Proc. Estonian Acad. Sci. Eng., 2005, **11**, 4, 313–328
https://doi.org/10.3176/eng.2005.4.05

Transfinite semantics in program slicing

Härmel Nestra

Institute of Computer Science, University of Tartu, J. Liivi 2, 50409 Tartu, Estonia; harmel.nestra@ut.ee

Received 4 August 2005, in revised form 23 September 2005

Abstract. This paper studies mathematically some special kinds of transfinite trace semantics and investigates program slicing w.r.t. these semantics. Several general facts about slicing, which hold for a wide class of programming languages and their transfinite semantics, are proven. The principal part of the work is done on control flow graphs keeping the treatment abstracted from any concrete programming language. Structured control flow is not assumed but programs written in standard programming languages with structured control flow are among those to which our theory applies.

Key words: program slicing, transfinite semantics, transfinite iteration.

1. INTRODUCTION

Program slicing is a kind of program transformation where the aim is to find an executable subset of the set of atomic statements of a program which is responsible for computing all the values important to the user. Slicing was introduced and its significance was explained first by Weiser [1]; summaries of its techniques and applications can be found in Tip [2] and Binkley and Gallagher [3].

A standard example of program slicing is the following (small numbers are short denotations of program points):

```
0sum := 0; 1prod := 1;
2i := 0;
 while 3i < n do (
   4i := i + 1;
   5sum := sum + i;
   6prod := prod * i
 );
7
```

$\longrightarrow$

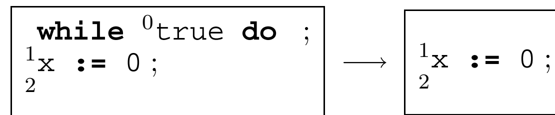
```
0sum := 0;
2i := 0;
 while 3i < n do (
   4i := i + 1;
   5sum := sum + i;

 );
7
```

The first program computes both the sum and the product of the first n positive integers (where n is the initial value of n). The second program computes the sum; all statements concerning the product only are *sliced away*. If sum is the only interesting value, the two programs are equally good.

The specification of which variables are important at which program points is called *slicing criterion*. It can be given mathematically as a binary relation between program points and variables. The essential property of a slice – being equally good to the original program in computing the values of user's interest – is then more precisely formulated as follows: for arbitrary initial values of variables, the slice and the original program compute the same sequence of values for every program point and variable related by the criterion.

The slice above has been found w.r.t. criterion $\{(7, \texttt{sum})\}$ saying that the user is interested in the value of variable sum at program point 7. As control reaches program point 7 just once (at the end of execution) and, when this happened, the value of sum computed by both programs is the same, the crucial property is met. If the criterion were $\{(5, \texttt{sum})\}$, the property would mean that the sequence of values aquired by sum at point 5 be the same in both programs. This is also true since both programs compute values $0, 1, 3, \ldots, (n-1)n/2$ for sum at 5. These observations together imply the property also for the criterion $\{(5, \texttt{sum}), (7, \texttt{sum})\}$.

If our concern is to prove correctness of slicing algorithms, we need a formalization of the crucial property. Clearly this must involve a semantics. It has been noticed earlier that standard semantics are not completely satisfactory for this purpose because slicing can produce terminating programs from nonterminating ones, which implies that the program points of interest can be reached more times in the slice than in the original program and the later reachings correspond to computation, never undertaken by the original program. As an illustration, consider the following example:



The second program is a slice of the first w.r.t. criterion $\{(2, \texttt{x})\}$. The loop is sliced away since no influence to x at point 2 can be detected. This causes the program point 2 to be reached once during the run of the slice while not being reached at all during the run of the original program.

This phenomenon is called *semantic anomaly* [4, 5]. It is a fundamental issue since no slicing algorithm can decide whether a loop terminates. Therefore non-trivial slicing algorithms, in particular the standard ones based on data flow analysis, cannot be correct w.r.t. standard semantics in all cases. (Reps and Yang [6] prove correctness of their notion of slice w.r.t. standard semantics under the restriction that the original program terminates.) Hence, for obtaining a working version of the notion of correctness here, one must abstract from termination.

Giacobazzi and Mastroeni [5] investigate transfinite semantics with the aim of solving this problem; the idea has been proposed already by Cousot [7]. By *transfinite semantics* one means a semantics, according to which computation may continue after an infinite number of steps from some limit state, determined somehow by the infinite computation performed.

Our paper follows up transfinite semantics and program slicing in the context of them. In Section 2, we argue that two special kinds of transfinite semantics, which we call iterative and corecursive, are particularly interesting and we carry out a mathematical investigation of them. Regarding deterministic transfinite semantics, this improves the theory of transfinite corecursion theory, reported by us in [8] (non-determinism is not investigated here). In Section 3, a transfinite trace semantics for while-loops is described together with further explanations why transfinite semantics works for program slicing. It also discusses defining transfinite semantics for programs with unstructured control flow. Section 4 defines a class of transfinite semantics we call escaping. A theorem, estimating the length of transfinite computation in escaping semantics, is proven; it both refines and generalizes an analogous result of Giacobazzi and Mastroeni [5]. Section 5 proves that the problem of finding statement minimal slice w.r.t. transfinite semantics is undecidable. For standard semantics, this result is known. Section 6 discusses related work and points out a few important differences of our approach from that of Giacobazzi and Mastroeni [5]. Section 7 refers to some problems with using transfinite semantics in program slicing and hints at the work yet to be done.

2. TRANSFINITE SEMANTICS ABSTRACTLY

A trace semantics of a program expresses its execution behaviour step by step. It is basically a set of sequences of items representing execution states. In standard trace semantics, the sequences are finite lists or streams whose components are indexed with natural numbers. In transfinite trace semantics, the sequences are transfinite, the components are indexed with ordinals. Call them *transfinite lists*.

The notion of ordinal is obtained as a generalization of the notion of natural number by adding transfinite elements. So we have all the natural numbers $0, 1, 2, \ldots$, as well as $\omega, \omega+1, \omega+2$ etc., $\omega+\omega$ and a lot of greater elements, among ordinals. Standardly, ordinals are defined as isomorphism classes of well-ordered sets. The essence of ordinals together with their standard order is expressed by the fact that, for arbitrary set of ordinals, there exists the least ordinal greater than any element of this set. There are many books giving profound introductions to ordinal theory; [9, 10] represent just two different approaches.

We treat transfinite lists over A as functions which take ordinals into A and whose domain is downward closed. So a transfinite list over A is a function $l : \mathbb{O}_o \to A$ for some o where $\mathbb{O}_o$ denotes the set of all ordinals less than o; in this case, o is called *length* of l and denoted by $|l|$. Denote the empty list – the only list of length 0 – by nil.

For a transfinite list l and $\alpha < |l|$, $l(\alpha)$ (or l_α) is the *αth component* of l. For simplicity, we allow writing $l(\alpha)$ also for $\alpha \geqslant |l|$ and count $l(\alpha) = \bot \notin A$ in this case. The first component, $l(0)$, is also denoted by $\mathrm{head}\, l$.

For every transfinite list l and $o \leqslant |l|$, let $\mathrm{take}\, o\, l$ and $\mathrm{drop}\, o\, l$ denote the transfinite list which is obtained from l by taking and dropping, respectively, the first o elements from it. So, for any ordinal π,

$$(\mathrm{take}\, o\, l)(\pi) = \left\{ \begin{array}{ll} l(\pi) & \text{if } \pi < o \\ \bot & \text{otherwise} \end{array} \right\}, \qquad (\mathrm{drop}\, o\, l)(\pi) = l(o + \pi) \ .$$

Thereby, $|\mathrm{take}\, o\, l| = o$ and $|\mathrm{drop}\, o\, l| = |l| - o$.

If $o > |l|$ or l is not a list then $\mathrm{take}\, o\, l = \bot = \mathrm{drop}\, o\, l$. All operations considered in this paper, including function application, are strict, i.e. a subexpression with value $\bot$ turns the value of the whole expression to $\bot$.

Lemma 1. *Let l be any transfinite list.*
(i) *For ordinals o and π*, $l(o + \pi) = (\mathrm{drop}\, o\, l)(\pi)$;
(ii) *For ordinals o and π*, $\mathrm{drop}(o + \pi)\, l = \mathrm{drop}\, \pi(\mathrm{drop}\, o\, l)$;
(iii) *For ordinals o and π*, $\mathrm{take}\, \pi(\mathrm{drop}\, o\, l) = \mathrm{drop}\, o(\mathrm{take}(o + \pi)\, l)$.

This lemma was proven in [8]. The claims are rather intuitive and we are going to use them without any reference; however, note that the claims hold also for cases where some expressions evaluate to $\bot$.

A transfinite list is typically defined using transfinite recursion. This means that every element of the list is expressed in terms of all the preceding elements. In the case of semantics, this is not completely satisfactory. In a deterministic standard trace semantics, every execution state is completely determined by its single predecessor and it would be unnecessarily burdening or misguiding to carry all the preceding states along in definition.

When defining a transfinite computation, we similarly prefer to express every state in terms of as few preceding states as possible. However, if the number of the preceding states is a limit ordinal then one cannot find the last among them on which the new state could solely depend.

For example, if one is defining $l(\omega)$ then at least ω elements backward must be taken into account. In defining $l(\omega + k)$ for a positive natural k, it suffices to consider the last element only. But when defining $l(\omega + \omega)$, there is no last element again; at least ω elements backward must be studied.

This consideration leads to our notion of selfish ordinal. In [10], these ordinals are called *additive principal numbers*; we like our shorter term more.

Definition 1. *We call an ordinal $\gamma > 0$* selfish *if $\gamma - o = \gamma$ for every $o < \gamma$.*

In other words, γ is selfish iff the well-order of the part, remaining when cutting out any proper initial part from the well-order Γ representing γ, is isomorphic to Γ itself. One more characterization is as follows: $\gamma > 0$ is selfish iff it cannot be

expressed as the sum of two ordinals less than γ (i.e. the set of ordinals less than γ is closed under finite sums).

For example, ω is selfish. If one cuts out any proper initial part of the well-order representing ω (see figure), the remaining part represents ω itself.

$$\bullet - \bullet - \bullet - \bullet - \bullet - \bullet - \bullet - \bullet - \ldots$$

The ordinals $\omega + \omega$, $\omega + \omega + \omega$ etc. are not selfish because removing the initial ω leads to a smaller number. However, the limit of this sequence is selfish.

Note that 1 is selfish – the least, the only finite and the only successor ordinal among them.

Proposition 1.
(i) *Every ordinal $o > 0$ is uniquely representable in the form $o = \alpha + \gamma$ where γ is selfish and α is the least ordinal for which $o - \alpha$ is selfish.*
(ii) *Every ordinal $o > 0$ is uniquely representable in the form $o = \lambda + \beta$ where λ is selfish and $\beta < o$.*

Proposition 1 implies that every ordinal can be uniquely expressed as the sum of the elements of a finite non-increasing list of selfish ordinals. This fact can also be deduced from the classical theorem of ordinal theory about representations on base since it can be proven that an ordinal is selfish if and only if it is a power of ω; the representation on base ω is also called *Cantor normal form* [10, 11].

In the rest, we call the representation $o = \alpha + \gamma$ where γ being selfish and α minimized (the representation of Proposition 1(i)) the *principal representation* of o. If the Cantor normal form of o is written as a sum of powers of ω like in [10] then adding all summands but the last of this sum gives the first component of the principal representation of o and the last summand equals to the other component.

Suppose we are defining $l(o)$ in terms of elements preceding it in list l. The selfish ordinal in the principal representation of o coincides with the number of elements, inevitable to study backward in the list l. For example, the principal representation of ω is $0 + \omega$; the principal representation of $\omega + k$ with any positive natural number k is $(\omega + (k - 1)) + 1$; the principal representation of $\omega \cdot k = \underbrace{\omega + \ldots + \omega}_{k}$ with any positive natural number k is $\omega \cdot (k - 1) + \omega$.

Let $\propto$ be a fixed selfish ordinal and let $\mathrm{TList}\, A$ denote the set of all transfinite lists over A of length not exceeding $\propto$. Let $\mathrm{STList}\, A$ denote the subset of $\mathrm{TList}\, A$ consisting of lists by which next elements are defined, i.e. lists of length being both selfish and less than $\propto$ (lists of length $\propto$ do not have next elements). So

$$\mathrm{TList}\, A = \bigcup_{o \leqslant \propto} (\mathbb{O}_o \to A), \qquad \mathrm{STList}\, A = \bigcup_{\substack{\gamma < \propto \\ \gamma \text{ selfish}}} (\mathbb{O}_\gamma \to A).$$

Definition 2. *Let X, A be sets. Assume $\varphi : X \to 1 + A = A \cup \{\perp\}$ and $\psi : \mathrm{STList}\, A \to X$. We say that a function $h : X \to \mathrm{TList}\, A$ is* iterative on φ and ψ *iff, for each $x \in X$, the following two conditions hold*:

1. $h(x)(0) = \varphi(x)$;
2. $h(x)(o) = \varphi(\psi(\mathrm{take}\,\gamma(\mathrm{drop}\,\alpha(h(x)))))$ *for every* $o < \propto$ *with principal representation* $o = \alpha + \gamma$.

This notion captures the desire described above: oth component of a list $h(x)$ is defined in terms of γ preceding components where γ is the selfish ordinal from the principal representation of o. As 1 is one particular selfish ordinal, the iteration schema handles finite and infinite steps uniformly.

Example 1. *Take* $A = \mathbb{N}$, $X = \mathbb{Z}$. *For* $x \in \mathbb{Z}$ *and* $l \in \mathrm{STList}\,\mathbb{N}$, *define*

$$\varphi(x) = \left\{ \begin{matrix} x & \textit{if } x \in \mathbb{N} \\ \bot & \textit{otherwise} \end{matrix} \right\}, \qquad \psi(l) = \left\{ \begin{matrix} \mathrm{head}\,l & \textit{if } |l| = 1 \\ n+1 & \textit{if } l \textit{ stabilizes to } n \\ -1 & \textit{otherwise} \end{matrix} \right\}.$$

Provided $\propto \geqslant \omega^2$, *the following function* h *is iterative on* φ *and* ψ:

$$h(x) = \left\{ \begin{matrix} \underbrace{(\underbrace{x, x, \ldots}_{\omega}, \underbrace{x+1, x+1, \ldots}_{\omega}, \underbrace{x+2, x+2, \ldots}_{\omega}, \ldots\ldots)}_{\omega} & \textit{if } x \in \mathbb{N} \\ \mathrm{nil} & \textit{otherwise} \end{matrix} \right\}.$$

Theorem 1. *Let* X, A *be sets. For every* $\varphi : X \to 1 + A = A \cup \{\bot\}$ *and* $\psi : \mathrm{STList}\,A \to X$, *there exists a unique function* $h : X \to \mathrm{TList}\,A$ *being iterative on* φ *and* ψ.

Proof. Essentially done in [8] at the beginning of the proof of corecursion theorem for deterministic semantics. □

Theorem 1 asserts that, for defining a transfinite semantics "by iteration", it suffices to provide just φ and ψ.

Standard deterministic trace semantics have the nice property that the part of the computation, starting from an intermediate state s, is independent of the computation performed before reaching s. This is because state s alone uniquely determines all the following computation. For transfinite semantics, even if defined by transfinite iteration, this property may not hold.

Here we find a weaker condition holding also for iterative transfinite semantics; furthermore, we find a natural restriction on ψ in case of which the corresponding transfinite semantics satisfies also the desired stronger property. We call the two conditions weak corecursivity and corecursivity, respectively. We chose such word because the conditions are to some extent analogous to traditional stream corecursion (the analogy will be explained below).

Definition 3. *Let* X, A *be sets.*

(i) *Call any function* $\psi : \mathrm{STList}\,A \to X$ limit operator *if* $\psi(l) = \psi(\mathrm{drop}\,\lambda\,l)$ *for all selfish ordinals* λ, γ *with* $\lambda < \gamma < \propto$ *and transfinite lists* $l \in \mathbb{O}_\gamma \to A$.

(ii) *Assume* $\varphi : X \to 1{+}A = A \cup \{\bot\}$, $\psi : \mathrm{STList}\,A \to X$ *and* $h : X \to \mathrm{TList}\,A$. *Consider the following properties*:

1. *if* $\varphi(x) = a \in A$ *then* $\mathrm{head}(h(x)) = a$, *and*
   if $\varphi(x) = \bot$ *then* $h(x) = \mathrm{nil}$;
2. *if* $|h(x)| \geqslant \lambda$ *and* λ, μ *are consecutive selfish ordinals with* $\lambda < \mu \leqslant \propto$ *then, for every ordinal* $o < \mu$,

$$\mathrm{drop}\,\lambda(h(x))(o) = h(\psi(\mathrm{take}\,\lambda(h(x))))(o);$$

3. *if* $|h(x)| \geqslant \lambda$ *and* $\lambda < \propto$ *is selfish then*

$$\mathrm{drop}\,\lambda(h(x)) = h(\psi(\mathrm{take}\,\lambda(h(x)))).$$

We say that $h : X \to \mathrm{TList}\,A$ *is* weakly corecursive on φ and ψ *iff the conditions* 1 *and* 2 *hold. We say that* $h : X \to \mathrm{TList}\,A$ *is* corecursive on φ and ψ *iff the conditions* 1 *and* 3 *hold.*

Limit operators are analogous to limits in calculus by certain properties (the limit of a sequence equals to the limit of its every subsequence; all sequences obtained as a final part of a diverging sequence also diverge). In the case of semantics, ψ being a limit operator means that the limit state, which appears immediately after an infinite computation, does not depend on the actual starting point of the final part of selfish length, i.e. one does not need to use the principal representation to determine the final part to rely on, but may equivalently use any final part of the same length. This stricture on ψ seems natural but we will see below that, when inventing transfinite semantics, appropriate for describing slicing programs with unstructured control flow, ψ may not be a limit operator.

Condition 3 of Definition 3(ii) obviously implies condition 2 (2 requires something to hold for every $o < \mu$ while 3 requires essentially the same thing to hold for all o), hence corecursivity implies weak corecursivity.

Taking $\lambda = 1$ in the definition gives a construction similar to stream corecursion in the sense that the result list is defined by giving its head and expressing its tail as the value of the same function which is being defined. (Conditions 2 and 3 are equivalent in stream case since $\lambda = 1$ implies $\mu = \omega$; so both conditions apply to the whole stream). In transfinite corecursion, the breaking point can be after any initial part of selfish length rather than after the head only. Unlike in the traditional corecursion, any component of any list being a value of a function corecursive in our sense determines all the following components uniquely.

With Theorem 2(i) below, we obtain the equivalence of iterativity and weak corecursivity. Theorem 2(ii) was proven already in [8] but there we gave a direct proof while the proof presented here relies on weak corecursivity. In [8], also an analogous theorem for non-deterministic semantics was proven.

Theorem 2. *Let* X, A *be sets. Let* $\varphi : X \to A \cup \{\bot\} = 1 + A$, $\psi : \mathrm{STList}\,A \to X$ *and* $h : X \to \mathrm{TList}\,A$.
(i) *Then* h *is iterative on* φ *and* ψ *iff* h *is weakly corecursive on* φ *and* ψ.
(ii) *If* h *is iterative on* φ *and* ψ *and* ψ *is a limit operator then* h *is corecursive on* φ *and* ψ.

Proof.
(i) Similar to the proof of corecursion theorem for deterministic semantics in [8] whereby the uniqueness part there corresponds to the if-part here.
(ii) Our h is weakly corecursive by part (i). It remains to prove condition 3 from Definition 3(ii). Prove by transfinite induction on o that

$$\forall \lambda < \propto \forall x \in X \ (\mathrm{drop}\, \lambda(h(x))(o) = h(\psi(\mathrm{take}\, \lambda(h(x))))(o)) \ ,$$

where λ ranges over selfish ordinals only. If $o = 0$ then the claim holds by weak corecursivity. If $o > 0$, let $o = \kappa + \beta$ with selfish κ and $\beta < o$ (possible by Proposition 1(ii)). Fix λ and let μ be the next selfish ordinal. If $\mu > \kappa$ then $o = \kappa + \beta < \mu$ (because otherwise $\beta \geqslant \mu$ implying $\beta = \kappa + \beta = o$) and the claim holds again by weak corecursivity. Hence assume $\mu \leqslant \kappa$. So $\lambda < \kappa$ and $\lambda + \kappa = \kappa$. The induction hypothesis implies $\mathrm{take}\, \kappa(\mathrm{drop}\, \lambda(h(x))) = \mathrm{take}\, \kappa(h(\psi(\mathrm{take}\, \lambda(h(x)))))$, as well as $\mathrm{drop}\, \kappa(h(y))(\beta) = h(\psi(\mathrm{take}\, \kappa(h(y))))(\beta)$ for all $y \in X$. Using this knowledge together with the assumption that ψ is a limit operator, we obtain

$$\begin{aligned}
\mathrm{drop}\, \lambda(h(x))(\kappa+\beta) &= h(x)(\lambda+\kappa+\beta) = h(x)(\kappa+\beta) = \mathrm{drop}\, \kappa(h(x))(\beta) \\
&= h(\psi(\mathrm{take}\, \kappa(h(x))))(\beta) \\
&= h(\psi(\mathrm{drop}\, \lambda(\mathrm{take}\, \kappa(h(x)))))(\beta) \\
&= h(\psi(\mathrm{drop}\, \lambda(\mathrm{take}(\lambda+\kappa)(h(x)))))(\beta) \\
&= h(\psi(\mathrm{take}\, \kappa(\mathrm{drop}\, \lambda(h(x)))))(\beta) \\
&= h(\psi(\mathrm{take}\, \kappa(h(\psi(\mathrm{take}\, \lambda(h(x)))))))(\beta) \\
&= \mathrm{drop}\, \kappa(h(\psi(\mathrm{take}\, \lambda(h(x)))))(\beta) \\
&= h(\psi(\mathrm{take}\, \lambda(h(x))))(\kappa+\beta) \ . \qquad \square
\end{aligned}$$

Function ψ of Example 1 is a limit operator, so Theorem 2(ii) gives that h of that example is corecursive. It is also easy to check this directly. We showed in [8] that, without the restriction on ψ, Theorem 2(ii) breaks.

We will need the following corollary in Section 4.

Corollary 1. *Let X, A be sets. Let $\varphi : X \to A \cup \{\bot\} = 1 + A$, $\psi : \mathrm{STList}\, A \to X$ and let $h : X \to \mathrm{TList}\, A$ be iterative on φ and ψ. Denote function composition by ; (function in the left is applied first). Let λ, μ be consecutive selfish ordinals with $\lambda < \mu \leqslant \propto$. Then, for every natural number n,*

$$h \ ; \mathrm{drop}(\lambda \cdot n) \ ; \mathrm{take}\, \mu = (h \ ; \mathrm{take}\, \lambda \ ; \psi)^n \ ; h \ ; \mathrm{take}\, \mu.$$

Proof. By weak corecursivity,

$$h \ ; \mathrm{drop}\, \lambda \ ; \mathrm{take}\, \mu = h \ ; \mathrm{take}\, \lambda \ ; \psi \ ; h \ ; \mathrm{take}\, \mu.$$

By $\lambda < \mu$ and μ being selfish,

$$\mathrm{drop}\, \lambda \ ; \mathrm{take}\, \mu = \mathrm{take}(\lambda + \mu) \ ; \mathrm{drop}\, \lambda = \mathrm{take}\, \mu \ ; \mathrm{drop}\, \lambda.$$

Argue by induction on n. If $n = 0$, both sides of the desired equation reduce to h ; $\mathrm{take}\,\mu$. If the claim holds for n, we get

$$\begin{aligned} h \,;\, \mathrm{drop}(\lambda \cdot (n+1)) \,;\, \mathrm{take}\,\mu &= h \,;\, \mathrm{drop}(\lambda \cdot n + \lambda) \,;\, \mathrm{take}\,\mu \\ &= h \,;\, \mathrm{drop}(\lambda \cdot n) \,;\, \mathrm{drop}\,\lambda \,;\, \mathrm{take}\,\mu \\ &= h \,;\, \mathrm{drop}(\lambda \cdot n) \,;\, \mathrm{take}\,\mu \,;\, \mathrm{drop}\,\lambda \\ &= (h \,;\, \mathrm{take}\,\lambda \,;\, \psi)^n \,;\, h \,;\, \mathrm{take}\,\mu \,;\, \mathrm{drop}\,\lambda \\ &= (h \,;\, \mathrm{take}\,\lambda \,;\, \psi)^n \,;\, h \,;\, \mathrm{drop}\,\lambda \,;\, \mathrm{take}\,\mu \\ &= (h \,;\, \mathrm{take}\,\lambda \,;\, \psi)^n \,;\, h \,;\, \mathrm{take}\,\lambda \,;\, \psi \,;\, h \,;\, \mathrm{take}\,\mu \\ &= (h \,;\, \mathrm{take}\,\lambda \,;\, \psi)^{n+1} \,;\, h \,;\, \mathrm{take}\,\mu \ . \qquad \square \end{aligned}$$

3. CONFIGURATION TRACE SEMANTICS

We work as much as possible on control flow graphs to obtain uniform results for a wide class of programming languages. Just say we have an imperative language *Prog* whose programs are all finite and involve neither recursion (direct or mutual) nor non-determinism. In examples, we use ubiquitous syntactic constructs belonging to the most popular imperative programming languages.

Program points of a program S are potential locations of control during executions of S. Assume that the set of all program points of any program S is finite and contains a fixed *initial* program point i_S.

A *configuration* is a pair of a program point and a *state*, the latter containing an evaluation of variables. Let State and *Conf* denote the set of all states and the set of all configurations, respectively. The configuration with program point p and state s is denoted by $\langle \mathsf{p} \mid s \rangle$; the program point occurring in configuration c is denoted by $\mathrm{pp}\, c$.

We are going to study semantics where the meaning of a program is a function whose values are sequences of configurations expressing the computation process step by step. The states of Section 2 are actually abstractions of configurations.

Suppose a transition function $\mathrm{next} : \mathit{Conf} \to 1 + \mathit{Conf} = \mathit{Conf} \cup \{\bot\}$ is fixed; applying next represents making an atomic computation step. The *control flow graph* of a program S, denoted by $\mathrm{cfg}\, S$, is a directed graph whose vertices are all the program points of S and arcs represent atomic computations (usually assignment, predicate test etc.). The set of all program points of S can therefore be denoted by $V(\mathrm{cfg}\, S)$. So $\mathsf{i}_S \in V(\mathrm{cfg}\, S)$, and if $\mathrm{next}\langle \mathsf{p} \mid s \rangle = \langle \mathsf{q} \mid t \rangle$ where $\mathsf{p} \in V(\mathrm{cfg}\, S)$ then $\mathsf{q} \in V(\mathrm{cfg}\, S)$ and $\mathrm{cfg}\, S$ has an arc from p to q. Every computation with a program S redounds as a walk in $\mathrm{cfg}\, S$.

To see how transfinite trace semantics helps to avoid semantic anomaly of program slicing (see Section 1), consider the following way to define transfinite semantics for a program, containing a while-loop. Besides the transition function, we must provide principles for finding limit configurations of endless sequences

of them. As explained in Section 2, it suffices to provide rules for lists of selfish length (in terms of Definition 2 and Theorem 1, we must define φ and ψ).

For the limit program point $\lim \mathsf{p}$ of a transfinite list p, coming up as the sequence of program points, visited during a repetition of the body of a while-loop for ω times, take the immediate postdominator of the program point, corresponding to the head of the loop in the control flow graph. Typically, this corresponds to the part of code, lexically following the loop.

This ensures that, after executing the body of a loop for ω times, we reach a configuration where we have "overcome" the loop. A loop **while** B **do** T in this semantics means "while B keeps holding, do T, but never more than ω times".

In the limit state $\lim s$ of a state list s, a variable X has value a if the transfinite list of the values of X during the transfinite computation represented by s stabilizes to a; if the list does not stabilize then the value of X is ambiguous ($\top$). This choice is to some extent arbitrary; some non-stabilizing sequences of values may possess limits of some other kind being natural to use instead of $\top$. (Giacobazzi and Mastroeni [5] have an example where the limit of the non-stabilizing sequence $1, 2, 3, \ldots$ is taken to be ω.)

For every transfinite configuration list $c = (\langle \mathsf{p}_o \mid s_o \rangle : o < \gamma)$ with selfish length γ, define

$$\psi(c) = \begin{Bmatrix} \mathrm{next}(\mathrm{head}\, c) & \text{if } \gamma = 1 \\ \langle \lim \mathsf{p} \mid \lim s' \rangle & \text{otherwise} \end{Bmatrix},$$

where s' is the transfinite list obtained from s by keeping only those states which occur when control passes through the head of the while-loop causing the infinite computation c. Then we have $\psi : \mathrm{STList}\, \mathit{Conf} \to 1 + \mathit{Conf}$.

By Theorem 1, there exists a function $h : 1 + \mathit{Conf} \to \mathrm{TList}\, \mathit{Conf}$ being iterative on $\mathrm{id} : 1 + \mathit{Conf} \to 1 + \mathit{Conf}$ and ψ. The desired transfinite semantics $\mathcal{T} : \mathit{Prog} \to \mathit{State} \to \mathrm{TList}\, \mathit{Conf}$ is achieved by defining $\mathcal{T}(S)(s) = h\langle i_S \mid s \rangle$ for every program S and initial state s. It is easy to verify that ψ is a limit operator in the sense of Definition 3(i); hence the semantics is even corecursive.

Being strict, applying Theorem 1 needs fixing an ordinal $\propto$ which is the upper bound of lengths of all transfinite lists obtained as values of the iterative functions. We can choose $\propto$ arbitrarily; Giacobazzi and Mastroeni [5] prove for a simple language IMP with structured control flow that taking $\propto = \omega^{\omega+1}$ ensures any program being executed to the end of its code. In Section 4, we improve the result achieving ω^ω as the bound and generalize it to a wider class of languages.

In this semantics, the execution of the program in the last example of Section 1 with initial state $\{ \mathtt{x} \to 1 \}$ goes as follows:

$$\langle 0 \mid \{ \mathtt{x} \to 1 \} \rangle \to \langle 0 \mid \{ \mathtt{x} \to 1 \} \rangle \to \langle 0 \mid \{ \mathtt{x} \to 1 \} \rangle \to \underbrace{\ldots\ldots}_{\omega \text{ steps}}$$
$$\to \langle 1 \mid \{ \mathtt{x} \to 1 \} \rangle \to \langle 2 \mid \{ \mathtt{x} \to 0 \} \rangle .$$

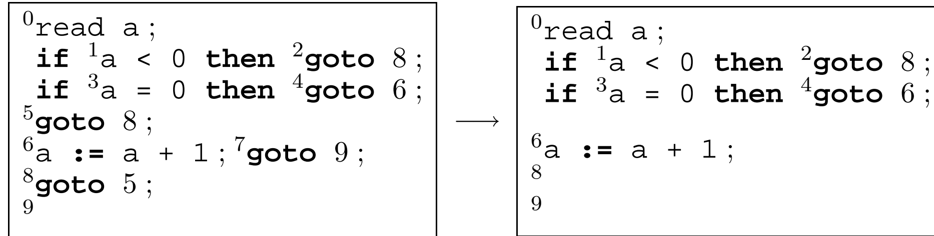
It visits program point 2 once like the slice and computes the same value for $\mathtt{x}$.

For every $\psi : \mathrm{STList}\ \mathit{Conf} \to 1 + \mathit{Conf}$, denote the function being iterative on $\mathrm{id} : 1 + \mathit{Conf} \to 1 + \mathit{Conf}$ and ψ by $\mathrm{iter}\,\psi$ and the corresponding transfinite semantics by $\mathcal{T}_\psi$. So we have the formula $\mathcal{T}_\psi(S)(s) = \mathrm{iter}\,\psi\langle i_S \mid s\rangle$.

In the case of while-loops, defining limit configurations does not make much trouble. The choice of the limit program point is particularly straightforward because there is just one natural way to escape from the loop – following the arc of the control flow graph used when the predicate evaluates to false.

If the control flow is unstructured, such an obvious choice need not exist. Obscurity can arise also in the case of structured control flow with statements like `break` in C as they can cause more than one arc escaping from a loop. To ensure a transfinite semantics being in harmony with program slicing, a general guideline for defining a limit point could be choosing the point where control would fall if the loop were removed.

In the following example with unstructured control flow, we use our abstract program point notation in goto-statements since the code is primarily intended to be illustrative. Each if-statement incorporates only one row in the program.



Suppose the slicing criterion is $\{(9, \mathtt{a})\}$. The loop, consisting of statements 5 and 8, does not affect the value of a, therefore it is sliced away. As a result of this transformation, control reaches program point 6 also in the case $a > 0$ (where a is the input value of a). If $a < 0$, control bypasses this program point.

To be consistent with such a way of slicing, a transfinite semantics of the original program must jump to 6 after the infinite loop if it started at 5 (the case $a > 0$) and to 9 if it started at 8 (the case $a < 0$). This way, the limit point of the loop depends on how far backward we observe it. Thus if the semantics is of the form $\mathcal{T}_\psi$ then ψ is not a limit operator and the semantics is not corecursive.

4. ESCAPING TRANSFINITE SEMANTICS

Irrespective of the possible universal rules for choosing limit points, we can notice a natural property, desired in probably all situations. Namely, the limit point must be outside the loop, causing non-termination as the idea behind the transfinite semantics to be able to overcome non-terminating parts of programs.

This observation leads to the kind of transfinite semantics we call escaping.

Definition 4.
(i) *Let* $l \in \mathrm{TList}\,\mathit{Conf} \setminus \{\mathrm{nil}\}$. *We call a program point* p looping in l *iff, for every ordinal* $o < |l|$, *there exists an ordinal* π, $o < \pi < |l|$, *such that* $\mathrm{pp}\, l_\pi = \mathsf{p}$. *The set of all program points looping in* l *is denoted by* $\mathrm{loop}\, l$.
(ii) *Call a function* $\psi : \mathrm{STList}\,\mathit{Conf} \to 1 + \mathit{Conf}$ escaping *iff, for every* $c \in \mathit{Conf}$ *and selfish ordinal* γ *satisfying* $1 < \gamma < |\,\mathrm{iter}\,\psi\, c|$, *if we define* $l = \mathrm{take}\,\gamma(\mathrm{iter}\,\psi\, c)$ *then* $\psi(l) \in \mathit{Conf}$ *and* $\mathrm{pp}(\psi(l)) \notin \mathrm{loop}\, l$. *Call a transfinite configuration trace semantics* escaping *iff it is of form* $\mathcal{T}_\psi$ *for some escaping* ψ.

Clearly a computation l contains looping program points only if the length of l is a limit ordinal. Note also that, for every non-empty computation l, there exists an ordinal $o < |l|$ such that $\mathrm{pp}\, l_\pi \in \mathrm{loop}\, l$ for every π, $o < \pi < |l|$. This holds because, for every non-looping program point, either l does not visit it or a visit of it is the last in l – as a program has a finite number of program points only, one can find o so that no visits of non-looping program points occur after the oth step. A semantics is escaping if, after any endless computation, control reaches a program point which it has not visited during an infinite final part of this computation. The transfinite semantics for while-loops considered in Section 3 is obviously escaping by the definition of $\lim \mathsf{p}$ for program point lists p.

Next we prove that the theorem of Giacobazzi and Mastroeni [5] on estimation of the length of transfinite computation of an IMP program holds for all escaping semantics, irrespective of the language. We achieve also a bit better estimation.

Denote the set of all program points visited by computation c by $\mathrm{occur}\, c$.

Lemma 2. *Let* $\psi : \mathrm{STList}\,\mathit{Conf} \to 1 + \mathit{Conf}$ *be escaping. For every natural number* k *and arbitrary* $c \in \mathit{Conf}$,

$$|\,\mathrm{iter}\,\psi\, c| \geqslant \omega^k \;\Rightarrow\; |\,\mathrm{loop}\,(\mathrm{take}\,\omega^k(\mathrm{iter}\,\psi\, c))| \geqslant k,$$
$$|\,\mathrm{iter}\,\psi\, c| > \omega^k \;\Rightarrow\; |\,\mathrm{occur}\,(\,\mathrm{take}(\omega^k + 1)(\mathrm{iter}\,\psi\, c))| > k.$$

Proof. Prove by induction on k. The case $k = 0$ is trivial.

Suppose the claim holding for k and assume

$$|\,\mathrm{iter}\,\psi\, c| \geqslant \omega^{k+1} = \omega^k \cdot \omega = \underbrace{\omega^k + \omega^k + \ldots}_{\omega}.$$

Thus the list $\mathrm{take}\,\omega^{k+1}(\mathrm{iter}\,\psi\, c)$ divides into ω subparts, each of length ω^k. Each subpart is of the form $\mathrm{take}\,\omega^k(\mathrm{drop}(\omega^k \cdot n)(\mathrm{iter}\,\psi\, c))$ for a natural number n.

Apply Corollary 1 with $h = \mathrm{iter}\,\psi$ and $\lambda = \omega^k$, $\mu = \omega^{k+1}$ (note that being selfish is equivalent to being a power of ω). We obtain

$$\mathrm{take}\,\omega^{k+1}(\mathrm{drop}(\omega^k \cdot n)(\mathrm{iter}\,\psi\, c)) = \mathrm{take}\,\omega^{k+1}(\mathrm{iter}\,\psi\, d), \tag{1}$$

where $d = (\mathrm{iter}\,\psi\ ;\ \mathrm{take}\,\omega^k\ ;\ \psi)^n(c)$. Both sides of (1) are different from $\perp$ since our assumption $|\,\mathrm{iter}\,\psi\, c| \geqslant \omega^{k+1}$ implies $|\,\mathrm{drop}(\omega^k \cdot n)(\mathrm{iter}\,\psi\, c)| \geqslant \omega^{k+1}$. This

allows to conclude $|\,\mathrm{iter}\,\psi\, d| \geqslant \omega^{k+1} > \omega^k$ and $\mathrm{take}\, o(\mathrm{drop}(\omega^k \cdot n)(\mathrm{iter}\,\psi\, c)) = \mathrm{take}\, o(\mathrm{iter}\,\psi\, d)$ for all $o \leqslant \omega^{k+1}$. Now the induction hypothesis gives

$$\begin{aligned} &|\,\mathrm{occur}\,(\mathrm{take}(\omega^k+1)(\mathrm{drop}(\omega^k \cdot n)(\mathrm{iter}\,\psi\, c)))| \\ &\qquad = |\,\mathrm{occur}\,(\mathrm{take}(\omega^k+1)(\mathrm{iter}\,\psi\, d))| > k. \end{aligned} \tag{2}$$

Let $m = |\,\mathrm{loop}\,\big(\mathrm{take}\,\omega^{k+1}(\mathrm{iter}\,\psi\, c)\big)\,|$. It is possible to find n such that the computation $\mathrm{drop}(\omega^k \cdot n)(\mathrm{take}\,\omega^{k+1}(\mathrm{iter}\,\psi\, c))$ visits these m looping program points only. Therefore $m \geqslant k+1$ since, by (2), the first ω^k+1 steps of this computation visit more than k program points.

Finally, if $|\,\mathrm{iter}\,\psi\, c| > \omega^{k+1}$ then $\omega^{k+1} < \propto$. The representation $\omega^{k+1} = 0 + \omega^{k+1}$ is principal, hence, by iterativity and escapement,

$$\begin{aligned} \mathrm{pp}((\mathrm{iter}\,\psi\, c)(\omega^{k+1})) &= \mathrm{pp}(\psi(\mathrm{take}\,\omega^{k+1}(\mathrm{drop}\,0(\mathrm{iter}\,\psi\, c)))) \\ &= \mathrm{pp}(\psi(\mathrm{take}\,\omega^{k+1}(\mathrm{iter}\,\psi\, c))) \\ &\notin \mathrm{loop}\,(\mathrm{take}\,\omega^{k+1}(\mathrm{iter}\,\psi\, c)). \end{aligned}$$

Therefore $|\,\mathrm{occur}\,\big(\mathrm{take}(\omega^{k+1}+1)(\mathrm{iter}\,\psi\, c)\big)\,| > k+1$. □

Theorem 3. *Let $\mathcal{T}$ be an escaping semantics. Let l be a transfinite list of configurations obtained as a computation process according to a program S in semantics $\mathcal{T}$. Then $|l| \leqslant \omega^{|V(\mathrm{cfg}\, S)|} < \omega^\omega$.*

Proof. By conditions, $l = \mathcal{T}(S)(s) = \mathrm{iter}\,\psi\langle i_S \mid s\rangle$ for some state s and escaping operator ψ. Suppose $|l| > \omega^{|V(\mathrm{cfg}\, S)|}$. Then Lemma 2 implies that l visits more program points than there is in $\mathrm{cfg}\, S$, a contradiction. □

For every $n \in \mathbb{N}$, the length of the transfinite computation of the program

$$\underbrace{\texttt{while true do} \;\ldots\ldots\; \texttt{while true do}}_{n}$$

is ω^n. The least common upper bound of the numbers ω^n is ω^ω. Hence Theorem 3 achieves the best conservative estimation common to all programs.

5. UNDECIDABILITY RESULTS

When slicing programs in practice, our natural desire is to compute slices having as few statements as possible. Such slices are called *statement minimal*.

Weiser [1] has shown that the problem of finding statement minimal slices is undecidable but he considers slicing w.r.t. standard semantics. The same argument fails for transfinite semantics. Therefore, it is natural to ask whether the minimal slice problem is decidable w.r.t. transfinite semantics of our style.

The answer to this question is also negative. We prove this for while-loops, hence the result holds also in general case. The idea of our proof is similar to Weiser's: reduce the halting problem to the minimal slice problem.

Let S be an arbitrary program in our language. Assume that no branching predicate in S has any side-effect. This assumption in no way loses the generality. For each loop of shape **while** B **do** T occurring in S, replace it with code

```
X := B ;
while X do (T ; X := B) ;
Z := X || Z
```

where X, Z are variables not occurring in S. Let the resulting program be S'.

Denote the truth values by tt and ff. As predicates B have no side-effect, the change of the loops affects neither their termination/nontermination status nor the values assigned to the variables of S. Thus S' and S either both terminate or both loop. If the body of a loop in S' is executed a finite number of times then, before exiting the loop, X gets value ff. If the body is executed for ω times then X has always value tt when control reaches the head of the loop, hence the value of X after leaving the loop is tt. In this way, the running value of Z tells whether the computation has already looped or not.

Consider finding a minimal slice of the program Z `:= false` ; S' w.r.t. Z at the final point. If S' terminates then Z has value ff at the final point, therefore S' can be sliced away. Note that there is no other statement in the program which would alone guarantee Z having value ff at the end, thus a hypothetical solver of minimal slice problem is required to output Z `:= false`. If S' does not terminate then Z has value tt at the final point, therefore the solver must output something else. Altogether, this solver would decide also the halting problem. Thus the minimal slice problem is undecidable.

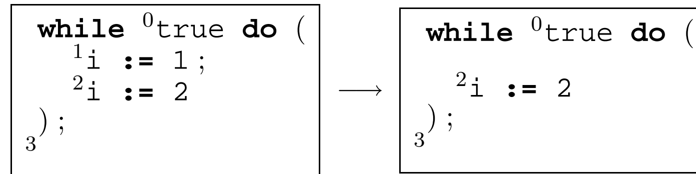
Note that the difficulty actually lies in checking whether one program is a slice of another w.r.t. given criterion. If we were able to perform this check, we would solve the minimal slice problem by checking all subsets of the given program and outputting one of the smallest subsets among those, which turn out to be slices. So whatever semantics we have, if the programs are finite and minimal slice problem is undecidable then "slice checking" problem is also undecidable.

Note also that the argument, used to prove the undecidability of minimal slice problem, simultaneously proves the undecidability of constant propagation as, in the construction above, determining whether Z after the run of Z `:= false` ; S' is constantly ff would solve the halting problem for S. Constant propagation is known to be undecidable also in context of standard semantics.

6. RELATED WORK

Transfinite semantics have been studied first for functional programming [12]. Paper [6] is a fundamental theoretical work on program slicing in the context of

structured control flow and standard semantics. Besides transfinite semantics used in [5], other ways to handle semantic anomaly exist [4, 13]. It is worth to note that paper [5] almost bypasses the problem of determining program points where control jumps after an infinite loop. In this sense, our work improves their approach. Moreover, we define also the limit states differently from [5]; their treatment could be achieved by replacing s' with s in our definition of ψ in Section 3. In other words, the limit state of [5] depends on all the states observed during the infinite computation while our limit state depends only on the states observed at the top point of the loop. The following example shows the need for this change.



The second program is a slice of the first w.r.t. criterion $\{(3, \mathtt{i})\}$. But in the transfinite semantics of [5], the value of `i` at 3 is $\top$ in the first program but 2 in the second. Hence the essential property of slicing is still not met. Our way to define limit states helps.

The *lazy semantics* of Danicic et al. [13] does not have this problem as they handle the body of a loop as a unit when defining semantics of the loop.

7. CONCLUSIONS

In this paper, we have theoretically studied transfinite semantics in program slicing – the method first used by Giacobazzi and Mastroeni [5]. We may conclude that, at least in simple cases like those considered in this paper, transfinite semantics are appropriate for semantics-based description of program slicing leading to a definition consistent with standard slicing algorithms.

In general case, suitability of transfinite semantics in the form of [5] or of this paper is not so clear. Firstly, recursion is not involved. With recursive procedures, one can obtain a new kind of loops due to infinitely deep recursion which results also in infinitely long call stack. There is no obvious way to define limits of such infinite computations. A promising idea is to replace transfinite semantics based on ordinals with a more general semantics allowing also “backward infinity”. This would enable one to handle escaping infinitely deep recursion in the seemingly most natural way: unloading the infinite call stack level by level starting from infinity.

Secondly, even the usual branching according to a predicate can raise doubts when the value of the predicate happens to be $\top$. To be consistent with the lazy semantics of [13], both branches should be entered and processed independently of each other and, after the end of both computations, the resulting states should be

merged into one. The mathematical structures used in this paper do not enable this. Note, however, that the undecidability proof in Section 5 is valid also in this case.

Another possibility is to count $\top$ equivalent to $f\!f$ in branching, so keeping the semantics in our framework. The suitability of this approach for our aims is unclear. Our work in progress shows the existence of a wide class of deterministic transfinite semantics for which standard slicing algorithms are correct. However, one cannot be sure that all the programs that are intuitively considered as slices while not being producible via standard algorithms are slices w.r.t. any of these semantics.

REFERENCES

1. Weiser, M. Program slicing. *IEEE Trans. Softw. Eng.*, 1984, **10**, 352–357.
2. Tip, F. A survey of program slicing techniques. *J. Program. Lang.*, 1995, **3**, 121–181.
3. Binkley, D. W. and Gallagher, K. B. Program slicing. *Adv. Computers*, 1996, **43**, 1–50.
4. Reps, T. and Turnidge, T. Program specialization via program slicing. In *Proc. Dagstuhl Seminar of Partial Evaluation* (Danvy, O., Glueck, R. and Thiemann, P., eds.). *Lecture Notes in Computer Science*, 1996, **1110**, 409–429.
5. Giacobazzi, R. and Mastroeni, I. Non-standard semantics for program slicing. *Higher-Order Symb. Comput.*, 2003, **16**, 297–339.
6. Reps, T. and Yang, W. The semantics of program slicing and program integration. *Lecture Notes in Computer Science*, 1989, **352**, 360–374.
7. Cousot, P. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Electron. Notes Theor. Comput. Sci.*, 1997, **6**, 25 p.
8. Nestra, H. Transfinite Corecursion. *Nordic J. Comput.* Forthcoming.
9. Moschovakis, Y. N. *Notes on Set Theory*. Undergraduate Texts in Mathematics. Springer-Verlag, New York, 1994.
10. Schütte, K. *Proof Theory*. Grundlehren der matematischen Wissenschaften. Springer-Verlag, Berlin, 1977.
11. Poizat, B. *A Course in Model Theory: an Introduction to Contemporary Mathematical Logic*. Springer-Verlag, New York, 2000.
12. Kennaway, R., Klop, J. W., Sleep, R. and Vries, F.-J. de. Transfinite reductions in orthogonal term rewriting systems. *Inf. Comput.*, 1995, **119**, 18–38.
13. Danicic, S., Harman, M., Howroyd, J. and Ouarbya, L. A lazy semantics for program slicing. In *Proc. 1st International Workshop on Programming Language Interference and Dependence*. http://profs.sci.univr.it/~mastroen/download/PLID/Proceedings/Proceedings.html (2004)

Transfiniitsed semantikad programmide viilutamisel

Härmel Nestra

Artikkel sisaldab teatavate transfiniitsete jälitussemantikate matemaatilise esituse ja on uuritud programmide viilutamist nende kontekstis. On tõestatud mõned üldised faktid viilutamisest, mis kehtivad paljude programmeerimiskeelte ja nende transfiniitsete semantikate kohta. Põhiline teemakäsitlus on arendatud juhtvoograafide jaoks, et abstraheeruda konkreetsetest programmeerimiskeeltest. Juhtvoo struktuursust ei ole eeldatud, kuid arendatud teooria rakendub kõigile standardsetele struktuurse juhtvooga programmeerimiskeeltele.