaluates to bottom. If the value of x depends on the guard then x is mapped to $\perp$. On the other hand, if 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. 1. in $\{x \mapsto 1, y \mapsto 1, z \mapsto \perp\}$, x has final value 1, whereas y has final value $\perp$.

For example given an initial state $\sigma = \begin{cases} x = 1 \\ y = 1 \\ z = \perp \end{cases}$ in $\Sigma^{\perp}$, the value *if* predicate in the program in Figure 1 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 1 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 $\perp$. The final values of both variables ,x and y, when using the lazy semantics by Cartwright and Felleisen [9] is $\perp$. This is a result of the fact that their semantics lose all information about the variables defined in the *then* or *else* parts of an *if* statement in states where its predicate is undefined.

The lazy semantics of a *while* loop, *while* (B) S is defined in terms of the loop's unfoldings $\mathcal{W}_i(B,S)$ where$\mathcal{W}_0(B,S) = \texttt{skip}$ and $\mathcal{W}_{n+1}(B,S) =$ *if* (B) *then* $\{S; \mathcal{W}_n(B,S)\}$ *else* `skip`. The lazy meaning of a *while* loop is then defined as follows:

$$\mathcal{M}_{lazy}[\![\textit{while}\ (B)\ S]\!] = \lambda\sigma \cdot \bigsqcup_{i=0}^{\infty} G_i\sigma \text{ where } G_i\sigma = \bigsqcap_{n=i}^{\infty} \mathcal{M}_{lazy}[\![\mathcal{W}_n(B,S)]\!]\sigma$$

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 $\mathcal{M}_{lazy}[\![\mathcal{W}_n(B,S)]\!]$ are 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 $G_i \sqsubseteq G_{i+1}$, hence the least upper bound of the G_i exists.

In states where the *while* loop does not terminate or the guard evaluates to $\perp$, if the value of a 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 2 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.

Substitutivity of the semantics greatly simplifies correctness proofs for the sorts of transformations described in this paper and others for example amorphous slicing [15] which is another transformation system where a program's transformations are expressed in terms of the transformations of its sub-components. We showed that our lazy semantics is substitutive. Furthermore, a semantics definition of *variable dependence*(*neededness*) is given in terms of it, that is semantically discriminating, Weiser-consistent and sub-sequential.

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

Fig. 2. The lazy value of x is 1 and of y is $\perp$.

As a demonstration of the applicability of our lazy semantics, Hausler's Denotational Slicing Algorithm is proved correct with respect to the lazy semantic definition of a slice. Since our lazy definition of a slice is stronger than the standard one, this proves that Hausler's Algorithm [11] is correct with respect to the standard definition too.

Future work will explore the extension of our lazy semantics and Hausler's slicing algorithm to handle programs with procedures.

References

1. M. S. Hecht, Flow Analysis of Computer Programs, Elsevier, 1977.
2. D. J. Kuck, The structure of computers and computations 12 (1) (1990) 26–60.
3. 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.
4. M. Harman, L. Hu, M. Munro, X. Zhang, D. W. Binkley, S. Danicic, M. Daoudi, L. Ouarbya, Syntax-directed amorphous slicing, Journal of Automated Software Engineering 11 (1) (2004) 27–61.
5. 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.
6. M. Daoudi, S. Danicic, J. Howroyd, M. Harman, C. Fox, L. Ouarbya, M. Ward, ConSUS: A scalable approach to conditioned slicing, in: IEEE Working Conference on Reverse Engineering (WCRE 2002), IEEE Computer Society Press, Los Alamitos, California, USA, Richmond, Virginia, USA, 2002, pp. 109 – 118, invited for special issue of the Journal of Systems and Software as best paper from WCRE 2002.
7. L. Ouarbya, S. Danicic, D. M. Daoudi, M. Harman, C. Fox, A denotational interprocedural program slicer, in: IEEE Working Conference on Reverse Engineering (WCRE 2002), IEEE Computer Society Press, Los Alamitos, California, USA, Richmond, Virginia, USA, 2002, pp. 181 – 189.
8. M. Harman, L. Hu, X. Zhang, M. Munro, S. Danicic, M. Daoudi, L. Ouarbya, An interprocedural amorphous slicer for WSL, in: IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002), IEEE Computer Society Press, Los Alamitos, California, USA, Montreal, Canada, 2002, pp. 105–114, selected for consideration for the special issue of the Journal of Automated Software Engineering.

9. R. Cartwright, M. Felleisen, The semantics of program dependence, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, 1989, pp. 13–27.
10. 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.
11. P. A. Hausler, Denotational program slicing, in: 22^{nd}, Annual Hawaii International Conference on System Sciences, Volume II, 1989, pp. 486–495.
12. M. Weiser, Program slicing, IEEE Transactions on Software Engineering 10 (4) (1984) 352–357.
13. R. Giacobazzi, I. Mastroeni, Non-standard semantics for program slicing, Higher-Order and Symbolic Computation(HOSC) 16 (4) (2003) 297–339.
14. J. Kennaway, J. Klop, M. Sleep, F. Vries, Transfinite reduction in orthogonal term rewriting systems., Information and computation 119 (1) (1995) 18–38.
15. 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.