DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

ITERATIVELY DEFINED TRANSFINITE TRACE SEMANTICS AND PROGRAM SLICING WITH RESPECT TO THEM

HÄRMEL NESTRA

TARTU 2006

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

ITERATIVELY DEFINED TRANSFINITE TRACE SEMANTICS AND PROGRAM SLICING WITH RESPECT TO THEM

HÄRMEL NESTRA

TARTU 2006

Faculty of Mathematics and Computer Science, University of Tartu, Estonia

Dissertation accepted for public defense of the degree of Doctor of Philosophy (PhD) on June 30, 2006 by the Council of the Faculty of Mathematics and Computer Science, University of Tartu.

Supervisor:

PhD, associate professor Varmo Vene
Tartu Ülikool, arvutiteaduse instituut
Tartu, Estonia

Opponents:

PhD, professor Helmut Seidl
Technische Universität München, Institut für Informatik
Munich, Germany

PhD, senior researcher Tarmo Uustalu
Tallinna Tehnikaülikool, Küberneetika Instituut
Tallinn, Estonia

The public defense will take place on Oct 13, 2006.

The publication of this dissertation was financed by Institute of Computer Science, University of Tartu.

Tartu Ülikooli Kirjastuse trükikoda
Tiigi 78, 50410 Tartu
Tellimus nr. 460

CONTENTS

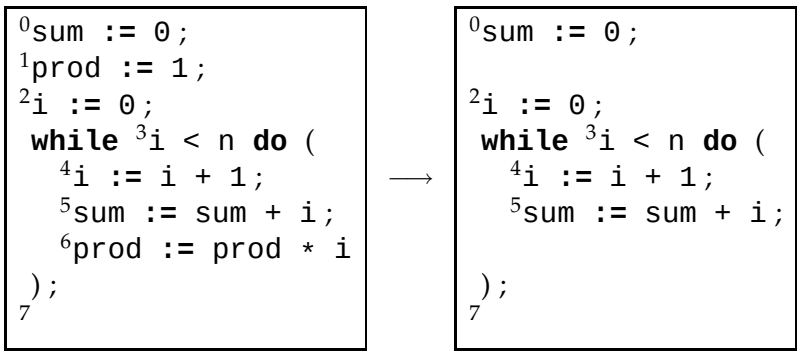
CHAPTER 1

INTRODUCTION

1.1 Program Slicing

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. Program slicing was introduced and its significance was explained first by Weiser [20]; summaries of its techniques and applications can be found in Tip [18] and in Binkley and Gallagher [2].

Example 1.1.1. A standard example of program slicing is the following:



(The small numbers are short notations of program points.) 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 be-

tween program points and variables. The essential property of 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 program point p and variable X related by the criterion and for arbitrary initial values of variables, the sequence of values of X occurring when control of the execution of the original program goes through program point p equals to the sequence of values of X occurring when control of the execution of the slice goes through the program point corresponding to p in the slice.

The slice in Example 1.1.1 has been found with respect to 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 $0, 1, 3, \ldots, \frac{(n-1)n}{2}$ for `sum` at 5. These observations together imply the property also for criterion $\{(5, \texttt{sum}), (7, \texttt{sum})\}$.

If our concern is to prove correctness of slicing algorithms, we need a formalization of the important property. Clearly this must involve a trace semantics $\mathcal{S}$. Assume that $\mathcal{S}$ takes a program and an initial state of variables as arguments and provides a computation trace as value whereby the computation traces are modelled by sequences of configurations, each consisting of the current program point and variable state. Then the straightforward formalization would be as follows. For all programs P and slicing criterions C, a slice of P w.r.t. C is any program Q for which the following holds:

1. Q is obtained from P by deletion of statements.
2. Let $(p, X) \in C$ and $s \in \mathit{State}$. Let $\tilde{p}$ be the program point of Q corresponding to p in P. Then

$$\text{map}(\text{val}\, X)(\text{filter}(\text{at}\, p)(\mathcal{S}(P)(s))) \tag{1.1}$$
$$= \text{map}(\text{val}\, X)(\text{filter}(\text{at}\, \tilde{p})(\mathcal{S}(Q)(s))) \tag{1.2}$$

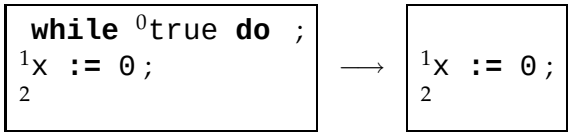
   where

   - $\text{val}\, X\, c$ is the value of variable X in the state of configuration c,
   - $\text{at}\, p$ is the predicate which is true just for configurations with program point p,
   - map and filter are the list functions known from functional programming carried over to traces (i.e. $\text{map}\, f\, l$ applies f to every configuration

of trace l and filter $p\,l$ forms the list of elements of l satisfying p (keeping the order)).

It has been noticed earlier that standard semantics are not completely satisfactory for formalizing the notion of slice 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.

Example 1.1.2. The second program is a slice of the first w.r.t. criterion $\{(2, \mathsf{x})\}$:



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 being reached no times during the run of the original program. □

This phenomenon is called *semantic anomaly* [13, 5]. It is a fundamental issue since no slicing algorithm can decide whether a loop terminates. Therefore nontrivial slicing algorithms, the standard ones based on data flow analysis in particular, cannot be correct w.r.t. standard semantics in all cases. (Reps and Yang [14] 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.

One possibility to handle the semantic anomaly is to modify the definition slightly, allowing Eq. 1.1 to be a prefix of Eq. 1.2 in condition 2 and requiring them to be equal only for cases when both programs terminate [2, 5]. This is also not an ideal solution since, if the original program loops, one can delete any collection of statements from the part of a slice following the infinite loop and the result is a slice again. This makes the notion of slice too wide.

A better solution is to reinterpret the original equation in context of transfinite semantics.

1.2 Transfinite Trace Semantics

Standard semantics consider computations doing at most ω steps (i.e. computations whose any proper initial part is finite). This choice has been artless since no real computation process can ever do more.

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. Transfinite trace semantics of a program is basically a set of transfinite lists of states or configurations satisfying certain conditions. Transfinite list is a function whose domain is a downward closed set of ordinals (i.e. a set O containing all elements less than any element belonging to O).

The first study of transfinite semantics has been done for functional programming, see [7]. The necessity arises from the fact that there are (finite) expressions whose value is an infinite data structure which can not be reached using any standard reduction strategy with the first ω reductions.

Giacobazzi and Mastroeni [5] investigate transfinite semantics with the aim of solving the problem of semantic anomaly; the idea has been proposed already by Cousot [3]. The principle of transfinite semantics is that everything observed in the code should be reflected by the semantics. A loop followed by an assignment in the code should be a loop followed by an assignment also in the semantics, irrespectively of whether the loop terminates. The assignment after an infinite loop being never reached during real processes only shows the deficiency of our implementation and is not a reason for omitting the assignment from the semantics.

In transfinite semantics, changing a nonterminating statement S to a terminating one does not necessarily cause the problem considered above since control can reach the statements following statement S anyway.

In [11], we showed the naturalness of transfinite semantics by expressing both standard and transfinite trace semantics of a simple structured language in a uniform way (more to the point, in fixpoint form) so that the choice between standard and transfinite semantics comes up from the values of a few global parameters of the semantics definition schema.

However, transfinite semantics works well only if there are no recursive procedures. Looping caused by infinitely deep recursion result in infinitely long call stack. There is no obvious way to define limits of such infinite computations. The most natural way to escape from infinitely deep recursion is unloading the infinite call stack level by level starting from infinity. This would require infinitely long backward subsequences of traces which is impossible within transfinite semantics since infinite decreasing sequences of ordinals do not exist.

1.3 Outline and Structure of the Thesis

In this thesis, we find out a class of transfinite semantics w.r.t. which standard slicing algorithms turn out to be correct. This assures transfinite semantics being

a way to overcome “semantic anomaly”. The correctness of slicing of terminating programs w.r.t. standard semantics, proven earlier by Reps and Yang [14], can be deduced as a corollary from the correctness w.r.t. transfinite semantics.

The theory is developed for control flow graphs to keep the treatment abstracted from any concrete programming language. One of the purposes of choosing this approach has been the desire to capture also slicing of unstructured control flow. Therefore, our results hold uniformly for a wide range of deterministic imperative languages without recursion. Programs written in standard programming languages with structured control flow are among those to which our theory applies.

We find that transfinite semantics enable one to prove correctness of slicing via correctness of a cognate transformation where the irrelevant statements are replaced with other irrelevant ones rather than removed. We call this program approximation. As replacing with **`skip`** (a statement doing nothing) is a special case of it, program approximation is roughly a generalization of program slicing. We can treat slicing as a two-step process: first replacing the irrelevant statements with **`skip`**, thereby not affecting the control structure, and then removing the new statements, thereby not affecting the data flow, and study the correctness of these steps separately.

Program approximation can produce termination from nontermination like program slicing but also nontermination from termination. The process of slicing away a loop can consist of replacing all the statements of its body with **`skip`**, so introducing nontermination, followed by removing the loop, abolishing this nontermination. Consequently, this kind of correctness proof would not be possible within the standard (i.e. not transfinite) semantics framework even for terminating programs.

A noteworthy part of the thesis is devoted to studying transfinite semantics in their own. As standard trace semantics are usually defined iteratively, we develop a transfinite counterpart of iteration and investigate properties of transfinite iteration. Our transfinite iteration does not coincide with the widely used transfinite recursion but is a special case of it. The difference is that, for defining a function by transfinite iteration, the iteration step does not have to be given for initial parts of arbitrary length like in traditional recursion but only for some possible lengths (e.g. 1, ω, ω^2 etc.). We define two different variants of transfinite corecursion as analogues to the traditional stream corecursion and find connections between transfinite iteration and transfinite corecursion.

The thesis is structured as follows.

In Chapter 2, we provide a brief introduction to the part of graph theory we need in the thesis. This chapter does not pretend to containing new results.

Chapter 3 contains abstract theory of transfinite trace semantics. It starts with a ba-

sic introduction to ordinals (Section 3.1) followed by a mathematical framework for handling transfinite sequences (Section 3.2). Then, transfinite iteration and two variants of transfinite corecursion are defined and their connections are studied (Sections 3.3–3.4). We prove that, under certain conditions, a given transfinite corecursion schema determines a unique function with transfinite lists as values. So one can define deterministic transfinite trace semantics using these schemata. Furthermore, we prove an analogous theorem for defining non-deterministic transfinite trace semantics and investigate the connections between the corecursions for deterministic and non-deterministic case (Sections 3.5–3.6). Most of the content of Sections 3.1–3.4 has been published in [9, 10], the material of Sections 3.5–3.6 can be found in [9]. Up to the author's knowledge, the content of Sections 3.3–3.6 is fully original. The author also has met no theory of transfinite lists like that developed in Section 3.2 before.

Chapter 4 contains the main contribution of the thesis, i.e. the proof of correctness of two standard slicing algorithms w.r.t. a class of transfinite semantics. In Sect. 4.1, an introduction to the field together with a few examples are given. In Sections 4.2–4.5, the mathematical framework is developed and many auxiliary lemmas are proven. In Sect. 4.6, semantic correctness of program approximation — the first step of the schema described above — is proven. In Sect. 4.7, the second step which we call program simplification is studied. In Sect. 4.8, semantic correctness of program slicing is deduced as a corollary of the semantic correctness of program approximation and program simplification. The two algorithms whose correctness we obtain are also briefly described there. The content of this chapter is mostly unpublished but fragments of Sections 4.1 and 4.3 can be found in [10].

Chapter 5 contains discussion on various related issues. In Sect. 5.1, it is realized that several undecidability results which are widely known have been stated and proved in principle w.r.t. standard semantics and the proofs not necessarily apply to transfinite interpretation. We give proofs for transfinite case. Most of this study can be found in [10]. In Sect. 5.2, we discuss a promising direction of further work where transfinite semantics are replaced by fractional semantics, meaning that items of computation traces are indexed by rational numbers rather than ordinals. This framework overcomes the principal inability of transfinite semantics to model unloading infinitely deep recursion. This approach was introduced in our latest paper [11]. In Sect. 5.3, we point out a common trait appearing in some definitions of operations on transfinite lists given earlier and discuss the nature of it. The other sections refer to related work and conclude.

CHAPTER 2

PRELIMINARIES FROM GRAPH THEORY

This chapter contains definitions of some notions and proofs of some basic facts concerning control flow graphs. It is likely that all this can be found in the literature but we provide this chapter for easy reference of the notation and exact meaning of terms of graph theory used throughout the thesis. All definitions and theorems have been put into words by the author; all proofs have been constructed by the author without using any reference material; also the choice of the notions and facts has been done by the author.

In Sect. 2.1, directed graph is defined (we do not use others) and some basic properties mainly concerning subgraphs and reachability are proved. Section 2.2 studies postdominance order and Sect. 2.3 studies dependence which, in context of control flow graphs, is usually called control dependence. Definitions of postdominance and control dependence in context of control flow graphs and program slicing can be found in Tip [18].

2.1 Directed Graphs

Definition 2.1.1. *A* directed graph *is a triple* $G = (V, E, (\sigma, \tau))$ *where* V *and* E *are sets whose elements are called* vertices *and* arcs*, respectively, and* $\sigma \in E \to V$, $\tau \in E \to V$ *are functions giving the* initial *and* terminal *vertex for any arc, respectively.*

For generality, we do not assume that an arc is a pair of its initial and terminal vertex. This enables us to keep the graphs with multiple arcs under consideration.

Definition 2.1.2. *Let* $G = (V, E, (\sigma, \tau))$ *be any directed graph.*

(i) *A* walk in G *is any sequence* $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$ *where* $e_i \in E$ *for all* $i = 1, \ldots, l$, $v_i \in V$ *for all* $i = 0, \ldots, l$ *and* $\sigma(e_i) = v_{i-1}$, $\tau(e_i) = v_i$ *for all* $i = 1, \ldots, l$. *Thereby, the number* l *is called the* length *of walk* w.

(ii) *For any walk* $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$, *denote* $\sigma(w) = v_0$ *and* $\tau(w) = v_l$ *and call* w *a walk* from v_0 to v_l. *For any* $w \in V$, *say that walk* w passes through w *iff* $w = v_i$ *for some* $i = 1, \ldots, l$; *for any* $d \in E$, *say that walk* w uses d *iff* $d = e_i$ *for some* $i = 1, \ldots, l$.

(iii) *If* $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$ *and* $v = (v_l, e_{l+1}, v_{l+1}, \ldots, e_{l+k}, v_{l+k})$ *are walks in* G *then denote* $wv = (v_0, e_1, v_1, \ldots, e_{l+k}, v_{l+k})$ *— the joined walk.*

(iv) *Let* v, w *be vertices in* G. *If there exists a walk* w *in* G *from* v *to* w *then* w *is called* successor of v *and* v *is called* predecessor of w. *If, thereby, the length of* w *is* 1, *i.e. there exists an arc* e *in* G *such that* $\sigma(e) = v$ *and* $\tau(e) = w$, *then the successor* w *of* v *is called* immediate, *likewise the predecessor* v *of* w *is called* immediate. *The latter situation is denoted by* $v \to w$.

Note that, according to Definition 2.1.2(ii), a walk w always passes through $\tau(w)$ but generally does not pass through $\sigma(w)$.

Proposition 2.1.3. *Let* G *be a directed graph.*

(i) *Let* w *be a walk from* x *to* y *in* G. *Then there exists a walk from* x *to* y *in* G *which does not pass through* x.

(ii) *Let* w, v *be walks such that* $\tau(w) = \sigma(v)$. *If* wv *passes through* x *then either* w *passes through* x *or* v *passes through* x.

Proof.

(i) Let $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$; then $x = v_0$. Let i be the largest integer for which $x = v_i$. Then $(v_i, e_{i+1}, v_{i+1}, \ldots, e_l, v_l)$ is a walk from x to y which does not pass through x.

(ii) Let $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$ and $v = (v_l, e_{l+1}, v_{l+1}, \ldots, e_{l+k}, v_{l+k})$. By assumption, $v_i = x$ for some $i = 1, \ldots, l + k$. If $i > l$ then v passes through x; otherwise, w passes through x. □

Proposition 2.1.4. *Let* $G = (V, E, (\sigma, \tau))$ *be a directed graph. Let* $X \subseteq V$, $A \subseteq E$ *be such that, for every* $e \in E$,

$$e \in A \;\Rightarrow\; \sigma(e) \in X \;\wedge\; \tau(e) \in X \;.$$

Then $(X, A, (\sigma\big|_A, \tau\big|_A))$ *is a directed graph.*

Proof. Straightforward. □

Definition 2.1.5. *Let $G = (V, E, (\sigma, \tau))$ be a directed graph. Let $X \subseteq V$, $A \subseteq E$ such that, for every $e \in E$,*

$$e \in A \;\Rightarrow\; \sigma(e) \in X \;\wedge\; \tau(e) \in X \;.$$

Then the directed graph $(X, A, (\sigma\big|_A, \tau\big|_A))$ is called subgraph of G. *If, for all $e \in E$, also*

$$\sigma(e) \in X \;\wedge\; \tau(e) \in X \;\Rightarrow\; e \in A$$

then this subgraph is called induced.

Proposition 2.1.6. *Let G be a directed graph and H its subgraph. Let x, y be vertices of H. If w is a walk from x to y in H then w is a walk from x to y in G.*

Proof. Straightforward. □

Definition 2.1.7. *Let $G = (V, E, (\sigma, \tau))$ be a directed graph.*

(i) *Let $x \in V$. For every $y \in V$, call y* reachable from *x iff there exists a walk $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$ such that $v_0 = x$ and $v_i = y$ for some $i = 0, \ldots, l$. For every $a \in E$, call a* reachable from *x iff there exists a walk $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$ such that $v_0 = x$ and $e_i = a$ for some $i = 1, \ldots, l$.*

(ii) *Let $S \subseteq V$. For every vertex or arc of G, call it* reachable from *S iff it is reachable from some vertex $x \in S$.*

Proposition 2.1.8. *Let $G = (V, E, (\sigma, \tau))$ be a directed graph. Let $S \subseteq V$. Let X and A be the set of all vertices and arcs, respectively, of G being reachable from S. Then $(X, A, (\sigma\big|_A, \tau\big|_A))$ is an induced subgraph of G.*

Proof. Take $d \in E$ arbitrarily.

If $d \in A$ then there exists a walk $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$ such that $v_0 \in S$ and $d = e_i$ for some $i = 1, \ldots, l$. Obviously $\sigma(d) = v_{i-1} \in X$ and $\tau(d) = v_i \in X$.

If $\sigma(d) \in X$ then there exists a walk $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$ such that $v_0 \in S$ and $\sigma(d) = v_i$ for some $i = 0, \ldots, l$. Then $(v_0, e_1, v_1, \ldots, e_i, v_i)(v_i, d, \tau(d))$ is a walk in G starting from $v_0 \in S$. Consequently, $d \in A$. □

Definition 2.1.9. *Let $G = (V, E, (\sigma, \tau))$ be a directed graph. Let $X \subseteq V$. Denote by $G\big|_S$ the induced subgraph of G consisting of all vertices and arcs of G reachable from S.*

Proposition 2.1.10. *Let $G = (V, E, (\sigma, \tau))$ be a directed graph and $S \subseteq V$. Let w be a walk in G such that $\sigma(w) \in S$. Then w is a walk in $G\big|_S$.*

Proof. Let $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$. By assumption, there is a walk v in G from some $s \in S$ to $\sigma(w) = v_0$. Concatenating w to the end of v, we see that all v_i and e_i are reachable from s. Therefore w is a walk in $G\big|_S$. □

2.2 Postdominance

Definition 2.2.1. *A* flow graph *is any pair (G, f) where $G = (V, E, (\sigma, \tau))$ is a directed graph with both V and E being finite and $f \in V$ is a vertex called* final *being reachable from every vertex in G.*

Definition 2.2.1 is dual to Weiser's definition [20] which required the existence of an *initial* vertex from which there is a walk to every vertex. As we do not need Weiser's variant, we can adopt this notion to our context in this slightly different form.

Definition 2.2.2. *Let (G, f) be a flow graph. For arbitrary vertices v and w, the vertex w is called a* postdominator *of v in G iff every walk from v to f in G passes through w. If w is a postdominator of v then one also says that w* postdominates *v.*

Clearly f postdominates every vertex except f itself.

Proposition 2.2.3. *Let (G, f) be a flow graph with $G = (V, E, (\sigma, \tau))$. Let $S \subseteq V$ be non-empty.*

(i) *Then $(G|_S, f)$ is a flow graph.*

(ii) *For arbitrary vertices x, y of $G|_S$, y postdominates x in $G|_S$ if and only if y postdominates x in G.*

Proof.

(i) By assumptions, there is a vertex $s \in S$ and f is reachable from s in G. Therefore f is a vertex of $G|_S$.
Take arbitrary vertex x of $G|_S$. By assumption, there is a walk w in G from x to f. By Proposition 2.1.10, w is a walk in $G|_S$. Consequently, f is reachable from x in $G|_S$.

(ii) By part (i), f is a vertex of $G|_S$. By Propositions 2.1.6, 2.1.8 and 2.1.10, w is a walk from x to f in G iff w is a walk from x to f in $G|_S$. Hence the claim follows. □

Theorem 2.2.4. *The postdominance relation is a strict order in any flow graph.*

Proof. For antireflexivity, suppose that x postdominates x in some flow graph. By Definition 2.2.1, there is a walk from x to f. By Proposition 2.1.3(i), there exists a walk from x to f which does not pass through x. This contradicts the supposition.
For transitivity, suppose both y postdominating x and z postdominating y. Consider any walk $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$ from x to f. It passes through y as y

postdominates x, so $v_i = y$ for some $i > 0$. Then $(v_i, e_{i+1}, v_{i+1}, \ldots, e_l, v_l)$ is a walk from y to f, so passing through z as z postdominates y, thus $v_j = z$ for some $j > i > 0$. Consequently, w passes through z which implies z postdominating x. □

In the following, let $x < y$ denote that y postdominates x. Let $\leqslant$ denote the corresponding non-strict order (i.e. $x \leqslant y$ means that y postdominates x or $x = y$).

Lemma 2.2.5. *Let x, y, z be vertices in a flow graph.*

(i) *If $x < z$ and $y \not< z$ then every walk from x to y passes through z.*

(ii) *If $x < z$ and there exists a walk from x to y which does not pass through z then $y < z$.*

Proof.

(i) Let f be the final vertex. Since $y \not< z$, there exists a walk w from y to f which does not pass through z. Let v be any walk from x to y. Then vw is a walk from x to f. As z postdominates x, this walk passes through z. By Proposition 2.1.3(ii), either v or w passes through z. Hence v passes through z. Consequently, every walk from x to y passes through z.

(ii) The contrapositive of Lemma 2.2.5(i). □

Theorem 2.2.6. *The vertices postdominating one fixed vertex are linearly ordered w.r.t. $\leqslant$.*

Proof. Suppose that y and z both postdominate x while $y \neq z$. We must prove that $y < z$ or $z < y$. For this, assume $y \not< z$. By Lemma 2.2.5(i), every walk from x to y passes through z.

By Definition 2.2.1, there exists a walk $v = (v_0, e_1, v_1, \ldots, e_l, v_l)$ from x to f. As y postdominates x, it must pass through y. Let i be the least positive integer such that $v_i = y$. By the last paragraph, the walk $(v_0, e_1, v_1, \ldots, e_i, v_i)$ from x to y passes through z, so $v_j = z$ for some $j > 0$, $j < i$ ($j \neq i$ since $y \neq z$).

By construction, the walk $u = (v_0, e_1, v_1, \ldots, e_j, v_j)$ from x to z does not pass through y. So, by Lemma 2.2.5(ii), $z < y$. □

Definition 2.2.7. *Let x and y be vertices of a flow graph with $x < y$. Then postdominator y of x is called* immediate *iff any other postdominator of x postdominates y.*

In other words, y immediately postdominates x iff y is the least element w.r.t. $\leqslant$ in the set of all vertices $z > x$.

Theorem 2.2.8. *In every flow graph (G, f), every vertex except f has the immediate postdominator.*

Proof. By Theorem 2.2.6, the set of all vertices $z > x$ is linearly ordered w.r.t. $\leqslant$. By Definition 2.2.1, this set must be finite and hence it has the least element whenever it is non-empty. Every vertex $x \neq f$ is postdominated by at least f. Thus the claim follows. □

Theorem 2.2.9. *Let w be a walk starting from vertex x in a flow graph. Let both y, z be postdominators of x and assume w passing through z. Then the following are equivalent:*

1. *$y < z$;*
2. *w passes through y and the first occurrence of y in w is before the first occurrence of z.*

Proof. Let $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$ and let i be the least index for which $v_i = z$. Suppose $y < z$. Then $z \not< y$ and $z \neq y$. Lemma 2.2.5(i) gives every walk from x to z passing through y. So $(v_0, e_1, v_1, \ldots, e_i, v_i)$ passes through y before reaching z. This implies statement 2.

Suppose now statement 2. It means $(v_0, e_1, v_1, \ldots, e_l, v_l)$ passing through y before reaching z; thus there is a walk from x to y without passing through z. Lemma 2.2.5(i) gives $y < z$. □

This theorem states that every walk reaches the postdominators of the starting vertex in their postdominance order.

Corollary 2.2.10. *Let x and y be vertices with $x < y$.*

(i) *Let w be a walk from x to y which passes through no vertices postdominating x except y. Then y is the immediate postdominator of x.*

(ii) *If y is an immediate successor of x then it is the immediate postdominator of x.*

Proof.

(i) Suppose z being the immediate postdominator of x. If $z < y$ then Theorem 2.2.9 gives w reaching z before y. This contradicts the assumption about w. Hence $z = y$.

(ii) By assumption, there is an arc d from x to y. So (x, d, y) is a walk from x to y. Thereby, y is the only vertex postdominating x through which this walk passes. Thus Proposition 2.2.10(i) gives the desired result. □

Corollary 2.2.11. *Among the immediate successors of a vertex x, at most one postdominates x.*

Proof. By Corollary 2.2.10(ii), all the immediate successors of x which postdominate x are immediate postdominators of x. By Theorem 2.2.8, x has at most one immediate postdominator. Consequently, there can be at most one postdominator of x among the immediate successors of x. □

2.3 Dependence

This section investigates abstractly the relation known as control dependence.

Definition 2.3.1. *Let (G, f) be a flow graph.*

(i) *Let x and y be arbitrary vertices of G. Then y is said to be* dependent on x *iff $x \not\leqslant y$ and there exists an immediate successor z of x in G such that $z \leqslant y$.*

(ii) *We denote by* dep *the relation in which vertices x and y are if and only if y is dependent on x.*

Theorem 2.3.2. *Let x and y be vertices in a flow graph. Then x dep y iff $x \not\leqslant y$ and there exists a non-empty walk w from x to y such that $w \leqslant y$ for every vertex w through which w passes.*

Proof. Suppose x dep y. Then $x \not\leqslant y$ by Definition 2.3.1. Also, there exists an immediate successor z of x such that $z \leqslant y$. Let d be an arc going from x to z. If $z = y$ then (x, d, y) is a walk satisfying the desired property. So assume that $z < y$. By Definition 2.2.1, there exists a walk $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$ from z to f; it must pass through y. Let i be the least number for which $v_i = y$. We show that the walk $(x, d, z)(v_0, e_1, v_1, \ldots, e_i, v_i)$ from x to y has the desired property. Clearly it is non-empty. Furthermore, for every $j < i$, $j > 0$, Lemma 2.2.5(ii) implies $v_j < y$ since $(v_0, e_1, v_1, \ldots, e_j, v_j)$ is a walk from z to v_j which does not pass through y and $z < y$. This proves the "only if" part.

For the other part, suppose that $x \not< y$ and there exists a non-empty walk $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$ from x to y such that $v_i \leqslant y$ for every $i = 1, \ldots, l$. As w is non-empty, v_1 exists and is an immediate successor of x satisfying $v_1 \leqslant y$. Thus x dep y. □

The criterion for dependence provided by Theorem 2.3.2 is used as definition in [18].

Lemma 2.3.3. *Let x, y be vertices of a flow graph. If x dep y then there exists a walk from x to y which passes through no postdominator of x.*

Proof. Assume $x \operatorname{dep} y$. By Theorem 2.3.2, $x \not< y$ and there exists a walk $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$ from x to y such that $v_i \leqslant y$ for every $i = 1, \ldots, l$. If $x < v_i$ for some $i = 1, \ldots, l$, the transitivity of $\leqslant$ would give a contradiction. So w is the desired walk. □

Theorem 2.3.4. *Let $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$, $l > 0$, be a walk from x to y in a flow graph. If w passes through no vertex postdominating x then there exists an $i < l$ such that $v_i \operatorname{dep} y$ and $v_{i+1} \leqslant y$.*

Proof. Let i be the least number such that $v_{i+1} \leqslant y$. If $i > 0$ then $v_i = v_{(i-1)+1} \not\leqslant y$ implying $v_i \not< y$. If $i = 0$ then $v_i = x \not< y$ by assumption. As $v_i \to v_{i+1}$, it gives $v_i \operatorname{dep} y$. □

We denote by dep^* the reflexive transitive closure of dep and by dep^+ the transitive closure of dep, i.e.,

$$\operatorname{dep}^* = \bigcup_{n \in \mathbb{N}} \operatorname{dep}^n \qquad \text{and} \qquad \operatorname{dep}^+ = \bigcup_{\substack{n \in \mathbb{N} \\ n > 0}} \operatorname{dep}^n$$

where dep^n denotes the n-times composition of dep.

Theorem 2.3.5. *Let x and y be vertices in a flow graph. Then $x \operatorname{dep}^* y$ iff there exists a walk from x to y which passes through no postdominator of x.*

Proof. Assume $x \operatorname{dep}^* y$. Then there exists a chain $w_0 \operatorname{dep} w_1 \operatorname{dep} \ldots \operatorname{dep} w_k$ where $w_0 = x$, $w_k = y$. Proceed by induction on k. In the case $k = 0$, the claim holds trivially (take the empty walk from x to x). Assume the claim holding for k and consider a chain with length $k + 1$. By induction hypothesis, there exists a walk w from x to w_k which passes through no postdominator of x. By Lemma 2.3.3, there exists a walk v from w_k to y which passes through no postdominator of w_k. Suppose the walk wv passes through some postdominator z of x. Then v passes through z and so $w_k \not< z$. Lemma 2.2.5(i) now states that every walk from x to w_k passes through z. This leads to a contradiction since w does not pass through z. Consequently, wv is a walk with the desired property.

For the other direction of the equivalence, assume that there exists a walk $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$ from x to y passing through no postdominator of x. Argue by induction on l. If $l = 0$ then $x = y$, so $x \operatorname{dep}^0 y$. Assume now $l > 0$ and the claim holding for naturals less than l. By Theorem 2.3.4, there exists an $i < l$ such that $v_i \operatorname{dep} y$. By the induction hypothesis, $x \operatorname{dep}^* v_i$. Altogether, $x \operatorname{dep}^+ y$. □

Proposition 2.3.6.

(i) *Any immediate successor of a vertex x in a flow graph either postdominates x or is dependent on x.*

(ii) *For any vertex x in a flow graph, at most one of the immediate successors of x is not dependent on x.*

Proof.

(i) Let y be any immediate successor of x. Suppose y is not dependent on x. By Definition 2.3.1, either $x < y$ or $z \leqslant y$ for no immediate successor z of x. As y is an immediate successor of x and $y \leqslant y$, the latter is not the case. Consequently, $x < y$.

(ii) By Corollary 2.2.11, at most one of the immediate successors of x postdominates x. By Proposition 2.3.6(i), all the others are dependent on x. □

Definition 2.3.7. *Let $A = ((V, E, (\sigma, \tau)), f)$ be a flow graph. Call a set $S \subseteq V$* dependence system of A *iff both following conditions hold:*

1. $f \in S$;
2. *for every $x, y \in V$, if $y \in S$ and $x \operatorname{dep} y$ then $x \in S$.*

Theorem 2.3.8. *Let $A = (G, f)$ be a flow graph. Let S be a dependence system of A. Let w be a walk from x to y such that $x \notin S$ and the only vertex of S passed through by w is y. Then $x < y$.*

Proof. Let $w = (v_0, e_1, v_1, \ldots, e_l, v_l)$ and let i be the least index for which $v_i = y$.
Suppose the contrary, i.e. $x \not< y$. As $x \notin S$ and $y \in S$, we have $x \neq y$, so $x \nleqslant y$. Let $j < i$ be the largest index for which $x \leqslant v_j$. By transitivity of $\leqslant$, the vertices v_k for $j < k \leqslant i$ do not postdominate v_j. By Theorem 2.3.5, $v_j \operatorname{dep}^* y$. By Definition 2.3.7, we get $v_j \in S$ which contradicts the choice of i. Hence the claim follows. □

CHAPTER 3

THEORY OF TRANSFINITE TRACE SEMANTICS

A trace semantics of a program expresses its execution behaviour step by step. It is basically a set of sequences of elements representing execution states. In standard trace semantics, the sequences are finite lists or streams; their components therefore correspond to natural numbers. In the case of transfinite trace semantics, the sequences are transfinite, i.e. the components correspond to ordinal numbers. We call them *transfinite lists*.

3.1 Ordinal numbers

In this section, we give a short introduction to ordinals. The definitions and facts listed here are generally those we need in this thesis. There are many books giving profound introductions to ordinal theory; [8, 15] represent just two different approaches.

The notion of ordinal is obtained as a generalization of the notion of natural number by adding infinite elements. So we have all the natural numbers $0, 1, 2, \ldots$, as well as ω and a lot of greater elements, among ordinals. This notion differs from the notion of cardinal in that ordinals can be distinguished by the order of elements in set while cardinals express only the size.

Being precise, an *ordinal* is an isomorphism class of well-ordered sets. (A *well-ordered set* is an ordered set whose every non-empty subset has a least element.) As all the well-orders of a fixed finite set are isomorphic, there exists just one ordinal for any size of a finite set. For countable sets, for instance, there are many (actually uncountably many) in principle different well-orders. The standard order of natural numbers (representing ω) is among them; one of the others is the order of $\mathbb{N} \cup \{\infty\}$ where ∞ is greater than any natural number.

There is a natural order $\leqslant$ on ordinals: $o \leqslant \pi$ iff, for any well-ordered sets A and B corresponding to o and π, respectively, A is isomorphic to a cut of B. (A *cut* of a well-ordered set C is a subset D of it containing all elements of C being less than any given element of D. Cuts are also called *downward closed* subsets.) Every set of ordinals is well-ordered w.r.t. $\leqslant$. Clearly 0 is less than any other ordinal. The set of all ordinals less than o is denoted by $\mathbb{O}_o$; it turns out that $(\mathbb{O}_o;\ \leqslant)$ is a representative of o.

Any ordinal o has a unique immediate successor w.r.t. $\leqslant$; we denote it o'. If A is a well-ordered set representing o, a set representing o' is obtained by adding a new greatest element to A.

If an ordinal has an immediate predecessor, i.e. if it is a successor of some ordinal, it is called *successor ordinal*. Otherwise it is called *limit ordinal*. However, 0 is often considered as a neither successor nor limit ordinal.

For example, all naturals but 0 are successor ordinals while ω — the least infinite ordinal — is a limit ordinal (the least greater than 0). Then there are countably many successor ordinals ω', ω'' etc., followed by the next limit ordinal which of course is followed by countably many successor ordinals.

Let o, π be ordinals. Let A, B be some well-ordered sets representing o and π, respectively. An ordinal ϱ is called *sum* of o and π iff it corresponds to the well-ordered set obtained from A and B by finding their disjoint union and considering every element of A less than every element of B. The sum of o and π is denoted by $o + \pi$. Obviously $o' = o + 1$ for any ordinal o. The operation $+$ is associative and, for any o, $o+0 = 0+o = o$. For ordinals o and π, $o \leqslant \pi$ iff $o+\varrho = \pi$ for some ordinal ϱ. If $o \leqslant o'$ and $\pi \leqslant \pi'$ then always $o + \pi \leqslant o' + \pi'$ (addition is monotone w.r.t. both its arguments).

For example, $\omega + 1$ corresponds to the well-order of $\mathbb{N} \cup \{\infty\}$ introduced before. The ordinal $\omega + \omega$ corresponds to the limit of the sequence $\omega, \omega', \omega'', \ldots$, being the least limit ordinal greater than ω. We can construct the infinite sequence $\omega, \omega + \omega, \omega + \omega + \omega, \ldots$ of limit ordinals. There exists a limit of this sequence, followed by its successor etc. Ordinals form a "very infinite" biome in the sense that no set of ordinals can ever be complete.

If $o + \pi = o + \varrho$ then always $\pi = \varrho$. This allows to define *subtraction* of ordinals. If $o \leqslant \pi$ then $\pi - o$ is the ordinal ϱ such that $o + \varrho = \pi$.

In the following, we assume the reader having a solid knowledge on elementary ordinal theory.

3.2 Transfinite Lists

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 $l \in \mathbb{O}_o \to A$ for some 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 $\mathrm{head}\, l$. All operations considered in the theory are strict, i.e. a subexpression with value $\bot$ turns the value of the whole expression 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 preceding elements. For the case of semantics, this is unnecessarily general. In a deterministic standard trace semantics, every execution state is completely determined by its single predecessor and carrying all preceding states along in the definition could be burdening or misguiding. In other words, semantics are defined by iteration which is a special case of recursion.

The desire to express every computation state in terms of the previous one could be called "locality principle". It requires the behaviour of every atomic statement not depending on the computation occurred before reaching this statement, i.e. all information for performing the computation step being encoded locally in the last state. In transfinite case, defining every element in terms of its single predecessor is generally impossible since if the number of preceding states is a limit ordinal then there is no last element among them. Analogously to the atomic step case, the locality principle now requires that limit state reached due to an infinite loop should be determined by the computation during this loop only. We would like to have a transfinite iteration schema generalizing the usual iteration and respecting the locality principle.

Our main results are proven for semantics where the limit state does not depend on the exact place where we start counting the final part of the endless computation. This restriction is natural as the rest of the thesis shows. (Note that this final part definitely containes an infinite repetition of the body of the loop causing the endless computation but a finite number of first runs of the body possibly have remained outside.) In this case, the locality principle equivalently demands that every state during a computation is determined by a proper final part of the computation performed so far which is as short as it is possible to extract. This length is determined solely by the ordinal index of the component being defined.

For example, if one is defining $l(\omega)$ then ω elements backward must be taken into account. In defining $l(\omega+k)$ for a positive natural number k, it suffices to consider

the last element only. But when defining $l(\omega + \omega)$, there is no last element again; ω elements backward must be studied.

This consideration leads to our notion of selfish ordinal. In [15], these ordinals are called *additive principal numbers*; we like our shorter term more.

Definition 3.2.1. *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). Definition 3.2.1 implies that the lengths of the possible final parts having to be considered backwards when defining a new element of a transfinite list by recursion are precisely the selfish ordinals.

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.

The ordinals $\omega + \omega$, $\omega + \omega + \omega$ etc. are not selfish because removing the initial ω leads to a smaller number. However, the limit ω^2 of this sequence is selfish. Similarly, the limit ω^3 of the sequence $\omega^2, \omega^2 + \omega^2, \omega^2 + \omega^2 + \omega^2, \ldots$ is selfish. This observation can be continued infinitely. We obtain an infinite sequence $\omega, \omega^2, \omega^3, \ldots$ of selfish ordinals. The limit of this sequence is ω^ω which is also selfish. Now we can construct the sequence $\omega^\omega, \omega^\omega + \omega^\omega, \omega^\omega + \omega^\omega + \omega^\omega, \ldots$ whose limit is $\omega^{\omega+1}$, again selfish.

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

Proposition 3.2.2.

(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 $\beta < o$.*

Proof.

(i) If o is selfish, the representation $o = 0 + o$ obviously meets the requirements. Suppose o not being selfish. Then o can be represented as the sum of two ordinals both less than o. For any such representation, both ordinals are non-zero because otherwise the other would equal to o. Let γ be the least non-zero ordinal for which $o = \alpha + \gamma$ is possible. If $\gamma = \pi + \varrho$ for some π and ϱ both less than γ then

$o = \alpha + (\pi + \varrho) = (\alpha + \pi) + \varrho$ contradicts the choice of γ. Thus γ is selfish. Minimize α for this γ. We state that this results in the desired representation. For that, it suffices to prove that $o - \alpha$, even for varying α, can evaluate to at most one selfish ordinal.

Assume $o = \alpha_1 + \gamma_1 = \alpha_2 + \gamma_2$ with selfish γ_1, γ_2. W.l.o.g., $\gamma_1 \leqslant \gamma_2$. Suppose $\gamma_1 < \gamma_2$. It is easy to see that $\alpha_1 > \alpha_2$ (supposing $\alpha_1 \leqslant \alpha_2$ would give $o = \alpha_1 + \gamma_1 \leqslant \alpha_2 + \gamma_1 < \alpha_2 + \gamma_2 = o$, a contradiction). So $\alpha_1 = \alpha_2 + \varepsilon$ for some ε. Now $\alpha_2 + \varepsilon + \gamma_1 = o = \alpha_2 + \gamma_2$ implying $\varepsilon + \gamma_1 = \gamma_2$, a contradiction with selfishness of γ_1 and γ_2. Hence $\gamma_1 = \gamma_2$ and the result follows.

(ii) Let λ be the least ordinal for which $o = \lambda + \beta$ for some $\beta < o$; then $\lambda > 0$. If $\lambda = \pi + \varrho$ for some π and ϱ both less than λ then $o = (\pi + \varrho) + \beta = \pi + (\varrho + \beta)$. Then $\varrho + \beta = o$ by the choice of λ which gives a contradiction with the choice of λ. Hence λ is selfish.

Now assume $o = \lambda_1 + \beta_1 = \lambda_2 + \beta_2$ with selfish λ_1, λ_2 and $\beta_1 < o$, $\beta_2 < o$. W.l.o.g., $\lambda_1 \leqslant \lambda_2$. Suppose $\lambda_1 < \lambda_2$. Then $\lambda_2 = \lambda_1 + \delta$ for some δ; actually, $\delta = \lambda_2$ since λ_2 is selfish. Now $\lambda_1 + \beta_1 = o = \lambda_1 + \lambda_2 + \beta_2$ giving $\beta_1 = \lambda_2 + \beta_2 = o$ which contradicts $\beta_1 < o$. So $\lambda_1 = \lambda_2$ implying also $\beta_1 = \beta_2$. □

Proposition 3.2.2 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* [12, 15].

In the rest, we call the representation $o = \alpha + \gamma$ where γ being selfish and α minimized (the representation of Prop. 3.2.2(i)) the *principal representation* of o. 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 $\omega \cdot (k - 1) + \omega$. If the Cantor normal form of o is written as a sum of powers of ω like in [15] 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.

Principal representations classify ordinals according to the second summand: successor ordinals are "1-ordinals" while ω, $\omega + \omega$ etc. are "ω-ordinals", ω^2 is "ω^2-ordinal" etc.

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. Even if $l(o)$ is represented in terms of this selfish number of preceding elements, the length of the remaining initial part

not necessarily coincides with the other number in the principal representation; the length can be larger. However, the principal representation provides a way to formalize uniformly the kind of recursion we desire.

There is no set of all ordinals and hence also no set of all transfinite lists over a non-empty set. Let $\propto$ be a fixed selfish ordinal "large enough" 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 $\propto$ cannot be continued). So

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

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 (because of its domain not being a cut) then $\mathrm{take}\,o\,l = \bot = \mathrm{drop}\,o\,l$.

Lemma 3.2.3. *Let l be any transfinite list.*

(i) *For ordinals o and π, $l(o + \pi) = (\mathrm{drop}\,o\,l)(\pi)$.*

(ii) *For ordinal o, $l(o) = \mathrm{head}(\mathrm{drop}\,o\,l)$.*

(iii) *For ordinals o and π, $\mathrm{drop}(o + \pi)\,l = \mathrm{drop}\,\pi(\mathrm{drop}\,o\,l)$.*

(iv) *For ordinals o and π, $\mathrm{take}\,\pi(\mathrm{drop}\,o\,l) = \mathrm{drop}\,o(\mathrm{take}(o + \pi)\,l)$.*

(v) *For ordinals o and π, if $\pi \leqslant o \leqslant |l|$ then $\mathrm{take}\,\pi\,l = \mathrm{take}\,\pi(\mathrm{take}\,o\,l)$.*

Proof.

(i) Trivial because if $o + \pi > |l|$ then $l(o + \pi) = \bot = \mathrm{drop}\,o\,l(\pi)$.

(ii) We have $\mathrm{head}(\mathrm{drop}\,o\,l) = (\mathrm{drop}\,o\,l)(0) = l(o + 0) = l(o)$.

(iii) For any ordinal α,

$$\begin{aligned} (\mathrm{drop}\,\pi(\mathrm{drop}\,o\,l))(\alpha) = (\mathrm{drop}\,o\,l)(\pi + \alpha) &= l(o + \pi + \alpha) \\ &= (\mathrm{drop}(o + \pi)\,l)(\alpha) \ . \end{aligned}$$

(iv) If $o + \pi > |l|$, both sides of the desired equality are $\bot$. Otherwise, both are

defined giving, for any ordinal α,

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

(v) For any ordinal α,

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

□

The claims of Lemma 3.2.3 are rather intuitive and we are going to use them without any reference.

For arbitrary $l, k \in \mathrm{TList}\, A$, let $l +\!\!+ k$ denote the transfinite list which is obtained by concatenating k to the end of l. So, for any ordinal π,

$$(l +\!\!+ k)(\pi) = \left\{\begin{array}{ll} l(\pi) & \text{if } \pi < |l| \\ k(\pi - |l|) & \text{otherwise}\end{array}\right\}\ .$$

Thereby, $|l +\!\!+ k| = |l| + |k|$.

Lemma 3.2.4. *Let l, k be transfinite lists.*

(i) $l = k$ *iff* $\mathrm{take}\, o\, l = \mathrm{take}\, o\, k$ *and* $\mathrm{drop}\, o\, l = \mathrm{drop}\, o\, k$ *for some ordinal o with $o \leqslant |l|$, $o \leqslant |k|$.*

(ii) $\mathrm{take}\, |l|(l +\!\!+ k) = l$ *and* $\mathrm{drop}\, |l|(l +\!\!+ k) = k$.

Proof.

(i) Consider the "if" part (the other is trivial). Suppose $o \leqslant |l|$, $o \leqslant |k|$ and $\mathrm{take}\, o\, l = \mathrm{take}\, o\, k$, $\mathrm{drop}\, o\, l = \mathrm{drop}\, o\, k$. Take any ordinal α. If $\alpha < o$ then

$$l(\alpha) = (\mathrm{take}\, o\, l)(\alpha) = (\mathrm{take}\, o\, k)(\alpha) = k(\alpha)\ .$$

If $\alpha \geqslant o$ then

$$l(\alpha) = (\mathrm{drop}\, o\, l)(\alpha - o) = (\mathrm{drop}\, o\, k)(\alpha - o) = k(\alpha)\ .$$

Hence $l = k$.

(ii) Let α be any ordinal. If $\alpha < |l|$ then

$$(\text{take}\,|l|(l +\!\!+ k))(\alpha) = (l +\!\!+ k)(\alpha) = l(\alpha)\ ,$$

otherwise both sides of the desired equality are $\perp$. If $\alpha < |k|$ then

$$(\text{drop}\,|l|(l +\!\!+ k))(\alpha) = (l +\!\!+ k)(|l| + \alpha) = k(|l| + \alpha - |l|) = k(\alpha)\ ,$$

otherwise both sides of the desired equality are $\perp$. □

Let $\mathbb{T} = \{\text{tt}, \text{ff}\}$ be the set of truth values. We will denote by $\text{map} \in (A \to B) \to (\text{TList}\,A \to \text{TList}\,B)$ and $\text{filter} \in (A \to \mathbb{T}) \to (\text{TList}\,A \to \text{TList}\,A)$ the transfinite counterparts of the namesake functions known from functional programming. More precisely, if $f \in A \to B$ while $l \in \text{TList}\,A$ then $|\,\text{map}\,f\,l| = |l|$ and $(\text{map}\,f\,l)(o) = f(l(o))$ for every $o < |l|$. If $p \in A \to \mathbb{T}$ while $l \in \text{TList}\,A$ then $|\,\text{filter}\,p\,l| = \kappa$ and $(\text{filter}\,p\,l)(\varrho) = l(o_\varrho)$ for all $\varrho < \kappa$ where $(o_\varrho : \varrho < \kappa)$ is the ascending family of all indices corresponding to components of l satisfying p. Denote function composition by ; (function in the left is applied first). Lemma 3.2.5 states properties of map and filter which will be used later. They are easy to prove and, in the case of finite lists and streams, also well known.

Lemma 3.2.5. *Let A, B, C be sets.*

(i) *If $f \in A \to B$ and $g \in B \to C$ then* $\text{map}\,f\,;\,\text{map}\,g = \text{map}(f\,;\,g)$.

(ii) *If $p, q \in A \to \mathbb{T}$ then* $\text{filter}\,p\,;\,\text{filter}\,q = \text{filter}(\lambda a.\, p(a)\ \wedge\ q(a))$.

(iii) *If $p \in A \to \mathbb{T}$ and $l \in \text{TList}\,A$ such that p is constantly true on components of l then* $\text{filter}\,p\,l = l$.

(iv) *If $f \in A \to B$ and $p \in B \to \mathbb{T}$ then* $\text{map}\,f\ ;\ \text{filter}\,p = \text{filter}(f\ ;\ p)\ ;\ \text{map}\,f$.

(v) *If $p \in A \to \mathbb{T}$ and $l \in \text{TList}\,A$ then* $|\,\text{filter}\,p\,l| \leqslant |l|$.

3.3 Transfinite Iteration

Transfinite iteration based on principal representations is defined as follows.

Definition 3.3.1. *Let X, A be sets. Assume $\varphi \in X \to 1 + A = A \cup \{\perp\}$ and $\psi \in \text{STList}\,A \to X$. We say that a function $h \in X \to \text{TList}\,A$ is* iterative on φ and ψ *iff, for each $x \in X$, the following two conditions hold:*

1. $h(x)(0) = \varphi(x)$*;*
2. $h(x)(o) = \varphi(\psi(\text{take}\,\gamma(\text{drop}\,\alpha(h(x)))))$ *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.

Call a transfinite list l with limit ordinal length *stabilizing to* v iff there is an $o < |l|$ such that $l(\pi) = v$ for every π satisfying $o \leqslant \pi < |l|$. Note that l is stabilizing to v iff $\operatorname{drop}\alpha\, l$ is stabilizing to v for every $\alpha < |l|$.

Example 3.3.2. Take $A = \mathbb{N}$, $X = \mathbb{Z}$. For $x \in \mathbb{Z}$ and $l \in \operatorname{STList}\mathbb{N}$, define

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

Then

$$h(x) = \left\{ \begin{array}{ll} (\underbrace{x, x, \ldots}_{\omega}, \underbrace{x+1, x+1, \ldots}_{\omega}, \underbrace{x+2, x+2, \ldots}_{\omega}, \ldots\ldots) & \text{if } x \in \mathbb{N} \\ \underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\omega} & \\ \text{nil} & \text{otherwise} \end{array} \right\}$$

is iterative on φ and ψ (provided $\propto \geqslant \omega^2$). □

Theorem 3.3.3. *Let X, A be sets. For every $\varphi \in X \to 1 + A = A \cup \{\bot\}$ and $\psi \in \operatorname{STList} A \to X$, there exists a unique function $h \in X \to \operatorname{TList} A$ being iterative on φ and ψ.*

Proof. The conditions in Definition 3.3.1 serve as transfinite recursion schema since $h(x)(o)$ is expressed in terms of values of function $h(x)$ on arguments less than $\alpha + \gamma = o$ only. Hence there exists a unique $h \in X \to \mathbb{O}_{\propto} \to A \cup \{\bot\}$ satisfying these conditions.

It remains to make clear that $h(x) \in \operatorname{TList} A$ for every x. Let $h(x)(\pi) = \bot$ and $\pi < o$. Since $o > 0$, there is a principal representation $o = \alpha + \gamma$. As

$$\operatorname{take}\gamma(\operatorname{drop}\alpha(h(x))) = \operatorname{drop}\alpha(\operatorname{take} o(h(x))) = \bot \ ,$$

we get $h(x)(o) = \bot$ by the definition of h. Thus the domain of $h(x)$ is a cut and $h(x) \in \operatorname{TList} A$. □

Theorem 3.3.3 asserts that, for defining a transfinite semantics "by iteration", it suffices to provide just φ and ψ.

3.4 Transfinite Corecursion

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, it is not relevant whether there was any computation before reaching s or was it the initial state. For transfinite semantics, even if defined by transfinite iteration, this property need not hold.

However, there exists a similar weaker condition holding also for iterative transfinite semantics. Furthermore, it is possible to put 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 choosed such word because the conditions are to some extent analogous to traditional stream corecursion (the analogy will be explained below).

Definition 3.4.1. *Let X, A be sets.*

(i) *If $\psi \in \mathrm{STList}\, A \to X$ is such that $\psi(l) = \psi(\mathrm{drop}\, \lambda\, l)$ for all selfish ordinals λ, γ, $\lambda < \gamma < \propto$, and $l \in \mathrm{TList}\, A$ with $|l| = \gamma$, then we call ψ* limit operator.

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

1. *if $\varphi(x) = a \in A$ then $\mathrm{head}(h(x)) = a$, and*
   if $\varphi(x) \in 1$ 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 $h \in X \to \mathrm{TList}\, A$ is weakly corecursive on φ and ψ *iff the conditions 1 and 2 hold. We say that $h \in X \to \mathrm{TList}\, A$ 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, into which the computation falls after an infinite computation , does not depend on the actual starting point

of the final part of selfish length. This implies that one need not use the principal representation to determine the final part to rely on but may equivalently use any final part of the same length (as every ordinal has a Cantor normal form, every function of form $\mathrm{drop}\,\alpha$ can be expressed as a finite composition of functions of form $\mathrm{drop}\,\lambda$ with selfish λ). Example 4.6.4 will show the inevitability of it in context of our approach. In Sect. 4.1, we will provide also examples of deriving transfinite semantics for programs with unstructured control flow where ψ need not be a limit operator.

Condition 3 of Definition 3.4.1(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.

Like recursion, corecursion in general is a special way to define functions. When one has to define functions whose values are streams, corecursion is often the neatest choice.

Consider an example from number theory of defining continued fractions. The function c which takes real numbers to their representations as continued fractions satisfies the corecurrent equation

$$c(x) = \lfloor x \rfloor : \left\{ \begin{array}{ll} c\left(\frac{1}{\langle x \rangle}\right) & \text{if } x \notin \mathbb{Z} \\ \mathrm{nil} & \text{otherwise} \end{array} \right\}$$

where : and nil are the cons and empty-list constructor, respectively, and $\lfloor x \rfloor$, $\langle x \rangle$ denote the integral and fractional part of x, respectively. Obviously, this equation determines the function c uniquely despite containing c on the right-hand side. In number theory books, continued fractions are usually defined recursively rather than corecursively, resulting in a more complicated formulation because the length of the result is not known a priori, it clears up as the computation reaches the end.

The soundness of both recursive and corecursive definitions in the case of their simplest forms is obvious; however, proving it is surprisingly hard. There are works trying to give proofs of corecursion theorems at a very general level; the reader being interested in is recommended to study [1].

When defining a standard trace semantics of a programming language, one usually gives a plenty of elementary transition rules to be used for different syntactic constructions. In other words, one has a transition function or, if a non-deterministic semantics is desired, a transition relation. It is trivial to argue by stream coinduction that a transition function gives rise to a unique deterministic semantics, i.e. it determines the unique function from initial states to streams of states such that the first state of the result is the initial state and any following state is obtained from its predecessor by applying the transition function. In the case of non-determinism, the circumstances are similar.

The corecursion theorems being proved in this chapter play a similar role in the case of transfinite trace semantics. In this case, one has to define transfinite lists rather than usual streams. While the corecurrent equations of usual kind relate a list with its tail, tail of tail etc. only, the form of corecurrent equations must enable to relate transfinite lists with their arbitrarily (transfinitely) deep substructures.

Usual corecursion is called corecursion because it is dual to recursion [6]. We have not found such kind of connection between transfinite recursion and our transfinite corecursions. The name "transfinite corecursion" has been choosed solely by the analogy with usual corecursion where one writes equations relating a (possibly infinite) structure with its substructures which are expressed as values of the same function.

Taking $\lambda = 1$ in Definition 3.4.1(ii) 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.

Theorem 3.4.2(i) states the equivalence of iterativity and weak corecursivity.

Theorem 3.4.2. *Let X, A be sets. Let $\varphi \in X \to A \cup \{\bot\} = 1 + A$, $\psi \in \mathrm{STList}\, A \to X$ and $h \in X \to \mathrm{TList}\, A$.*

(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) Consider the "only if" part first. Let h be iterative on φ and ψ.

We have $\mathrm{head}(h(x)) = h(x)(0) = \varphi(x)$; thereby, if $\varphi(x) \notin A$ then $h(x)(0) = \bot$ implying $h(x) = \mathrm{nil}$. Thus condition 1 of Definition 3.4.1(ii) holds.

It remains to prove condition 2. Take x and λ, μ consecutive selfish ordinals such that $\lambda < \mu \leqslant \propto$ and $|h(x)| \geqslant \lambda$. We are going to show by transfinite induction that $h(x)(\lambda + o) = h(\psi(\mathrm{take}\, \lambda(h(x))))(o)$ for every ordinal $o < \mu$. Assume the equality being valid for all ordinals $\pi < o$, so $\mathrm{take}\, o(\mathrm{drop}\, \lambda(h(x))) = \mathrm{take}\, o(h(\psi(\mathrm{take}\, \lambda(h(x)))))$.

If $o = 0$, we get

$$\begin{aligned} h(x)(\lambda + 0) &= h(x)(0 + \lambda) = \varphi(\psi(\mathrm{take}\, \lambda(\mathrm{drop}\, 0(h(x))))) \\ &= \varphi(\psi(\mathrm{take}\, \lambda(h(x)))) = h(\psi(\mathrm{take}\, \lambda(h(x))))(0) \quad . \end{aligned}$$

If $o > 0$, let $o = \alpha + \gamma$ be the principal representation. Then $\gamma \leqslant o < \mu$, implying $\gamma \leqslant \lambda$. Using the induction hypothesis, we get

$$\begin{aligned}\mathrm{take}\,\gamma(\mathrm{drop}(\lambda+\alpha)(h(x))) &= \mathrm{take}\,\gamma(\mathrm{drop}\,\alpha(\mathrm{drop}\,\lambda(h(x))))\\ &= \mathrm{drop}\,\alpha(\mathrm{take}\,o(\mathrm{drop}\,\lambda(h(x))))\\ &= \mathrm{drop}\,\alpha(\mathrm{take}\,o(h(\psi(\mathrm{take}\,\lambda(h(x))))))\\ &= \mathrm{take}\,\gamma(\mathrm{drop}\,\alpha(h(\psi(\mathrm{take}\,\lambda(h(x))))))\ .\end{aligned}$$

By Lemma 3.4.5(i), $(\lambda + \alpha) + \gamma$ is the principal representation of $\lambda + o$. Thus

$$\begin{aligned}h(x)(\lambda+o) &= \varphi(\psi(\mathrm{take}\,\gamma(\mathrm{drop}(\lambda+\alpha)(h(x)))))\\ &= \varphi(\psi(\mathrm{take}\,\gamma(\mathrm{drop}\,\alpha(h(\psi(\mathrm{take}\,\lambda(h(x))))))))\\ &= h(\psi(\mathrm{take}\,\lambda(h(x))))(o)\ .\end{aligned}$$

Thus h is weakly corecursive on φ and ψ.

For the "if" part, suppose h being weakly corecursive on φ and ψ. Let $\tilde{h}$ be the function iterative on φ and ψ. It suffices to show $h = \tilde{h}$.

By the "only if" part, $\tilde{h}$ is weakly corecursive on φ and ψ. We are going to prove that

$$\forall x \in X\ (h(x)(o) = \tilde{h}(x)(o)) \tag{3.1}$$

for every $o < \propto$. Argue by transfinite induction. For $o = 0$, one obtains

$$h(x)(0) = \mathrm{head}(h(x)) = \varphi(x) = \mathrm{head}(\tilde{h}(x)) = \tilde{h}(x)(0)\ .$$

Consider now any non-zero $o < \propto$ and assume Eq. 3.1 for all $\pi < o$. Take $o = \lambda + \beta$ with λ selfish and $\beta < o$. The induction hypothesis implies $|h(x)| \geqslant \lambda \iff |\tilde{h}(x)| \geqslant \lambda$, as well as $\mathrm{take}\,\lambda(h(x)) = \mathrm{take}\,\lambda(\tilde{h}(x))$, as well as $h(y)(\beta) = \tilde{h}(y)(\beta)$ for every $y \in X$. Using these, we get

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

(ii) Our h is weakly corecursive by part (i). It remains to prove condition 3 from Definition 3.4.1(ii). Prove by transfinite induction on o that

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

(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 3.2.2(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

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

□

Theorem 3.4.2(ii) can be proven also without the reference to Theorem 3.4.2(i), simply supplementing the proof of the "only if" part of the latter with the case $o \geqslant \mu$. It suffices to consider the case $\alpha = 0$ together with $\lambda < \gamma$ (this is the case in which the argumentation given in the proof fails). In this case, using the assumption that ψ is a limit operator together with the induction hypothesis, we would get

$$\begin{aligned}
\psi(\mathrm{take}\,\gamma(\mathrm{drop}\,0(h(x)))) &= \psi(\mathrm{take}\,\gamma(h(x)))\\
&= \psi(\mathrm{drop}\,\lambda(\mathrm{take}\,\gamma(h(x))))\\
&= \psi(\mathrm{drop}\,\lambda(\mathrm{take}(\lambda+\gamma)(h(x))))\\
&= \psi(\mathrm{take}\,\gamma(\mathrm{drop}\,\lambda(h(x))))\\
&= \psi(\mathrm{take}\,\gamma(h(\psi(\mathrm{take}\,\lambda(h(x))))))\\
&= \psi(\mathrm{take}\,\gamma(\mathrm{drop}\,0(h(\psi(\mathrm{take}\,\lambda(h(x))))))\ .
\end{aligned}$$

Note that $0 + \gamma$ is the principal representation of both o and $\lambda + o$. Thus

$$\begin{aligned}
h(x)(\lambda+o) &= \varphi(\psi(\mathrm{take}\,\gamma(\mathrm{drop}\,0(h(x)))))\\
&= \varphi(\psi(\mathrm{take}\,\gamma(\mathrm{drop}\,0(h(\psi(\mathrm{take}\,\lambda(h(x))))))))\\
&= h(\psi(\mathrm{take}\,\lambda(h(x))))(o)
\end{aligned}$$

and we would have done.

Function ψ of Example 3.3.2 in Sect. 3.3 is a limit operator. Hence one can deduce by Theorem 3.4.2(ii) that h defined in that example is corecursive. It is also easy to check this directly.

As Proposition 3.4.3(ii) together with Example 3.4.4 show, Theorem 3.4.2(ii) would break down without the assumption that ψ is a limit operator.

Proposition 3.4.3. *Let X, A be sets. Let $\varphi \in X \to 1 + A$ and $\psi \in \mathrm{STList}\, A \to X$. Assume that $f \in X \to \mathrm{TList}\, A$ is corecursive on φ and ψ. Let λ, γ be selfish ordinals less than $\propto$ and take $x \in X$.*

(i) *Then*

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

(ii) *Moreover, if $\lambda < \gamma$ then*

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

Proof.

(i) If $|f(x)| \geqslant \lambda + \gamma$ then

$$\begin{aligned}\mathrm{drop}\,\gamma(\mathrm{drop}\,\lambda(f(x))) &= \mathrm{drop}\,\gamma(f(\psi(\mathrm{take}\,\lambda(f(x))))) \\ &= f(\psi(\mathrm{take}\,\gamma(f(\psi(\mathrm{take}\,\lambda(f(x))))))) \\ &= f(\psi(\mathrm{take}\,\gamma(\mathrm{drop}\,\lambda(f(x))))) \ .\end{aligned}$$

If $|f(x)| < \lambda + \gamma$ then both sides of the equality are $\perp$.

(ii) Note that $\lambda + \gamma = \gamma$ since γ is selfish. If $|f(x)| \geqslant \gamma$ then

$$\begin{aligned}f(\psi(\mathrm{take}\,\gamma(f(x)))) &= \mathrm{drop}\,\gamma(f(x)) = \mathrm{drop}(\lambda + \gamma)(f(x)) \\ &= \mathrm{drop}\,\gamma(\mathrm{drop}\,\lambda(f(x))) \\ &= f(\psi(\mathrm{take}\,\gamma(\mathrm{drop}\,\lambda(f(x))))) \\ &= f(\psi(\mathrm{drop}\,\lambda(\mathrm{take}(\lambda + \gamma)(f(x))))) \\ &= f(\psi(\mathrm{drop}\,\lambda(\mathrm{take}\,\gamma(f(x))))) \ .\end{aligned}$$

If $|f(x)| < \gamma$ then both sides of the equality are $\perp$. □

Example 3.4.4. Take $X = A = \{0, 1\}$, $\varphi(x) = x$ for both $x \in X$, and

$$\psi(l) = \left\{ \begin{array}{ll} 1 & \text{if } |l| = 1 \text{ or } \forall o < |l|\ (l(o) = 1) \\ 0 & \text{otherwise} \end{array} \right\} \ .$$

Suppose that f is corecursive on φ and ψ. It is easy to see that

$$\mathrm{take}\,\omega(f(0)) = 0 : 1 : 1 : 1 : \ldots \ .$$

Thus the equality stated by Proposition 3.4.3(ii) is broken:

$$\begin{aligned}\mathrm{head}(f(\psi(0 : 1 : 1 : 1 : \ldots))) &= \mathrm{head}(f(0)) = \varphi(0) = 0 \\ &\neq 1 = \varphi(1) = \mathrm{head}(f(1)) \\ &= \mathrm{head}(f(\psi(1 : 1 : 1 : \ldots))) \\ &= \mathrm{head}(f(\psi(\mathrm{drop}\,1(0 : 1 : 1 : 1 : \ldots)))) \ .\end{aligned}$$

□

Before ending this section, we prove some facts we need later on in the thesis.

Lemma 3.4.5. *Let o be an ordinal with principal representation $o = \alpha + \gamma$.*

(i) *If $\lambda \geqslant \gamma$ and λ is selfish then the principal representation of $\lambda + o$ is $(\lambda + \alpha) + \gamma$.*

(ii) *If $\alpha > 0$ or $\gamma = 1$ then, for arbitrary ordinal π, the principal representation of $\pi + o$ is $(\pi + \alpha) + \gamma$.*

Proof.

(i) For the "if" part of the lemma, assume that $\alpha > 0$ or $\lambda \geqslant \gamma$ holds. Take an arbitrary β such that $\lambda + o = \beta + \gamma$ (such β exists since $\lambda + o = \lambda + \alpha + \gamma$).
Show $\beta \geqslant \lambda$ by contradiction. If $\beta < \lambda$ then $\lambda = \beta + \lambda$, leading to $\beta + \lambda + \alpha + \gamma = \beta + \gamma$, thus also $\lambda + \alpha + \gamma = \gamma$ and $\lambda + \alpha < \gamma$. In the case $\alpha > 0$, we have $\alpha \geqslant \gamma$ (otherwise $\alpha + \gamma = \gamma = 0 + \gamma$ which contradicts the minimality of α), a contradiction with $\lambda + \alpha < \gamma$. The case $\lambda \geqslant \gamma$ contradicts the same inequality.
Let $\beta = \lambda + \varepsilon$. Then $\lambda + \alpha + \gamma = \lambda + \varepsilon + \gamma$ and thus $\alpha + \gamma = \varepsilon + \gamma$. As the lhs is a principal representation, this implies $\alpha \leqslant \varepsilon$. So $\lambda + \alpha \leqslant \lambda + \varepsilon = \beta$. Consequently, $(\lambda + \alpha) + \gamma$ is the principal representation of $\lambda + o$.
For the "otherwise" case, assume $\alpha = 0$ and $\lambda < \gamma$. Then $\lambda + \alpha + \gamma = \lambda + \gamma = \gamma = 0 + \gamma$ where the last sum is a principal representation.

(ii) Take β such that $\pi + o = \beta + \gamma$ (it is possible since $\pi + o = \pi + \alpha + \gamma$). We have to show that $\pi + \alpha \leqslant \beta$. Suppose the contrary, i.e. $\beta < \pi + \alpha$.
If $\beta < \pi$ then $\pi = \beta + \varepsilon$ for some ε. We obtain $\beta + \gamma = \pi + o = \beta + \varepsilon + o$, implying $\gamma = \varepsilon + o$. Hence $o = \gamma$ (as γ is selfish and $o \neq 0$), giving $\alpha = 0$ and therefore $\gamma = 1$. So we have $\pi + 1 = \beta + 1$ which leads to $\pi = \beta$, a contradiction.
If $\beta \geqslant \pi$ then $\beta = \pi + \varepsilon$ for some ε. As $\pi + \varepsilon = \beta < \pi + \alpha$, we have $\varepsilon < \alpha$. Then $\pi + o = \beta + \gamma = \pi + \varepsilon + \gamma$, implying $o = \varepsilon + \gamma$. Hence $\alpha \leqslant \varepsilon$, a contradiction. □

Theorem 3.4.6. *Let X, A be sets. Let $\varphi \in X \to A \cup \{\bot\} = 1 + A$, $\psi \in \mathrm{STList}\, A \to X$ and let $h \in X \to \mathrm{TList}\, A$ be iterative on φ and ψ. Let $o < \propto$ be an arbitrary ordinal with principal representation $o = \alpha + \gamma$.*

(i) *Let μ be the selfish ordinal next to γ. Then, for every ordinal $\pi \leqslant \mu$,*

$$h \,;\, \mathrm{drop}\, o \,;\, \mathrm{take}\, \pi = h \,;\, \mathrm{drop}\, \alpha \,;\, \mathrm{take}\, \gamma \,;\, \psi \,;\, h \,;\, \mathrm{take}\, \pi \ .$$

(ii) *If ψ is a limit operator then*

$$h \,;\, \mathrm{drop}\, o = h \,;\, \mathrm{drop}\, \alpha \,;\, \mathrm{take}\, \gamma \,;\, \psi \,;\, h \ .$$

Proof. Both statements are proved by induction on o. For $o = 0$, the claims are true vacuously. So assume that the claims hold for ordinals less than o.

(i) Weak corecursivity implies

$$h \,;\, \mathrm{drop}\,\gamma \,;\, \mathrm{take}\,\pi = h \,;\, \mathrm{take}\,\gamma \,;\, \psi \,;\, h \,;\, \mathrm{take}\,\pi \ .$$

Hence the desired claim follows for the case $\alpha = 0$.
Assume now $\alpha > 0$, let $\alpha = \beta + \delta$ be the principal representation. If $\delta < \gamma$ then $o = \alpha + \gamma = \beta + \delta + \gamma = \beta + \gamma$ — a contradiction since $\beta < \alpha$. Thus $\delta \geqslant \gamma$. As $\gamma + \pi \leqslant \gamma + \mu = \mu$, this implies that $\gamma + \pi$ does not exceed the selfish ordinal next to δ. Note furthermore that $\mathrm{take}\,\pi \,;\, \mathrm{take}\,\pi = \mathrm{take}\,\pi$. Now compute

$$\begin{aligned}
h \,;\, \mathrm{drop}\,o \,;\, \mathrm{take}\,\pi &= h \,;\, \mathrm{drop}(\alpha + \gamma) \,;\, \mathrm{take}\,\pi \\
&= h \,;\, \mathrm{drop}\,\alpha \,;\, \mathrm{drop}\,\gamma \,;\, \mathrm{take}\,\pi \,;\, \mathrm{take}\,\pi \\
&= h \,;\, \mathrm{drop}\,\alpha \,;\, \mathrm{take}(\gamma + \pi) \,;\, \mathrm{drop}\,\gamma \,;\, \mathrm{take}\,\pi \\
&= h \,;\, \mathrm{drop}\,\beta \,;\, \mathrm{take}\,\delta \,;\, \psi \,;\, h \,;\, \mathrm{take}(\gamma + \pi) \,;\, \mathrm{drop}\,\gamma \,;\, \mathrm{take}\,\pi \\
&= h \,;\, \mathrm{drop}\,\beta \,;\, \mathrm{take}\,\delta \,;\, \psi \,;\, h \,;\, \mathrm{drop}\,\gamma \,;\, \mathrm{take}\,\pi \,;\, \mathrm{take}\,\pi \\
&= h \,;\, \mathrm{drop}\,\beta \,;\, \mathrm{take}\,\delta \,;\, \psi \,;\, h \,;\, \mathrm{take}\,\gamma \,;\, \psi \,;\, h \,;\, \mathrm{take}\,\pi \,;\, \mathrm{take}\,\pi \\
&= h \,;\, \mathrm{drop}\,\alpha \,;\, \mathrm{take}\,\gamma \,;\, \psi \,;\, h \,;\, \mathrm{take}\,\pi \ .
\end{aligned}$$

(ii) Corecursivity implies

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

Hence the desired claim follows for the case $\alpha = 0$.
Assume now $\alpha > 0$, let $\alpha = \beta + \delta$ be the principal representation. Now compute

$$\begin{aligned}
h \,;\, \mathrm{drop}\,o &= h \,;\, \mathrm{drop}(\alpha + \gamma) = h \,;\, \mathrm{drop}\,\alpha \,;\, \mathrm{drop}\,\gamma \\
&= h \,;\, \mathrm{drop}\,\beta \,;\, \mathrm{take}\,\delta \,;\, \psi \,;\, h \,;\, \mathrm{drop}\,\gamma \\
&= h \,;\, \mathrm{drop}\,\beta \,;\, \mathrm{take}\,\delta \,;\, \psi \,;\, h \,;\, \mathrm{take}\,\gamma \,;\, \psi \,;\, h \\
&= h \,;\, \mathrm{drop}\,\alpha \,;\, \mathrm{take}\,\gamma \,;\, \psi \,;\, h \ .
\end{aligned}$$

□

Corollary 3.4.7. *Let X, A be sets. Let $\varphi \in X \to A \cup \{\bot\} = 1 + A$, $\psi \in \mathrm{STList}\,A \to X$ and let $h \in X \to \mathrm{TList}\,A$ be iterative on φ and ψ. Let λ, μ be consecutive selfish ordinals with $\lambda < \mu \leqslant \infty$. Then, for every natural number n and ordinal $\pi \leqslant \mu$,*

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

Proof. Argue by induction on n. If $n = 0$, both sides of the desired equation reduce to $h \,;\, \mathrm{take}\,\pi$. Assume now that the claim holds for n. As the principal

representation of $\lambda\cdot(n+1)$ is $\lambda\cdot n+\lambda$, Theorem 3.4.6(i) together with the induction hypothesis give

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

□

Corollary 3.4.8. *Let X, A be sets. Let $\varphi \in X \to A \cup \{\bot\} = 1 + A$, $\psi \in \mathrm{STList}\,A \to X$ and let $h \in X \to \mathrm{TList}\,A$ be iterative on φ and ψ. Let $\alpha \leqslant |h(x)|$ for some $x \in X$. Then there exists an element $z \in X$ such that all the following holds:*

1. $h(x)(\alpha) = \varphi(z)$;
2. *if γ is such that $\alpha + \gamma$ is a principal representation then*

$$\mathrm{take}\,\gamma(\mathrm{drop}\,\alpha(h(x))) = \mathrm{take}\,\gamma(h(z)) \ ;$$

3. *if ψ is a limit operator then* $\mathrm{drop}\,\alpha(h(x)) = h(z)$.

Proof. If $\alpha = 0$ then take $z = x$. By iterativity, $h(x)(0) = \varphi(x)$, so the first statement follows. The other two hold because $\mathrm{drop}\,0$ is the identity.
Consider the case $\alpha > 0$; let $\alpha = \beta + \delta$ be the principal representation. Define $z = \psi(\mathrm{take}\,\delta(\mathrm{drop}\,\beta(h(x)))) \in X$. By iterativity, $h(x)(\alpha) = \varphi(z)$. To prove the second equality, note that $\delta < \gamma$ would give $\alpha + \gamma = \beta + \delta + \gamma = \beta + \gamma$ with $\beta < \alpha$, contradicting the assumption that $\alpha + \gamma$ is a principal representation. Hence $\gamma \leqslant \delta$ which implies $\gamma < \mu$ where μ is the selfish ordinal next to δ. Thus, by Theorem 3.4.6(i), $\mathrm{take}\,\gamma(\mathrm{drop}\,\alpha(h(x))) = \mathrm{take}\,\gamma(h(z))$. If ψ is a limit operator then Theorem 3.4.6(ii) immediately gives $\mathrm{drop}\,\alpha(h(x)) = h(z)$. □

3.5 Non-Deterministic Transfinite Corecursion

We start with providing a new definition of corecursion which is the counterpart of Definition 3.4.1(ii) in the case where functions with sets of transfinite lists as values are considered. Denote the set of all subsets of S by $\wp S$. We use the set comprehension syntax of the well-known Z notation [17].

Definition 3.5.1. *Let X, A be sets. Take $\varphi \in X \to 1 + A$ and $\Psi \in \mathrm{STList}\,A \to \wp X$. We say that a function $F \in X \to \wp(\mathrm{TList}\,A)$ is* corecursive on φ and Ψ *iff the following two* corecursion conditions *hold:*

1. *if* $\varphi(x) = a \in A$ *then, for every* $l \in F(x)$, $\mathrm{head}\, l = a$, *and if* $\varphi(x) \in 1$ *then* $F(x) = \{\mathrm{nil}\}$;

2. *for any selfish* $\lambda < \propto$ *and transfinite list* $l \in (\mathbb{O}_\lambda \to A)$ *such that there exists an* $m \in F(x)$ *for which* $\mathrm{take}\, \lambda\, m = l$,

$$\{m \in F(x) \mid \mathrm{take}\, \lambda\, m = l \bullet \mathrm{drop}\, \lambda\, m\} = \bigcup_{z \in \Psi(l)} F(z) \ .$$

Lemma 3.5.2. *Let* X, A *be sets. Let* $\varphi \in X \to 1 + A$ *and* $\Psi \in \mathrm{STList}\, A \to \wp X$. *Assume that* $F \in X \to \wp(\mathrm{TList}\, A)$ *is corecursive on* φ *and* Ψ. *Suppose* o *being an ordinal with principal representation* $\alpha + \gamma$. *Let* $x \in X$ *and* $l \in \mathrm{TList}\, A$, $|l| = o$ *such that there exists an* $m \in F(x)$ *such that* $\mathrm{take}\, o\, m = l$. *Then*

$$\{m \in F(x) \mid \mathrm{take}\, o\, m = l \bullet \mathrm{drop}\, o\, m\} \subseteq \bigcup_{z \in \Psi(\mathrm{drop}\, \alpha\, l)} F(z) \ .$$

Proof. If $\alpha = 0$, the claim follows directly from the premises. So suppose $\alpha > 0$. Proceed by induction on o. As $\gamma > 0$, we have $\alpha < o$, so the induction hypothesis holds for α. Let $\beta + \delta$ be the principal representation of α.
Then, for arbitrary $k \in \mathrm{TList}\, A$,

$$\begin{aligned}
& k \in \{m \in F(x) \mid \mathrm{take}\, o\, m = l \bullet \mathrm{drop}\, o\, m\} \\
\iff\ & \exists m \in F(x)\, (\mathrm{take}\, o\, m = l \ \wedge\ \mathrm{drop}\, o\, m = k) \\
\iff\ & \exists m \in F(x)\, (\\
& \qquad \mathrm{take}\, \alpha(\mathrm{take}\, o\, m) = \mathrm{take}\, \alpha\, l \ \wedge \\
& \qquad \mathrm{drop}\, \alpha(\mathrm{take}\, o\, m) = \mathrm{drop}\, \alpha\, l \ \wedge \\
& \qquad \mathrm{drop}\, \gamma(\mathrm{drop}\, \alpha\, m) = k \\
&)\ (\\
\iff\ & \exists m \in F(x)\, (\\
& \qquad \mathrm{take}\, \alpha\, m = \mathrm{take}\, \alpha\, l \ \wedge \\
& \qquad \mathrm{take}\, \gamma(\mathrm{drop}\, \alpha\, m) = \mathrm{drop}\, \alpha\, l \ \wedge \\
& \qquad \mathrm{drop}\, \gamma(\mathrm{drop}\, \alpha\, m) = k \\
&) \\
\iff\ & \exists d \in \{m \in F(x) \mid \mathrm{take}\, \alpha\, m = \mathrm{take}\, \alpha\, l \bullet \mathrm{drop}\, \alpha\, m\} \\
& \qquad (\mathrm{take}\, \gamma\, d = \mathrm{drop}\, \alpha\, l \ \wedge\ \mathrm{drop}\, \gamma\, d = k) \\
\implies\ & \exists d \in \bigcup_{z \in \Psi(\mathrm{drop}\, \beta(\mathrm{take}\, \alpha\, l))} F(z) \\
& \qquad (\mathrm{take}\, \gamma\, d = \mathrm{drop}\, \alpha\, l \ \wedge\ \mathrm{drop}\, \gamma\, d = k)
\end{aligned}$$

$$
\begin{aligned}
&\iff \exists z \in \Psi(\mathrm{drop}\,\beta(\mathrm{take}\,\alpha\,l))\,\exists d \in F(z) \\
&\qquad (\mathrm{take}\,\gamma\,d = \mathrm{drop}\,\alpha\,l \ \wedge\ \mathrm{drop}\,\gamma\,d = k) \\
&\iff \exists z \in \Psi(\mathrm{drop}\,\beta(\mathrm{take}\,\alpha\,l)) \\
&\qquad (k \in \{d \in F(z) \mid \mathrm{take}\,\gamma\,d = \mathrm{drop}\,\alpha\,l \bullet \mathrm{drop}\,\gamma\,d\}) \\
&\implies \exists z \in \Psi(\mathrm{drop}\,\beta(\mathrm{take}\,\alpha\,l))\,(k \in \bigcup_{w \in \Psi(\mathrm{drop}\,\alpha\,l)} F(w)) \\
&\iff k \in \bigcup_{w \in \Psi(\mathrm{drop}\,\alpha\,l)} F(w)\ .
\end{aligned}
$$

This completes the proof. □

Theorem 3.5.3 claims the existence of a *largest* function meeting certain conditions rather than a unique function like it was in Theorem 3.4.2(ii). The order of functions is defined componentwise, being based on the set inclusion order.

Theorem 3.5.3. *Let X, A be sets. Let $\varphi \in X \to 1+A$ and $\Psi \in \mathrm{STList}\,A \to \wp X$. Assume $\Psi(l) = \Psi(\mathrm{drop}\,\lambda\,l)$ for all selfish ordinals λ, γ with $\lambda < \gamma < \propto$ and transfinite lists $l \in (\mathbb{O}_\gamma \to A)$. Then there is a largest function $F \in X \to \wp(\mathrm{TList}\,A)$ being corecursive on φ and Ψ.*

Proof. Let function $H \in X \to \wp(\mathrm{TList}\,A)$ be defined with

$$
H(x) = \left\{ \begin{array}{ll} \{\mathrm{nil}\} & \text{if } \varphi(x) \notin A \\ \{l \in \mathrm{TList}\,A \mid \mathrm{head}\,l = a \ \wedge\ P(l) \bullet l\} & \text{if } \varphi(x) = a \in A \end{array} \right\}
$$

where $P(l)$ means that

for all $o < |l|$ with principal representation $o = \alpha + \gamma$,
$\exists z \in \Psi(\mathrm{drop}\,\alpha(\mathrm{take}\,o\,l))\,(\varphi(z) = l(o))$
and,
with principal representation $|l| = \alpha + \gamma$,
$\exists z \in \Psi(\mathrm{drop}\,\alpha\,l)\,(\varphi(z) \notin A)$.

We are going to show that H is the largest function being corecursive on φ and Ψ. The proof is divided into many cases and subcases.

1. *Show that H is corecursive on φ and Ψ.*

1.1. *Corecursion condition 1.*

If $\varphi(x) = a \in A$ and $l \in H(x)$ then $\mathrm{head}\,l = a$ by definition of H. If $\varphi(x) \notin A$ then $H(x) = \{\mathrm{nil}\}$ by definition of H.

1.2. *Corecursion condition 2.*

Let $x \in X$. Fix a selfish ordinal $\lambda < \propto$ and a transfinite list $l \in (\mathbb{O}_\lambda \to A)$ such that $\mathsf{take}\,\lambda\,m = l$ for some $m \in H(x)$. Then $H(x) \neq \varnothing$, hence $\varphi(x) = a \in A$. Take an arbitrary $k \in \mathrm{TList}\,A$. We are going to show that

$$k \in \{m \in H(x) \mid \mathsf{take}\,\lambda\,m = l \bullet \mathsf{drop}\,\lambda\,m\} \iff \exists z \in \Psi(l)\,(k \in H(z)) \ .$$

1.2.a. *The case* $|k| = 0$, i.e. $k = \mathrm{nil}$.

1.2.a.($\subseteq$). Assume $\mathrm{nil} = \mathsf{drop}\,\lambda\,m$ for $m \in H(x)$ with $\mathsf{take}\,\lambda\,m = l$. Then $l = m \in H(x)$. By $P(l)$, we get $\varphi(z) \notin A$ for some $z \in \Psi(\mathsf{drop}\,0\,l) = \Psi(l)$. Then $H(z) = \{\mathrm{nil}\} \ni k$ for the same z.

1.2.a.($\supseteq$). Assume $\mathrm{nil} \in H(z)$ for some $z \in \Psi(l)$. Then $\varphi(z) \notin A$. Choose $m \in H(x)$ such that $\mathsf{take}\,\lambda\,m = l$. Then $\mathsf{head}\,l = \mathsf{head}\,m = a$ by definition of H. Take an arbitrary ordinal $o < |l|$ with principal representation $o = \alpha + \gamma$. As $m \in H(x)$ and $o < |m|$, there exists a $w \in \Psi(\mathsf{drop}\,\alpha(\mathsf{take}\,o\,m))$ such that $\varphi(w) = m(o)$. As $m(o) = l(o)$ and $\mathsf{take}\,o\,m = \mathsf{take}\,o\,l$, we have shown the first part of $P(l)$. The other part follows from $\varphi(z) \notin A$ and $z \in \Psi(l)$.

1.2.b. *The case* $|k| > 0$.

1.2.b.($\subseteq$). Fix $m \in H(x)$ such that $l = \mathsf{take}\,\lambda\,m$ and $k = \mathsf{drop}\,\lambda\,m$. Then $\lambda < |m|$ with principal representation $\lambda = 0 + \lambda$, so we have $\varphi(z) = m(\lambda)$ for some $z \in \Psi(\mathsf{take}\,\lambda\,m) = \Psi(l)$. It suffices to show $k \in H(z)$. As $\varphi(z) = m(\lambda) \in A$, we must show $\mathsf{head}\,k = m(\lambda)$ and $P(k)$. The former comes from $\mathsf{head}\,k = (\mathsf{drop}\,\lambda\,m)(0) = m(\lambda + 0)$. We now prove $P(k)$.

1.2.b.($\subseteq$).1. Take $o < |k|$ with $o = \alpha + \gamma$ being its principal representation. Then

$$\begin{aligned}\Psi(\mathsf{drop}\,\alpha(\mathsf{take}\,o\,k)) &= \Psi(\mathsf{take}\,\gamma(\mathsf{drop}\,\alpha\,k)) \\ &= \Psi(\mathsf{take}\,\gamma(\mathsf{drop}\,\alpha(\mathsf{drop}\,\lambda\,m))) \\ &= \Psi(\mathsf{take}\,\gamma(\mathsf{drop}(\lambda + \alpha)\,m)) \\ &= \Psi(\mathsf{drop}(\lambda + \alpha)(\mathsf{take}(\lambda + o)\,m)) \ .\end{aligned}$$

Note that $\lambda + o < \lambda + |k| = |m|$. If $\alpha > 0$ or $\lambda \geqslant \gamma$ then, using $P(m)$, we get $k(o) = m(\lambda + o) = \varphi(w)$ for some $w \in \Psi(\mathsf{drop}(\lambda + \alpha)(\mathsf{take}(\lambda + o)\,m)) = \Psi(\mathsf{drop}\,\alpha(\mathsf{take}\,o\,k))$. If $\alpha = 0$ and $\lambda < \gamma$ then, using $P(m)$ together with the restriction imposed on Ψ, we get $k(o) = k(\gamma) = m(\gamma) = \varphi(w)$ for some $w \in \Psi(\mathsf{take}\,\gamma\,m) = \Psi(\mathsf{take}\,\gamma\,k)$. The first clause of $P(k)$ follows.

1.2.b.($\subseteq$).2. Take $|k| = \alpha + \gamma$, the sum being a principal representation. Then

$$\Psi(\mathsf{drop}\,\alpha\,k) = \Psi(\mathsf{drop}\,\alpha(\mathsf{drop}\,\lambda\,m)) = \Psi(\mathsf{drop}(\lambda + \alpha)m) \ .$$

Note that $\lambda + |k| = |m|$. If $\alpha > 0$ or $\lambda \geqslant \gamma$ then, using $P(m)$, we get $\varphi(w) \notin A$ for some $w \in \Psi(\mathsf{drop}(\lambda + \alpha)\,m) = \Psi(\mathsf{drop}\,\alpha\,k)$. If $\alpha = 0$ and $\lambda < \gamma$ then, using

$P(m)$ together with the restriction imposed on Ψ, we get $\varphi(w) \notin A$ for some $w \in \Psi(m) = \Psi(k)$. The second clause of $P(k)$ follows.

1.2.b.($\supseteq$). Assume $k \in H(z)$ for some $z \in \Psi(l)$. Fix $m \in H(x)$ such that take $\lambda\, m = l$. It suffices to show $l +\!\!+ k \in H(x)$. As $\varphi(x) = a \in A$, we have to check head$(l +\!\!+ k) = a$ and $P(l +\!\!+ k)$. The former comes from head$(l +\!\!+ k) =$ head l = head $m = a$. We now prove $P(l +\!\!+ k)$.

1.2.b.($\supseteq$).1. Let $o < |l +\!\!+ k| = |l| + |k|$ with principal representation $o = \beta + \delta$.

If $o < \lambda$ then, as $m \in H(x)$, we have $\varphi(w) = m(o) = l(o) = (l +\!\!+ k)(o)$ for $w \in \Psi(\text{drop}\,\beta(\text{take}\, o\, m)) = \Psi(\text{drop}\,\beta(\text{take}\, o\, l)) = \Psi(\text{drop}\,\beta(\text{take}\, o(l +\!\!+ k)))$.

Assume $o \geqslant \lambda$ now; let $o = \lambda + \pi$. If $\pi = 0$ then, as $k \in H(z)$, we have $\varphi(z) = k(0) = (l +\!\!+ k)(o)$. In this case, $\beta = 0$ and $o = \lambda$, so

$$\Psi(\text{drop}\,\beta(\text{take}\, o(l +\!\!+ k))) = \Psi(l) \ni z \ .$$

It remains to consider the case $\pi > 0$. Let $\pi = \alpha{+}\gamma$ be the principal representation. If $\alpha > 0$ or $\lambda \geqslant \gamma$ then $\beta = \lambda + \alpha$ and $\delta = \gamma$. Note that

$$\begin{aligned}\Psi(\text{drop}\,\alpha(\text{take}\,\pi\, k)) &= \Psi(\text{drop}\,\alpha(\text{take}\,\pi(\text{drop}\,\lambda(l +\!\!+ k))))\\ &= \Psi(\text{drop}\,\alpha(\text{drop}\,\lambda(\text{take}\, o(l +\!\!+ k))))\\ &= \Psi(\text{drop}(\lambda + \alpha)(\text{take}\, o(l +\!\!+ k)))\\ &= \Psi(\text{drop}\,\beta(\text{take}\, o(l +\!\!+ k)))\ .\end{aligned}$$

Thus we have $\varphi(w) = k(\pi) = (l +\!\!+ k)(o)$ for $w \in \Psi(\text{drop}\,\beta(\text{take}\, o(l +\!\!+ k)))$. If $\alpha = 0$ and $\lambda < \gamma$ then $\beta = 0$ and $\delta = \gamma$. Note that

$$\begin{aligned}\Psi(\text{take}\,\gamma\, k) &= \Psi(\text{take}\,\gamma(\text{drop}\,\lambda(l +\!\!+ k))) = \Psi(\text{drop}\,\lambda(\text{take}\,\gamma(l +\!\!+ k)))\\ &= \Psi(\text{take}\,\gamma(l +\!\!+ k)) = \Psi(\text{take}\, o(l +\!\!+ k))\ .\end{aligned}$$

Thus we have $\varphi(w) = k(\pi) = k(\gamma) = (l +\!\!+ k)(\gamma) = (l +\!\!+ k)(o)$ for some $w \in \Psi(\text{take}\, o(l +\!\!+ k))$.

In all cases, we have proven the first condition of $P(l +\!\!+ k)$.

1.2.b.($\supseteq$).2. Let $|k| = \alpha + \gamma$ be the principal representation. If $\alpha > 0$ or $\lambda \geqslant \gamma$ then, since $k \in H(z)$, we have $\varphi(w) \notin A$ for some $w \in \Psi(\text{drop}\,\alpha\, k) = \Psi(\text{drop}(\lambda + \alpha)(l +\!\!+ k))$. If $\alpha = 0$ and $\lambda < \gamma$ then analogously we have $\varphi(w) \notin A$ for some $w \in \Psi(k) = \Psi(\text{drop}\,\lambda(l +\!\!+ k)) = \Psi(l +\!\!+ k)$. In both cases, the second clause of $P(l +\!\!+ k)$ follows.

2. *Show that if some F is corecursive on φ and Ψ then $F \sqsubseteq H$ (i.e. $F(x) \subseteq H(x)$ for every $x \in X$).*

If $\varphi(x) \notin A$ then $F(x) = \{\text{nil}\} = H(x)$. Let now $\varphi(x) = a \in A$ and take $l \in F(x)$. Then head $l = a$ by corecursion condition 1.

Take $o \leqslant |l|$ with principal representation $\alpha + \gamma$. By Lemma 3.5.2,

$$\{m \in F(x) \mid \mathrm{take}\, o\, m = \mathrm{take}\, o\, l \bullet \mathrm{drop}\, o\, m\} \subseteq \bigcup_{z \in \Psi(\mathrm{drop}\, \alpha(\mathrm{take}\, o\, l))} F(z) \ .$$

Thus $\mathrm{drop}\, o\, l \in F(z)$ for some $z \in \Psi(\mathrm{drop}\, \alpha(\mathrm{take}\, o\, l))$.
If $o < |l|$ then obviously $\mathrm{drop}\, o\, l \neq \mathrm{nil}$. Thus $F(z) \neq \{\mathrm{nil}\}$ implying $\varphi(z) = b \in A$. Now $\mathrm{head}(\mathrm{drop}\, o\, l) = b$, i.e. $\varphi(z) = l(o)$. If $o = |l|$, we get $z \in \Psi(\mathrm{drop}\, \alpha\, l)$ and $\mathrm{nil} \in F(z)$. Then $\varphi(z) \in A$ would contradict F satisfying corecursion condition 1, hence $\varphi(z) \notin A$. Altogether, this gives $P(l)$. Consequently, $l \in H(x)$. □

To finish this section, we show that the function H constructed in the proof of Theorem 3.5.3 is not the only one being corecursive on φ and Ψ.

Note that the corecursion conditions hold trivially whenever

$$F(x) = \left\{ \begin{array}{ll} \{\mathrm{nil}\} & \text{if } \varphi(x) \notin A \\ \varnothing & \text{otherwise} \end{array} \right\}$$

for every $x \in X$. Let $X = \{0, 1\}$, $A = \{0\}$ and $\varphi(0) = 0$, $\varphi(1) \notin A$. Let $\Psi(l) = X$ for every l. Then $F(0) = \varnothing$ while $H(0) \ni 0 : \mathrm{nil}$.

3.6 Connections between Two Corecursions

The corecursion theorems (Theorems 3.4.2(ii) and 3.5.3) were proven independently on each other. Nevertheless, the two corecursions are analogous. This section points out the relation between them.

The following result states that using our corecursion of the second type for defining one-element sets is equivalent to the corecursion of the first type.

Theorem 3.6.1. *Let X, A be sets. Take $\varphi \in X \to 1 + A$, as well as $\psi \in \mathrm{STList}\, A \to X$ and $\Psi \in \mathrm{STList}\, A \to \wp X$ such that $\Psi(l) = \{\psi(l)\}$ for every $l \in \mathrm{STList}\, A$. Assume that $f \in X \to \mathrm{TList}\, A$ is corecursive on φ and ψ. Then $F(x) = \{f(x)\}$ is the largest function corecursive on φ and Ψ.*

Proof. We show first that F is corecursive on φ and Ψ. Take an arbitrary $x \in X$. If $\varphi(x) \in A$ then, for every $l \in F(x)$, obviously $\mathrm{head}\, l = \mathrm{head}(f(x)) = \varphi(x)$. If $\varphi(x) \notin A$ then $F(x) = \{f(x)\} = \{\mathrm{nil}\}$. Take now a selfish λ and a transfinite list $l \in (\mathbb{O}_\lambda \to A)$ such that $\mathrm{take}\, \lambda\, m = l$ for some $m \in F(x)$. This obviously implies $\mathrm{take}\, \lambda(f(x)) = l$ and

$$\begin{aligned} \{m \in F(x) \mid \mathrm{take}\, \lambda\, m = l \bullet \mathrm{drop}\, \lambda\, m\} &= \{\mathrm{drop}\, \lambda(f(x))\} \\ = \{f(\psi(\mathrm{take}\, \lambda(f(x))))\} &= \{f(\psi(l))\} \\ &= F(\psi(l)) = \bigcup_{z \in \Psi(l)} F(z) \ . \end{aligned}$$

So F is indeed corecursive on φ and Ψ.
Let now $G \in X \to \wp(\mathrm{TList}\, A)$ be any function corecursive on φ and Ψ. We show by transfinite induction on o that

$$\forall x \in X\, \forall l \in G(x)\, (f(x)(o) = l(o)) \tag{3.2}$$

for every ordinal $o < \infty$. Firstly, we obtain

$$f(x)(0) = \mathrm{head}(f(x)) = \left\{ \begin{array}{ll} \varphi(x) & \text{if } \varphi(x) \in A \\ \bot & \text{otherwise} \end{array} \right\} = \mathrm{head}\, l = l(0)$$

for every $x \in X$ and $l \in G(x)$. Consider now any non-zero $o < \infty$ and assume Eq. 3.2 holding for all $\pi < o$. Take $o = \gamma + \beta$ with γ selfish and $\beta < o$. Fix $x \in X$ and $l \in G(x)$ arbitrarily. The induction hypothesis implies $|l| \geqslant \gamma \iff |f(x)| \geqslant \gamma$, as well as $\mathrm{take}\,\gamma(f(x)) = \mathrm{take}\,\gamma\, l$, as well as $f(z)(\beta) = k(\beta)$ for every $z \in X$ and $k \in G(z)$. By corecursiveness of G,

$$\begin{aligned} \{m \in G(x) \mid \mathrm{take}\,\gamma\, m = \mathrm{take}\,\gamma\, l \bullet \mathrm{drop}\,\gamma\, m\} &= \bigcup_{z \in \Psi(\mathrm{take}\,\gamma\, l)} G(z) \\ &= G(\psi(\mathrm{take}\,\gamma\, l))\ . \end{aligned}$$

So $\mathrm{drop}\,\gamma\, l \in G(\psi(\mathrm{take}\,\gamma\, l))$. Hence

$$\begin{aligned} f(x)(o) &= f(x)(\gamma + \beta) = (\mathrm{drop}\,\gamma(f(x)))(\beta) = (f(\psi(\mathrm{take}\,\gamma(f(x)))))(\beta) \\ &= (f(\psi(\mathrm{take}\,\gamma\, l)))(\beta) = (\mathrm{drop}\,\gamma\, l)(\beta) = l(\gamma + \beta) = l(o)\ . \end{aligned}$$

This shows that $F(x) = \{f(x)\} \supseteq G(x)$ for every $x \in X$, i.e. $G \sqsubseteq F$. □

There is also a result of more general kind relating the two corecursions.

Lemma 3.6.2. *Let X, A be sets. Take $\varphi \in X \to 1 + A$, as well as $\Psi_1, \Psi_2 \in \mathrm{STList}\, A \to X$ such that $\Psi_1 \sqsubseteq \Psi_2$ (i.e. $\Psi_1(l) \subseteq \Psi_2(l)$ for every $l \in \mathrm{STList}\, A$). Assume both $\Psi_1(l) = \Psi_1(\mathrm{drop}\,\lambda\, l)$ and $\Psi_2(l) = \Psi_2(\mathrm{drop}\,\lambda\, l)$ for all selfish ordinals λ, γ with $\lambda < \gamma < \infty$ and transfinite lists $l \in (\mathbb{O}_\gamma \to A)$. Let $F_1 \in X \to \wp(\mathrm{TList}\, A)$ be the largest function corecursive on φ and Ψ_1, and $F_2 \in X \to \wp(\mathrm{TList}\, A)$ be the largest function corecursive on φ and Ψ_2. Then $F_1 \sqsubseteq F_2$.*

Proof. Let P_1 and P_2 denote the condition P from the proof of Theorem 3.5.3 defined for Ψ_1 and Ψ_2, respectively. Let H_1 and H_2 be the corresponding functions H defined in the same proof. The proof shows that $F_1 = H_1$ and $F_2 = H_2$. So it suffices to check $H_1(x) \subseteq H_2(x)$ for all $x \in X$ which is an easy case study by the definition of H using the assumption $\Psi_1 \sqsubseteq \Psi_2$. □

Theorem 3.6.3. *Let X, A be sets. Take $\varphi \in X \to 1 + A$, as well as $\psi \in \mathrm{STList}\, A \to X$ and $\Psi \in \mathrm{STList}\, A \to \wp X$ such that $\psi(l) \in \Psi(l)$ for every $l \in \mathrm{STList}\, A$. Assume both $\psi(l) = \psi(\mathrm{drop}\, \lambda\, l)$ and $\Psi(l) = \Psi(\mathrm{drop}\, \lambda\, l)$ for all selfish ordinals λ, γ with $\lambda < \gamma < \propto$ and transfinite lists $l \in (\mathbb{O}_\gamma \to A)$. Let $f \in X \to \mathrm{TList}\, A$ be the function corecursive on φ and ψ and $F \in X \to \wp(\mathrm{TList}\, A)$ be the largest function corecursive on φ and Ψ. Then $f(x) \in F(x)$ for every $x \in X$.*

Proof. Define $\Psi'(l) = \{\psi(l)\}$ for all $l \in \mathrm{STList}\, A$. By Theorem 3.6.1, $F'(x) = \{f(x)\}$ is the largest function corecursive on φ and Ψ'. Since $\Psi'(l) = \{\psi(l)\} = \{\psi(\mathrm{drop}\, \lambda\, l)\} = \Psi'(\mathrm{drop}\, \lambda\, l)$ for all selfish ordinals λ, γ with $\lambda < \gamma < \propto$ and lists $l \in (\mathbb{O}_\lambda \to A)$, Lemma 3.6.2 applies and gives the desired result. □

Theorem 3.6.3 made all the assumptions which are needed to prove any of the two corecursion theorems. In contrast to it, Theorem 3.6.1 assumes only the corecursivness of f on φ and ψ, no further assumptions except the special shape of Ψ are needed.

CHAPTER 4

PROGRAM SLICING WITH RESPECT TO TRANSFINITE SEMANTICS

As the main contribution of the thesis, this chapter contains an expansion of the mathematical framework of transfinite trace semantics and a proof of correctness of standard algorithms of program slicing w.r.t. a class of transfinite semantics.

The theory is developed for control flow graphs to keep the treatment abstracted from syntactic details and to drop the need for assuming structured control flow. We generalize the traditional notion of control flow graph to transfinite control flow graph which is obtained from traditional one by adding "transfinite arcs" representing possible escaping from infinite loops.

4.1 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.

To describe program slicing, the first demand to semantics is that it must trace movement of control, as well as changing of evaluation of variables. *Program points* of a program S are potential locations of control during executions of S. To achieve iterative semantics, locations of control in a procedure must be distinguished by call string (e.g. the starting point of a procedure P when called from Q and when called from $R \neq Q$ are different program points). As there is no recursion, the set of all program points of a fixed program is finite. There is one special

program point f called *final* corresponding to the empty (or finished) computation. For every program S, there is a program point i_S among the program points of S — the initial point.

A *configuration* is a pair of a program point and a *state*, the latter containing an evaluation of variables. Let PP, $State$ and $Conf$ denote the set of all program points of all programs, the set of all states, and the set of all configurations, respectively; so $Conf = PP \times State$. The configuration with program point p and state s is denoted by $\langle p \mid s \rangle$. Let Var be the set of all variables and Val denote the set of all possible values of variables.

This section copes with semantics where the meaning of a program is a function whose values are sequences of configurations expressing the step-by-step computation process. Hence the states of Chapter 3 are actually abstractions of configurations. This principle holds also in the remaining sections with the difference that the configurations are a little more complicated.

As transfinite configuration trace semantics differ from standard configuration trace semantics only by the occurrence of "transfinite steps", we concentrate to the latter in our discussion here. For ordinary steps, just say we have fixed a transition function $\mathrm{next} \in Conf \to 1 + Conf = Conf \cup \{\bot\}$ such that $\mathrm{next}\langle p \mid s \rangle = \bot$ iff $p = f$. Applying next represents making an atomic computation step and just the final program point enables no further computation.

Consider the following way to define transfinite semantics for a program which contains a while-loop. One has to provide principles for finding limit configurations of endless sequences of them. It means that one must show both the limit program point and the limit state. As explained in Sect. 3.3, it suffices to provide rules for lists of selfish length (in terms of Definition 3.3.1 and Theorem 3.3.3, we are defining ψ).

For the limit program point $\lim p$ 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 program point where the control would go if the predicate on top of this loop would evaluate to ff. For other transfinite lists of program points, define the limit point to be constantly f.

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 holds, do T, but never more than ω times".

In the limit state $\lim s$ of a state list s, a variable X has value v if the transfinite list of the values of X during the transfinite computation represented by s stabilizes to v; 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.

Now for every transfinite configuration list $c = (\langle p_o \mid s_o \rangle : o < \gamma)$ with selfish length γ, define

$$\psi(c) = \left\{ \begin{array}{ll} \text{next}(\text{head}\, c) & \text{if } \gamma = 1 \\ \langle \lim p \mid \lim s' \rangle & \text{otherwise} \end{array} \right\} \tag{4.1}$$

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

By Theorem 3.3.3, there exists a function $h \in 1 + \mathit{Conf} \to \text{TList}\, \mathit{Conf}$ being iterative on $\text{id} \in 1 + \mathit{Conf} \to 1 + \mathit{Conf}$ and ψ. The desired transfinite semantics $\mathcal{T} \in \mathit{Prog} \to \mathit{State} \to \text{TList}\, \mathit{Conf}$ is achieved by defining $\mathcal{T}(S)(i) = h\langle i_S \mid i \rangle$ for every program S and initial state i. It is easy to verify that ψ is a limit operator; hence the semantics is even corecursive.

Definition 4.1.1. *Let C be a set of configurations such that $\langle i_S \mid i \rangle \in C$ for every $S \in \mathit{Prog}$ and $i \in \mathit{State}$. (We do not require $C \subseteq \mathit{Conf}$ since, in the following sections, we use a wider kind of configurations.) Denote the function being iterative on $\text{id} \in 1 + C \to 1 + C$ and $\psi \in \text{STList}\, C \to 1 + C$ by $\text{iter}\, \psi$ and, for all $S \in \mathit{Prog}$ and $i \in \mathit{State}$, define transfinite configuration trace semantics corresponding to ψ by*

$$\mathcal{T}_\psi(S)(i) = \text{iter}\, \psi \langle i_S \mid i \rangle \ .$$

Being strict, applying Theorem 3.3.3 needs fixing an ordinal $\propto$ which is an upper bound of lengths of all transfinite lists obtained as values of our iterative functions. We can choose $\propto$ arbitrarily; Theorem 4.3.8 shows that taking $\propto = \omega^\omega$ ensures any program being executed to the end of its code.

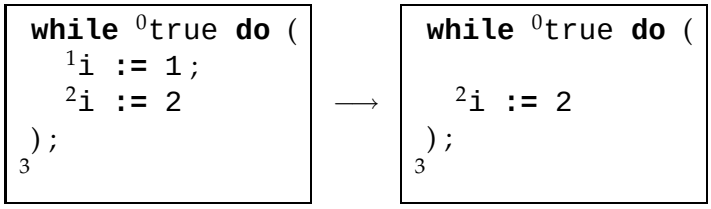
In this semantics, the execution of the program in Example 1.1.2 with initial state $\{\, \mathsf{x} \to 1 \,\}$ goes as follows:

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

So it reaches program point 2 once just like the slice and computes the same value (0) for x. Semantic anomaly has disappeared.

Note that replacing s' with s in the definition of ψ (Eq. 4.1) would cause another kind of trouble.

Example 4.1.2. The second program is a slice of the first w.r.t. criterion $\{(3, \texttt{i})\}$:



Having s at place of s' in Eq. 4.1 means that the sequence of values whose stabilization determines the value of a variable after the loop involves all values this variable has during the infinite execution, not only the values observed at top of the loop. In the example, this would mean that 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. Using s' in Eq. 4.1 ensures the value of `i` at 3 being 2 also in the first program. □

The problem observed here arises because the intuitive way of understanding slicing, followed also by the standard slicing algorithms, assumes that the values of variables immediately after a loop are computable according to their values at the head point of while-loop. Transfinite semantics must follow this principle.

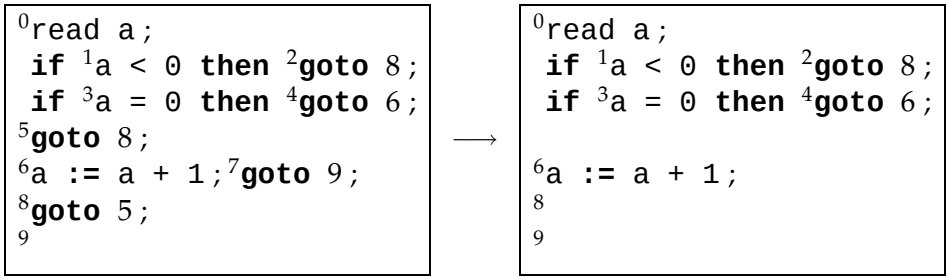
In the case of while-loops, defining limit configurations does not make much trouble. The choice of the limit program point is particularly straightforward because there is just one natural way to escape from the loop — going to the point where control would fall if the predicate evaluated to false.

If the control flow is unstructured, such an obvious choice need not exist. Obscurity can arise also in the case of structured control flow, for example, with statements like **`break`** in C as, in the presence of such statements in the loop, more than one natural way to leave the loop exists. But if our language allows a priori infinite loop constructions like **`loop`** S, there is no natural ways to leave at all. The intuition tells us that one should choose the program point lexically following the loop, taking the structure of the program into account (e.g., if the loop-construct is the only statement in another loop then we should fall to the beginning of the outer loop).

The latter intuition is based on the following general principle. To ensure a transfinite semantics being in harmony with program slicing, the limit program point should be the point where control would fall if the loop were removed.

We illustrate this principle on two examples on unstructured control flow. We use our abstract program point notation in goto-statements since the code is primarily intended to be illustrative rather than strictly following some syntax rules of a fixed language. Each if-statement incorporates only one row in the program.

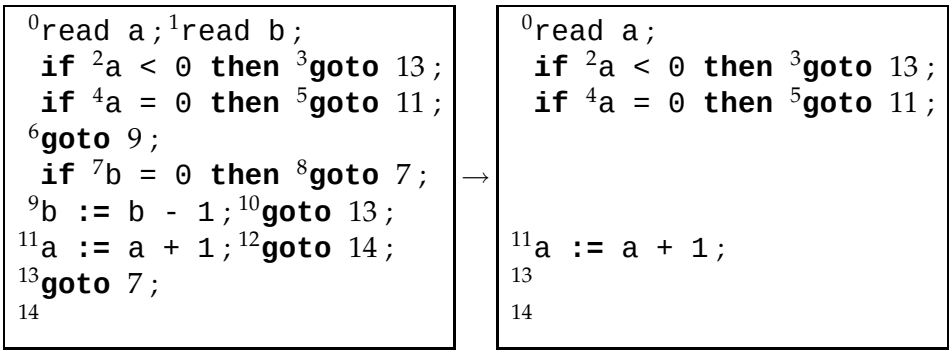
Example 4.1.3.



Suppose the slicing criterion is $\{(9, \mathsf{a})\}$. 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 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 form $\mathcal{T}_\psi$ then ψ is not a limit operator and the semantics is not corecursive. □

Example 4.1.4. Consider the following modification of Example 4.1.3:



Some stuff concerning a new variable b has been added in comparison with Example 4.1.3 (the new stuff is at points 1 and 6–9). If the slicing criterion is still $\{(14, \mathsf{a})\}$ then our aim is to slice this away resulting in the same program as in Example 4.1.3.

Denote by a the initial value of a again. To justify this slicing with transfinite semantics, control still must reach program point 11 during the transfinite run of the left-hand program if $a > 0$ and bypass 11 if $a < 0$. Things are more complicated than in Example 4.1.3 because of the new loop at 7–8 into which control falls whenever $a \neq 0$.

What should the limit point of this new loop be? The most natural choice seems to be 9. But then, after control has reached 9, it starts looping between program points 7, 9–10, 13, and the entire sequence of length ω of the program points visited during this looping does not depend on whether a is positive or negative.

There are two imaginable ways out from this trouble. One is that the limit program point of the loop at 7–8 depends on the program points through which control has reached it — even if these program points are visited only finite number of times. Then we can declare that the limit point is 9 if control came to the loop via 6 and 14 otherwise. Another approach is that the limit program point of the second loop (including 7, 9, 10, 13) depends on some information embedded into the computation which occurred before control even reached this part of length ω. This would mean that the semantics is not even iterative in the sense of Definition 3.3.1.

None of these solutions are captured by our theory developed in this thesis. To obtain a semantics to which our theory would apply, program point 10 must be added into the slice and the limit point of the loop at 7, 9, 10, 13 must be 14 irrespective of a. □

4.2 Transfinite Control Flow Graphs

To go on, we take a slightly different view to transfinite trace semantics. Firstly, we must describe the transition in terms of ψ rather than ψ in terms of the transition like in Sect. 4.1. This enables one to formulate all our properties of semantics in terms of properties of ψ. Secondly, we must augment our configurations with additional information about the atomic computation step executed just before reaching this configuration. In particular, this holds for the configurations reached via jumping out from an endless computation. Explicit encoding of such "transfinite steps" into the semantics simplifies formalization of some useful properties later.

We will define the augmented configurations formally in the next section; this section introduces the appropriate context for doing it. Our graphs are all directed; many well-known notions of graph theory are used, the necessary definitions and properties were given in Chapter 2.

Let *PP* be a set of formal objects called *program points* where one point $f \in PP$ is called *final*. This in principle coincides with that of Sect. 4.1. Additionally, let AS be a set of formal objects encoding all conceivable atomic computation steps, e.g. assignments, predicate tests etc., including the transfinite steps seceding from loops. The transfinite steps form a subset AS_∞ of AS.

We require each $e \in AS$ incorporating also information about the program point

to which control reaches after making this step; call this program point *target of* e and denote it by $\tau(e)$. Hence the elements of AS cannot be just code fragments, they must enable one to locate the code in terms of our program points.

Assume that each $e \in AS$ determines also the *source* point of e which we denote by $\sigma(e)$. For ordinary steps e, it is the program point from which step e starts. For transfinite steps e, the source of e is meant to be the program point at which the values of variables are recorded for finding limit state when doing the transfinite step e. Relying on Example 4.1.2, the source point of the transfinite step escaping from a while-loop should be the head point of this loop.

This way, the system $TCFG = (PP, AS, (\sigma, \tau))$ forms a (possibly infinite) directed graph (Definition 2.1.1). We call it *global transfinite control flow graph* and the elements of AS_∞ *transfinite arcs*. This graph is a formal object like code. The explanations in the preceding paragraphs take into account also the purpose the vertices and arcs can obtain via associating with a semantics.

Assume further that f is reachable from every program point in $TCFG$ while no arc starts from f. The former means that transfinite arcs must in principle allow to escape from any place the computation has driven and enable finishing the run. So $(TCFG, f)$ forms a flow graph in the sense of Definition 2.2.1.

Denote by CFG the directed graph obtained from $TCFG$ by removing transfinite arcs; call it *global control flow graph*. The word 'global' is used in both cases with the aim of reflecting the property of the graph to incorporate all programs. To handle control flow graphs of specific programs, we introduce the parallel notions of local flow graphs.

Call *local transfinite control flow graph* any system (G, i) where G is any subgraph of $TCFG$ being closed w.r.t. finding reachable vertices and arcs, and i is any vertex of G. The vertex i is called *initial*. A graph is called *local control flow graph* if can be obtained from a local transfinite control flow graph by removing all transfinite arcs.

Note that any local transfinite control flow graph contains f and forms a flow graph of Definition 2.2.1 together with f. Any local control flow graph also must contain f but not necessarily forms a flow graph in the sense of Definition 2.2.1. The initial vertex can be different for different local flow graphs while the final vertex f is common for all.

For every program $S \in Prog$, let $\mathrm{tcfg}\, S$ be a local transfinite control flow graph called *transfinite control flow graph of* S. Let $\mathrm{cfg}\, S$, the *control flow graph of* S, be the local control flow graph obtained from $\mathrm{tcfg}\, S$ by removing transfinite arcs. The initial point of $\mathrm{tcfg}\, S$ is denoted by i_S; call it *the initial point of* S. The set of all program points of a program S can therefore be denoted by $V(\mathrm{tcfg}\, S)$ (or, equivalently, by $V(\mathrm{cfg}\, S)$).

The treatment may seem mysterious at first glance since we have fixed control flow graphs for programs before fixing a semantics. The common mind tells that finding control flow graph of a program cannot be done without knowing the semantics of this program.

The idea is that we take a big graph (the global transfinite) which enfolds all possible transition systems ever needed, define semantics in terms of this graph and then programs can get their semantics via an appropriate mapping $S \mapsto \mathrm{tcfg}\, S$. In our theoretical study, we just say that some mapping is fixed, omitting the details of its definition, and are therefore able to introduce it independently of semantics.

Another reason for this approach is that control flow graphs, like they are traditionally computed according to the code, are not semantically precise, they are conservative approximations. Finding a semantically precise control flow graph is not decidable because a program doing it would decide whether a test statement really involves branching or whether a while-loop can terminate normally. To obtain results about slicing algorithms, the theory must capture decidable control flow graphs. In particular, this holds for transfinite arcs which also must be decidable because the analyses being preformed in order to find slices must be able to follow them.

Definition 4.2.1. *Let S be any program.*

(i) *Call S* finite *iff* tcfg S *contains only a finite number of vertices and arcs.*

(ii) *Call S* regular *iff S is finite and, for every transfinite arc e in* tcfg S, $\tau(e)$ *is the immediate postdominator of $\sigma(e)$ in* tcfg S.

The notion of finiteness of programs places the traditional finiteness of programs into the context of control flow graphs. The definition of regularity refers to the postdominance order which is well-known but also explained by us in Sect. 2.2 for abstract flow graphs. Informally, a program point q postdominates a program point p if control definitely reaches q whenever it has reached p (provided the computation finishes at f). The immediate postdominator of p is the least w.r.t. postdominance order point postdominating p. The notion of regularity is not intended to be a counterpart of any standard notion. It states the transfinite arcs respecting some order in the global graph. This condition has been formed keeping the treatment of while-loops from Sect. 4.1 in mind. There, the limit program point of a while-loop was defined in such a way that it coincided with the immediate postdominator of the head point of the while-loop.

4.3 Augmented Configuration Trace Semantics

Augmented configurations are pairs of form $\langle \mathsf{a} \mid s \rangle$ where $\mathsf{a} \in PP + AS$ and $s \in State$. If $e \in AS$, the configuration $\langle e \mid s \rangle$ encodes the situation where state s has been obtained by performing atomic step e. For $p \in PP$, the configuration $\langle p \mid i \rangle$ means that computation starts at p with initial state i. To achieve uniformity, the left components of augmented configurations which belong to PP may be called *entrance steps*.

Denote the set of all augmented configurations by $AConf$, so $AConf = (PP + AS) \times State$ and taking $C = AConf$ in Definition 4.1.1 satisfies its conditions. Denote also

$$\begin{aligned}
\mathrm{st}\langle \mathsf{a} \mid s \rangle &= s \ , \\
\mathrm{pp}\langle \mathsf{a} \mid s \rangle &= \left\{ \begin{array}{ll} \tau(\mathsf{a}) & \text{if } \mathsf{a} \in AS \\ \mathsf{a} & \text{otherwise} \end{array} \right\} \ , \\
\mathrm{arc}\langle \mathsf{a} \mid s \rangle &= \left\{ \begin{array}{ll} \mathsf{a} & \text{if } \mathsf{a} \in AS \\ \bot & \text{otherwise} \end{array} \right\} \ , \\
\mathrm{conf}\, c &= \langle \mathrm{pp}\, c \mid \mathrm{st}\, c \rangle \ .
\end{aligned}$$

So $\mathrm{st} \in AConf \to State$, $\mathrm{pp} \in AConf \to PP$, $\mathrm{arc} \in AConf \dashrightarrow AS$, $\mathrm{conf} \in AConf \to Conf$ (with $\dashrightarrow$, we denote partial functions).

In the rest of this chapter, we deal with transfinite semantics of form $\mathcal{T}_\psi$ for operators $\psi \in \mathrm{STList}\, AConf \to 1 + AConf$. Not every ψ gives rise to a reasonable semantics $\mathcal{T}_\psi$ because different augmented configurations in one list contain overlapping information which can be contradictory. To be ready for defining soundness of a given operator ψ, we at first have to be able to make clear for every looping computation which program points the loop causing this computation actually consists of. The following definition states it.

Definition 4.3.1. *Let $l \in \mathrm{TList}\, AConf \setminus \{\mathrm{nil}\}$. We call a program point p* looping *in l iff, for every ordinal $o < |l|$, there exists an ordinal ϱ, $o < \varrho < |l|$, such that $\mathrm{pp}(l(\varrho)) = p$. The set of all program points looping in l is denoted by $\mathrm{loop}\, l$.*

Clearly a computation l contains looping program points only if $|l|$ is a limit ordinal. Note that not necessarily all program points of an infinitely running while-loop are looping in this infinite computation (there can be branches of conditionals in the body of the loop being used a finite number of times only). However, the head point of an infinitely running while-loop (meaning the body being executed infinitely many times) definitely is looping.

The writing $(0 \mapsto c)$ denotes the transfinite list of length 1 whose single element is c.

Definition 4.3.2. *Let* $\psi \in \mathrm{STList}\, AConf \to 1 + AConf$.

(i) *Call the operator* ψ sound *iff, for every* $c \in AConf$*, both following conditions hold:*

1. *if* $\psi(0 \mapsto c) \neq \bot$ *then* $\mathrm{arc}(\psi(0 \mapsto c)) \notin AS_\infty$ *and* $\sigma(\mathrm{arc}(\psi(0 \mapsto c))) = \mathrm{pp}(c) \neq f$*, otherwise* $\mathrm{pp}(c) = f$*;*
2. *for every* $\tilde{c} \in AConf$, $\mathrm{conf}\,\tilde{c} = \mathrm{conf}\,c$ *implies* $\psi(0 \mapsto \tilde{c}) = \psi(0 \mapsto c)$.

(ii) *Call the operator* ψ transfinitely sound *iff it is sound and, for every* $c \in AConf$ *and selfish ordinal* γ *satisfying both* $1 < \gamma \leqslant |\mathrm{iter}\,\psi\,c|$ *and* $\gamma < \infty$*, if we denote* $l = \mathrm{take}\,\gamma(\mathrm{iter}\,\psi\,c)$ *then both following conditions hold:*

1. $\psi(l) \neq \bot$, $\mathrm{arc}(\psi(l)) \in AS_\infty$ *and* $\sigma(\mathrm{arc}(\psi(l))) \in \mathrm{loop}\,l$*;*
2. *for every* $\tilde{l} \in \mathrm{STList}\, AConf$, $\mathrm{map\,conf}\,\tilde{l} = \mathrm{map\,conf}\,l$ *implies* $\psi(\tilde{l}) = \psi(l)$.

(iii) *For every* $c \in AConf$*, define* $\mathrm{next}_\psi\, c = \psi(0 \mapsto c)$.

Soundness of ψ guarantees that the program point of any component of a list representing a computation according to ψ coincides with the source of the atomic step of the next component (provided it exists). Soundness also states that f corresponds to the finished computation. Transfinite soundness states that every endless initial part of any computation can be continued with a transfinite step and the source of this step is looping in the computation observed. These and some other facts are more precisely stated and proven in Lemma 4.3.3 below.

It is also required that ordinary steps use ordinary arcs while transfinite steps use transfinite arcs. This is a matter of simplicity. In the case of constructs like **loop** S discussed in Sect. 4.1, it is obvious that escaping the loop must involve a new arc, but it is semantically neat to demand that actually all infinite loops use some special kind of arcs for transfinite escapement even if there exists an ordinary arc between the same vertices. Hence an ordinary arc starting in a program point of predicate test is still used only if the predicate evaluated to the truth value corresponding to this arc. For while-loops, it means that the ordinary arc from the beginning point to the point immediately after the loop is dubbed with a transfinite arc between the same program points.

The second conditions of soundness and transfinite soundness together demand that the additional information in configurations in comparison to Sect. 4.1 have no influence on the computation process. The meaning of function next_ψ is the same as in Sect. 4.1. It is indexed with ψ to emphasize the dependence on ψ.

By $p \to q$, we denote that q is an immediate successor of p, i.e. there is an arc from p to q, in *TCFG*.

Lemma 4.3.3. *Let* $\psi \in \mathrm{STList}\, AConf \to 1 + AConf$.

(i) *Let* $l = \mathrm{iter}\,\psi\,c$ *for some* $c \in AConf$. *Take an ordinal* $o \leqslant |l|$ *and let* $o = \alpha + \gamma$ *be the principal representation. Then* $\mathrm{loop}\,(\mathrm{take}\,o\,l) = \mathrm{loop}\,(\mathrm{take}\,\gamma(\mathrm{drop}\,\alpha\,l))$.

(ii) *Let* $l = \mathrm{iter}\,\psi\,c$ *for some* $c \in AConf$. *For every* $o < |l|$, $l(o+1) = \mathrm{next}_\psi(l(o))$.

(iii) *Assume* ψ *being sound. For every* $c \in AConf$, $\mathrm{next}_\psi\,c = \bot$ *iff* $\mathrm{pp}\,c = f$.

(iv) *Assume* ψ *being sound. Let* $c \in AConf$ *be arbitrary. If* $\mathrm{pp}(\mathrm{next}_\psi\,c) = q$ *and* $\mathrm{pp}\,c = p$ *then* $\mathrm{arc}(\mathrm{next}_\psi\,c)$ *goes from* p *to* q *in TCFG. Moreover, if* $p \in V(\mathrm{cfg}\,S)$ *for some program* S *then* q *and* $\mathrm{arc}(\mathrm{next}_\psi\,c)$ *belong to* $\mathrm{cfg}\,S$.

(v) *Assume* ψ *being sound. Let* $l = \mathrm{iter}\,\psi\,c$ *for some* $c \in AConf$. *For every ordinal* o *with* $o + 1 < |l|$, $\mathrm{arc}(l(o+1))$ *goes from* $\mathrm{pp}(l(o))$ *to* $\mathrm{pp}(l(o+1))$ *in TCFG*.

(vi) *Assume* ψ *being transfinitely sound. Let* $l = \mathrm{iter}\,\psi\,c$ *for some* $c \in AConf$ *whereby* $|l| < \infty$. *Then* $|l| = o + 1$ *and* $\mathrm{pp}(l(o)) = f$ *for some ordinal* o.

(vii) *Assume* ψ *being transfinitely sound. Let* $l = \mathrm{iter}\,\psi\,c$ *for some* $c \in AConf$ *whereby* $|l| < \infty$. *Let* $\lambda \leqslant |l|$ *be a limit ordinal. Then* $\lambda < |l|$, $\mathrm{arc}(l(\lambda)) \in AS_\infty$ *and* $\sigma(\mathrm{arc}(l(\lambda))) \in \mathrm{loop}\,(\mathrm{take}\,\lambda\,l)$.

Proof.

(i) Assume $p \in \mathrm{loop}\,(\mathrm{take}\,o\,l)$. Take arbitrary $\pi < |\,\mathrm{take}\,\gamma(\mathrm{drop}\,\alpha\,l)| = \gamma$. Then $\alpha + \pi < \alpha + \gamma = o$ and there exists a ϱ satisfying $\alpha + \pi < \varrho < o$ such that $(\mathrm{take}\,o\,l)(\varrho) = p$. Then $\pi < \varrho - \alpha < \gamma$ and

$$(\mathrm{take}\,\gamma(\mathrm{drop}\,\alpha\,l))(\varrho - \alpha) = (\mathrm{drop}\,\alpha\,l)(\varrho - \alpha) = l(\varrho) = (\mathrm{take}\,o\,l)(\varrho) = p\ .$$

Assume now $p \in \mathrm{loop}\,(\mathrm{take}\,\gamma(\mathrm{drop}\,\alpha\,l))$. Take arbitrary $\pi < |\,\mathrm{take}\,o\,l| = o$. Define $\tilde{\pi} = \pi - \alpha$ if $\pi \geqslant \alpha$ and $\tilde{\pi} = 0$ otherwise. Then $\tilde{\pi} < \gamma$ and there exists a ϱ satisfying $\tilde{\pi} < \varrho < \gamma$ such that $(\mathrm{take}\,\gamma(\mathrm{drop}\,\alpha\,l))(\varrho) = p$. Then $\alpha < \alpha + \varrho < o$ and

$$(\mathrm{take}\,o\,l)(\alpha + \varrho) = (\mathrm{drop}\,\alpha(\mathrm{take}\,o\,l))(\varrho) = (\mathrm{take}\,\gamma(\mathrm{drop}\,\alpha\,l))(\varrho) = p\ .$$

(ii) By iterativity and Definitions 4.1.1 and 4.3.2(iii),

$$\begin{aligned} l(o+1) &= \mathrm{id}(\psi(\mathrm{take}\,1(\mathrm{drop}\,o\,l))) = \psi(\mathrm{take}\,1(\mathrm{drop}\,o\,l)) \\ &= \psi(0 \mapsto (\mathrm{drop}\,o\,l)(0)) = \psi(0 \mapsto l(o)) = \mathrm{next}_\psi(l(o))\ . \end{aligned}$$

(iii) Straightforward by Definitions 4.3.2(i) and 4.3.2(iii).

(iv) Denote $e = \mathrm{arc}(\mathrm{next}_\psi\,c) = \mathrm{arc}(\psi(0 \mapsto c))$. As $\psi(0 \mapsto c) = \mathrm{next}_\psi\,c \neq \bot$, soundness gives $\sigma(e) = \mathrm{pp}\,c = p$ and $e \notin AS_\infty$. We also have $\tau(e) = \tau(\mathrm{arc}(\mathrm{next}_\psi\,c)) = \mathrm{pp}(\mathrm{next}_\psi\,c) = q$. Therefore $p \to q$, implying together with $p \in V(\mathrm{cfg}\,S)$ that q and $\mathrm{arc}(\mathrm{next}_\psi\,c)$ both are in $\mathrm{cfg}\,S$.

(v) Straightforward by parts (ii) and (iv).

(vi) Let $|l| = \alpha + \gamma$ be the principal representation. If $\gamma > 1$ then, by iterativity and transfinite soundness used together with Corollary 3.4.8, we obtain

$$l(|l|) = \psi(\mathrm{take}\,\gamma(\mathrm{drop}\,\alpha\,l)) \neq \bot$$

which contradicts the concept of length. Hence $\gamma = 1$ and, taking $o = \alpha$, we obtain the desired form. Using part (ii), we get

$$\bot = l(|l|) = l(o+1) = \mathrm{next}_\psi(l(o)) \ ,$$

hence, by part (iii), $\mathrm{pp}(l(o)) = \mathsf{f}$.

(vii) By (vi), $|l|$ is a successor ordinal, so $\lambda = |l|$ cannot be the case. Let $\lambda = \alpha + \gamma$ be the principal representation; then $\gamma > 1$. By transfinite soundness and Corollary 3.4.8, $\mathsf{e} = \mathrm{arc}(l(\lambda)) = \mathrm{arc}(\psi(\mathrm{take}\,\gamma(\mathrm{drop}\,\alpha\,l))) \in AS_\infty$ and $\sigma(\mathsf{e})$ is looping in $\mathrm{take}\,\gamma(\mathrm{drop}\,\alpha\,l)$, hence, by (i), also in $\mathrm{take}\,\lambda\,l$. □

Traditionally, any computation with a program redounds as a walk in its control flow graph. Lemma 4.3.4 states that, in the case of transfinite semantics based on a transfinitely sound operator, a similar property holds also for transfinite control flow graphs: any transfinite computation according to a finite program S can be traced by a walk in tcfg S.

Lemma 4.3.4. *Let $\psi \in \mathrm{STList}\,AConf \to 1 + AConf$ be transfinitely sound and let $S \in Prog$ be finite. Let $l = \mathcal{T}_\psi(S)(i)$ for an $i \in State$ and let o, π be ordinals, $o \leqslant \pi < |l|$. Denote $\mathsf{p} = \mathrm{pp}(l(o))$ and $\mathsf{q} = \mathrm{pp}(l(\pi))$. Then p and q are vertices in* tcfg S*; furthermore, there exists a walk $w = (\mathsf{v}_0, \mathsf{e}_1, \mathsf{v}_1, \ldots, \mathsf{e}_n, \mathsf{v}_n)$ from p to q in* tcfg S *such that the following holds:*

1. *there exist ordinals $\varrho_1, \ldots, \varrho_n$ such that $o < \varrho_1 < \ldots < \varrho_n \leqslant \pi$ and, for every $i = 1, \ldots, n$,* $\mathrm{arc}(l(\varrho_i)) = \mathsf{e}_i$ *and* $\mathrm{pp}(l(\varrho_i)) = \mathsf{v}_i$*;*
2. *for every ordinal ϱ with $o < \varrho \leqslant \pi$, there exists an $i = 1, \ldots, n$ such that* $\mathrm{arc}(l(\varrho)) = \mathsf{e}_i$.

Proof. Argue by transfinite induction on (o, π) ordered lexicographically.

If $o = \pi = 0$ then $\mathsf{p} = \mathsf{q} = i_S$ and the desired result follows trivially (one can take the empty walk from i_S to itself).

If $0 < o = \pi$ then applying the induction hypothesis for $o \leftarrow 0$, $\pi \leftarrow o$ gives that p is a vertex in tcfg S. Thus the empty walk from p to p works.

Let finally be $0 \leqslant o < \pi$. Let $\pi - o = \alpha + \gamma$ be the principal representation of $\pi - o$ and let $\pi = \beta + \gamma$ be the principal representation of π. By iterativity, $l(\pi) = \psi(m)$ where $m = \mathrm{take}\,\gamma(\mathrm{drop}\,\beta\,l)$.

If $\gamma = 1$ then, by the induction hypothesis, find a walk v from $\mathrm{pp}(l(o))$ to $\mathrm{pp}(l(o+\alpha))$ in tcfg S which meets the two properties. By soundness,

$$\sigma(\mathrm{arc}(l(\pi))) = \sigma(\mathrm{arc}(\psi(m))) = \mathrm{pp}(\mathrm{head}\, m) = \mathrm{pp}(\mathrm{head}(\mathrm{drop}\, \beta\, l)) = \mathrm{pp}(l(\beta)) \ .$$

By Lemma 3.4.5(ii), $\beta = o + \alpha$. Thus the desired walk can be obtained by appending $\mathrm{arc}(l(\pi))$ and $\mathrm{pp}(l(\pi))$ to the end of v.

Consider the case $\gamma > 1$ now. Let M be the set of all arcs used by computation $\mathrm{drop}(o+1)(\mathrm{take}\, \pi\, l)$. The induction hypothesis implies that every arc in M is in tcfg S. By assumptions, M is finite. For every $e \in M$, let ξ_e be the least ordinal such that $\mathrm{arc}((\mathrm{drop}(o+1)(\mathrm{take}\, \pi\, l))(\xi_e)) = e$. By transfinite soundness,

$$\sigma(\mathrm{arc}(l(\pi))) = \sigma(\mathrm{arc}(\psi(m))) \in \mathrm{loop}\, m \ .$$

Find $\varrho \geqslant \beta$, $\varrho \geqslant \max_{e \in M}(o + 1 + \xi_e)$ such that $\sigma(\mathrm{arc}(l(\pi))) = \mathrm{pp}(m(\varrho - \beta))$. As $o \leqslant \varrho < \pi$, the induction hypothesis implies that there is a walk v from $\mathrm{pp}(l(o))$ to $\mathrm{pp}(l(\varrho))$ meeting the two properties. The desired walk can be obtained by appending $\mathrm{arc}(l(\pi))$ and $\mathrm{pp}(l(\pi))$ to the end of v. □

Lemma 4.3.4 implies that all executions of any fixed finite program in a transfinitely sound semantics use only finitely many program points and atomic steps altogether. The Lemma 4.3.5 states that, under similar assumptions, every endless computation has a non-empty final part during which it visits looping program points only.

Lemma 4.3.5. *Let $\psi \in \mathrm{STList}\, AConf \to 1 + AConf$ be transfinitely sound and let $S \in Prog$ be finite. Let $l = \mathcal{T}_\psi(S)(i)$ for an $i \in State$. For every ordinal o satisfying $0 < o \leqslant |l|$, there exists an ordinal $\pi < o$ such that $\mathrm{pp}(l(\varrho)) \in \mathrm{loop}(\mathrm{take}\, o\, l)$ for every ϱ satisfying $\pi < \varrho < o$.*

Proof. Let P be the set of all program points p being reachable from i_S in *TCFG* and satisfying $p \notin \mathrm{loop}(\mathrm{take}\, o\, l)$. For every $p \in P$, choose $\xi_p < o$ in such a way that $\mathrm{pp}(l(\varrho)) = p$ for no ϱ satisfying $\xi_p < \varrho < o$. This definition is sound for every p as p is not looping in $\mathrm{take}\, o\, l$ and $0 < o$.

As $P \subseteq V(\mathrm{tcfg}\, S)$, it is finite and we can find $\pi = \max_{p \in P} \xi_p < o$. Choose any ϱ such that $\pi < \varrho < o$. By construction, $\mathrm{pp}(l(\varrho)) \notin P$. By Lemma 4.3.4, $\mathrm{pp}(l(\varrho))$ is reachable from $\mathrm{pp}(l(0)) = i_S$ in *TCFG*. Thus $\mathrm{pp}(l(\varrho)) \in \mathrm{loop}(\mathrm{take}\, o\, l)$. □

We introduce some more restrictions to be imposed on semantics w.r.t. which we are going to work. Like soundness and transfinite soundness, they are formulated as properties of 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 transfinite semantics is to be able to overcome non-terminating parts of programs. This observation leads to the kind of transfinite semantics we call escaping.

Definition 4.3.6. *Call a transfinitely sound operator* $\psi \in \mathrm{STList}\, AConf \to 1 + AConf$ escaping *iff, for every* $c \in AConf$ *and selfish* γ *satisfying* $1 < \gamma < |\operatorname{iter}\psi\, c|$*, if we denote* $l = \operatorname{take}\gamma(\operatorname{iter}\psi\, c)$ *then* $\operatorname{pp}(\psi(l)) \notin \operatorname{loop} l$.

After any infinite computation according to a finite program in an escaping semantics, 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 Sect. 4.1 is obviously escaping by the definition of $\lim p$ for program point lists p.

Next we are going to prove that ω^ω is an upper bound of the lengths of transfinite computation in escaping semantics, irrespective of the language. This is achieved by Theorem 4.3.8. Almost the same result ($\omega^{\omega+1}$) was obtained by Giacobazzi and Mastroeni [5] for IMP programs.

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

Lemma 4.3.7. *Let* $\psi : \mathrm{STList}\, AConf \to 1 + AConf$ *be an escaping operator. For every natural number* k *and arbitrary* $c \in Conf$*,*

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

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

Suppose that the claim holds for k and assume

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

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

Apply Corollary 3.4.7 with $h = \operatorname{iter}\psi$, $\lambda = \omega^k$, $\mu = \omega^{k+1}$ (note that being selfish is equivalent to being a power of ω), and $\pi = \omega^{k+1}$. We obtain

$$\operatorname{take}\omega^{k+1}(\operatorname{drop}(\omega^k \cdot n)(\operatorname{iter}\psi\, c)) = \operatorname{take}\omega^{k+1}(\operatorname{iter}\psi\, d) \tag{4.2}$$

where $d = (\operatorname{iter}\psi\ ;\ \operatorname{take}\omega^k\ ;\ \psi)^n(c)$. Both sides of (4.2) are different from $\bot$ since our assumption $|\operatorname{iter}\psi\, c| \geqslant \omega^{k+1}$ implies $|\operatorname{drop}(\omega^k \cdot n)(\operatorname{iter}\psi\, c)| \geqslant \omega^{k+1}$. This allows to conclude $|\operatorname{iter}\psi\, d| \geqslant \omega^{k+1} > \omega^k$ and

$$\operatorname{take} o(\operatorname{drop}(\omega^k \cdot n)(\operatorname{iter}\psi\, c)) = \operatorname{take} o(\operatorname{iter}\psi\, d)$$

for all $o \leqslant \omega^{k+1}$. Now the induction hypothesis gives

$$\begin{aligned} &|\operatorname{occur}\big(\operatorname{take}(\omega^k+1)(\operatorname{drop}(\omega^k\cdot n)(\operatorname{iter}\psi\, c))\big)| \\ &\qquad = |\operatorname{occur}\big(\operatorname{take}(\omega^k+1)(\operatorname{iter}\psi\, d)\big)| > k\ . \end{aligned} \tag{4.3}$$

Let $m = |\operatorname{loop}\big(\operatorname{take}\omega^{k+1}(\operatorname{iter}\psi\, c)\big)|$. It is possible to find n such that the computation $\operatorname{drop}(\omega^k\cdot n)(\operatorname{take}\omega^{k+1}(\operatorname{iter}\psi\, c))$ visits these m looping program points only. Therefore $m \geqslant k+1$ since, by (4.3), the first ω^k+1 steps of this computation visit more than k program points.

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

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

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

Theorem 4.3.8. *Let $\psi \in \operatorname{STList} AConf \to 1 + AConf$ be an escaping operator and let S be a finite program. For every $i \in State$, we have $|\mathcal{T}_\psi(S)(i)| \leqslant \omega^{|V(\operatorname{cfg} S)|} < \omega^\omega$.*

Proof. By conditions, $l = \mathcal{T}_\psi(S)(s) = \operatorname{iter}\psi\langle i_S \mid s\rangle$ for some state s. Suppose $|l| > \omega^{|V(\operatorname{cfg} S)|}$. Then Lemma 4.3.7 implies that l visits more program points than there is in $\operatorname{cfg} S$ which is impossible. Hence the first inequality follows. By finiteness, $|V(\operatorname{cfg} S)| < \omega$, implying the second inequality. □

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 4.3.8 achieves the best conservative estimation common to all programs (provided our language is powerful enough to enable arbitrary finite depth of nested loops).

The following definitions refer to control dependence. This notion is well-known; however, it is also explained in Sect. 2.3 for abstract flow graphs; it is called simply dependence there.

Informally, a program point q is control dependent on p if there is a computation starting at p and finishing at f which avoids q but, after some possible atomic computation step, it reaches a program point where it is already impossible to avoid

q later. (Note that, in the case of transfinite semantics, finishing computation is not the same as terminating computation since also non-terminating computations can finish.)

For every $P \subseteq PP$, we denote by at P the predicate being true on configurations c with $\mathrm{pp}\, c \in P$. For every $X \in Var$ and $c \in AConf$, the value of X at state $\mathrm{st}\, c$ is denoted by $\mathrm{val}\, X\, c$.

Definition 4.3.9. *Let* $\psi \in \mathrm{STList}\, AConf \to 1 + AConf$.

(i) *Call the operator* ψ regular *iff* ψ *is transfinitely sound and, for every* $c \in AConf$ *and selfish* γ *satisfying* $1 < \gamma < |\mathrm{iter}\, \psi\, c|$, *if* $l = \mathrm{take}\, \gamma(\mathrm{iter}\, \psi\, c)$ *then there is exactly one program point* $p \in \mathrm{loop}\, l$ *with transfinite arc from* p *to* $\mathrm{pp}(\psi(l))$ *and* $\mathrm{pp}(\psi(l))$ *postdominates all program points in* $\mathrm{loop}\, l$.

(ii) *Call the operator* ψ intuitive *iff both following conditions hold:*

1. *for every* $c, d \in AConf$, *selfish ordinals* γ, δ *satisfying* $1 < \gamma < |\mathrm{iter}\, \psi\, c|$, $1 < \delta < |\mathrm{iter}\, \psi\, d|$ *and* $X \in Var$, *if we denote* $l = \mathrm{take}\, \gamma(\mathrm{iter}\, \psi\, c)$, $k = \mathrm{take}\, \delta(\mathrm{iter}\, \psi\, d)$ *then*

$$\begin{aligned}&\mathrm{map}(\mathrm{val}\, X)(\mathrm{filter}(\mathrm{at}\, \{\sigma(\mathrm{arc}(\psi(l)))\})\, l)\\ &\quad = \mathrm{map}(\mathrm{val}\, X)(\mathrm{filter}(\mathrm{at}\, \{\sigma(\mathrm{arc}(\psi(k)))\})\, k)\end{aligned}$$

   implies $\mathrm{val}\, X(\psi(l)) = \mathrm{val}\, X(\psi(k))$*;*

2. *for every* $c \in AConf$, *selfish ordinal* γ *satisfying* $1 < \gamma < |\mathrm{iter}\, \psi\, c|$, $X \in Var$ *and* $v \in Val$, *if we denote* $l = \mathrm{take}\, \gamma(\mathrm{iter}\, \psi\, c)$ *then if there exists an* $o < |l|$ *such that* $\mathrm{val}\, X(l(\pi)) = v$ *for every* $\pi > o$, $\pi < |l|$, *then* $\mathrm{val}\, X(\psi(l)) = v$.

By construction of the transfinite semantics of while-loops in Sect. 4.1, that semantics is regular. Note that every regular operator is escaping since all postdominators of a vertex differ from it (Theorem 2.2.4).

The notion of intuitivity (Definition 4.3.9(ii)) formalizes two natural desires about limits of endless sequences of states. The first is that the limit value of every variable depends only on the values this variable possesses at program point where the transfinite arc escapes the loop. This is what Example 4.1.2 suggested and what we demanded in our semantics for while-loops in Sect. 4.1. The second is that if the sequence of values of a variable stabilizes then the limit equals to this stable value — again a condition demanded for while-loops. In other words, it states that, if the value of a variable is changed, this must be caused by a finite atomic step.

This way, all the properties of operators ψ defined in this section have been chosen having in mind the transfinite semantics for while-loops constructed in Sect. 4.1. The facts proven about semantics possessing these properties apply to other semantics inasmuch as they behave similarly.

Lemma 4.3.10. *Let* $\psi : \mathrm{STList}\, \mathit{AConf} \to 1 + \mathit{AConf}$ *be an operator.*

(i) *Let* ψ *be regular and* $l = \mathrm{iter}\, \psi\, c$ *for a* $c \in \mathit{AConf}$*. For every limit ordinal* $\lambda < |l|$*, there is exactly one program point* $p \in \mathrm{loop}(\mathrm{take}\, \lambda\, l)$ *with transfinite arc from* p *to* $\mathrm{pp}(l(\lambda))$ *and* $\mathrm{pp}(l(\lambda))$ *postdominates every program point in* $\mathrm{loop}(\mathrm{take}\, \lambda\, l)$*.*

(ii) *Let* ψ *be intuitive and* $l = \mathrm{iter}\, \psi\, c$ *for a* $c \in \mathit{AConf}$*. For every limit ordinal* $\lambda < |l|$ *and* $X \in \mathit{Var}$*, if* $\mathrm{map}(\mathrm{val}\, X)(\mathrm{take}\, \lambda\, l)$ *stabilizes to* v *then* $\mathrm{val}\, X(l(\lambda)) = v$*.*

(iii) *Let* ψ *be an intuitive limit operator and* $l = \mathrm{iter}\, \psi\, c$*,* $k = \mathrm{iter}\, \psi\, d$ *for* $c, d \in \mathit{AConf}$*. For every* $X \in \mathit{Var}$*, limit ordinals* $\lambda < |l|$*,* $\kappa < |k|$*, and ordinals* α*,* β *such that* $\lambda - \alpha$ *and* $\kappa - \beta$ *both are selfish limit ordinals, if*

$$\begin{aligned}&\mathrm{map}(\mathrm{val}\, X)(\mathrm{filter}(\mathrm{at}\, \{\sigma(\mathrm{arc}(l(\lambda)))\})(\mathrm{drop}\, \alpha(\mathrm{take}\, \lambda\, l)))\\ &\quad = \mathrm{map}(\mathrm{val}\, X)(\mathrm{filter}(\mathrm{at}\, \{\sigma(\mathrm{arc}(k(\kappa)))\})(\mathrm{drop}\, \beta(\mathrm{take}\, \kappa\, k)))\end{aligned}$$

then $\mathrm{val}\, X(l(\lambda)) = \mathrm{val}\, X(k(\kappa))$*.*

Proof.

(i) Straightforward by regularity and Corollary 3.4.8.

(ii) Straightforward by intuitivity and Corollary 3.4.8.

(iii) Denote $\gamma = \lambda - \alpha$, $\delta = \kappa - \beta$ and $\tilde{l} = \mathrm{drop}\, \alpha(\mathrm{take}\, \lambda\, l) = \mathrm{take}\, \gamma(\mathrm{drop}\, \alpha\, l)$, $\tilde{k} = \mathrm{drop}\, \beta(\mathrm{take}\, \kappa\, l) = \mathrm{take}\, \delta(\mathrm{drop}\, \beta\, k)$. By Corollary 3.4.8, $\mathrm{drop}\, \alpha\, l = \mathrm{iter}\, \psi\, x$ and $\mathrm{drop}\, \beta\, k = \mathrm{iter}\, \psi\, y$ for some $x, y \in \mathit{AConf}$, therefore $\tilde{l} = \mathrm{take}\, \gamma(\mathrm{iter}\, \psi\, x)$ and $\tilde{k} = \mathrm{take}\, \delta(\mathrm{iter}\, \psi\, y)$. Thus

$$\begin{aligned}l(\lambda) &= (\mathrm{drop}\, \alpha\, l)(\gamma) = (\mathrm{iter}\, \psi\, x)(\gamma) = \psi(\mathrm{take}\, \gamma(\mathrm{iter}\, \psi\, x)) = \psi(\tilde{l})\ ,\\ k(\kappa) &= (\mathrm{drop}\, \beta\, k)(\delta) = (\mathrm{iter}\, \psi\, y)(\delta) = \psi(\mathrm{take}\, \delta(\mathrm{iter}\, \psi\, y)) = \psi(\tilde{k})\ .\end{aligned}$$

Hence, by intuitivity, $\mathrm{val}\, X(\psi(\tilde{l})) = \mathrm{val}\, X(\psi(\tilde{k}))$, implying the desired claim. □

By $p < q$, we denote that q postdominates p in *TCFG*. By Theorem 2.2.4, $<$ is a strict order on program points. The corresponding non-strict order is denoted by $\leqslant$.

Lemma 4.3.11. *Let* $\psi \in \mathrm{STList}\, \mathit{AConf} \to 1 + \mathit{AConf}$ *be transfinitely sound and let* $S \in \mathit{Prog}$ *be finite. Let* $l = \mathcal{T}_\psi(S)(i)$ *for an* $i \in \mathit{State}$ *and let* $\mathrm{pp}(l(o)) = p \in \mathit{PP}$ *for an ordinal* o*. Let* q*,* r *be postdominators of* p *both visited by* $m = \mathrm{drop}\, o\, l$*. If* $q < r$ *then the first visit of* q *by* m *occurs before the first visit of* r*.*

Proof. Let ϱ be the least ordinal for which $\mathrm{pp}(m(\varrho)) = r$; then

$$r = \mathrm{pp}((\mathrm{drop}\, o\, l)(\varrho)) = \mathrm{pp}(l(o + \varrho))\ .$$

By Lemma 4.3.4, there exists a walk w from p to r in *TCFG* using only the arcs occurring in the list $\mathrm{drop}(o+1)(\mathrm{take}(o+\varrho+1)\,l)$. As $q < r$ and both q and r postdominate p, walk w passes through q (Theorem 2.2.9). Thus we find an ordinal π such that $\pi+1 < |\,\mathrm{drop}(o+1)(\mathrm{take}(o+\varrho+1)\,l)| = (o+\varrho+1)-(o+1) = \varrho+1-1 = \varrho-1+1$ and

$$\begin{aligned} q &= \mathrm{pp}((\mathrm{drop}(o+1)(\mathrm{take}(o+\varrho+1)\,l))(\pi)) \\ &= \mathrm{pp}((\mathrm{take}(o+\varrho+1)\,l)(o+1+\pi)) \\ &= \mathrm{pp}(l(o+1+\pi)) \\ &= \mathrm{pp}((\mathrm{drop}\,o\,l)(1+\pi)) \\ &= \mathrm{pp}(m(1+\pi))\ . \end{aligned}$$

Hence m visits q before r as $\pi+1 < \varrho-1+1$ gives $\pi < \varrho-1$ and $1+\pi < \varrho$. □

Lemma 4.3.12. *Let $\psi \in \mathrm{STList}\,AConf \to 1 + AConf$ be regular and let $S \in Prog$ be regular. Let $l = \mathcal{T}_\psi(S)(i)$ for an $i \in State$. Take $\pi < |l|$ and assume* $\mathrm{arc}(l(\pi))$ *going from p to q. Then every program point in* $\mathrm{loop}\,(\mathrm{take}\,\pi\,l)$ *is transitively control dependent on p.*

Proof. If π is a successor ordinal, the result holds vacuously as $\mathrm{loop}\,(\mathrm{take}\,\pi\,l) = \varnothing$. Assume therefore π being a limit ordinal. Then, by transfinite soundness, $\mathrm{arc}(l(\pi)) \in AS_\infty$ and $p \in \mathrm{loop}\,(\mathrm{take}\,\pi\,l)$.

By Lemma 4.3.5, there exists an $o < \pi$ such that $\mathrm{pp}(l(o)) = p$ and $\mathrm{pp}(l(\varrho)) \in \mathrm{loop}\,(\mathrm{take}\,\pi\,l)$ for every ϱ satisfying $o \leqslant \varrho < \pi$. By Lemma 4.3.4, find a walk w from p to q containing precisely the arcs used by $\mathrm{drop}(o+1)(\mathrm{take}(\pi+1)\,l)$.

As S is regular, q immediately postdominates p. As ψ is regular, q postdominates all program points in $\mathrm{loop}\,(\mathrm{take}\,\pi\,l)$, i.e. the program points visited by $\mathrm{drop}\,o(\mathrm{take}\,\pi\,l)$. Thus q does not occur in w except at the end. So w is a walk from point p to its immediate postdominator whereby no intermediate program points postdominate p. Thus all intermediate program points are transitively control dependent on p (Theorem 2.3.5). By construction, w passes through all program points in $\mathrm{loop}\,(\mathrm{take}\,\pi\,l)$. Hence the claim follows. □

4.4 Data Flow Approximation

Definition 4.4.1. *Let $\psi \in \mathrm{STList}\,AConf \to 1 + AConf$ be a sound operator. For every arc $e \in AS$, let* $\mathrm{def}_\psi\, e \subseteq Var$ *be given by*

$$X \in \mathrm{def}_\psi\, e \iff \exists c \in AConf\ \big(\mathrm{arc}(\mathrm{next}_\psi\, c) = e \ \wedge\ \mathrm{val}\,X(\mathrm{next}_\psi\, c) \neq \mathrm{val}\,X\,c\big)\ .$$

Informally, the set $\mathrm{def}_\psi\, e$ consists of all variables whose value can be affected by the atomic computation step e. The definition implies that if $e \in AS_\infty$ then $\mathrm{def}_\psi\, e = \varnothing$.

Lemma 4.4.2. *Let $\psi \in \mathrm{STList}\, AConf \to 1 + AConf$ be a sound intuitive operator. Let $l = \mathrm{iter}\, \psi\, c$ for some $c \in AConf$ and let ordinals o, π be such that $o \leqslant \pi < |l|$. Let $X \in Var$ be such that $X \in \mathrm{def}_\psi(\mathrm{arc}(l(\varrho)))$ for no ordinal ϱ satisfying $o < \varrho \leqslant \pi$. Then $\mathrm{val}\, X(l(\varrho)) = \mathrm{val}\, X(l(o))$ for every ordinal ϱ satisfying $o \leqslant \varrho \leqslant \pi$.*

Proof. Suppose the contrary. Let ϱ be the least ordinal not less than o such that $\mathrm{val}\, X(l(\varrho)) \neq \mathrm{val}\, X(l(o))$. Clearly $o < \varrho$.

Consider the case $\varrho = \alpha + 1$ for some α. Then $o \leqslant \alpha$, giving

$$\mathrm{val}\, X(\mathrm{next}_\psi(l(\alpha))) = \mathrm{val}\, X(l(\varrho)) \neq \mathrm{val}\, X(l(\alpha)) \ ,$$

hence $X \in \mathrm{def}_\psi(\mathrm{arc}(\mathrm{next}_\psi(l(\alpha)))) = \mathrm{def}_\psi(\mathrm{arc}(l(\varrho)))$. As $o < \varrho \leqslant \pi$, this contradicts the assumption.

It remains to study the case ϱ being a limit ordinal. By the choice of ϱ, the sequence of values of X in computation $\mathrm{take}\, \varrho\, l$ stabilizes to $v = \mathrm{val}\, X(l(o))$. By intuitivity, $\mathrm{val}\, X(l(\varrho)) = v$. This contradicts the choice of ϱ. □

Denote by dep the control dependence relation in *TCFG*, i.e. $r\ \mathrm{dep}\ q$ means that q is control dependent on r in *TCFG*.

For a binary relation P on a set A, denote by P^* the reflexive transitive closure of P. For a binary relation P between sets A and B, let $P^{\rightarrowtail} \in A \to \wp(B)$ be the function defined by $b \in P^{\rightarrowtail}(a) \iff a\ P\ b$. The part of state s incorporating only variables in set $\mathcal{X} \subseteq Var$ is denoted by $s\big|_{\mathcal{X}}$.

Next we define the approximating def-sets and ref-sets of program points. Our treatment is more scrupulous than usual ones: we take into account, for example, that computation of the values of different variables at the same program point may refer to different sets of variables.

Definition 4.4.3. *Let $\psi \in \mathrm{STList}\, AConf \to 1 + AConf$ be a sound operator. We call a pair* $(\mathrm{def}_\vee, \mathrm{ref})$ data-flow approximation system for ψ *iff*

$$\mathrm{def}_\vee \in AS \to \wp(Var) \ \textit{and}\ \ \mathrm{ref} \in \left(\sum_{e \in AS} \mathrm{def}_\vee\, e + \sum_{p \in PP} \mathrm{dep}^{\rightarrowtail}(p) \right) \to \wp(Var)$$

satisfy the following conditions:

1. *for every* $e \in AS$,

$$\mathrm{def}_\psi\, e \subseteq \mathrm{def}_\vee\, e \;\wedge\; (e \in AS_\infty \;\Rightarrow\; \mathrm{def}_\vee\, e = \varnothing) \;;$$

2. *for every* $e \in AS$, $X \in \mathrm{def}_\vee\, e$ *and* $c, d \in AConf$ *with* $\mathrm{arc}(\mathrm{next}_\psi\, c) = \mathrm{arc}(\mathrm{next}_\psi\, d) = e$,

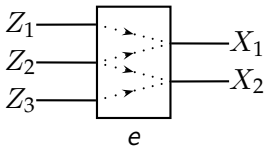
$$\mathrm{st}\, c\big|_{\mathrm{ref}(e,X)} = \mathrm{st}\, d\big|_{\mathrm{ref}(e,X)} \;\Rightarrow\; \mathrm{val}\, X(\mathrm{next}_\psi\, c) = \mathrm{val}\, X(\mathrm{next}_\psi\, d) \;;$$

3. *for every* $p \in PP$, $q \in \mathrm{dep}^{\rightarrowtail}(p)$ *and* $c, d \in AConf$ *with* $\mathrm{pp}\, c = \mathrm{pp}\, d = p$,

$$\mathrm{st}\, c\big|_{\mathrm{ref}(p,q)} = \mathrm{st}\, d\big|_{\mathrm{ref}(p,q)} \;\Rightarrow\; \big(\mathrm{pp}(\mathrm{next}_\psi\, c) \leqslant q \iff \mathrm{pp}(\mathrm{next}_\psi\, d) \leqslant q\big) \;.$$

The set $\mathrm{def}_\vee\, p$ is an upper approximation to $\mathrm{def}_\psi\, p$ for every p. These are needed because, in real situation, we are able to compute conservative approximations only. Checking whether the value of a variable can change at a certain program point is generally undecidable. The statement $Z \in \mathrm{def}_\vee\, e$ could be read as "Z may be updated by the atomic computation step e".
The domain of ref, $\sum_{e \in AS} \mathrm{def}_\vee\, e + \sum_{p \in PP} \mathrm{dep}^{\rightarrowtail}(p)$, consists of pairs (e, X) with $e \in AS$, $X \in \mathrm{def}_\vee\, e$ and pairs (p, q) with $p \in PP$, $q \in \mathrm{dep}^{\rightarrowtail}(p)$. The statement $Z \in \mathrm{ref}(e, X)$ can be read as "the value of Z may influence the value of X in the atomic computation step e". The condition in the definition tells that if the differences between states at $\sigma(e)$ remain outside $\mathrm{ref}(e, X)$ then they do not affect the value of X after the atomic computation step e on these states. Similarly, $Z \in \mathrm{ref}(p, q)$ can be read as "the value of Z at p may decide whether control reaches q". It is defined for branching program points p only as no program point can be control dependent on a non-branching point. The condition in the definition tells that if the differences between given states at p remain outside $\mathrm{ref}(p, q)$ then control, when starting from p, certainly reaches q either in both cases or in no case.

Example 4.4.4.



This figure illustrates an atomic step e which refers to variables Z_1, Z_2, Z_3 and updates variables X_1, X_2, whereby the new value of X_1 is computed by Z_1 and Z_2 only and the new value of X_2 is computed by Z_2 and Z_3 only. In this situation, one may take $\mathrm{ref}(e, X_1) \supseteq \{Z_1, Z_2\}$ and $\mathrm{ref}(e, X_2) \supseteq \{Z_2, Z_3\}$. □

Lemma 4.4.5. *Let $\psi \in \mathrm{STList}\, AConf \to 1+AConf$ be a sound operator and let* $(\mathrm{def}_\vee, \mathrm{ref})$ *be a data flow approximation system for it. Let $e \in AS$ and take $c, d \in AConf$ such that* $\mathrm{arc}(\mathrm{next}_\psi\, c) = \mathrm{arc}(\mathrm{next}_\psi\, d) = e$, $\mathrm{st}\, c\big|_{\mathrm{ref}(e,X)} = \mathrm{st}\, d\big|_{\mathrm{ref}(e,X)}$ *for every $X \in \mathrm{def}_\vee\, e$, and* $\mathrm{st}\, c\big|_{Var\setminus \mathrm{def}_\vee\, e} = \mathrm{st}\, d\big|_{Var\setminus \mathrm{def}_\vee\, e}$. *Then* $\mathrm{val}\, X(\mathrm{next}_\psi\, c) = \mathrm{val}\, X(\mathrm{next}_\psi\, d)$ *for every $X \in Var$.*

Proof. If $X \in \mathrm{def}_\vee\, e$ then $\mathrm{st}\, c\big|_{\mathrm{ref}(e,X)} = \mathrm{st}\, d\big|_{\mathrm{ref}(e,X)}$ by assumption and, by Definition 4.4.3, $\mathrm{val}\, X(\mathrm{next}_\psi\, c) = \mathrm{val}\, X(\mathrm{next}_\psi\, d)$. If $X \notin \mathrm{def}_\vee\, e$ then, by Definition 4.4.1, $\mathrm{val}\, X(\mathrm{next}_\psi\, c) = \mathrm{val}\, X\, c = \mathrm{val}\, X\, d = \mathrm{val}\, X(\mathrm{next}_\psi\, d)$. □

Lemma 4.4.6. *Let $\psi \in \mathrm{STList}\, AConf \to 1 + AConf$ be a sound operator and let* $(\mathrm{def}_\vee, \mathrm{ref})$ *be a data flow approximation system for it. Let p be a program point and take $c, d \in AConf$ such that* $\mathrm{pp}\, c = \mathrm{pp}\, d = p$ *and* $\mathrm{st}\, c\big|_{\mathrm{ref}(p,q)} = \mathrm{st}\, d\big|_{\mathrm{ref}(p,q)}$ *for every $q \in \mathrm{dep}^{\rightarrowtail}(p)$ with $p \to q$. Then* $\mathrm{pp}(\mathrm{next}_\psi\, c) = \mathrm{pp}(\mathrm{next}_\psi\, d)$.

Proof. Denote $r_1 = \mathrm{pp}(\mathrm{next}_\psi\, c)$ and $r_2 = \mathrm{pp}(\mathrm{next}_\psi\, d)$. Clearly $p \to r_1$ and $p \to r_2$. If neither of r_1 and r_2 is control dependent on p then $r_1 = r_2$ (Proposition 2.3.6(ii)).

Consider the case where one of them, say r_2, is control dependent on p. Then $\mathrm{st}\, c\big|_{\mathrm{ref}(p,r_2)} = \mathrm{st}\, d\big|_{\mathrm{ref}(p,r_2)}$ by assumption and $r_1 \leqslant r_2$ by Definition 4.4.3. It follows that also r_1 is control dependent on p since otherwise $p < r_1 \leqslant r_2$ (Proposition 2.3.6(i)) contradicting the assumption $p\ \mathrm{dep}\ r_2$. Thus $\mathrm{st}\, c\big|_{\mathrm{ref}(p,r_1)} = \mathrm{st}\, d\big|_{\mathrm{ref}(p,r_1)}$ by assumption and $r_2 \leqslant r_1$ by Definition 4.4.3. Consequently, $r_1 = r_2$. □

4.5 Program Approximation

In the rest, we often need the notion of isomorphism for flow graphs.

Definition 4.5.1. *Let $(G, i_G), (H, i_H)$ be local flow graphs (either transfinite or not). A bijection $\overline{\cdot}$ both between vertices of G and H and between arcs of G and H is called* isomorphism *iff all the following holds:*

1. $\overline{\sigma(e)} = \sigma(\overline{e})$ *and* $\overline{\tau(e)} = \tau(\overline{e})$ *for every $e \in E(G)$;*
2. $\overline{f} = f$;
3. $\overline{i_G} = i_H$;
4. *transfinite arcs and only these are mapped to transfinite arcs.*

Denote this situation $\overline{\cdot} \in (G, i_G) \to (G, i_H)$*. In the case an isomorphism exists, the graphs are called* isomorphic.

For any program S, denote by $\mathrm{aconf}\, S$ the set of all augmented configurations of form $\langle \mathsf{a} \mid s \rangle$ with either $\mathsf{a} \in V(\mathrm{tcfg}\, S)$ or $\mathsf{a} \in E(\mathrm{tcfg}\, S)$. If f is any operator on vertices and arcs of *TCFG* then we extend it to vertex sets and augmented configurations naturally by letting $f(\mathsf{A}) = \{\mathsf{a} \mid \mathsf{a} \in \mathsf{A} \bullet f(\mathsf{a})\}$ and $f\langle \mathsf{a} \mid s \rangle = \langle f(\mathsf{a}) \mid s \rangle$.

Definition 4.5.2. *Let* $\psi \in \mathrm{STList}\, \mathit{AConf} \to 1 + \mathit{AConf}$ *be a transfinitely sound operator and let* $D = (\mathrm{def}_\vee, \mathrm{ref})$ *be a data flow approximation system for it. Let* S *be a program.*

(i) *A pair* $(\mathsf{S}, \mathrm{rel})$ *is called* relevance system of S *w.r.t.* D *iff* $\mathsf{S} \subseteq V(\mathrm{tcfg}\, S)$ *and* $\mathrm{rel} \in V(\mathrm{tcfg}\, S) \to \wp(\mathit{Var})$ *such that the following holds:*

1. $\mathsf{f} \in \mathsf{S}$*;*
2. $\mathrm{rel}\, \mathsf{q} \setminus \mathrm{def}_\vee\, \mathsf{e} \subseteq \mathrm{rel}\, \mathsf{p}$ *for every* $\mathsf{e} \in E(\mathrm{tcfg}\, S)$*,* $\mathsf{p} = \sigma(\mathsf{e})$*,* $\mathsf{q} = \tau(\mathsf{e})$*;*
3. *if* $\mathsf{p} \in V(\mathrm{tcfg}\, S)$ *and* $X \in \mathrm{def}_\vee\, \mathsf{e} \cap \mathrm{rel}\, \mathsf{q}$ *for* $\mathsf{e} \in E(\mathrm{tcfg}\, S)$ *and* $\mathsf{p} = \sigma(\mathsf{e})$*,* $\mathsf{q} = \tau(\mathsf{e})$*, then* $\mathsf{p} \in \mathsf{S}$ *and* $\mathrm{ref}(\mathsf{e}, X) \subseteq \mathrm{rel}\, \mathsf{p}$*;*
4. *if* $\mathsf{p} \in V(\mathrm{tcfg}\, S)$*,* $\mathsf{q} \in \mathsf{S}$ *and* $\mathsf{p}\ \mathrm{dep}\ \mathsf{q}$ *then* $\mathsf{p} \in \mathsf{S}$ *and* $\mathrm{ref}(\mathsf{p}, \mathsf{q}) \subseteq \mathrm{rel}\, \mathsf{p}$*.*

(ii) *Take* $\mathsf{S} \subseteq V(\mathrm{tcfg}\, S)$ *and let* $\overline{S}$ *be a program. We say that* $\overline{S}$ approximates S on base S and D *iff there exists an isomorphism* $\overline{\cdot} \in \mathrm{tcfg}\, S \to \mathrm{tcfg}\, \overline{S}$ *such that the following holds:*

1. *for every* $c \in \mathrm{aconf}\, S$*, if* $\mathrm{pp}\, c \in \mathsf{S}$ *then* $\mathrm{next}_\psi\, \overline{c} = \overline{\mathrm{next}_\psi\, c}$*;*
2. *for every* $\mathsf{e} \in E(\mathrm{tcfg}\, S)$*, always* $\mathrm{def}_\vee\, \overline{\mathsf{e}} \subseteq \mathrm{def}_\vee\, \mathsf{e}$ *and if* $\sigma(\mathsf{e}) \in \mathsf{S}$ *then also* $\mathrm{def}_\vee\, \mathsf{e} \subseteq \mathrm{def}_\vee\, \overline{\mathsf{e}}$*.*

A claim $Z \in \mathrm{rel}\, \mathsf{p}$ can be read as "Z is relevant at p". A relevance system consists of a possible result of Relevant Sets analysis on S together with the corresponding slice of S; this analysis forms the basis of a classic way of automatic program slicing (see, e.g., Binkley and Gallagher [2] or Sect. 4.8 of this thesis).

Program approximation, i.e. finding a program approximating a given program, is the transformation we consider as the first step in program slicing where irrelevant statements have been replaced with other irrelevant statements. "Irrelevance" of a statement is equivalent to starting from a point outside S here. Item 2 of Definition 4.5.2(ii) requires that a step $\overline{\mathsf{e}}$ may update only variables potentially updated also by e, hence the irrelevance of the updates by e implies the irrelevance of the updates by $\overline{\mathsf{e}}$.

Proposition 4.5.3. *Let $\psi \in \mathrm{STList}\, AConf \to 1{+}AConf$ be a transfinitely sound operator and let $D = (\mathrm{def}_\vee, \mathrm{ref})$ be a data flow approximation system for it. Let (S, rel) be a relevance system for a program S w.r.t. D.*

(i) *S approximates S on base S and D.*

(ii) *Let $\overline{S}$ approximate S on base S and D with isomorphism $\overline{\cdot}$. For every $e \in E(\mathrm{tcfg}\, S)$, if $\sigma(e) \in \mathsf{S}$ then $\mathrm{def}_\psi\, \overline{e} = \mathrm{def}_\psi\, e$.*

Proof.

(i) Take $\overline{p} = p$ and $\overline{e} = e$ for each $p \in V(\mathrm{tcfg}\, S)$ and $e \in E(\mathrm{tcfg}\, S)$. Then $\overline{f} = f$ and $\overline{i_S} = i_S$ like required. The other two conditions also hold trivially.

(ii) Let $e \in E(\mathrm{tcfg}\, S)$ such that $\sigma(e) \in \mathsf{S}$ be fixed. Choose an arbitrary $c \in \mathrm{aconf}\, S$ such that $\mathrm{pp}\, c = \sigma(e)$. By condition 1 of program approximation (Definition 4.5.2(ii)), $\mathrm{next}_\psi\, \overline{c} = \overline{\mathrm{next}_\psi\, c}$, hence $\mathrm{arc}(\mathrm{next}_\psi\, \overline{c}) = \overline{\mathrm{arc}(\mathrm{next}_\psi\, c)}$ and $\mathrm{st}(\mathrm{next}_\psi\, \overline{c}) = \mathrm{st}(\mathrm{next}_\psi\, c)$. The former equality gives $\mathrm{arc}(\mathrm{next}_\psi\, c) = e \iff \mathrm{arc}(\mathrm{next}_\psi\, \overline{c}) = \overline{e}$ and the latter gives $\mathrm{val}\, X(\mathrm{next}_\psi\, \overline{c}) = \mathrm{val}\, X(\mathrm{next}_\psi\, c)$ for all $X \in \mathit{Var}$ implying further $\mathrm{val}\, X(\mathrm{next}_\psi\, c) = \mathrm{val}\, X\, c \iff \mathrm{val}\, X(\mathrm{next}_\psi\, \overline{c}) = \mathrm{val}\, X\, \overline{c}$. Altogether, this shows that, for all $X \in \mathit{Var}$, the claims

$$\forall c \in \mathrm{aconf}\, S\, \big(\mathrm{pp}\, c = \sigma(e) \wedge \mathrm{arc}(\mathrm{next}_\psi\, c) = e \;\Rightarrow\; \mathrm{val}\, X(\mathrm{next}_\psi\, c) = \mathrm{val}\, X\, c\big)\ ,$$
$$\forall c \in \mathrm{aconf}\, S\, \big(\mathrm{pp}\, c = \sigma(e) \wedge \mathrm{arc}(\mathrm{next}_\psi\, \overline{c}) = \overline{e} \;\Rightarrow\; \mathrm{val}\, X(\mathrm{next}_\psi\, \overline{c}) = \mathrm{val}\, X\, \overline{c}\big)$$

are equivalent. As $\mathrm{pp}\, c = \sigma(e) \iff \mathrm{pp}\, \overline{c} = \sigma(\overline{e})$ and $\overline{\cdot}$ is a bijection between $\mathrm{aconf}\, S$ and $\mathrm{aconf}\, \overline{S}$, we obtain that the claims

$$\forall c \in \mathrm{aconf}\, S\, \big(\mathrm{pp}\, c = \sigma(e) \wedge \mathrm{arc}(\mathrm{next}_\psi\, c) = e \;\Rightarrow\; \mathrm{val}\, X(\mathrm{next}_\psi\, c) = \mathrm{val}\, X\, c\big)\ ,$$
$$\forall c \in \mathrm{aconf}\, \overline{S}\, \big(\mathrm{pp}\, c = \sigma(\overline{e}) \wedge \mathrm{arc}(\mathrm{next}_\psi\, c) = \overline{e} \;\Rightarrow\; \mathrm{val}\, X(\mathrm{next}_\psi\, c) = \mathrm{val}\, X\, c\big)$$

are also equivalent. Note that, for every $c \in \mathit{AConf}$ such that $\mathrm{pp}\, c = \sigma(e)$, there exists a $d \in \mathrm{aconf}\, S$ such that $\mathrm{pp}\, d = \sigma(e)$ and $\mathrm{st}\, d = \mathrm{st}\, c$ (we can simply choose $\sigma(e) \in \mathit{AS}$ to be the first component). By soundness, $\mathrm{next}_\psi\, d = \mathrm{next}_\psi\, c$ in such case. An analogous observation can be carried on for $\overline{e}$ and $\mathrm{aconf}\, \overline{S}$. It follows that the claims

$$\forall c \in \mathit{AConf}\, \big(\mathrm{pp}\, c = \sigma(e) \wedge \mathrm{arc}(\mathrm{next}_\psi\, c) = e \;\Rightarrow\; \mathrm{val}\, X(\mathrm{next}_\psi\, c) = \mathrm{val}\, X\, c\big)\ ,$$
$$\forall c \in \mathit{AConf}\, \big(\mathrm{pp}\, c = \sigma(\overline{e}) \wedge \mathrm{arc}(\mathrm{next}_\psi\, c) = \overline{e} \;\Rightarrow\; \mathrm{val}\, X(\mathrm{next}_\psi\, c) = \mathrm{val}\, X\, c\big)$$

are equivalent as equivalents to the corresponding claims from the previous pair. Note further that, by soundness,

$$\mathrm{arc}(\mathrm{next}_\psi\, c) = e \;\Rightarrow\; \mathrm{pp}\, c = \sigma(e) \quad \text{and} \quad \mathrm{arc}(\mathrm{next}_\psi\, c) = \overline{e} \;\Rightarrow\; \mathrm{pp}\, c = \sigma(\overline{e})\ .$$

So we can omit the left components of the premise conjunctions. By Definition 4.4.1, the result tells that $X \notin \mathrm{def}_\psi\, e \iff X \notin \mathrm{def}_\psi\, \overline{e}$. Hence $\mathrm{def}_\psi\, e = \mathrm{def}_\psi\, \overline{e}$. □

Lemma 4.5.4. *Let $\psi \in \mathrm{STList}\, AConf \to 1 + AConf$ be a transfinitely sound operator and let $D = (\mathrm{def}_\vee, \mathrm{ref})$ be a data flow approximation system for it. Let S be a program and (S, rel) be its relevance system w.r.t. D. Let $\overline{S}$ approximate S on base S and D with isomorphism $\overline{\cdot}$. Let $p, q \in V(\mathrm{tcfg}\, S)$ and let $w = (v_0, e_1, v_1, \ldots, e_n, v_n)$, $n > 0$, be a walk from p to q in $\mathrm{tcfg}\, S$. Let $X \in \mathrm{rel}\, q$. Let $i \in \mathbb{N}$ be the least number such that $X \in \mathrm{def}_\vee \overline{e_k}$ for no k with $i+1 < k \leqslant n$. Then $X \in \mathrm{rel}\, v_{i+1}$ and i is also the least natural number such that $X \in \mathrm{def}_\vee e_k$ for no k with $i+1 < k \leqslant n$. Moreover, if $X \in \mathrm{def}_\vee e_{i+1}$ then $v_i \in S$.*

Proof. Let $j \in \mathbb{N}$ be the least number such that $X \in \mathrm{def}_\vee e_k$ for no k with $j+1 < k \leqslant n$. An easy induction shows that $X \in \mathrm{rel}\, v_{j+1}$ (using the assumption $X \in \mathrm{rel}\, v_n$ as base case; the step follows from $\mathrm{rel}\, v_{s+1} \setminus \mathrm{def}_\vee e_{s+1} \subseteq \mathrm{rel}\, v_s$ holding for all s by Definition 4.5.2(i)).

For every k with $j+1 < k \leqslant n$, we have $X \notin \mathrm{def}_\vee \overline{e_k}$ since $\mathrm{def}_\vee e_k \supseteq \mathrm{def}_\vee \overline{e_k}$ by Definition 4.5.2(ii). Hence $i \leqslant j$.

If $X \in \mathrm{def}_\vee e_{j+1}$ then $X \in \mathrm{def}_\vee e_{j+1} \cap \mathrm{rel}\, v_{j+1}$ implying $v_j \in S$ by Definition 4.5.2(i). But then $X \in \mathrm{def}_\vee e_{j+1} = \mathrm{def}_\vee \overline{e_{j+1}}$ by Definition 4.5.2(ii) giving $j \leqslant i$.

If $X \notin \mathrm{def}_\vee e_{j+1}$ then, by construction, $j = 0$ which also leads to $j \leqslant i$. Altogether, $i = j$ and the claims follow. □

Lemma 4.5.5. *Let $\psi \in \mathrm{STList}\, AConf \to 1 + AConf$ be a transfinitely sound intuitive operator and let $D = (\mathrm{def}_\vee, \mathrm{ref})$ be a data flow approximation system for it. Let S be a finite program and (S, rel) be its relevance system w.r.t. D. Let $\overline{S}$ approximate S on base S and D with isomorphism $\overline{\cdot}$. Let $l = \mathcal{T}_\psi(\overline{S})(s)$ for an $s \in State$ and let $o, \pi < |l|$, $o < \pi$ be such that $\mathrm{pp}(l(\varrho)) \in \overline{S}$ for no ordinal ϱ satisfying $o < \varrho < \pi$. Let $\overline{p} = \mathrm{pp}(l(o))$, $\overline{q} = \mathrm{pp}(l(\pi))$, $\overline{x} = \mathrm{pp}(l(o+1))$, $\overline{d} = \mathrm{arc}(l(o+1))$ and let $X \in \mathrm{rel}\, q$.*

(i) *Then $\mathrm{val}\, X(l(o+1)) = \mathrm{val}\, X(l(\pi))$, $X \in \mathrm{rel}\, x$ and if $X \in \mathrm{def}_\vee d$ then $p \in S$.*

(ii) *If $\mathrm{val}\, X(l(o)) \neq \mathrm{val}\, X(l(\pi))$ or $X \notin \mathrm{rel}\, p$ then $X \in \mathrm{def}_\vee d$ and $p \in S$.*

Proof.

(i) Let $w = (v_0, e_1, v_1, \ldots, e_n, v_n)$ be a walk from p to q in $\mathrm{tcfg}\, S$ such that $\overline{e_1} = \mathrm{arc}(l(o+1)) = \overline{d}$ and $(v_1, e_2, v_2, \ldots, e_n, v_n)$ uses precisely the arcs whose $\overline{\cdot}$-image occurs in $\mathrm{drop}(o+2)(\mathrm{take}(\pi+1)\, l)$.

Suppose $i \in \mathbb{N}$ is the least number such that $X \in \mathrm{def}_\vee \overline{e_k}$ for no k with $i+1 < k \leqslant n$. By Lemma 4.5.4, $X \in \mathrm{rel}\, v_{i+1}$. If $X \in \mathrm{def}_\vee e_{i+1}$ then Lemma 4.5.4 also gives $v_i \in S$ which together with our assumptions and construction implies $i = 0$. If $X \notin \mathrm{def}_\vee e_{i+1}$ then $i = 0$ by choice. So $i = 0$.

Hence $X \in \mathrm{def}_\vee \overline{e_k}$ for no k with $1 < k \leqslant n$. Therefore if ϱ is any ordinal satisfying $o+1 < \varrho \leqslant \pi$ then $X \notin \mathrm{def}_\vee(\mathrm{arc}(l(\varrho)))$. Consequently, $X \in \mathrm{def}_\psi(\mathrm{arc}(l(\varrho)))$ for no ϱ satisfying $o+1 < \varrho \leqslant \pi$. By assumptions, $o+1 \leqslant \pi$. So Lemma 4.4.2 applies and gives $\mathrm{val}\, X(l(o+1)) = \mathrm{val}\, X(l(\pi))$.

Furthermore, note that $X \in \mathrm{rel}\, v_1$ and $v_1 = x$ by construction. The last claim of this part of the lemma follows directly from the last part of Lemma 4.5.4.

(ii) Suppose $\mathrm{val}\, X(l(o)) \neq \mathrm{val}\, X(l(\pi))$. Using Lemma 4.5.5(i), we get

$$\mathrm{val}\, X(l(o)) \neq \mathrm{val}\, X(l(o+1)) = \mathrm{val}\, X(\mathrm{next}_\psi(l(o)))\ .$$

Consequently, $X \in \mathrm{def}_\psi \overline{d} \subseteq \mathrm{def}_\vee \overline{d} \subseteq \mathrm{def}_\vee d$.

Suppose $X \notin \mathrm{rel}\, p$. By Lemma 4.5.5(i), $X \in \mathrm{rel}\, x$. By Definition 4.5.2(i), $\mathrm{rel}\, x \setminus \mathrm{def}_\vee d \subseteq \mathrm{rel}\, p$. Hence $X \in \mathrm{def}_\vee d$.

By Lemma 4.5.5(i), we have also $p \in S$. □

Lemma 4.5.6. *Let $\psi \in \mathrm{STList}\, AConf \to 1 + AConf$ be an escaping operator with $\propto \geqslant \omega^\omega$. Let $D = (\mathrm{def}_\vee, \mathrm{ref})$ be a data flow approximation system for it. Let (S, rel) be a relevance system of a finite program S w.r.t. D and let $\overline{S}$ be an approximation of S on base S and D with isomorphism $\overline{\cdot}$. Let $l = \mathcal{T}_\psi(\overline{S})(s)$ for some $s \in State$ and let $o, \pi < |l|$, $o < \pi$ be such that $\mathrm{pp}(l(\varrho)) \in \overline{S}$ for no ordinal ϱ satisfying $o < \varrho < \pi$ but $\mathrm{pp}(l(\pi)) \in \overline{S}$. Let $\overline{p} = \mathrm{pp}(l(o))$, $\overline{q} = \mathrm{pp}(l(\pi))$.*

(i) *Let $\overline{x} = \mathrm{pp}(l(o+1))$; then $x \leqslant q$.*

(ii) *Let r be the immediate postdominator of p. Then $r \not\leqslant q$ only if p dep q.*

Proof. By Theorem 4.3.8 and transfinite soundness, computation l reaches the final point f since, due to $|l| < \propto$, the only way to end is reaching f. Note that S, by Definition 4.5.2(i), is a dependence system of $(\mathrm{tcfg}\, S, f)$ in the sense of Definition 2.3.7. As $\mathrm{tcfg}\, S$ and $\mathrm{tcfg}\, \overline{S}$ are isomorphic, $\overline{S}$ is a dependence system of $(\mathrm{tcfg}\, \overline{S}, f)$.

(i) Finding a walk from $\overline{x}$ to $\overline{q}$ satisfying Lemma 4.3.4, we obtain $\overline{x} \leqslant \overline{q}$ (see also Theorem 2.3.8), hence the first desired claim follows.

(ii) Find a walk from $\overline{p}$ to $\overline{f} = f$ satisfying Lemma 4.3.4; it must pass through $\overline{r}$ since $p < r$ gives $\overline{p} < \overline{r}$. Thus there exists an ordinal ϱ such that $\mathrm{pp}(l(\varrho)) = \overline{r}$.

If $\varrho \leqslant \pi$ then a train of thought analogous to the one in Lemma 4.5.6(i) gives $\overline{r} \leqslant \overline{q}$. Thus $r \not\leqslant q$ implies $\pi < \varrho$.

Finally, find a walk $w = (v_0, e_1, v_1, \ldots, e_n, v_n)$ from $\overline{p}$ to $\overline{q}$ such that $v_1 = \overline{x}$ and $(v_1, e_2, v_2, \ldots, e_n, v_n)$ satisfies Lemma 4.3.4. By $\pi < \varrho$ and Lemma 4.3.11, w passes through no postdominators of $\overline{p}$. Thus there exists a v_i such that v_i dep $\overline{q}$ and $v_{i+1} \leqslant \overline{q}$ (Theorem 2.3.4), the former condition implying $v_i \in \overline{S}$. By assumptions, $i = 0$. Hence $\overline{p}$ dep $\overline{q}$ giving also p dep q. □

4.6 Semantic Correctness of Program Approximation

In this section, we prove preservation of some semantic properties under program approximation (Theorem 4.6.3). The result basically states that a program S and a program $\overline{S}$ approximating S, when both start running from the same initial state, compute the same transfinite sequence of values of relevant variables.

The proof idea is transfinite induction using Lemmas 4.6.1 and 4.6.2.

Lemma 4.6.1. *Let $\psi \in \mathrm{STList}\, AConf \to 1 + AConf$ be a transfinitely sound escaping intuitive operator with $\propto \geqslant \omega^\omega$. Let $D = (\mathrm{def}_\vee, \mathrm{ref})$ be a data flow approximation system for it. Let (S, rel) be a relevance system of a finite program S w.r.t. D and let $\overline{S}$ approximate S on base S and D with isomorphism $\overline{\cdot}$. Let $l = \mathcal{T}_\psi(S)(i)$ and $\overline{l} = \mathcal{T}_\psi(\overline{S})(\overline{i})$ for some states $i, \overline{i}$. Take ordinals o, $\overline{o}$ such that $\mathrm{conf}(l(o)) = \langle p \mid s\rangle$, $\mathrm{conf}(\overline{l}(\overline{o})) = \langle \overline{p} \mid \overline{s}\rangle$ for a program point p and states s, $\overline{s}$ such that $s\big|_{\mathrm{rel}\, p} = \overline{s}\big|_{\mathrm{rel}\, p}$. Let π be the least ordinal greater than o such that $\mathrm{pp}(l(\pi)) \in S$; denote $q = \mathrm{pp}(l(\pi))$. Then there exists the least ordinal $\overline{\pi}$ greater than $\overline{o}$ satisfying $\mathrm{pp}(\overline{l}(\overline{\pi})) = \overline{q}$; thereby, both following conditions hold:*

1. $\mathrm{pp}(\overline{l}(\varrho)) \in \overline{S}$ *for no* ϱ *with* $\overline{o} < \varrho < \overline{\pi}$,
2. $\mathrm{val}\, X(l(\pi)) = \mathrm{val}\, X(\overline{l}(\overline{\pi}))$ *for every* $X \in \mathrm{rel}\, q$.

Proof. Let r be the immediate postdominator of p in $TCFG$. Let $x = \mathrm{pp}(l(o+1))$ and $\overline{y} = \mathrm{pp}(\overline{l}(\overline{o}+1))$. Note that $x \leqslant r$ and $\overline{y} \leqslant \overline{r}$ (Lemma 2.2.5(ii)), the latter giving also $y \leqslant r$. As $p \in S$, Definition 4.5.2(ii) together with soundness gives

$$\overline{y} = \mathrm{pp}(\mathrm{next}_\psi(\overline{l}(\overline{o}))) = \mathrm{pp}(\mathrm{next}_\psi\langle \overline{p} \mid \overline{s}\rangle) = \overline{\mathrm{pp}(\mathrm{next}_\psi\langle p \mid \overline{s}\rangle)}\ . \tag{4.4}$$

Lemma 4.5.6(i) and Proposition 4.5.3(i) together imply $x \leqslant q$. If $r \leqslant q$ then, by transitivity, $y \leqslant q$. If $r \not\leqslant q$ then applying Lemma 4.5.6(ii) and Proposition 4.5.3(i) together gives p dep q. As $q \in S$, the latter implies $p \in S$ and $\mathrm{ref}(p, q) \subseteq \mathrm{rel}\, p$. Hence $s\big|_{\mathrm{ref}(p,q)} = \overline{s}\big|_{\mathrm{ref}(p,q)}$ by assumptions. Thus $\mathrm{pp}(\mathrm{next}_\psi\langle p \mid s\rangle) \leqslant q$ iff $\mathrm{pp}(\mathrm{next}_\psi\langle p \mid \overline{s}\rangle) \leqslant q$ by Definition 4.4.3 and soundness. Using Eq. 4.4, we obtain $x \leqslant q$ iff $\overline{y} \leqslant \overline{q}$. Thus $y \leqslant q$ also in this case.

Let $\overline{\pi}$ be the least ordinal greater than $\overline{o}$ such that $\mathrm{pp}(\overline{l}(\overline{\pi})) = \overline{q}$ (the computation $\mathrm{drop}(\overline{o}+1)\,\overline{l}$ reaches $\overline{q}$ since it starts from $\overline{y}$ and $\overline{y} \leqslant \overline{q}$). Let $\overline{\sigma}$ be the least ordinal greater than $\overline{o}$ such that $\mathrm{pp}(\overline{l}(\overline{\sigma})) \in \overline{S}$; denote $\overline{s} = \mathrm{pp}(\overline{l}(\overline{\sigma}))$. Then $\overline{\sigma} \leqslant \overline{\pi}$ since $\overline{q} \in \overline{S}$.

Lemma 4.5.6(i) now implies $y \leqslant s$. If $r \leqslant s$ then, by transitivity, $x \leqslant s$. If $r \not\leqslant s$ then applying Lemma 4.5.6(ii) gives p dep s. As $\overline{s} \in \overline{S}$ which is equivalent to $s \in S$, the latter gives $p \in S$ and $\mathrm{ref}(p, s) \subseteq \mathrm{rel}\, p$. Hence $s\big|_{\mathrm{ref}(p,s)} = \overline{s}\big|_{\mathrm{ref}(p,s)}$ by

assumptions. Thus $\mathrm{pp}(\mathrm{next}_\psi\langle p \mid s\rangle) \leqslant s$ iff $\mathrm{pp}(\mathrm{next}_\psi\langle p \mid \overline{s}\rangle) \leqslant s$ by Definition 4.4.3 and soundness. Using Eq. 4.4, we obtain $x \leqslant s$ iff $\overline{y} \leqslant \overline{s}$. Thus $x \leqslant s$ also in this case.

Altogether, we have got that both q and s are common non-strict postdominators of x and y. So either $q \leqslant s$ or $s \leqslant q$ (Theorem 2.2.6). As q occurs in $\mathrm{drop}(o+1)\,l$ not later than s, Lemma 4.3.11 implies $q \leqslant s$. Hence $\overline{q} \leqslant \overline{s}$ and Lemma 4.3.11 gives $\overline{\pi} \leqslant \overline{\sigma}$. Consequently, $\overline{\sigma} = \overline{\pi}$ giving also $\overline{s} = \overline{q}$. This proves the first part of the lemma.

To prove the second part, choose $X \in \mathrm{rel}\, q$ arbitrarily. Assumptions enable $\mathrm{pp}(l(\varrho)) \in S$ for no ϱ with $o < \varrho < \pi$. By the first part, $\mathrm{pp}(\overline{l}(\varrho)) \in \overline{S}$ for no ϱ with $\overline{o} < \varrho < \overline{\pi}$. Lemma 4.5.5(i) together with Proposition 4.5.3(i) give $X \in \mathrm{rel}(\mathrm{pp}(l(o+1))) = \mathrm{rel}\, x$. Denote $e = \mathrm{arc}(l(o+1))$.

Consider the case $X \in \mathrm{def}_\vee e$. By Definition 4.5.2(i), $X \in \mathrm{rel}\, x$ gives $p \in S$ and $\mathrm{ref}(e, X) \subseteq \mathrm{rel}\, p$. So $s\big|_{\mathrm{ref}(e,X)} = \overline{s}\big|_{\mathrm{ref}(e,X)}$ by assumptions. By soundness, Definition 4.4.3, and Definition 4.5.2(ii),

$$\begin{aligned}
\mathrm{val}\, X(l(o+1)) &= \mathrm{val}\, X(\mathrm{next}_\psi(l(o))) = \mathrm{val}\, X(\mathrm{next}_\psi\langle p \mid s\rangle) \\
&= \mathrm{val}\, X(\mathrm{next}_\psi\langle p \mid \overline{s}\rangle) = \mathrm{val}\, X(\overline{\mathrm{next}_\psi\langle p \mid \overline{s}\rangle}) \\
&= \mathrm{val}\, X(\mathrm{next}_\psi\langle \overline{p} \mid \overline{s}\rangle) = \mathrm{val}\, X(\mathrm{next}_\psi(\overline{l}(\overline{o}))) \\
&= \mathrm{val}\, X(\overline{l}(\overline{o}+1)) \ .
\end{aligned}$$

By Lemma 4.5.5(i) and Proposition 4.5.3(i),

$$\mathrm{val}\, X(l(\pi)) = \mathrm{val}\, X(l(o+1)) \quad \text{and} \quad \mathrm{val}\, X(\overline{l}(\overline{\pi})) = \mathrm{val}\, X(\overline{l}(\overline{o}+1)) \ .$$

Consequently, $\mathrm{val}\, X(l(\pi)) = \mathrm{val}\, X(\overline{l}(\overline{\pi}))$.

If $X \notin \mathrm{def}_\vee e$ then, by Lemma 4.5.5(ii) and Proposition 4.5.3(i), $\mathrm{val}\, X(l(\pi)) = \mathrm{val}\, X(l(o))$ and $\mathrm{val}\, X(\overline{l}(\overline{\pi})) = \mathrm{val}\, X(\overline{l}(\overline{o}))$ whereby $X \in \mathrm{rel}\, p$. The latter implies $\mathrm{val}\, X(l(o)) = \mathrm{val}\, X(\overline{l}(\overline{o}))$ by assumptions. Hence the claim follows. □

Lemma 4.6.2. *Let $\psi \in \mathrm{STList}\, AConf \to 1 + AConf$ be a regular intuitive limit operator and let D be a data flow approximation system for it. Let (S, rel) be a relevance system for a regular program S w.r.t. D and let $\overline{S}$ approximate S on base S and D with isomorphism $\overline{\cdot}$. Let $l = \mathcal{T}_\psi(S)(i)$ and $\overline{l} = \mathcal{T}_\psi(\overline{S})(\overline{i})$ for some states i, $\overline{i}$. Let $\lambda > 1$ be a selfish ordinal. For each ordinal $\xi < \lambda$, let o_ξ, $\overline{o}_\xi$ be ordinals less than $|l|$ and $|\overline{l}|$, respectively. Denote $\langle p_\xi \mid s_\xi\rangle = \mathrm{conf}(l(o_\xi))$ for each $\xi < \lambda$. Assume the following:*

1. $\mathrm{pp}(l(o_0)) \in S$, $\mathrm{pp}(\overline{l}(\overline{o}_0)) \in \overline{S}$;
2. *for every ξ with $0 < \xi < \lambda$, o_ξ is the least ordinal greater than any ordinal o_η with $\eta < \xi$ such that $\mathrm{pp}(l(o_\xi)) \in S$;*

3. *for every* ξ *with* $0 < \xi < \lambda$, $\overline{o}_\xi$ *is the least ordinal greater than any ordinal* $\overline{o}_\eta$ *with* $\eta < \xi$ *such that* $\mathrm{pp}(\overline{l}(\overline{o}_\xi)) \in \overline{S}$*;*

4. *for each* $\xi < \lambda$, $\mathrm{conf}(\overline{l}(\overline{o}_\xi)) = \langle \overline{p_\xi} \mid \overline{s}_\xi \rangle$ *with state* $\overline{s}_\xi$ *such that* $s_\xi\big|_{\mathrm{rel}\, p_\xi} = \overline{s}_\xi\big|_{\mathrm{rel}\, p_\xi}$.

Let π *be the least ordinal greater than any of* o_ξ *for which* $\mathrm{pp}(l(\pi)) \in S$*; denote* $q = \mathrm{pp}(l(\pi))$*. Then there exists the least ordinal* $\overline{\pi}$ *greater than any of* $\overline{o}_\xi$ *for which* $\mathrm{pp}(\overline{l}(\overline{\pi})) = \overline{q}$*; thereby, both following conditions hold:*

1. $\mathrm{pp}(\overline{l}(\varrho)) \in \overline{S}$ *for no* $\varrho < \overline{\pi}$ *greater than any of* $\overline{o}_\xi$*;*

2. $\mathrm{val}\, X(l(\pi)) = \mathrm{val}\, X(\overline{l}(\overline{\pi}))$ *for every* $X \in \mathrm{rel}\, q$.

Proof. Let τ be the least ordinal greater than any of o_ξ and let $\overline{\tau}$ be the least ordinal greater than any of $\overline{o}_\xi$. Let $\tau = \alpha + \gamma$ and $\overline{\tau} = \overline{\alpha} + \overline{\gamma}$ be principal representations. As $\alpha < \tau$, there exists the least ordinal ζ such that $o_\zeta \geqslant \alpha$. Analogously, let $\overline{\zeta}$ be the least ordinal such that $\overline{o}_{\overline{\zeta}} \geqslant \overline{\alpha}$. As λ is selfish, there are λ many ordinals $o_\xi \geqslant \alpha$ and as many ordinals $\overline{o}_\xi \geqslant \overline{\alpha}$. Hence $\mathrm{drop}\, \alpha\, l$ visits at least λ program points in S, the first of them being $\mathrm{pp}(l(o_\zeta))$. Analogously, $\mathrm{drop}\, \overline{\alpha}\, \overline{l}$ visits at least λ program points in $\overline{S}$, the first of them being $\mathrm{pp}(\overline{l}(\overline{o}_{\overline{\zeta}}))$.

Let $t = \mathrm{pp}(l(\tau))$ and $\tilde{t} = \mathrm{pp}(\overline{l}(\overline{\tau}))$. We claim that $\tilde{t} = \overline{t}$. For proving it, suppose the contrary, i.e. $\tilde{t} \neq \overline{t}$. As the semantics is regular, t postdominates every program point in $\mathrm{loop}\,(\mathrm{take}\, \tau\, l)$ and $\tilde{t}$ postdominates every program point in $\mathrm{loop}\,\big(\mathrm{take}\, \overline{\tau}\, \overline{l}\big)$. As S is finite and every non-empty final part of $\mathrm{take}\, \tau\, l$ visits program points of S, there is at least one program point $s \in \mathrm{loop}\,(\mathrm{take}\, \tau\, l) \cap S$. By assumptions, $\overline{s} \in \mathrm{loop}\,\big(\mathrm{take}\, \overline{\tau}\, \overline{l}\big)$. So, in particular, $s < t$ and $\overline{s} < \tilde{t}$. Hence $\overline{t}$ and $\tilde{t}$ are common postdominators of $\overline{s}$. As $\overline{t} \neq \tilde{t}$, either $\overline{t} < \tilde{t}$ or $\tilde{t} < \overline{t}$ must hold (Theorem 2.2.6). Suppose the former; the proof continues analogously in the other case.

By regularity, $\tilde{t}$ is the immediate postdominator of a program point $\overline{y}$ looping in $\mathrm{take}\, \overline{\tau}\, \overline{l}$ (see also Corollary 2.2.10(ii)). For arbitrary $\eta < \overline{\tau}$, there is a subcomputation of $\mathrm{drop}\, \eta(\mathrm{take}\, \overline{\tau}\, \overline{l})$ driving control from $\overline{s}$ to $\overline{s}$ through $\overline{y}$; find a corresponding walk w_η from s to s through y satisfying Lemma 4.3.4.

Let w_η be the last program point on the part of w_η starting with y such that $y \leqslant w_\eta$. Then the program points passed through by the part of w starting with the last occurrence of w_η do not postdominate w_η since otherwise they would postdominate y contradicting the choice of w_η. Therefore all these program points are transitively control dependent on w_η (Theorem 2.3.5). In particular, $w_\eta\ \mathrm{dep}^*\ s$. As $s \in S$, we have also $w_\eta \in S$. As S is finite, we have an unbounded set of ordinals η for which w_η is the same, say, v. So $\overline{v} \in \mathrm{loop}\,\big(\mathrm{take}\, \overline{\tau}\, \overline{l}\big) \cap \overline{S}$ implying

$v \in \operatorname{loop}(\operatorname{take}\tau\, l) \cap S$. Hence $\overline{y} \leqslant \overline{v} < \overline{t} < \tilde{t}$, a contradiction since $\tilde{t}$ was supposed to be the immediate postdominator of $\overline{y}$.

Consequently, $\tilde{t} = \overline{t}$.

As $\tau \leqslant \pi$, there is a walk from t to q according to Lemma 4.3.4. Thus $t \leqslant q$ (Theorem 2.3.8). Let $\overline{\pi}$ be the least ordinal greater than or equal to $\overline{\tau}$ such that $\operatorname{pp}(\overline{l}(\overline{\pi})) = \overline{q}$ (the computation $\operatorname{drop}\overline{\tau}\,\overline{l}$ reaches $\overline{q}$ since it starts from $\overline{t}$ and $\overline{t} \leqslant \overline{q}$). Let $\overline{\sigma}$ be the least ordinal greater than or equal to $\overline{\tau}$ such that $\operatorname{pp}(\overline{l}(\overline{\sigma})) \in \overline{S}$; denote $\overline{s} = \operatorname{pp}(\overline{l}(\overline{\sigma}))$. Then $\overline{\sigma} \leqslant \overline{\pi}$ since $\overline{q} \in \overline{S}$.

We have also $\overline{t} \leqslant \overline{s}$ (Theorem 2.3.8). So either $q \leqslant s$ or $s \leqslant q$ (Theorem 2.2.6). As the first visit of q occurs not later than the first visit of s by $\operatorname{drop}\tau\, l$, Lemma 4.3.11 implies $q \leqslant s$. Thus $\overline{q} \leqslant \overline{s}$ and Lemma 4.3.11 gives $\overline{\pi} \leqslant \overline{\sigma}$. Consequently, $\overline{\sigma} = \overline{\pi}$ giving also $\overline{s} = \overline{q}$. This proves the first part of the lemma.

To prove the second part, choose $X \in \operatorname{rel} q$ arbitrarily. Let $x = \sigma(\operatorname{arc}(l(\tau))) \in \operatorname{loop}(\operatorname{take}\tau\, l)$. By Lemma 4.3.12, x is transitively control depended on by every program point in $\operatorname{loop}(\operatorname{take}\tau\, l)$. As there are program points of S among them, it follows that $x \in S$. Thus $\overline{x} \in \operatorname{loop}\big(\operatorname{take}\overline{\gamma}(\operatorname{drop}\overline{\alpha}\,\overline{l})\big)$. As tcfg S and tcfg $\overline{S}$ are isomorphic, there is a transfinite arc $\overline{e}$ from $\overline{x}$ to $\overline{t}$. By regularity, the loop $\operatorname{take}\overline{\tau}\,\overline{l}$ is escaped from using $\overline{e}$.

If $t = q$ then $X \in \operatorname{rel} t$; if $t \neq q$ then $\tau < \pi$, $t \notin S$ and Lemma 4.5.5(ii) together with Proposition 4.5.3(i) also imply $X \in \operatorname{rel} t$. Hence $X \in \operatorname{rel} x$ since $\operatorname{rel} t \subseteq \operatorname{rel} x$ by Definition 4.5.2(i). Thus by assumptions, $s_\xi\big|_{\{X\}} = \overline{s}_\xi\big|_{\{X\}}$ for every ordinal $\xi < \lambda$ such that $p_\xi = x$. This covers all places where $\operatorname{take}\gamma(\operatorname{drop}\alpha\, l)$ visits x and $\operatorname{take}\overline{\gamma}(\operatorname{drop}\overline{\alpha}\,\overline{l})$ visits $\overline{x}$. This means

$$\begin{aligned}&\operatorname{map}(\operatorname{val} X)(\operatorname{filter}(\operatorname{at}\{x\})(\operatorname{drop} o_{\max(\zeta,\overline{\zeta})}(\operatorname{take}\tau\, l)))\\ &\qquad = \operatorname{map}(\operatorname{val} X)(\operatorname{filter}(\operatorname{at}\{\overline{x}\})(\operatorname{drop}\overline{o}_{\max(\zeta,\overline{\zeta})}(\operatorname{take}\overline{\tau}\,\overline{l})))\ .\end{aligned}$$

Lemma 4.3.10(iii) gives $\operatorname{val} X(l(\tau)) = \operatorname{val} X(\overline{l}(\overline{\tau}))$. As $X \in \operatorname{rel} q$, using Lemma 4.5.5(ii) together with Proposition 4.5.3(i) gives $\operatorname{val} X(l(\tau)) = \operatorname{val} X(l(\pi))$ and $\operatorname{val} X(\overline{l}(\overline{\tau})) = \operatorname{val} X(\overline{l}(\overline{\pi}))$. Hence the desired claim follows. □

Theorem 4.6.3. *Let $\psi \in \operatorname{STList} AConf \to 1 + AConf$ be an regular intuitive limit operator with $\propto \geqslant \omega^\omega$ and D be a data flow approximation system for it. Let (S, rel) be a relevance system for a regular program S w.r.t. D and let $\overline{S}$ approximate S on base S and D with isomorphism $\overline{\cdot}$. Let $l = \mathcal{T}_\psi(S)(s)$ and $\overline{l} = \mathcal{T}_\psi(\overline{S})(s)$ for some $s \in State$. Denote $m = \operatorname{filter}(\operatorname{at} S)\, l$, $\overline{m} = \operatorname{filter}(\operatorname{at}\overline{S})\,\overline{l}$. Then $\overline{S}$ is regular, $\operatorname{map}(\operatorname{pp}\,;\overline{\cdot})\, m = \operatorname{map}\operatorname{pp}\overline{m}$ and, for every $\xi < |m|$ and $X \in \operatorname{rel}(\operatorname{pp}(m(\xi)))$, the equality $\operatorname{val} X(m(\xi)) = \operatorname{val} X(\overline{m}(\xi))$ holds.*

Proof. $\overline{S}$ is regular as S is regular and the conditions of program regularity are stated in terms of transfinite control flow graphs and preserved by isomorphism.

Let $(o_\xi : \xi < |m|)$ be the increasing family of all ordinals indexing the components of l satisfying $\mathrm{at}\, S$; then $m(\xi) = l(o_\xi)$ for every $\xi < |m|$. Analogously, let $(\overline{o}_\eta : \eta < |\overline{m}|)$ be the increasing family of all ordinals indexing the components of $\overline{l}$ satisfying $\mathrm{at}\, \overline{S}$. Denote $p_\xi = \mathrm{pp}(m(\xi))$.

We start with showing by transfinite induction on ξ that $\mathrm{pp}(\overline{l}(\overline{o}_\xi)) = \overline{p_\xi}$ and $\mathrm{val}\, X(l(o_\xi)) = \mathrm{val}\, X(\overline{l}(\overline{o}_\xi))$ for every $X \in \mathrm{rel}\, p_\xi$.

Consider the case $\xi = 0$. As $\mathrm{pp}(\overline{l}(0)) = i_{\overline{S}} = \overline{i_S} = \overline{\mathrm{pp}(l(0))}$ and $s = s$, we have $o_0 = 0 \iff \overline{o}_0 = 0$ and the desired claim holds if $o_0 = 0$. If $o_0 > 0$, the claim is implied by Lemma 4.6.1 for $o \leftarrow 0$, $\overline{o} \leftarrow 0$, $\pi \leftarrow o_0$.

For $\xi > 0$, assume the claim holding for smaller ordinals and let $\xi = \alpha + \gamma$ be the principal representation.

If $\gamma = 1$, apply Lemma 4.6.1 for $o \leftarrow o_\alpha$, $\overline{o} \leftarrow \overline{o}_\alpha$, $\pi \leftarrow o_\xi = o_{\alpha+1}$. The required assumptions about program points and states at o_αth step hold by induction hypothesis.

If $\gamma > 1$, apply Lemma 4.6.2 for $o_\eta \leftarrow o_{\alpha+\eta}$ and $\overline{o}_\eta \leftarrow \overline{o}_{\alpha+\eta}$ for every $\eta < \gamma$, $\pi \leftarrow o_\xi$. The required assumptions hold due to construction and induction hypothesis.

It remains to show $|m| = |\overline{m}|$. By Theorem 4.3.8, $|l| < \propto$. By Lemma 4.3.3(vi), computation l ends at f. By Definition 4.5.2(i), $f \in S$. So there is a ζ such that $\mathrm{pp}(m(\zeta)) = p_\zeta = f$. By the proof so far, $\mathrm{pp}(\overline{m}(\zeta)) = \mathrm{pp}(\overline{l}(\overline{o}_\zeta)) = \overline{f} = f$. Thus $|m| = |\overline{m}| = \zeta + 1$. □

The assumption of Lemma 4.6.2 and Theorem 4.6.3 that ψ is a limit operator (in the sense of Definition 3.4.1(i)) is mandatory as shown by Example 4.6.4.

Example 4.6.4. Consider the following transformation:

```
[0]x := true;
[1]i := 0;
 while [2]i >= 0 do (
   [3]while x do ;
   [4]x := false;
   [5]i := i + 1
 );
[6]
```

⇢

```
[0]x := false;
[1]i := 0;
 while [2]i >= 0 do (
   [3]while x do ;
   [4]x := false;
   [5]i := i + 1
 );
[6]
```

Take $S = \{1, 2, 5, 6\}$ and $\mathrm{rel}\,1 = \mathrm{rel}\,2 = \mathrm{rel}\,5 = \mathrm{rel}\,6 = \{\mathtt{i}\}$. Then (S, rel) is a relevance system in the original program w.r.t. naturally defined def-sets and ref-sets. It can be obtained by computing the slice w.r.t. criterion $\{(6, \mathtt{i})\}$ in the traditional way of relevant sets.

The transfinite control flow graphs of these two programs are clearly isomorphic; the isomorphism is reflected by the numeration of program points. The second program differs from the first by the first statement only. As the set of variables assigned is the same, the second program qualifies as an approximation of the first on base S.

The different value assigned to `x` involves a significant difference between the lengths of run of these programs. The first program works for $\omega + \omega + 1$ steps (the inner infinite loop is executed during the first execution of the body of the outer loop and skipped afterwards) while the second program works for $\omega + 1$ steps only (the inner infinite loop is always skipped).

The principal representation of $\omega + \omega$ is $\omega + \omega$ while the principal representation of ω is $0 + \omega$. Hence the state of variables at the final configuration of the run of the first program (where the computation reaches after entirely running the outer loop) is determined by the sequence of intermediate states occurring when control passes through 2, exclusive of the first passing as the latter remains inside the first ω steps. The state of variables at the final configuration of the run of the second program is determined by the sequence of all intermediate states occurring when control passes through 2.

In the first sequence, variable `i` obtains values $1, 2, 3, \ldots$; in the second sequence, it obtains values $0, 1, 2, 3, \ldots$. If ψ were not a limit operator, the limit state given by ψ might be different on these two sequences and hence the transformation would not be correct, Theorem 4.6.3 would not hold. □

Example 4.6.4 implies also that the correctness of standard algorithms for program slicing w.r.t. transfinite semantics (studied in Sect. 4.8) holds generally only if the semantics is corecursive. This conclusion may be done at least for semantics whose definition is grounded on principal representations of ordinals.

One may argue, relying on Example 4.6.4, that the way in which the principal representation splits the computation process is unnatural because this splitting does not respect the computation intervals corresponding to the composite (i.e. non-atomic) statements. For languages with structured control flow, one may forsake principal representations and ground on the syntax structure only (like in our recent paper [11]). In the theory developed in this thesis, we wanted to keep the theory abstracted from language details, therefore we had to ground on some other mechanism of structuring computation processes. If limits of processes are always defined via limit operators then this choice makes no essential difference.

4.7 Program Simplification

After finding a suitable approximation to a program, the irrelevant statements must be safely removed to obtain a slice. We call this step program simplification.

First of all, as simplifing means omission of computation steps and the latter correspond to arcs in control flow graphs in our treatment, the simplification operation on control flow graphs must take sets of arcs as arguments. Not all sets of arcs are allowed: one cannot remove a branching structure while maintaining the inner statements of it. The sets of arcs of control flow graphs whose members may be removed simultaneously will be called total here. The formal criterion is given by Definition 4.7.1(i).

Definition 4.7.1.

(i) *Call a set* $D \subseteq E(TCFG)$ total *iff, for every arc* $e \in D$*, all arcs starting from* $\sigma(e)$ *in* $TCFG$*, as well as from any vertex control dependent on* $\sigma(e)$*, belong to* D*.*

(ii) *For every total* $D \subseteq E(TCFG)$*, define a transformation* $\mathrm{fall}_D \in PP \to PP$ *as follows:*

$$\forall p \in PP\ (\mathrm{fall}_D\ p = \min\{x \in PP \mid p \leqslant x\ \wedge\ \forall e \in D\ (\sigma(e) \neq x) \bullet x\})$$

(minimum is found w.r.t. postdominance order).

Roughly, the idea behind the concept of totality is that a computation step can be omitted only if all computation steps control dependent on it are also removed. This condition on control dependence implies the same also for transitive control dependence.

Definition 4.7.1(ii) is correct as $f \in \{x \in PP \mid p \leqslant x\ \wedge\ \forall e \in D\ (\sigma(e) \neq x) \bullet x\}$ and the set of all postdominators of p is finite and linearly ordered for every p. It states that $\mathrm{fall}_D\ p$ is the least w.r.t. postdominance order program point non-strictly postdominating p from which no arc of D starts. Particularly, if no arc of D starts from p then $\mathrm{fall}_D\ p = p$. Intuitively, $\mathrm{fall}_D\ p$ is the program point where one falls through from p when the arcs of D disappear.

Proposition 4.7.2 characterizes the totality property and fall operator. Proposition 4.7.3(i) tells that a transfinite arc can belong to a total set only together with all arcs used in endless computations from which control escapes along this arc. The other claims of Proposition 4.7.3 are corollaries of this fact.

Proposition 4.7.2. *Let* D *be a total set of arcs.*

(i) *For every program point, either all or none of the arcs starting from it belong to* D*.*

(ii) *Let w be a walk from p to q in TCFG using arcs of D only. If no arc of D starts from q then* $\mathrm{fall}_D\, p = q$.

(iii) *For every p, there is a walk from p to* $\mathrm{fall}_D\, p$ *using arcs from D only.*

(iv) *For every p such that* $\mathrm{fall}_D = f$, *every walk from p to f uses arcs from D only.*

(v) *Let p, q, s be program points such that both p and q are reachable from s using arcs of D only. Then* $\mathrm{fall}_D\, p = \mathrm{fall}_D\, q$.

Proof.

(i) By Definition 4.7.1(i), if an arc starting from p belongs to D then all arcs starting from p belong to D.

(ii) Show first that $p \leqslant q$. If $q = f$ then this is the case trivially. Therefore assume $q \neq f$. Suppose $p \not\leqslant q$. Let r be the last vertex on w before reaching q such that $r \not\leqslant q$. As there is an arc starting from q not belonging to D while there is an arc starting from r belonging to D, Definition 4.7.1(i) implies that q is not transitively control dependent on r. Hence all walks from r to q pass through a postdominator of r (Theorem 2.3.5). This must hold also for the part of walk w from r to q. So there is an s passed through by that part of w such that $r < s$. By the choice of r, we have $s \leqslant q$. Therefore $r < q$ by transitivity, giving a contradiction.

Now $p \leqslant q$ implies $q \in \{x \in PP \mid p \leqslant x \ \wedge\ \forall e \in D\,(\sigma(e) \neq x) \bullet x\}$. Suppose the desired claim does not hold; then there is a program point $t < q$ such that $t \in \{x \in PP \mid p \leqslant x \ \wedge\ \forall e \in D\,(\sigma(e) \neq x) \bullet x\}$. If $p = t$ then w must be empty (as no arc of D starts from it), hence $p = q$, giving a contradiction to $p = t < q$. If $p < t$ then w passes through t (Theorem 2.2.9); but then there are arcs of D starting from t contradicting the choice of t.

(iii) Consider a walk from p to f. By Proposition 4.7.2(ii), its longest initial part using arcs from D only ends at $\mathrm{fall}_D\, p$. This is the desired walk.

(iv) Consider arbitrary walk w from p to f. By Proposition 4.7.2(ii), its longest initial part using arcs from D only ends at $\mathrm{fall}_D\, p = f$. As no arc starts from f, this initial part of w coincides with w. Hence all arcs of w belong to D.

(v) There are walks from s to p, from s to q, from p to $\mathrm{fall}_D\, p$, and from q to $\mathrm{fall}_D\, q$, all of which use arcs of D only. Putting together, we obtain walks from s to both $\mathrm{fall}_D\, p$ and $\mathrm{fall}_D\, q$ using arcs from D only. By Definition 4.7.1(ii), no arcs of D start from $\mathrm{fall}_D\, p$ or $\mathrm{fall}_D\, q$. Now by Proposition 4.7.2(ii), $\mathrm{fall}_D\, p = \mathrm{fall}_D\, s = \mathrm{fall}_D\, q$. □

Lemma 4.7.3. *Let $\psi \in \mathrm{STList}\, AConf \to 1 + AConf$ be a regular operator with $\propto \geqslant \omega^\omega$ and S a regular program. Let $l = \mathcal{T}_\psi(S)(i)$ for some $i \in State$. Let $D \subseteq E(\mathrm{tcfg}\, S)$ be total.*

(i) *For every limit ordinal* $\pi < |l|$*, all arcs between program points looping in* $\mathrm{take}\,\pi\,l$ *are in* D *iff* $\mathrm{arc}(l(\pi)) \in D$.

(ii) *For every limit ordinal* $\pi < |l|$*, if* $\mathrm{drop}\,\varrho(\mathrm{take}\,\pi\,l)$ *uses arcs outside* D *for every* $\varrho < \pi$ *then* $\mathrm{arc}(l(\pi)) \notin D$.

(iii) *For any ordinal* $o < |l|$*, if* σ *is the least ordinal greater than* o *such that* $\mathrm{arc}(l(\sigma)) \notin D$ *then* σ *is a successor ordinal. Thereby, if* $\sigma < |l|$ *then we have* $\mathrm{fall}_D(\mathrm{pp}(l(o))) = \sigma(\mathrm{arc}(l(\sigma)))$ *and otherwise* $\mathrm{fall}_D(\mathrm{pp}(l(o))) = f$.

(iv) *For any ordinal* $o < |l|$*, there exists a largest ordinal* π *such that* $\mathrm{arc}(l(\varrho)) \in D$ *for every* ϱ *satisfying* $o < \varrho \leqslant \pi$*. Thereby,* $\mathrm{fall}_D(\mathrm{pp}(l(o))) = \mathrm{pp}(l(\pi))$.

(v) *For every ordinal* $\pi < |l|$*, there exists a least ordinal* o *such that* $\mathrm{arc}(l(\varrho)) \in D$ *for every* ϱ *satisfying* $o < \varrho \leqslant \pi$*; thereby,* $\mathrm{arc}(l(o)) \notin D$.

(vi) *There exists a largest ordinal* $o < |l|$ *such that* $\mathrm{arc}(l(o)) \notin D$*. Thereby,* $\mathrm{fall}_D(\mathrm{pp}(l(o))) = f$.

Proof.

(i) Suppose that all arcs between program points looping in $\mathrm{take}\,\pi\,l$ are in D. As $\sigma(\mathrm{arc}(l(\pi)))$ is looping in this computation, an arc starting from it belongs to D. Hence, using Proposition 4.7.2(i), also $\mathrm{arc}(l(\pi)) \in D$.

Suppose $\mathrm{arc}(l(\pi)) \in D$. By Lemma 4.3.12, all program points looping in $\mathrm{take}\,\pi\,l$ are transitively control dependent on $\sigma(l(\pi))$. By Definition 4.7.1(i), all arcs starting from all these vertices are in D.

(ii) By Lemma 4.7.3(i), $\mathrm{arc}(l(\pi))$ belonging to D would imply all arcs between program points which are looping in $\mathrm{take}\,\pi\,l$ belonging to D. By Lemma 4.3.5, we can find $o < \pi$ such that all program points visited by $\mathrm{drop}\,o(\mathrm{take}\,\pi\,l)$ are looping in $\mathrm{take}\,\pi\,l$. All arcs $\mathrm{arc}(l(\varrho))$ for $o < \varrho < \pi$ would have to be in D contradicting the assumption.

(iii) Let o be fixed. Clearly $\sigma \leqslant |l|$.

Show the first claim. If $\sigma = |l|$ then σ is a successor ordinal and we are done. Suppose $\sigma < |l|$ and σ being a limit ordinal. By Lemma 4.7.3(i), there exists an arc outside D between program points looping in $\mathrm{take}\,\sigma\,l$. By Proposition 4.7.2(i), $\mathrm{take}\,\sigma\,l$ must use arcs outside D endlessly which contradicts the choice of σ. Consequently, σ is a successor ordinal.

Prove the second claim now. We have $\sigma = \pi + 1$ for a $\pi \geqslant o$. Find a walk w from $\mathrm{pp}(l(o))$ to $\mathrm{pp}(l(\pi))$ using precisely the arcs of $\mathrm{drop}(o+1)(\mathrm{take}\,\sigma\,l)$. All the arcs of w belong to D. No arc of D starts from $\mathrm{pp}(l(\pi))$ since $\mathrm{arc}(l(\sigma)) \notin D$. Hence $\mathrm{fall}_D(\mathrm{pp}(l(o))) = \mathrm{pp}(l(\pi))$ by Proposition 4.7.2(ii). If $\sigma < |l|$ then $\mathrm{pp}(l(\pi)) = \sigma(\mathrm{arc}(l(\sigma)))$, otherwise $\mathrm{pp}(l(\pi)) = f$. This concludes the proof.

(iv) Let σ be the least ordinal greater than o such that $\mathrm{arc}(l(\sigma)) \notin D$. By Lemma 4.7.3(iii), $\sigma = \pi + 1$ for a π. Then $\mathrm{arc}(l(\varrho)) \in D$ for every ϱ satisfying $o < \varrho \leqslant \pi$ and clearly π is the largest such ordinal.

If $\sigma < |l|$ then Lemma 4.7.3(iii) gives $\mathrm{fall}_D(\mathrm{pp}(l(o))) = \sigma(\mathrm{arc}(l(\sigma))) = \mathrm{pp}(l(\pi))$. If $\sigma = |l|$ then, again by Lemma 4.7.3(iii), $\mathrm{fall}_D(\mathrm{pp}(l(o))) = f = \mathrm{pp}(l(\pi))$.

(v) As $\mathrm{arc}(l(\varrho)) \in D$ for every ϱ satisfying $\pi < \varrho \leqslant \pi$, the set of ordinals where the minimal has to be found is non-empty. Hence the first claim follows.

For the second, suppose $\mathrm{arc}(l(o)) \in D$. If $o = \xi + 1$ for some ξ then $\xi < o$ while $\mathrm{arc}(l(\varrho)) \in D$ for every ϱ satisfying $\xi < \varrho \leqslant \pi$. If o is a limit ordinal then using Lemma 4.7.3(ii) contrapositively gives that $\mathrm{drop}\,\xi(\mathrm{take}\,o\,l)$ does not use arcs outside D for some $\xi < o$ for which we then have $\mathrm{arc}(l(\varrho)) \in D$ for every ϱ satisfying $\xi < \varrho \leqslant \pi$. Hence both cases contradict the choice of o.

(vi) Let $l(\sigma)$ be the last component of l. Define o to be the least ordinal such that $\mathrm{arc}(l(\varrho)) \in D$ for every ϱ satisfying $o < \varrho \leqslant \sigma$. By Lemma 4.7.3(v), this definition is correct and $\mathrm{arc}(l(o)) \notin D$. By construction, o is the largest ordinal less than $|l|$ such that $\mathrm{arc}(l(o)) \notin D$. For the second claim, note that $\tau = |l|$ is the least ordinal greater than o such that $\mathrm{arc}(l(\tau)) \notin D$. Thus, by Lemma 4.7.3(iii), $\mathrm{fall}_D(\mathrm{pp}(l(o))) = f$. □

We are going to define program simplification as a relation between two given programs like we did with program approximation. At place of isomorphism in the case of program approximation, we need another type of mapping in the case of program slicing. We call it sliceprojection. Like projections in general, it is a structure-preserving function losing some facets of its argument.

Definition 4.7.4. *Let $(G, i_G), (H, i_H)$ be local flow graphs (either transfinite or not). Let $D \subseteq E(G)$ be total. Let $\sim_D$ be the least equivalence relation on $V(G)$ containing $\{e \in D \mid \bullet(\sigma(e), \tau(e))\}$; for each $v \in V(G)$, denote its equivalence class by $v/\sim_D$. (The equivalence classes by $\sim_D$ are actually the weakly connected components of the graph whose vertices are that of G and arc set is D.)*

A mapping $\cdot^\circ$ from $V(G)$ to $V(H)$ and from $E(G) \setminus D$ to $E(H)$ is called sliceprojection from (G, i_G) to (H, i_H) w.r.t. D *iff all the following holds:*

1. *for arbitrary $p, q \in V(G)$,*

$$p^\circ = q^\circ \iff p \sim_D q \ ;$$

2. *on arcs, $\cdot^\circ$ is a bijection from $E(G) \setminus D$ to $E(H)$;*
3. *$(\sigma(e))^\circ = \sigma(e^\circ)$ and $(\tau(e))^\circ = \tau(e^\circ)$ for every $e \in E(G) \setminus D$;*
4. *$f^\circ = f$;*
5. *$i_G^\circ = i_H$;*
6. *transfinite arcs and only these are mapped to transfinite arcs.*

Note that $E(G) \setminus D$ can contain arcs between two program points equivalent by $\sim_D$. Such arcs transform to arcs of H whose source and target coincide. A practical example of this situation is removing the subgraph corresponding to S of the graph of a loop **while** B **do** S. The graph of S contains all program points of the loop (the top point of the loop is included as it is the end point of the flow of S) but it does not contain the arc going from the top of the loop to the beginning of S. Slicing S away means removing the atomic statements of S and all branching constructions inside S; this results in an empty loop **while** B **do** . When doing this on graphs, all program points of the graph of **while** B **do** S become equivalent, so being transformed to just one program point, but one arc remains and goes from this vertex to the very same vertex.

Proposition 4.7.2(iii) implies that p and $\mathrm{fall}_D\, p$ always belong to the same equivalence class of $\sim_D$ and therefore $p^\circ = (\mathrm{fall}_D\, p)^\circ$. More turns out: fall_D actually works like a canonical instance finder. Proposition 4.7.5 states this.

Proposition 4.7.5. *Let G, H be local (transfinite) control flow graphs. Let $D \subseteq E(G)$ be total. Let $\cdot^\circ$ be a sliceprojection from G to H. For every $p, q \in V(G)$,*

$$p^\circ = q^\circ \iff \mathrm{fall}_D\, p = \mathrm{fall}_D\, q \ .$$

Proof. Suppose $p^\circ = q^\circ$. As p and q belong to common equivalence class of $\sim_D$, there exists a finite sequence $t_0, s_1, t_1, \ldots, s_n, t_n$ such that $t_0 = p$, $t_n = q$ and, for each $i = 1, \ldots, n$, there are walks from s_i to t_{i-1} and t_i which use arcs of D only. Proceed by induction on n. If $n = 0$ then $p = q$ and $\mathrm{fall}_D\, p = \mathrm{fall}_D\, q$ follows trivially. Assume the claim for $n-1$ now. By induction hypothesis, $\mathrm{fall}_D\, p = \mathrm{fall}_D\, t_{n-1}$. By Proposition 4.7.2(v), $\mathrm{fall}_D\, t_{n-1} = \mathrm{fall}_D\, q$. Altogether, $\mathrm{fall}_D\, p = \mathrm{fall}_D\, q$.

On the other hand, if $\mathrm{fall}_D\, p = \mathrm{fall}_D\, q$ then $p^\circ = (\mathrm{fall}_D\, p)^\circ = (\mathrm{fall}_D\, q)^\circ = q^\circ$. □

Definition 4.7.6(i) introduces redundancy condition on arc set D requiring that the computation steps corresponding to the arcs of D have no influence to state. Program simplification introduced by Definition 4.7.6(ii) is basically program slicing on control flow graphs. The criterion of simplification tells that the action of an atomic step of the resulting program coincides with the action of the corresponding atomic step in the original program.

Definition 4.7.6. *Let $\psi \in \mathrm{STList}\, AConf \to 1 + AConf$ be a sound operator. Let S be a program and let $D \subseteq E(\mathrm{tcfg}\, S)$ be total.*

(i) *Call D* redundant on basis ψ *iff* $\mathrm{st}(\mathrm{next}_\psi\langle s \mid s\rangle) = s$ *for every $s \in V(\mathrm{tcfg}\, S)$ and $s \in State$ such that* $\mathrm{arc}(\mathrm{next}_\psi\langle s \mid s\rangle) \in D$.

(ii) *Assume* $D \subseteq E(\text{tcfg}\, S)$ *being total. For any prorgam* S°*, say that* S° simplifies S by D *iff there exists a sliceprojection from* $\text{tcfg}\, S$ *to* $\text{tcfg}\, S^\circ$ *such that, for every* $p \in V(\text{cfg}\, S)$ *and* $s \in \mathit{State}$*,*

$$\text{next}_\psi\langle p^\circ \mid s\rangle = \big(\text{next}_\psi\langle \text{fall}_D\, p \mid s\rangle\big)^\circ \quad .$$

In most cases, we will be restricted to simplifications by redundant sets in our theory. This does not lose generality as we assume we can always make an approximation step replacing the set of arcs having to be sliced away with a redundant set.

Lemma 4.7.7 shows that transfinite sequence of redundancy is a redundancy.

Lemma 4.7.7. *Let* $\psi \in \text{STList}\, \mathit{AConf} \to 1 + \mathit{AConf}$ *be a sound intuitive operator. Let* S *be a program and* $l = \mathcal{T}_\psi(S)(i)$ *for an* $i \in \mathit{State}$*. Let* $D \subseteq E(\text{tcfg}\, S)$ *be redundant on basis* ψ*. Let ordinals* o, π *be such that* $o \leqslant \pi < |l|$ *and* $\text{arc}(l(\varrho)) \in D$ *for every* ϱ *satisfying* $o < \varrho \leqslant \pi$*. Then* $\text{st}(l(o)) = \text{st}(l(\pi))$*.*

Proof. Choose $X \in \mathit{Var}$ arbitrarily. For any $e \in D$, suppose $c \in \mathit{AConf}$ being such that $\text{arc}(\text{next}_\psi\, c) = e$. Then $\text{st}(\text{next}_\psi\, c) = \text{st}(\text{next}_\psi\langle \sigma(e) \mid \text{st}\, c\rangle) = \text{st}\, c$ by redundancy. Therefore $\text{val}\, X(\text{next}_\psi\, c) = \text{val}\, X\, c$. By Definition 4.4.1, $X \notin \text{def}_\psi\, e$. Hence $X \in \text{def}_\psi(\text{arc}(l(\varrho)))$ for no ordinal ϱ satisfying $o < \varrho \leqslant \pi$. By Lemma 4.4.2, $\text{val}\, X(l(\varrho)) = \text{val}\, X(l(o))$ for every ϱ satisfying $o \leqslant \varrho \leqslant \pi$.

Hence $\text{st}(l(o)) = \text{st}(l(\varrho))$ for every ordinal ϱ satisfying $o \leqslant \varrho \leqslant \pi$. In particular, $\text{st}(l(o)) = \text{st}(l(\pi))$. □

For any $D \subseteq \mathit{AS}$, we denote by $\text{uses}\, D$ the predicate being true on augmented configurations c with $\text{arc}\, c \in D$. Thus $\text{uses}\, D\ ;\ \neg$ equals to the predicate being true on just the other configurations, i.e. $\text{uses}\, D\ ;\ \neg$ can be read "does not use D" in English.

The following lemma will be used as an auxiliary result in the rest. Note that k can be replaced with l in the claim since we can take $\alpha = 0$, $\beta = |l|$.

Lemma 4.7.8. *Let* $\psi \in \text{STList}\, \mathit{AConf} \to 1 + \mathit{AConf}$ *be a regular intuitive limit operator with* $\propto \geqslant \omega^\omega$*. Let* S *be a regular program. Let* $D \subseteq E(\text{tcfg}\, S)$ *be redundant on basis* ψ *and let* R *be a set of program points of* S *from which no arc of* D *starts. Let* $\cdot^\circ$ *be a sliceprojection from* $\text{tcfg}\, S$ *to some graph w.r.t.* D*. Let* $l = \mathcal{T}_\psi(S)(i)$ *and* $k = \text{drop}\, \alpha(\text{take}\, \beta\, l)$ *where* $\alpha \leqslant \beta \leqslant |l|$ *and* $\text{arc}(l(\alpha)) \notin D$*,* $\text{arc}(l(\beta)) \notin D$*. Then*

$$\begin{aligned}&\text{map}(\text{conf}\ ;\ \cdot^\circ)(\text{filter}(\text{pp}\ ;\ (\in R))\, k)\\ &\qquad = \text{map}(\text{conf}\ ;\ \cdot^\circ)(\text{filter}(\text{pp}\ ;\ \text{fall}_D\ ;\ (\in R))(\text{filter}(\text{uses}\, D\ ;\ \neg)\, k)) \quad .\end{aligned}$$

Proof. Let $o = |\,\mathrm{filter}(\mathrm{pp}\ ;\ (\in R))\,l|$ and let $(\varrho_\zeta : \zeta < o)$ be the increasing transfinite sequence of all ordinals ϱ_ζ such that $\mathrm{pp}(l(\varrho_\zeta)) \in R$. For each ζ, let π_ζ be the least ordinal such that $\mathrm{arc}(l(\xi)) \in D$ for every ξ satisfying $\pi_\zeta < \xi \leqslant \varrho_\zeta$. This definition is correct by Lemma 4.7.3(v) whereby never $\mathrm{arc}(l(\pi_\zeta)) \in D$.

Suppose $\zeta_1 < \zeta_2 < o$. Then $\varrho_{\zeta_1} < \varrho_{\zeta_2}$ by construction. As $\mathrm{arc}(l(\varrho_{\zeta_1}+1)) \notin D$ and $\varrho_{\zeta_1} + 1 \leqslant \varrho_{\zeta_2}$, it must be that $\varrho_{\zeta_1} + 1 \leqslant \pi_{\zeta_2}$. Hence $\pi_{\zeta_1} \leqslant \varrho_{\zeta_1} < \pi_{\zeta_2}$.

Take an ordinal $\tau < |l|$ such that both $\mathrm{arc}(l(\tau)) \notin D$ and $\mathrm{fall}_D(\mathrm{pp}(l(\tau))) \in R$ hold. Let ζ be the least ordinal such that $\tau \leqslant \varrho_\zeta$. Such ζ exists since a walk from $\mathrm{pp}(l(\tau))$ to f using all arcs of $\mathrm{drop}(\tau+1)l$ passes through $\mathrm{fall}_D(\mathrm{pp}(l(\tau))) \in R$. Then $\tau \leqslant \pi_\zeta$ since $\mathrm{arc}(l(\tau)) \notin D$. Let υ be the greatest ordinal such that $\mathrm{arc}(l(\upsilon)) \in D$ for every ξ satisfying $\tau < \xi \leqslant \upsilon$. By Proposition 4.7.3(iv), this definition is correct and $\mathrm{pp}(l(\upsilon)) = \mathrm{fall}_D(\mathrm{pp}(l(\tau))) \in R$ leading to $\varrho_\zeta \leqslant \upsilon$ by the choice of ζ. This inequality implies $\pi_\zeta \leqslant \tau$ since $\tau < \pi_\zeta$ would give $\upsilon < \pi_\zeta$. Hence $\tau = \pi_\zeta$.

Consequently, $(\pi_\zeta : \zeta < o)$ is the increasing transfinite sequence of ordinals π_ζ such that both $\mathrm{arc}(l(\pi_\zeta)) \notin D$ and $\mathrm{fall}_D(\mathrm{pp}(l(\pi_\zeta))) \in R$ hold. Filtering maintains components with indices ϱ_ζ on the left-hand side and components with indices π_ζ on the right-hand side.

Let μ be the least ordinal larger than each ζ satisfying $\pi_\zeta < \alpha$. Let $\tilde{\mu}$ be the least ordinal larger than each ζ satisfying $\varrho_\zeta < \alpha$. Fix a $\zeta < o$. If $\zeta < \tilde{\mu}$ then $\pi_\zeta \leqslant \varrho_\zeta < \alpha$, implying $\zeta < \mu$. If $\zeta \geqslant \tilde{\mu}$ then $\varrho_\zeta \geqslant \alpha$; the definition of π_ζ and the assumption $\mathrm{arc}(l(\alpha)) \notin D$ together give $\pi_\zeta \geqslant \alpha$, implying $\zeta \geqslant \mu$. Consequently, $\zeta < \mu \iff \zeta < \tilde{\mu}$ for each ζ, implying $\mu = \tilde{\mu}$.

Let ν be the least ordinal larger than each ζ satisfying $\pi_\zeta < \beta$. Analogously, ν equals to the least ordinal larger than each ζ satisfying $\varrho_\zeta < \beta$. Thus the sides of the desired equality are of equal length $\nu - \mu$.

It remains to show that $\big(\mathrm{conf}(l(\varrho_\zeta))\big)^\circ = \big(\mathrm{conf}(l(\pi_\zeta))\big)^\circ$ for every $\zeta < o$. Since $\mathrm{arc}(l(\varrho_\zeta + 1)) \notin D$, ϱ_ζ is the largest ordinal such that $\mathrm{arc}(l(\xi)) \in D$ for every ξ satisfying $\pi_\zeta < \xi \leqslant \varrho_\zeta$. By Lemma 4.7.3(iv), $\mathrm{fall}_D(\mathrm{pp}(l(\pi_\zeta))) = \mathrm{pp}(l(\varrho_\zeta))$; thus $\big(\mathrm{pp}(l(\varrho_\zeta))\big)^\circ = \big(\mathrm{pp}(l(\pi_\zeta))\big)^\circ$. By Lemma 4.7.7, $\mathrm{st}(l(\varrho_\zeta)) = \mathrm{st}(l(\pi_\zeta))$. Hence the desired claim follows. □

Theorem 4.7.9 states the semantic correctness of program simplification.

Theorem 4.7.9. *Let $\psi \in \mathrm{STList}\,AConf \to 1 + AConf$ be a regular intuitive limit operator with $\propto \geqslant \omega^\omega$. Let S be a regular program. Let $D \subseteq E(\mathrm{tcfg}\,S)$ be redundant on basis ψ. Let S° simplify S by D with sliceprojection $\cdot^\circ$. Let $l = \mathcal{T}_\psi(S)(i)$ for some $i \in \mathit{State}$. Denote*

$$l^\circ = \mathcal{T}_\psi(S^\circ)(i)\ , \qquad m = \mathrm{map}(\cdot^\circ)(\mathrm{filter}(\mathrm{uses}\,D\ ;\ \neg)\,l)\ .$$

Then $l^\circ = m$.

Proof. Program S is finite since it is regular; therefore S° is finite by construction. Hence, by the results obtained so far, both l and l° end at f.

Let $S = \{s \in V(\text{tcfg}\, S) \mid \forall e \in D\,(\sigma(e) \neq s) \bullet s\}$. Then $\text{fall}_D\, p \in S$ for every $p \in V(\text{tcfg}\, S)$. As $\text{pp}\ ;\ \text{fall}_D\ ;\ (\in S)$ is constantly true, and $\text{pp}\ ;\ \cdot^\circ$ and $\cdot^\circ\ ;\ \text{pp}$ work the same way on configurations c satisfying $\text{arc}\, c \notin D$, Lemma 4.7.8 implies

$$\begin{aligned}\text{map}(\text{pp}\ ;\ \cdot^\circ)(\text{filter}(\text{at}\, S)\, l) &= \text{map}(\text{pp}\ ;\ \cdot^\circ)(\text{filter}(\text{uses}\, D\ ;\ \neg)\, l)\\ &= \text{map}(\cdot^\circ\ ;\ \text{pp})(\text{filter}(\text{uses}\, D\ ;\ \neg)\, l)\\ &= \text{map}\,\text{pp}(\text{map}(\cdot^\circ)(\text{filter}(\text{uses}\, D\ ;\ \neg)\, l))\\ &= \text{map}\,\text{pp}\, m\ .\end{aligned}$$

Program point f occurs in l once at its end; as $f \in S$, it occurs also in $\text{filter}(\text{at}\, S)\, l$ once at its end. Thus f occurs once at the end also in $\text{map}(\text{pp}\ ;\ \cdot^\circ)(\text{filter}(\text{at}\, S)\, l)$ as $\cdot^\circ$ is injective on program points of S. Hence m visits f only once, at its end.

Now it remains to prove by induction on $o < |l^\circ|$ that $l^\circ(o) = m(o)$.

Consider the case $o = 0$. As the initial configurations do not use arcs, we obtain

$$l^\circ(0) = \langle i_{S^\circ} \mid i\rangle = \langle i_S \mid i\rangle^\circ = (l(0))^\circ = ((\text{filter}(\text{uses}\, D\ ;\ \neg)\, l)(0))^\circ = m(0).$$

Now let $o > 0$ with principal representation $o = \alpha + \gamma$. Supposing $o \geqslant |m|$, with help of the induction hypothesis, leads to $o = |m|$, $\gamma = 1$ and $\text{pp}(l^\circ(\alpha)) = \text{pp}(m(\alpha)) = f$ contradicting $o < |l^\circ|$. Hence $o < |m|$. Let τ be the ordinal such that the τth component of l corresponds to the oth component of m. Let σ be the ordinal such that the σth component of l corresponds to the αth component of m. Then $m(o) = (l(\tau))^\circ$ and $m(\alpha) = (l(\sigma))^\circ$.

Suppose $\gamma = 1$. By construction, $\xi = \tau$ is the least ordinal greater than σ such that $\text{arc}(l(\xi)) \notin D$. By Lemma 4.7.3(iii), $\tau = \varrho + 1$ for a ϱ whereby $\text{fall}_D(\text{pp}(l(\sigma))) = \sigma(\text{arc}(l(\tau))) = \text{pp}(l(\varrho))$. Hence, by the induction hypothesis together with simplification and Lemma 4.7.7,

$$\begin{aligned}l^\circ(o) &= \text{next}_\psi(l^\circ(\alpha)) = \text{next}_\psi(m(\alpha)) = \text{next}_\psi\,((l(\sigma))^\circ)\\ &= \text{next}_\psi\langle(\text{pp}(l(\sigma)))^\circ \mid \text{st}(l(\sigma))\rangle\\ &= \big(\text{next}_\psi\langle\text{fall}_D(\text{pp}(l(\sigma))) \mid \text{st}(l(\sigma))\rangle\big)^\circ\\ &= \big(\text{next}_\psi\langle\text{pp}(l(\varrho)) \mid \text{st}(l(\varrho))\rangle\big)^\circ = \big(\text{next}_\psi(l(\varrho))\big)^\circ = (l(\tau))^\circ = m(o)\ .\end{aligned}$$

Suppose at last that $\gamma > 1$. Denote $e = \text{arc}(l(\tau))$ and let it go from s to t. For every ordinal $\eta < o$, let ε_η be the ordinal such that the ε_ηth component of l corresponds to the ηth component of m. Let υ be the least ordinal greater than any of ε_η. By Lemma 4.7.3(ii), $\text{arc}(l(\upsilon)) \notin D$. Hence $\upsilon = \tau$ and e is transfinite. Let $\tau = \beta + \delta$ be the principal representation.

Let $(\pi_\zeta : \zeta < o)$ be the ascending family of indices of l at which components correspond to components of filter(uses D ; $\neg$)(take $\tau\, l$), i.e. also to components of take $o\, l^\circ$. Take $\zeta \geqslant \alpha$ such that $\beta \leqslant \pi_\zeta < \tau$ hold; this is possible since take $\tau\, l$ uses arcs outside D however far.
Denote $k = \mathrm{drop}\, \pi_\zeta(\mathrm{take}\, \tau\, l)$ and $k^\circ = \mathrm{drop}\, \zeta(\mathrm{take}\, o\, l^\circ)$. It is easy to see that $\mathrm{map}(\cdot^\circ)(\mathrm{filter}(\mathrm{uses}\, D\ ;\ \neg)\, k) = \mathrm{drop}\, \zeta(\mathrm{take}\, o\, m) = k^\circ$. Lemma 4.7.8 gives

$$\begin{aligned}
\mathrm{map}(\mathrm{conf}\ ;\ \cdot^\circ)(\mathrm{filter}(\mathrm{at}\, S)\, k) &= \mathrm{map}(\mathrm{conf}\ ;\ \cdot^\circ)(\mathrm{filter}(\mathrm{uses}\, D\ ;\ \neg)\, k) \\
&= \mathrm{map}(\cdot^\circ\ ;\ \mathrm{conf})(\mathrm{filter}(\mathrm{uses}\, D\ ;\ \neg)\, k) \\
&= \mathrm{map}\, \mathrm{conf}(\mathrm{map}(\cdot^\circ)(\mathrm{filter}(\mathrm{uses}\, D\ ;\ \neg)\, k)) \\
&= \mathrm{map}\, \mathrm{conf}\, k^\circ \ .
\end{aligned}$$

By transfinite soundness, s is looping in k. Thus, as $s \in S$, program point s occurs endlessly in $\mathrm{map}\, \mathrm{pp}(\mathrm{filter}(\mathrm{at}\, S)\, k)$. Hence s° is looping in k°, i.e., also in take $o\, l^\circ$. As e° is a transfinite arc from s° to t°, this arc is used by l° to escape from take $o\, l^\circ$ by regularity. This means $\mathrm{arc}(l^\circ(o)) = e^\circ = \mathrm{arc}(m(o))$.
Furthermore, as st = conf ; st = $\cdot^\circ$; st, as well as at $\{s\}$ = conf ; at $\{s\}$, and $\cdot^\circ$ is injective on S, we obtain

$$\begin{aligned}
&\mathrm{map}\, \mathrm{st}(\mathrm{filter}(\mathrm{at}\, \{s\})(\mathrm{take}\, \delta(\mathrm{drop}\, \pi_\zeta\, l))) \\
&\quad = \mathrm{map}\, \mathrm{st}(\mathrm{filter}(\mathrm{at}\, \{s\})\, k) \\
&\quad = \mathrm{map}\, \mathrm{st}(\mathrm{filter}(\mathrm{at}\, \{s\})(\mathrm{filter}(\mathrm{at}\, S)\, k)) \\
&\quad = \mathrm{map}\, \mathrm{st}(\mathrm{map}\, \mathrm{conf}(\mathrm{filter}(\mathrm{conf}\ ;\ \mathrm{at}\, \{s\})(\mathrm{filter}(\mathrm{at}\, S)\, k))) \\
&\quad = \mathrm{map}\, \mathrm{st}(\mathrm{filter}(\mathrm{at}\, \{s\})(\mathrm{map}\, \mathrm{conf}(\mathrm{filter}(\mathrm{at}\, S)\, k))) \\
&\quad = \mathrm{map}\, \mathrm{st}(\mathrm{map}(\cdot^\circ)(\mathrm{filter}(\cdot^\circ\ ;\ \mathrm{at}\, \{s^\circ\})(\mathrm{map}\, \mathrm{conf}(\mathrm{filter}(\mathrm{at}\, S)\, k)))) \\
&\quad = \mathrm{map}\, \mathrm{st}(\mathrm{filter}(\mathrm{at}\, \{s^\circ\})(\mathrm{map}(\mathrm{conf}\ ;\ \cdot^\circ)(\mathrm{filter}(\mathrm{at}\, S)\, k))) \\
&\quad = \mathrm{map}\, \mathrm{st}(\mathrm{filter}(\mathrm{at}\, \{s^\circ\})(\mathrm{map}\, \mathrm{conf}\, k^\circ)) \\
&\quad = \mathrm{map}\, \mathrm{st}(\mathrm{filter}(\mathrm{at}\, \{s^\circ\})\, k^\circ) \\
&\quad = \mathrm{map}\, \mathrm{st}(\mathrm{filter}(\mathrm{at}\, \{s^\circ\})(\mathrm{take}\, \gamma(\mathrm{drop}\, \zeta\, l^\circ))) \ .
\end{aligned}$$

By Lemma 4.3.10(iii), $\mathrm{st}(l^\circ(o)) = \mathrm{st}(l(\tau)) = \mathrm{st}(m(o))$.
Altogether, $l^\circ(o) = m(o)$. This concludes the proof. □

4.8 Correctness of Program Slicing

We are now going to account for correctness of two standard slicing algorithms. These algorithms are classically stated for classical control flow graphs but it is straightforward to adopt them to take transfinite arcs into account.

Let S be a fixed program and let C be a slicing criterion. In the case of both algorithms, the crucial point is that there exists a relevance system (S, rel) of S where S is the set of program points to be retained by the output slice of the algorithm. We show this first and then turn to the correctness issue.

At the first step, we will give informal description of both slicing algorithms. Both make use of classical def-sets and ref-sets which correspond, contrasting to our theory, to program points rather than computation steps. For each program point $p \in \mathrm{cfg}\, S$, its classical def-set $\underline{\mathrm{def}}\, p$ contains at least all variables possibly updated by a computation step starting from p and the classical ref-set $\underline{\mathrm{ref}}\, p$ contains at least all variables whose value is accessed by a computation step starting from p.

Firstly, consider the algorithm based on so-called relevant sets. This approach was the first in history, it occurred already in Weiser's works [20]. Originally, the computation process was formulated in rather complex way [20, 18] where the computation consists of iteration of analysis process (just one analysis of traditional form does not necessarily give the desired result).

More precisely, the domain of the analysis of the original algorithm is $\wp(\mathit{Var})$ (for each program point, a set of "relevant" variables is computed) together with inclusion order. Initially, every program point p is associated with set $i_{\mathrm{RS}}(p) = C^{\rightarrowtail}(p)$ (the variables declared by the slicing criterion are relevant). At any step of the analysis concerning an arc e from p to q in the control flow graph, variables X that meet either of the following conditions are added to the relevant set of p:

1. X is relevant at q while not belonging to $\underline{\mathrm{def}}\, p$ (i.e. the relevant value at q exists already at p);
2. $X \in \underline{\mathrm{ref}}\, p$ while $\underline{\mathrm{def}}\, p$ contains a variable already found to be relevant at q (i.e. the value of X at p can possibly influence, via the computation step corresponding to e, the value of some relevant variables).

This means that the transition functions $f_{\mathrm{RS}}(e)$ of the backward analysis are defined by

$$f_{\mathrm{RS}}(e)(\mathcal{Z}) = (\mathcal{Z} \setminus \underline{\mathrm{def}}(\sigma(e))) \cup \left\{ \begin{array}{ll} \underline{\mathrm{ref}}(\sigma(e)) & \text{if } \underline{\mathrm{def}}(\sigma(e)) \cap \mathcal{Z} \neq \varnothing \\ \varnothing & \text{otherwise} \end{array} \right\} .$$

All program points p such that $\underline{\mathrm{def}}\, p$ contains a variable relevant at some immediate successor q of p in $\mathrm{cfg}\, S$ are taken into the slice, as well as all program points p such that $C^{\rightarrowtail}(p) \neq \varnothing$. This is not the end; then all variables belonging to $\underline{\mathrm{ref}}\, p$ for any branching point p for which there is a program point q control dependent on it and being in the slice are added to the relevant set of p. The analysis together with this additional step is repeated until no more points are added into the slice.

It is possible to compute all the information needed with one backward analysis but control dependence arcs must then be added to the graph on which the analysis is performed. The domain of the new analysis is $\wp(Var) \times \mathbb{T}$ whereby ff $<$ tt and the order on pairs is defined componentwise. The truth value tells for each program point whether it has to be taken into the slice. The set components are initialized as in the previous variant of the algorithm; the Boolean component is initially true for f and all program points occurring in the slicing criterion. This means that the initial values associated to the program points are defined by

$$i_{\mathrm{RS}'}(p) = (C^{\rightarrowtail}(p), \left\{ \begin{array}{ll} \mathrm{tt} & \text{if } C^{\rightarrowtail}(p) \neq \varnothing \text{ or } p = f \\ \mathrm{ff} & \text{otherwise} \end{array} \right\}) \ .$$

For each normal arc e from p to q, the new transition function is defined by

$$f_{\mathrm{RS}'}(e)(\mathcal{Z}, b) = (\mathcal{Z} \setminus \underline{\mathrm{def}}(\sigma(e)), \mathrm{ff}) \vee \left\{ \begin{array}{ll} (\underline{\mathrm{ref}}(\sigma(e)), \mathrm{tt}) & \text{if } \underline{\mathrm{def}}(\sigma(e)) \cap \mathcal{Z} \neq \varnothing \\ (\varnothing, \mathrm{ff}) & \text{otherwise} \end{array} \right\} \ .$$

For any control dependence arc d, we have

$$f_{\mathrm{RS}'}(d)(\mathcal{Z}, b) = (\left\{ \begin{array}{ll} \underline{\mathrm{ref}}(\sigma(d)) & \text{if } b = \mathrm{tt} \\ \varnothing & \text{otherwise} \end{array} \right\}, b) \ .$$

This way, the information about program points taken into the slice are propagated around during the analysis process and no repetition of analysis is needed.
Algorithm 4.1 summarizes this briefly.

Input: a program S.

1. Compute tcfg S.
2. Compute control dependences and form a new graph tcfg$'$ S obtained from tcfg S by adding special arcs from p to q whenever q is control dependent on p.
3. Perform backward analysis RS$'$ defined by $i_{\mathrm{RS}'}$ and $f_{\mathrm{RS}'}$ on graph tcfg$'$ S.
4. Assign to S the set of all program points of S for which the analysis computed a pair with second component tt.

Output: The program obtained from S *by omitting all statements whose arc in* tcfg S *does not start from a vertex of* S.

Algorithm 4.1: Computing slices via relevant sets

The author developed the latter variant of slicing via relevant sets when he was writing a slicer within the DAEDALUS-project of program analysis [16]. The

advantage was the chance to reuse the modules of program analysis directly for slicing without modifying them (the analysis process was programmed to work on abstract graphs; computing control flow graphs was kept separate, so it was easy to replace the graph).

The idea behind the approach of relevant sets is to compute a relevance system directly. We can define a data flow approximation system, taking $\mathrm{def}_\vee\, e = \underline{\mathrm{def}}(\sigma(e))$ for every $e \in E(\mathrm{cfg}\, S)$, as well as $\mathrm{ref}(e, X) = \underline{\mathrm{ref}}(\sigma(e))$ for every $e \in E(\mathrm{cfg}\, S)$ together with $X \in \mathrm{def}_\vee\, e$ and $\mathrm{ref}(p, q) = \underline{\mathrm{ref}}\, p$ for every $p \in V(\mathrm{cfg}\, S)$ together with $q \in \mathrm{dep}^{\rightarrowtail} p$. For example, if e corresponds to the assignment X `:=` Y `+` Z then one obtains $\mathrm{def}_\vee\, e = \{X\}$ and $\mathrm{ref}(e, X) = \{Y, Z\}$. For any transfinite arc e, take $\mathrm{def}_\vee\, e = \varnothing$.

According to the result of the computation of the relevant sets and the slice, define $\mathrm{rel}\, p$ for each program point p to be the set of all variables decided as relevant at p. Let S be the set of all program points to be included into the slice, inclusive of f. It is straightforward to see that (S, rel) is a relevance system w.r.t. the data flow approximation system whereby $X \in \mathrm{rel}\, p$ for every $(p, X) \in C$.

The other classical way of slicing is reducing the task to a reachability problem in the data and control dependence graph [18] (i.e. the directed graph whose vertices are program points of S and every arc indicates either data or control dependence). The same classical def-sets and ref-sets as before may be taken as the starting point. Next, data dependence approximations are computed. A program point q is considered *data dependent on* p iff, for some variable X, the following conditions hold:

1. $X \in \underline{\mathrm{def}}\, p$;
2. $X \in \underline{\mathrm{ref}}\, q$ or $(q, X) \in C$;
3. there is a walk $w = (v_0, e_1, v_1, \ldots, e_n, v_n)$ from p to q in $\mathrm{cfg}\, S$ such that $X \in \underline{\mathrm{def}}\, v_i$ for no i satisfying $0 < i < n$.

The slice is then obtained as the set S of all program points from which there exists a directed path in data and control dependence graph to a vertex mentioned in the slicing criterion, together with f. (As computation steps correspond to arcs rather than program points, including f into the slice does not mean that some final statement is always included. The final vertex must be included just because it corresponds to the finished computation doing 0 steps.)

To adopt this approach safely to the case of transfinite arcs, replace the third condition of data dependence with the following:

3'. there is a walk $w = (v_0, e_1, v_1, \ldots, e_n, v_n)$ from p to q in $\mathrm{tcfg}\, S$ such that $e_1 \in E(\mathrm{cfg}\, S)$ and $X \in \underline{\mathrm{def}}\, v_i$ for no i satisfying both $0 < i < n$ and $e_i \in E(\mathrm{cfg}\, S)$.

For walks w laying completely in cfg S, conditions 3' and 3 coincide.
This method is summarized in Algorithm 4.2.

Input: a program S.

1. Compute tcfg S.
2. Compute both control dependences and data dependences. Form a new graph pdg S whose vertices coincide with vertices of tcfg S and arcs go from p to q iff q is either control or data dependent on p.
3. Compute reachability information for pdg S.
4. Assign to S the set of all program points of S from which some program point mentioned by criterion C is reachable in pdg S.

Output: The program obtained from S *by omitting all statements whose arc in* tcfg S *does not start from a vertex of* S.

Algorithm 4.2: Computing slices via data dependences

Define the data flow approximation system according to the classical def-sets and ref-sets as before. In this approach, relevance system is not computed but it can abstractly be attached to the result. This can be done, for example, as follows. For every program point $s \in \mathsf{S}$, let

$$\mathrm{rel}\, s = \underline{\mathrm{ref}}\, s \cup C^{\rightarrowtail}(s) \ . \tag{4.5}$$

For every program point $s \in V(\mathrm{cfg}\, S) \setminus \mathsf{S}$, let rel s consist of precisely the variables for which there is a walk $w = (v_0, e_1, v_1, \ldots, e_n, v_n)$ in tcfg S from s to some $r \in \mathsf{S}$ such that $X \in \mathrm{rel}\, r$ and $X \in \underline{\mathrm{def}}\, v_i$ for no i satisfying both $0 \leqslant i < n$ and $e_i \in E(\mathrm{cfg}\, S)$.
This way, $(\mathsf{S}, \mathrm{rel})$ is a relevance system w.r.t. the fixed data flow approximation system whereby $X \in \mathrm{rel}\, p$ for every $(p, X) \in C$. The latter condition holds by Eq. 4.5 because all the program points mentioned by C are in S. In the following a few paragraphs, we prove that $(\mathsf{S}, \mathrm{rel})$ is a relevance system.
By construction, we immediately have condition 1 of Definition 4.5.2(i).
Consider condition 2. Take arbitrary p and q such that there is an arc d from p to q in tcfg S. Choose arbitrary $X \in \mathrm{rel}\, q \setminus \mathrm{def}_{\vee}\, d$. If $q \notin \mathsf{S}$ then there exists a walk $w = (v_0, e_1, v_1, \ldots, e_n, v_n)$ from q to some $r \in \mathsf{S}$ such that $X \in \mathrm{rel}\, r$ and $X \in \underline{\mathrm{def}}\, v_i$ for no v_i satisfying both $0 \leqslant i < n$ and $e_i \in E(\mathrm{cfg}\, S)$. If $q \in \mathsf{S}$ then take $w = (q)$ (the empty path from q to q), it satisfies the same property with $r = q$. If $d \in E(\mathrm{cfg}\, S)$ then $\mathrm{def}_{\vee}\, d = \underline{\mathrm{def}}\, p$ and the walk v starting from p, going

to q via d, and continuing along w, meets the same property. If $d \notin E(\text{cfg}\, S)$ then v also satisfies this property. Hence $X \in \text{rel}\, p$.

Consider condition 3. Let $X \in \text{def}_\vee\, e \cap \text{rel}\, q$. As $X \in \text{rel}\, q$, there exists a walk $w = (v_0, e_1, v_1, \ldots, e_n, v_n)$ in tcfg S from q to some $r \in S$ such that $X \in \text{rel}\, r$ and $X \in \underline{\text{def}}\, v_i$ for no i satisfying both $0 \leqslant i < n$ and $e_i \in E(\text{cfg}\, S)$. Note that $e \in E(\text{cfg}\, S)$ because $\text{def}_\vee\, e \neq \varnothing$. Hence the walk starting from p, going to q via e, and continuing along w, makes evident that r is data dependent on p. Therefore $p \in S$ by construction of S. But $p \in S$ implies $\underline{\text{ref}}\, p \subseteq \text{rel}\, p$.

Consider condition 4 now. Suppose we have $p\ \text{dep}\ q$ and $q \in S$. If $q \neq f$ then $p \in S$ by definition of S. But $q = f$ contradicts $p\ \text{dep}\ q$ since the final point cannot be control dependent on any point. So $p \in S$ implying also $\underline{\text{ref}}\, p \subseteq \text{rel}\, p$.

This concludes the proof.

Both algorithms find a set S of program points which the slice should consist of. Of course, the arcs between them are the important part. The idea is that precisely the computation steps corresponding to the arcs starting from a vertex of S are important. All the other arcs could be eliminated via an appropriate sliceprojection. Proposition 4.8.1 states that this is possible: all arcs corresponding to irrelevant statements (according to a relevance system) form a total set.

Proposition 4.8.1. *Let $\psi \in \text{STList}\, AConf \to 1{+}AConf$ be a transfinitely sound operator. Let (S, rel) be a relevance system for S w.r.t. a data flow approximation system for ψ. Then $D = \{e \in E(\text{tcfg}\, S) \mid \sigma(e) \notin S \bullet e\}$ is total.*

Proof. Take $e \in D$ arbitrarily. Then $\sigma(e) \notin S$, hence all arcs starting from $\sigma(e)$ meet the criterion of belonging to D. Suppose q is control dependent on $\sigma(e)$. Then assuming $q \in S$ would lead to a contradiction with (S, rel) being a relevance system. Thus $q \notin S$ and therefore all arcs starting from q belong to D. □

In our terms, the slice constructed according to S is a simplification of S by $\{e \in E(\text{tcfg}\, S) \mid \sigma(e) \notin S \bullet e\}$.

Theorem 4.8.2 is a combination of Theorems 4.6.3 and 4.7.9 and touches consecutive approximation and simplification where the arcs of the intermediate graph with changed behaviour w.r.t. the originals form a redundant set. The point of the theorem can be given with the words of Reps and Yang [14], changing them a bit to accommodate to our case: “Slice captures a portion of a program’s behaviour in the sense that, for any initial state, the program and the slice compute the same transfinite sequence of values for each element of the slice.”

Theorem 4.8.2. *Let $\psi \in \text{STList}\, AConf \to 1{+}AConf$ be a regular intuitive limit operator with $\propto \geqslant \omega^\omega$ and let D be a data flow approximation system for ψ. Let (S, rel) be a relevance system for a regular program S w.r.t. D. Let $\overline{S}$ approximate*

S on base S and D with isomorphism $\overline{\cdot}$. Let $\overline{D} \subseteq \{e \in E(\mathrm{tcfg}\, S) \mid \sigma(e) \notin S \bullet \overline{e}\}$ be redundant on basis ψ. Let S° simplify $\overline{S}$ by $\overline{D}$ with sliceprojection $\cdot^\circ$. Let $l = \mathcal{T}_\psi(S)(i)$ and $l^\circ = \mathcal{T}_\psi(S^\circ)(i)$ for some $i \in$ State. Let $R \subseteq S$ and denote $k = \mathrm{filter}(\mathrm{at}\, R)\, l$ and $k^\circ = \mathrm{filter}(\mathrm{at}\, \overline{R}^\circ)\, l^\circ$. Then $\mathrm{map}(\mathrm{pp}\, ;\, \overline{\cdot}\, ;\, \cdot^\circ)\, k = \mathrm{map}\, \mathrm{pp}\, k^\circ$ and, for every $\zeta < |k|$ and $X \in \mathrm{rel}(\mathrm{pp}(k(\zeta)))$, one has $\mathrm{val}\, X(k(\zeta)) = \mathrm{val}\, X(k^\circ(\zeta))$.

Proof. Denote $\overline{l} = \mathcal{T}_\psi(\overline{S})(i)$ and $\overline{k} = \mathrm{filter}(\mathrm{at}\, \overline{R})\, \overline{l}$.
Note that

$$\begin{aligned}
\mathrm{at}\, \overline{R}^\circ c^\circ &\iff \exists r \in R\, (\mathrm{pp}\, c^\circ = \overline{r}^\circ) \\
&\iff \exists r \in R\, ((\mathrm{pp}\, c)^\circ = \overline{r}^\circ) \\
&\iff \exists r \in R\, (\mathrm{fall}_{\overline{D}}(\mathrm{pp}\, c) = \mathrm{fall}_{\overline{D}}\, \overline{r}) \\
&\iff \exists r \in R\, (\mathrm{fall}_{\overline{D}}(\mathrm{pp}\, c) = \overline{r}) \\
&\iff \mathrm{fall}_{\overline{D}}(\mathrm{pp}\, c) \in \overline{R}\ .
\end{aligned}$$

From Theorem 4.7.9,

$$l^\circ = \mathrm{map}(\cdot^\circ)(\mathrm{filter}(\mathrm{uses}\, \overline{D}\, ;\, \neg)\, \overline{l})\ ,$$

implying

$$\begin{aligned}
k^\circ &= \mathrm{filter}(\mathrm{at}\, \overline{R}^\circ)\, l^\circ \\
&= \mathrm{filter}(\mathrm{at}\, \overline{R}^\circ)(\mathrm{map}(\cdot^\circ)(\mathrm{filter}(\mathrm{uses}\, \overline{D}\, ;\, \neg)\, \overline{l})) \\
&= \mathrm{map}(\cdot^\circ)(\mathrm{filter}(\cdot^\circ\, ;\, \mathrm{at}\, \overline{R}^\circ)(\mathrm{filter}(\mathrm{uses}\, \overline{D}\, ;\, \neg)\overline{l})) \\
&= \mathrm{map}(\cdot^\circ)(\mathrm{filter}(\mathrm{pp}\, ;\, \mathrm{fall}_{\overline{D}}\, ;\, (\in \overline{R}))(\mathrm{filter}(\mathrm{uses}\, \overline{D}\, ;\, \neg)\overline{l}))\ .
\end{aligned}$$

Lemma 4.7.8 gives

$$\begin{aligned}
&\mathrm{map}(\mathrm{pp}\, ;\, \cdot^\circ)(\mathrm{filter}(\mathrm{at}\, \overline{R})\, \overline{l}) \\
&\qquad = \mathrm{map}(\mathrm{pp}\, ;\, \cdot^\circ)(\mathrm{filter}(\mathrm{pp}\, ;\, \mathrm{fall}_{\overline{D}}\, ;\, (\in \overline{R}))(\mathrm{filter}(\mathrm{uses}\, \overline{D}\, ;\, \neg)\, \overline{l}))\ , \\
&\mathrm{map}\, \mathrm{st}(\mathrm{filter}(\mathrm{at}\, \overline{R})\, \overline{l}) = \mathrm{map}\, \mathrm{st}(\mathrm{filter}(\mathrm{pp}\, ;\, \mathrm{fall}_{\overline{D}}\, ;\, (\in \overline{R}))(\mathrm{filter}(\mathrm{uses}\, \overline{D}\, ;\, \neg)\, \overline{l}))\ .
\end{aligned}$$

Let $m = \mathrm{filter}(\mathrm{at}\, S)\, l$ and $\overline{m} = \mathrm{filter}(\mathrm{at}\, \overline{S})\, \overline{l}$. Denote $\mathcal{R}(\zeta) = \mathrm{rel}(\mathrm{pp}(k(\zeta)))$ for every $\zeta < |k|$ and $\mathcal{Q}(\eta) = \mathrm{rel}(\mathrm{pp}(m(\eta)))$ for every $\eta < |m|$. Theorem 4.6.3 gives

$$\begin{aligned}
&\mathrm{map}(\mathrm{pp}\, ;\, \overline{\cdot})\, m = \mathrm{map}\, \mathrm{pp}\, \overline{m}\ , \\
&\forall \eta < |m|\, \left(\mathrm{st}(m(\eta))\big|_{\mathcal{Q}(\eta)} = \mathrm{st}(\overline{m}(\eta))\big|_{\mathcal{Q}(\eta)}\right)\ .
\end{aligned}$$

Observe that $k = \text{filter}(\text{at } R)\, m$ and $\overline{k} = \text{filter}(\text{at } \overline{R})\, \overline{m}$. We obtain

$$\begin{aligned}
\text{map}(\text{pp} \mathbin{;} \overline{\cdot})\, k &= \text{map}(\overline{\cdot})(\text{map pp}(\text{filter}(\text{at } R)\, m)) \\
&= \text{map}(\overline{\cdot})(\text{filter}(\in R)(\text{map pp}\, m)) \\
&= \text{filter}(\in \overline{R})(\text{map}(\overline{\cdot})(\text{map pp}\, m)) \\
&= \text{filter}(\in \overline{R})(\text{map}(\text{pp} \mathbin{;} \overline{\cdot})\, m) \\
&= \text{filter}(\in \overline{R})(\text{map pp}\, \overline{m}) \\
&= \text{map pp}(\text{filter}(\text{at } \overline{R})\, \overline{m}) \\
&= \text{map pp}\, \overline{k} \ .
\end{aligned}$$

Let $(o_\zeta : \zeta < |k|)$ be the increasing transfinite sequence of indices at which components of m satisfy at R. Then $k(\zeta) = m(o_\zeta)$ and $\mathcal{R}(\zeta) = \text{rel}(\text{pp}(k(\zeta))) = \text{rel}(\text{pp}(m(o_\zeta))) = \mathcal{Q}(o_\zeta)$ for each $\zeta < |k|$. But $\text{map}(\text{pp} \mathbin{;} \overline{\cdot})\, m = \text{map pp}\, \overline{m}$ implies that components of $\overline{m}$ which satisfy at $\overline{R}$ locate at the same positions o_ζ, $\zeta < |k|$, hence also $\overline{k}(\zeta) = \overline{m}(o_\zeta)$ for each $\zeta < |k|$. Thus

$$\text{st}(k(\zeta))\big|_{\mathcal{R}(\zeta)} = \text{st}(m(o_\zeta))\big|_{\mathcal{Q}(o_\zeta)} = \text{st}(\overline{m}(o_\zeta))\big|_{\mathcal{Q}(o_\zeta)} = \text{st}(\overline{k}(\zeta))\big|_{\mathcal{R}(\zeta)} \ .$$

The first desired claim is now proven by

$$\begin{aligned}
\text{map}(\text{pp} \mathbin{;} \overline{\cdot} \mathbin{;} \cdot^\circ)\, k &= \text{map}(\cdot^\circ)(\text{map}(\text{pp} \mathbin{;} \overline{\cdot})\, k) = \text{map}(\cdot^\circ)(\text{map pp}\, \overline{k}) \\
&= \text{map}(\cdot^\circ)(\text{map pp}(\text{filter}(\text{at } \overline{R})\, \overline{l})) \\
&= \text{map}(\text{pp} \mathbin{;} \cdot^\circ)(\text{filter}(\text{at } \overline{R})\, \overline{l}) \\
&= \text{map}(\text{pp} \mathbin{;} \cdot^\circ)(\text{filter}(\text{pp} \mathbin{;} \text{fall}_{\overline{D}} \mathbin{;} (\in \overline{R}))(\text{filter}(\text{uses } \overline{D} \mathbin{;} \neg)\overline{l})) \\
&= \text{map}(\cdot^\circ \mathbin{;} \text{pp})(\text{filter}(\text{pp} \mathbin{;} \text{fall}_{\overline{D}} \mathbin{;} (\in \overline{R}))(\text{filter}(\text{uses } \overline{D} \mathbin{;} \neg)\overline{l})) \\
&= \text{map pp}(\text{map}(\cdot^\circ)(\text{filter}(\text{pp} \mathbin{;} \text{fall}_{\overline{D}} \mathbin{;} (\in \overline{R}))(\text{filter}(\text{uses } \overline{D} \mathbin{;} \neg)\overline{l}))) \\
&= \text{map pp}\, k^\circ \ .
\end{aligned}$$

The second claim comes from

$$\begin{aligned}
&\text{st}(k(\zeta))\big|_{\mathcal{R}(\zeta)} \\
&\quad = \text{st}(\overline{k}(\zeta))\big|_{\mathcal{R}(\zeta)} \\
&\quad = \text{st}((\text{filter}(\text{at } \overline{R})\, \overline{l})(\zeta))\big|_{\mathcal{R}(\zeta)} \\
&\quad = \text{st}\left((\text{filter}(\text{pp} \mathbin{;} \text{fall}_{\overline{D}} \mathbin{;} (\in \overline{R}))(\text{filter}(\text{uses } \overline{D} \mathbin{;} \neg)\overline{l}))(\zeta)\right)\big|_{\mathcal{R}(\zeta)} \\
&\quad = \text{st}\left((\text{map}(\cdot^\circ)(\text{filter}(\text{pp} \mathbin{;} \text{fall}_{\overline{D}} \mathbin{;} (\in \overline{R}))(\text{filter}(\text{uses } \overline{D} \mathbin{;} \neg)\overline{l})))(\zeta)\right)\big|_{\mathcal{R}(\zeta)} \\
&\quad = \text{st}\left(k^\circ(\zeta)\right)\big|_{\mathcal{R}(\zeta)} \ .
\end{aligned}$$

□

Theorem 4.8.2 seemingly implies the desired semantic correctness of the two slicing algorithms, so that we have triumphed over the non-termination monster.

Roughly, this is true. However, there is one more concern which we have not discussed so far. The facts have been proven on flow graphs abstracting from details of programming languages but slicing is an operation on programs in a fixed language. To carry over the results to slicing programs, one should show that the transformations we considered on control flow graphs are indeed reflections of replacements and removals of atomic statements in program code. This would be a kind of result called *feasibility lemma* by Reps and Yang [14].

Feasibility can be problematic in the case of non-standard programming languages, for instance, those involving unstructured control flow. No proof uniform for all languages can be given. Proving feasibility remains out of scope of this thesis. We are satisfied with claiming that, for simple imperative programming languages, this is intuitively clear.

Under the assumption that all our transformations of flow graphs can be simulated on programs in a satisfactory manner, proving the desired correctness of program slicing algorithms is straightforward. This is done in Corollary 4.8.3(i). Note that if $R = \{r\}$ for a program point r then the result gives precisely the crucial property of slicing: computing the same sequence of values at r for every variable listed by the criterion as important at r.

Corollary 4.8.3. *Let $\psi \in \mathrm{STList}\, AConf \to 1 + AConf$ be a regular intuitive limit operator with $\propto \geqslant \omega^\omega$. Let S be a regular program and $\tilde{S}$ its slice w.r.t. criterion C found by one of the algorithms considered above. Let $\tilde{\cdot}$ denote the sliceprojection from* $\mathrm{tcfg}(S)$ *to* $\mathrm{tcfg}(\tilde{S})$ *w.r.t. the set of all omitted arcs. Let $l = \mathcal{T}_\psi(S)(i)$ and $\tilde{l} = \mathcal{T}_\psi(\tilde{S})(i)$ for some $i \in \mathit{State}$.*

(i) *Let R be arbitrary set of program points of S occurring in C. Denote $m =$* $\mathrm{filter}(\mathrm{at}\, R)\, l$ *and $\tilde{m} =$* $\mathrm{filter}(\mathrm{at}\, \tilde{R})\, \tilde{l}$*. Then* $\mathrm{map}(\mathrm{pp}\ ;\ \tilde{\cdot})\, m = \mathrm{map}\, \mathrm{pp}\, \tilde{m}$ *and, for every $\xi < |m|$ and $X \in C^{\rightarrowtail}(\mathrm{pp}(m(\xi)))$, one has* $\mathrm{val}\, X(m(\xi)) = \mathrm{val}\, X(\tilde{m}(\xi))$.

(ii) *The run of S° lasts at most as long as the run of S, i.e. $|\tilde{l}| \leqslant |l|$.*

Proof. By the analysis at the beginning of this section, there is a relevance system (S, rel) of S such that the computation steps maintained by the slicing are precisely those corresponding to the arcs which start from vertices of S; thereby, $\{p \in V(\mathrm{tcfg}\, S) \mid C^{\rightarrowtail}(p) \neq \varnothing \bullet p\} \subseteq S$ and $C^{\rightarrowtail}(p) \subseteq \mathrm{rel}(p)$ for every $p \in V(\mathrm{tcfg}\, S)$. Let $D = \{e \in E(\mathrm{tcfg}\, S) \mid \forall s \in S\, (\sigma(e) \notin S) \bullet e\}$.

Find an approximation $\overline{S}$ of S together with an isomorphism $\overline{\cdot}$ from $\mathrm{tcfg}\, S$ to $\mathrm{tcfg}\, \overline{S}$ such that every arc in $\overline{D}$ stands for a computation step with definitely no influence to data flow.

By Proposition 4.8.1, D is total, hence $\overline{D}$ is redundant. Let $\cdot^\circ$ be the mapping from tcfg $\overline{S}$ to tcfg $\tilde{S}$ w.r.t. $\overline{D}$ such that $\tilde{\cdot} = \overline{\cdot}\,;\,\cdot^\circ$. Then $\cdot^\circ$ is a sliceprojection w.r.t. $\overline{D}$.

By the choice of D, no arc of D starts from vertices of S. Furthermore, if no arc of D starts from a $p \in V(\text{tcfg}\,S)$ then if there exists an arc starting from p then $p \in S$ else $p = f \in S$. Thus $S = \{p \in V(\text{tcfg}\,S) \mid \forall e \in D\,(\sigma(e) \neq p) \bullet p\}$. Hence $\text{fall}_D\,p \in S$ for every $p \in V(\text{tcfg}\,S)$. Thus, by approximation,

$$\begin{aligned}\overline{\text{next}_\psi\langle\text{fall}_D\,p \mid s\rangle} = \text{next}_\psi\,\overline{\langle\text{fall}_D\,p \mid s\rangle} &= \text{next}_\psi\langle\overline{\text{fall}_D\,p} \mid s\rangle \\ &= \text{next}_\psi\langle\text{fall}_{\overline{D}}\,\overline{p} \mid s\rangle\ ,\end{aligned}$$

giving

$$\begin{aligned}\text{next}_\psi\langle\overline{p}^\circ \mid s\rangle &= \text{next}_\psi\langle\tilde{p} \mid s\rangle = \text{next}_\psi\widetilde{\langle\text{fall}_D\,p \mid s\rangle} \\ &= \left(\overline{\text{next}_\psi\langle\text{fall}_D\,p \mid s\rangle}\right)^\circ = \left(\text{next}_\psi\langle\text{fall}_{\overline{D}}\,\overline{p} \mid s\rangle\right)^\circ\ .\end{aligned}$$

Thus $\tilde{S}$ simplifies $\overline{S}$ by $\overline{D}$.

(i) Denote $\mathcal{R}(\zeta) = \text{rel}(\text{pp}(m(\zeta)))$ for every $\zeta < |m|$. By Theorem 4.8.2,

$$\begin{aligned}&\text{map}(\text{pp}\,;\,\tilde{\cdot})\,m = \text{map}\,\text{pp}\,\tilde{m}\ , \\ &\forall\zeta < |m|\,\left(\text{st}\,(m(\zeta))\,\big|_{\mathcal{R}(\zeta)} = \text{st}\,(\tilde{m}(\zeta))\,\big|_{\mathcal{R}(\zeta)}\right)\ .\end{aligned}$$

As $C^{\rightarrowtail}(p) \subseteq \text{rel}\,p$ for every p, this implies the desired claims.

(ii) By Theorem 4.8.2,

$$\text{map}(\text{pp}\,;\,\tilde{\cdot})(\text{filter}(\text{at}\,S)\,l) = \text{map}\,\text{pp}(\text{filter}(\text{at}\,\tilde{S})\,\tilde{l})\ .$$

Thus $|\,\text{filter}(\text{at}\,S)\,l| = |\,\text{filter}(\text{at}\,\tilde{S})\,\tilde{l}|$. By Theorem 4.7.9, $\tilde{l} = \text{map}(\cdot^\circ)\,\overline{m}$ where $\overline{m} = \text{filter}(\text{uses}\,\overline{D}\,;\,\neg)\,\overline{l}$.

Predicate $\text{at}\,\tilde{S} = \text{at}\,\overline{S}^\circ$ is constantly true on $V(\text{tcfg}\,\tilde{S})$ as $\text{fall}_{\overline{D}}\,\overline{p} \in \overline{S}$ for every $p \in V(\text{tcfg}\,S)$. Therefore $\text{filter}(\text{at}\,\tilde{S})\,\tilde{l} = \tilde{l}$. Hence

$$|\tilde{l}| = |\,\text{filter}(\text{at}\,\tilde{S})\,\tilde{l}| = |\,\text{filter}(\text{at}\,S)\,l| \leqslant |l|\ .$$

□

Corollary 4.8.3(ii) implies that if the original program terminates on an initial state (i.e. the length of its run is finite) then also the slice terminates on the same initial state. This means that it is possible to obtain correctness of slicing of terminating programs (a result like one proven by Reps and Yang [14]) as a corollary from correctness w.r.t. transfinite semantics. This can be done in so far as standard semantics are extensible to transfinite semantics meeting the requirements of our theory.

CHAPTER 5

DISCUSSION OF RELATED ISSUES

5.1 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 [20] has shown that the problem of finding statement minimal slices is undecidable but he considers slicing w.r.t. standard semantics. The argumentation he gives 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 **:= if** X **then** true **else** Z

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

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 different from tt. 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. So the value of X immediately after leaving a loop is an indicator of its termination/nontermination status.

By transfinite induction, it is easy to see that Z never takes value $\bot$ and, once having value tt, it keeps this value until the end of the run. 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 sits 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 we used to prove the undecidability of minimal slice problem simultaneously proves the undecidability of constant propagation. Constant propagation is the problem of determining, for a given variable X and program point p, whether the value of X at p depends on the way control reaches p and, if not, then finding the constant value. It is known to be undecidable in context of standard semantics. In the construction above, determining whether Z after the run of Z `:= false` ; S' is constantly ff would solve the halting problem for S, so constant propagation is undecidable also for our transfinite semantics.

We can give another proof to undecidability of minimal slice problem based on constant propagation. Let S be a program, X a variable in S and p a program point in S. Let c be a given value which a variable could have. Construct S' from S by inserting Y `:=` X ; Z `:=` Y directly before program point p where Y, Z are variables not occurring in S. Consider the task of finding minimal slice of the program Z `:=` E ; S' w.r.t. criterion $\{(p, Z)\}$ where E is an expression always evaluating to c. A hypothetical minimal slice problem solver would decide the constant value c of X at point p in S.

5.2 Fractional Semantics

The essential difference between standard trace semantics and transfinite trace semantics is that the states of traces of standard semantics can be indexed with natural numbers while those of transfinite semantics are indexed with ordinal numbers.

Many other kinds of numbers are used in mathematics; one may ask whether indexing the states with numbers of some other kind could be reasonable.

We have argued in [11] that indexing the trace components with rational numbers could enable passing by the difficulty to give transfinite semantics to recursive programs. Trace semantics where trace components are indexed with rational numbers are called *fractional* there. As all countable sets of ordinals can be mapped order-preservingly into any non-trivial interval of rationals, transfinite semantics in principle can be reformulated as fractional. Rational numbers form increasing, as well as decreasing infinite sequences, so the principal obstacle of using transfinite semantics for recursion which was pointed out at the end of Subsect. 1.2 does not appear in fractional semantics.

In [11], we defined a simple imperative language and a family of its fractional semantics in fixpoint form similar to usual definitions of denotational semantics. This family contains both a standard trace semantics and a transfinite trace semantics (in fractional form) which can be obtained by giving different values to a few parameters of the definition schema. Thus fractional semantics serves as a uniform framework for both standard and transfinite trace semantics.

The fractional semantics of the family studied in [11], however, can be only occasionally applied to recursive programs. Hence this approach has not yet been proved as a solver of the semantic anomaly problem for recursive programs.

As the work on fractional semantics is in progress yet and the exact framework developed in [11] will be more or less changed in the future works, we explain the behaviour of our fractional semantics only through examples in the thesis and do not go into precise details of the definition schema.

The fractional semantics defined in [11] are *binary* in the sense that all traces are built via interval bisection. As the starting interval is $[0;1]$, this means that only reduced fractions whose denominator is a power of 2 can occur as an index of a trace component. Another interesting property of our fractional semantics is that it associates the pieces of code statically with intervals of rationals. To a piece of code, the same interval of rationals is reserved irrespectively of the initial state. This is not so in standard or transfinite trace semantics since the components are enumerated with numbers without leaving gaps and the number of steps used by a part of a code depends on the initial state. We will observe this phenomenon in the examples following.

In the examples, traces are rational-indexed families of configurations. All indices belong to interval $[0;1]$. Each configuration is a pair of a program point and a variable state; we denote them like above. A program point must correspond to the rest of the code — the part of the program to be run yet. The latter must entail the current call-stack, including remainders of every pendent procedure. Relying on this observation, let program points be finite lists of code fragments where list

components in order correspond to unfinished procedures. The examples use only the most spread syntactic constructs; ε denotes the empty program.

Example 5.2.1. Running an assignment requires just one step. The trace must have two components: the initial state and the final one. No partition of the initial interval $[0;1]$ is therefore needed. The following are three examples about the meaning of assignment in our fractional semantics:

- the execution of `z := x` at initial state $(\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 0)$ gives trace

  $$0 \mapsto \langle [\mathsf{z} := \mathsf{x}] \mid (\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 0) \rangle \ ,$$
  $$1 \mapsto \langle [\varepsilon] \mid (\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1) \rangle \ ;$$

- the execution of `x := y` at initial state $(\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1)$ gives

  $$0 \mapsto \langle [\mathsf{x} := \mathsf{y}] \mid (\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1) \rangle \ ,$$
  $$1 \mapsto \langle [\varepsilon] \mid (\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1) \rangle \ ;$$

- the execution of `y := z` at initial state $(\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1)$ gives

  $$0 \mapsto \langle [\mathsf{y} := \mathsf{z}] \mid (\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1) \rangle \ ,$$
  $$1 \mapsto \langle [\varepsilon] \mid (\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 1, \mathsf{z} \mapsto 1) \rangle \ .$$

□

Example 5.2.2. Here we describe the semantics of sequential composition. In the trace of a run of S ; T, the traces of the runs of S and T occurring within it are compressed to twice shorter interval and joined together. Thereby, the code fragments in the trace of S are complemented with T.

This way, using the traces of Example 5.2.1, the execution trace of `x := y ; y := z` at initial state $(\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1)$ is

$$0 \mapsto \langle [\mathsf{x} := \mathsf{y} \,;\, \mathsf{y} := \mathsf{z}] \mid (\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1) \rangle \ ,$$
$$\tfrac{1}{2} \mapsto \langle [\mathsf{y} := \mathsf{z}] \mid (\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1) \rangle \ ,$$
$$1 \mapsto \langle [\varepsilon] \mid (\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 1, \mathsf{z} \mapsto 1) \rangle \ .$$

Note that the two compressed traces, the first of them complemented, have a common member at $\frac{1}{2}$. This double member fuses to one.

Analogously, the execution trace of the swap program `z := x ; (x := y ; y := z)` at initial state $(\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 0)$ is

$$0 \mapsto \langle [\mathsf{z} := \mathsf{x} \,;\, (\mathsf{x} := \mathsf{y} \,;\, \mathsf{y} := \mathsf{z})] \mid (\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 0) \rangle \ ,$$
$$\tfrac{1}{2} \mapsto \langle [\mathsf{x} := \mathsf{y} \,;\, \mathsf{y} := \mathsf{z}] \mid (\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1) \rangle \ ,$$
$$\tfrac{3}{4} \mapsto \langle [\mathsf{y} := \mathsf{z}] \mid (\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1) \rangle \ ,$$
$$1 \mapsto \langle [\varepsilon] \mid (\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 1, \mathsf{z} \mapsto 1) \rangle \ .$$

The assignments `z := x`, `x := y` and `y := z` are run within intervals $[0;\frac{1}{2}]$, $[\frac{1}{2};\frac{3}{4}]$ and $[\frac{3}{4};1]$, respectively. This is so independently of the initial state. □

Example 5.2.3. Let W = **`while`** `z > 0` **`do`** `z := z - 1` and consider the program `z := x` ; (W ; `y := z`). This program is obtained from the swap program of Example 5.2.2 by replacing the second assignment with W. Hence in the case of any execution of this program, the run of `z := x` lies in interval $[0;\frac{1}{2}]$, that of W in interval $[\frac{1}{2};\frac{3}{4}]$ and that of `y := z` in $[\frac{3}{4};1]$.

We place the run of W at variable state $(\mathsf{x}\mapsto 1,\mathsf{y}\mapsto 2,\mathsf{z}\mapsto 1)$ directly to $[\frac{1}{2};\frac{3}{4}]$. Running a while-loop consists of a predicate evaluation and something which depends on the result of this. In our case, the predicate evaluates to tt; thus the rest is not empty and thus the test step leads the trace to $\frac{5}{8}$ (the midpoint of $[\frac{1}{2};\frac{3}{4}]$). The rest of the run of the loop therefore lies within $[\frac{5}{8};\frac{3}{4}]$. Furthermore, the rest consists of the run of `z := z - 1` followed by a new run of W. Thus $[\frac{5}{8};\frac{3}{4}]$ is bisected, the run of `z := z - 1` is compressed to interval $[\frac{5}{8};\frac{11}{16}]$ and the run of W which consists of just one test this time is compressed to $[\frac{11}{16};\frac{3}{4}]$.

Thus the whole run looks as follows:

$$
\begin{aligned}
0 &\mapsto \langle [\mathsf{z := x}\,;(W\,;\mathsf{y := z})] \mid (\mathsf{x}\mapsto 1,\mathsf{y}\mapsto 2,\mathsf{z}\mapsto 0)\rangle\ ,\\
\tfrac{1}{2} &\mapsto \langle [W\,;\mathsf{y := z}] \mid (\mathsf{x}\mapsto 1,\mathsf{y}\mapsto 2,\mathsf{z}\mapsto 1)\rangle\ ,\\
\tfrac{5}{8} &\mapsto \langle [(\mathsf{z := z - 1}\,;W)\,;\mathsf{y := z}] \mid (\mathsf{x}\mapsto 1,\mathsf{y}\mapsto 2,\mathsf{z}\mapsto 1)\rangle\ ,\\
\tfrac{11}{16} &\mapsto \langle [W\,;\mathsf{y := z}] \mid (\mathsf{x}\mapsto 1,\mathsf{y}\mapsto 2,\mathsf{z}\mapsto 0)\rangle\ ,\\
\tfrac{3}{4} &\mapsto \langle [\mathsf{y := z}] \mid (\mathsf{x}\mapsto 1,\mathsf{y}\mapsto 2,\mathsf{z}\mapsto 0)\rangle\ ,\\
1 &\mapsto \langle [\varepsilon] \mid (\mathsf{x}\mapsto 1,\mathsf{y}\mapsto 0,\mathsf{z}\mapsto 0)\rangle\ .
\end{aligned}
$$

□

The next two examples describe the semantics of procedures and their calls.

Example 5.2.4. Consider procedure declaration

`proc` `q` **`is`** `x := y` .

A run of a procedure always ends with a return step. Therefore $[0;1]$ is bisected, the run of the body of the procedure is compressed to interval $[0;\frac{1}{2}]$ while interval $[\frac{1}{2};1]$ accommodates the return step. (The entrance step is handled together with the call.)

According to this principle, the execution trace of `q` at initial state $(\mathsf{x}\mapsto 1,\mathsf{y}\mapsto$

$2, \mathsf{z} \mapsto 1)$ is

$$0 \mapsto \langle [\mathsf{x} \texttt{ := } \mathsf{y}] \mid (\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1) \rangle \ ,$$
$$\tfrac{1}{2} \mapsto \langle [\varepsilon] \mid (\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1) \rangle \ ,$$
$$1 \mapsto \langle [] \mid (\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1) \rangle \ .$$

Note that the return step deletes the component corresponding to the finished procedure from the program point. □

Example 5.2.5. Consider procedure declaration

proc p **is** z **:=** x ; (**call** q ; y **:=** z) .

A call consists of the entrance step (lying in $[0; \frac{1}{2}]$) and run of the callee (compressed to $[\frac{1}{2}; 1]$). Thereby, all program points of the run of the callee are complemented with the rest of the caller to be executed after the return of the callee.

The execution trace of p at initial state $(\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 0)$, provided the semantics of q is as in Example 5.2.4, is

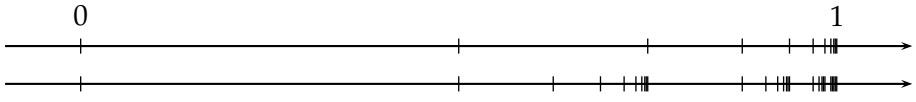
$$0 \mapsto \langle [\mathsf{z} \texttt{ := } \mathsf{x} \texttt{ ; (call } \mathsf{q} \texttt{ ; } \mathsf{y} \texttt{ := } \mathsf{z})] \mid (\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 0) \rangle \ ,$$
$$\tfrac{1}{4} \mapsto \langle [\texttt{call } \mathsf{q} \texttt{ ; } \mathsf{y} \texttt{ := } \mathsf{z}] \mid (\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1) \rangle \ ,$$
$$\tfrac{5}{16} \mapsto \langle [\mathsf{y} \texttt{ := } \mathsf{z}, \mathsf{x} \texttt{ := } \mathsf{y}] \mid (\mathsf{x} \mapsto 1, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1) \rangle \ ,$$
$$\tfrac{11}{32} \mapsto \langle [\mathsf{y} \texttt{ := } \mathsf{z}, \varepsilon] \mid (\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1) \rangle \ ,$$
$$\tfrac{3}{8} \mapsto \langle [\mathsf{y} \texttt{ := } \mathsf{z}] \mid (\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 2, \mathsf{z} \mapsto 1) \rangle \ ,$$
$$\tfrac{1}{2} \mapsto \langle [\varepsilon] \mid (\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 1, \mathsf{z} \mapsto 1) \rangle \ ,$$
$$1 \mapsto \langle [] \mid (\mathsf{x} \mapsto 2, \mathsf{y} \mapsto 1, \mathsf{z} \mapsto 1) \rangle \ .$$

Note that the comma between the code fragments in the program points is the separator of list components rather than sequential composition. □

To contrast the nature of fractional semantics to that of standard and transfinite trace semantics, one may call fractional traces *developing inward* while the traces of standard and transfinite semantics are *developing outward*. A fractional semantics of a non-base syntactic construct is an operation which rearranges and joins the traces of all child statements to the same space occupied by each of these traces. For looping constructs like while-loop and recursion, first such an operation is defined and then a fixpoint of this operation is found.

No example so far involved infinity. If infinity arises due to while-loops only, there exist fractional semantics of kind defined in [11] that copy transfinite trace semantics. The set of indices from $[0; 1]$ used by a trace of such fractional semantics can be represented as the image of an order-preserving mapping of some set of ordinals into $[0; 1]$.

Example 5.2.6. The components of the execution trace of **`while true do`** ε are numbered by ordinals $0, 1, 2, 3, \ldots$ and ω in transfinite semantics and by $0, \frac{1}{2}, \frac{3}{4}, \frac{7}{8}, \ldots$ and 1 in the fractional semantics under consideration. The latter set is depicted on the upper axis in the figure below. The components of double infinite loop **`while true do while true do`** ε are indexed by ordinals from 0 to ω^2 in transfinite semantics and by rational numbers shown on the lower axis in the fractional semantics.



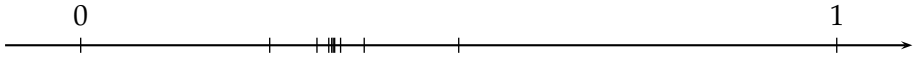
□

To handle recursion similarly, unloading infinitely deep recursion must be enabled. This involves chains with no least element and this is because transfinite semantics do not qualify. We bring two examples of infinite recursion in the case of which fractional semantics of our kind exists.

Example 5.2.7. The simplest example is obtained by declaration

```
proc p() is call p() .
```

The components of the execution trace of `p` are indexed by the rationals shown in the following picture:

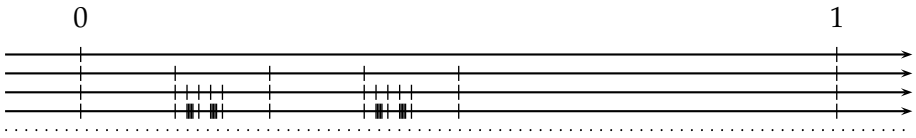


Two infinite sequences — one ascending and another descending — are both converging to $\frac{1}{3}$. □

Example 5.2.8. Consider the procedure declaration

```
proc q() is (call q();call q()) .
```

The set of indices used by the execution trace of `q` is the limit of the following step-by-step process:



Each step adds twice more points than the previous since the number of uninterpreted calls doubles every level.

The limit set forms a fractal structure. A rational number between 0 and 1 belongs to it iff its octal representation is finite and each its digit after octal point is either 1 or 3 except for the last one which can be also 2 or 4. The set of all possible limits of converging sequences of rationals in this set is uncountable. □

Example 5.2.8 shows that transfinite computations in the case of while-loop are analogues to fractal computations in the case of recursion.

In [11], we gave also example programs having no fractional semantics satisfying the definition in [11]. The programs in Examples 5.2.7 and 5.2.8 are trivial because the variable state does not change. In general, we have to show the values of variables at each point of the computation possibly forming a fractal structure. This is the main difficulty in defining an appropriate fractional semantics.

In programming theory, usually an operation is called *lazy* if it not necessarily leads to the error (a runtime error or nontermination) in which the evaluation of its arguments alone would result. Transfinite semantics is lazy in the sense that it enables overcoming looping computations: a transfinite semantics whose subcomputation is looping can be properly finished itself. The same applies to fractional semantics studied above. In the light of this, it is not really surprising that it is possible to implement fractional semantics in a lazy functional language like Haskell by just translating the mathematical definition directly to the language syntax and, in some cases, it is able to outpute proper results even for computation points occurring after looping parts or inside a fractal structure.

5.3 Triploids

In Subsect. 3.2, operators drop, take and also map, filter were defined for transfinite lists (for finite lists and streams, analogous operators with the same names are widely used in functional programming). The reader could notice from Lemmas 3.2.3 and 3.2.5 that operator pairs $(\mathrm{drop}, \mathrm{take})$ and $(\mathrm{map}, \mathrm{filter})$ possess very similar properties.

By making a few slight changes in definitions, we can make these pairs really instances of one algebraic structure type such that the similar properties become special cases of the axioms of it.

Firstly, according to the definition of drop and take given in Subsect. 3.2, both $\mathrm{drop}\, o\, l$ and $\mathrm{take}\, o\, l$ equal $\bot$ whenever $o > |l|$; change this to returning the empty list nil and the whole list l, respectively. (This makes the behaviour of drop and take analogous to the behaviour of the namesake functions in the functional programming language Haskell.) This change could be implemented also in the theory of this thesis without having to change the main results.

Secondly, restrict operator map to type $(A \to A) \to (\mathrm{TList}\, A \to \mathrm{TList}\, A)$, i.e. the new map can not be applied to functions whose domain and codomain differ.

The desired type of algebraic structures is given by Definition 5.3.1.

Definition 5.3.1. *Call* triploid *any* 3*-sorted structure*

$$((A;\,\cdot,1),(B;\,+,0),(C;\,{}_{\mathfrak{c}},\mathrm{I});\#,\mathsf{F},\mathsf{G})$$

that satisfies all the following:

1. $(A;\,\cdot,1)$, $(B;\,+,0)$, $(C;\,{}_{\mathfrak{c}},\mathrm{I})$ *are monoids;*
2. $\# : A \times B \to B$ *is an action of monoid* $(A;\,\cdot,1)$ *on* B*;*
3. $\mathsf{F} : (A;\,\cdot,1) \to (C;\,{}_{\mathfrak{c}},\mathrm{I})$ *and* $\mathsf{G} : (B;\,+,0) \to (C;\,{}_{\mathfrak{c}},\mathrm{I})$ *are monoid homomorphisms;*
4. 0 *is the zero of* #*, i.e.,*

$$\forall a \in A\,(a \,\#\, 0 = 0)\ ;$$

5. *#distributes over +, i.e.,*

$$\forall a \in A, b_1, b_2 \in B\,(a \,\#\, (b_1 + b_2) = (a \,\#\, b_1) + (a \,\#\, b_2))\ ;$$

6. *+ is both commutative and idempotent, i.e.,* $(B;\,+,0)$ *is actually a semilattice with the least element* 0*;*
7. *the following "quasi-commutativity" holds:*

$$\forall a \in A, b \in B\,(\mathsf{F}(a)\,{}_{\mathfrak{c}}\,\mathsf{G}(b) = \mathsf{G}(a \,\#\, b)\,{}_{\mathfrak{c}}\,\mathsf{F}(a))\ .$$

The word "triploid" is a simple derivation from "monoid" in the light of any triploid consisting of three monoids.

A preliminary glance to the triploid axioms indicates that there is a close relation between triploids and vector spaces. Monoid $(A;\,\cdot,1)$ plays the role of the structure of scalars, semilattice $(B;\,+,0)$ plays the role of the structure of vectors, # plays the role of multiplication of vectors by scalars. In vector spaces, scalars form a field and vectors form an Abelian group. The requirements about # are precisely the requirements about multiplication by scalars in vector spaces which can be formulated in terms of operations available in triploids. In addition, triploid has one more monoid $(C;\,{}_{\mathfrak{c}},\mathrm{I})$ within which both scalars and vectors can be interpreted via F and G, respectively, and which satisfies the "quasi-commutativity" property.

Let $\propto$ be a fixed selfish ordinal. Consider operators drop, take which are defined as suggeste above and such that

$$\mathrm{drop} \in \mathbb{O}_{\propto} \to \mathrm{TList}\,A \to \mathrm{TList}\,A\ ,$$
$$\mathrm{take} \in \mathbb{O}_{\propto'} \to \mathrm{TList}\,A \to \mathrm{TList}\,A\ .$$

In particular, one can take $\propto$ components from a transfinite list but dropping that many elements is precluded.
Then

$$
\begin{array}{l}
(\\
\quad (\mathbb{O}_{\propto};\, +, 0) \\
, \\
\quad (\mathbb{O}_{\propto'};\, \min, \propto) \\
, \\
\quad (\mathrm{TList}\, A \to \mathrm{TList}\, A;\; ;, \mathrm{id}) \\
; \\
\quad + \\
, \\
\quad \mathsf{drop} \\
, \\
\quad \mathsf{take} \\
)
\end{array}
$$

is a triploid. Both scalars and vectors are ordinal numbers. At place of both multiplications, there is ordinal addition; at place of vector addition, there is binary minimum operator. Unit scalar is 0, null vector is the upper limit $\propto$. Operators drop and take are the interpreting mappings of scalars and vectors, respectively.
Similarly, if map and filter are defined as in Subsect. 3.2 together with the restriction on map suggested above then

$$
\begin{array}{l}
(\\
\quad (A \to A;\; ;, \mathrm{id}) \\
, \\
\quad (A \to \mathbb{T};\, \diagup pq.\, \diagup a.\, p(a) \wedge q(a), \diagup a.\, \mathrm{tt}) \\
, \\
\quad (\mathrm{TList}\, A \to \mathrm{TList}\, A;\; ;, \mathrm{id}) \\
; \\
\quad ; \\
, \\
\quad \mathsf{map} \\
, \\
\quad \mathsf{filter} \\
)
\end{array}
$$

is a triploid. This time, scalars are transformations of A and vectors are predicates on A. At place of both multiplications, there is function composition; at place of vector addition, there is conjunction of predicates. Unit scalar is the identity, null

vector is the tautology. The interpreting mappings of scalars and vectors are map and filter, respectively.

In both triploids, the monoid in which the scalars and vectors were interpreted was the same, namely the monoid of all transformations of transfinite lists.

5.4 Related Work

Transfinite semantics have been studied first for functional programming, see Kennaway et al. [7].

A fundamental theoretical study of program slicing in the context of standard semantics has been done by Reps and Yang [14]. They prove that the sequence of values computed by any atomic statement of the slice during its run coincides with the sequence of values computed by the corresponding statement of the original program during its run whenever the initial states are equal and the original program terminates. They prove it by induction on the structure of the program. They obtain also a result they call "Feasibility Lemma" which states that slicing operation on control flow graphs can always be reflected as slicing of corresponding programs.

Besides transfinite semantics introduced by Giacobazzi and Mastroeni [5], there are some more approaches to handle semantic anomaly (see Reps and Turnidge [13], Danicic et al. [4]).

It is worth to note that we define the limit state into which the computation falls after an infinite number of steps differently from [5]; their treatment could be achieved by replacing s' with s in our definition of ψ in Sect. 4.1. In other words, the limit state of [5] depends on all states observed during the infinite computation while our limit state depends only on the states observed at the starting point of the loop which causes the divergence. Therefore, their semantics is not completely appropriate for describing program slicing (see Example 4.1.2). The *lazy semantics* of Danicic et al. [4] does not have this problem as the body of a loop is an undivided unit in the definition of the semantics of loops.

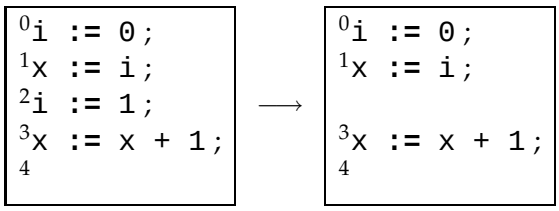
A predecessor of our Theorem 4.3.8 was proven in [5] for a fixed structured language.

A farther aim of [5] besides overcoming semantic anomaly is to provide a denotational semantics $\mathcal{D}$ with an order relation $\sqsubseteq$ on the corresponding semantic domain which would reflect the relation "being slice of" in the following sence: if $\tilde{S}$ is a slice of S then always $\mathcal{D}(\tilde{S}) \sqsubseteq \mathcal{D}(S)$. Unfortunately, the main theorem of this part (Theorem 6.4) of that paper is incorrect.

The wrong result is caused by making the assumption in the proof that all the variables of the slice have the same values at the end of execution of the slice as

they have at the end of execution of the original program on the same initial state. This does not hold in general.

Example 5.4.1. The right-hand program is a slice of the left-hand program w.r.t. criterion $\{(4, \mathtt{x})\}$. Note that this slice is to no extent exotic; it can be obtained using standard slicing algorithms, in particular via reachability in the data and control dependence graph as described in [5] and also in Sect. 4.8 of this thesis.



Variable `i` occurs in the slice; but at the end of execution of the slice, `i` has value 0 while, at the end of execution of the original program, it has value 1. □

The authors refer to Venkatesh [19] and Reps and Yang [14] as the sources of this assumption but it seems to be a misinterpretation of the results of these papers.

5.5 Conclusion of the Thesis and Suggestions of Further Work

In this thesis, we presented a proof of correctness of standard program slicing methods w.r.t. a class of transfinite semantics. Using transfinite semantics allows to omit assumptions about termination. The ground idea of our proof is to represent slicing as a composition of two transformations: the first removes the effect of irrelevant statements to data flow but maintains the control structure and the second simplifies the control structure by removing the statements with no influence to data-flow. The first transformation can produce nonterminating programs from terminating ones which implies that such kind of proof would not be possible in the case of standard semantics even for terminating programs.

The semantic correctness results of both transformations were proven for control flow graphs. This choice guarantees independence from syntactic details. In principle, the results can be even applied to languages with unstructured control flow. However, all the assumptions about the nature of the flow graphs and semantics made in our theorems were chosen so that they would hold in the case of simple structured control flow. Hence the results hold for unstructured control flow as far as it behaves like structured control flow to certain extent. We did not investigate the possible practical applications of our theory to unstructured control flow cases. This might be one direction of further work.

Another important limit of our theory is the exclusion of recursion. We made a step towards generalizing transfinite semantics to involve recursion in [11] by introducing fractional semantics. A brief introduction to this approach is provided also in Subsect. 5.2 of this thesis. This preliminary work shows that a natural generalization of transfinite trace semantics to recursive procedures would give rise to fractal computations. Finding out to which extent fractional semantics could be used for recursive programs, or finding a semantics appropriate for formalizing slicing of recursive programs, are also possible directions of future work.

There is one more concern about transfinite semantics: branching according to an ambiguous value. All examples of transfinite semantics defined so far involve ambiguous values which are aquired by variables in the first state after an endless computation where their value did not stabilize. A branching predicate may thus evaluate to the ambiguous value. Which branch should be chosen in this situation? Our theory holds if the ambiguous value is treated equivalently to false (or equivalently to true). There are other reasonable approaches like merging both branches. The lazy denotational semantics by Danicic et al. [4] is an instance of this approach. Merging branches means that both branches are run independently and, when both have finished, the result states are merged (this is very different from both concurrent execution where two threads may interact already during their run and non-determinism where no merging of different choices are performed). Defining such trace semantics could be somewhat problematic.

5.6 Acknowledgements

This work was partially supported by Estonian Science Foundation under grant no 6713.

I thank professor Helmut Seidl for enabling me to join his research group in Trier for half a year within the DAEDALUS-project. Some ideas which are carried out in this thesis originate from my work in Trier.

BIBLIOGRAPHY

[1] Barwise, J., Moss, L.: *Vicious Circles.* CSLI Lecture Notes No 60. CSLI Publications (1996)

[2] Binkley, D. W., Gallagher, K. B.: Program Slicing. *Advances in Computers* **43** (1996) 1–50

[3] Cousot, P.: Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Electronic Notes in Theoretical Computer Science* **6** (1997) 25 pp.

[4] Danicic, S., Harman, M., Howroyd, J., Ouarbya, L.: A Lazy Semantics for Program Slicing. (Extended abstract.) In *Proceedings of the 1st International Workshop on Programming Language Interference and Dependence*. Available at `http://profs.sci.univr.it/~mastroen/download/PLID/Proceedings/Proceedings.html` (2004)

[5] Giacobazzi, R., Mastroeni, I.: Non-Standard Semantics for Program Slicing. *Higher-Order Symbolic Computation* **16** (2003) 297–339

[6] Jacobs, B., Rutten, J.: A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bulletin* **62** (1997) 222–259

[7] Kennaway, R., Klop, J. W., Sleep, R., Vries, F.-J. de.: Transfinite Reductions in Orthogonal Term Rewriting Systems. *Information and Computation* **119**(1) (1995) 18–38

[8] Moschovakis, Y. N.: *Notes on Set Theory*. Undergraduate Texts in Mathematics. Springer-Verlag New York etc. (1994)

[9] Nestra, H.: Transfinite Corecursion. *Nordic Journal of Computing* **12**(2) (2005) 133–156

[10] Nestra, H.: Transfinite Semantics in Program Slicing. *Proceedings of the Estonian Academy of Sciences: Engineering* **11**(4) (2005) 313–328

[11] Nestra, H.: Fractional Semantics. In Johnson, M., Vene, V. (eds.): *Proceedings of AMAST 2006. Lecture Notes in Computer Science* **4019** (2006) 278–292

[12] Poizat, B.: *A Course in Model Theory: an Introduction to Contemporary Mathematical Logic.* Springer-Verlag New York (2000)

[13] Reps, T., Turnidge, T.: Program Specialization via Program Slicing. In Danvy, O., Glueck, R., Thiemann, P., (eds.): *Proceedings of the Dagstuhl Seminar of Partial Evaluation. Lecture Notes in Computer Science* **1110** (1996) 409–429

[14] Reps, T., Yang, W.: The Semantics of Program Slicing and Program Integration. *Lecture Notes in Computer Science* **352** (1989) 360–374

[15] Schütte, K.: *Proof Theory.* Grundlehren der matematischen Wissenschaften. Springer-Verlag Berlin etc. (1977)

[16] Seidl, H., Vene, V., Müller-Olm, M.: Global Invariants for Analysing Multi-Threaded Applications. *Proceedings of the Estonian Academy of Sciences: Physics; Mathematics* **52**(4) (2003) 413–436

[17] Spivey, M.: *The Z notation: A Reference Manual.* 2nd edition, Prentice Hall International Series in Computer Science (1992)

[18] Tip, F.: A Survey of Program Slicing Techniques. *Journal of Programming Languages* **3**(3) (1995) 121–181

[19] Venkatesh, G. A.: The Semantic Approach to Program Slicing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation. ACM SIGPLAN Notices* **26**(6) (1991) 107–119

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

ITERATIIVSELT DEFINEERITUD TRANSFINIITSED JÄLITUSSEMANTIKAD JA PROGRAMMISLITSEERIMINE NENDE SUHTES

Programmislitseerimine on niisugune programmide teisendamine, kus antud programmi järgi konstrueeritakse sliits, st tema elementaarlausetest koosnev (loodetavasti) väiksem ja rutem töötav programm, mis teatud muutujate väärtusi teatud punktides arvutab samamoodi kui originaalprogramm. Programmislitseerimine äratas arvutiteadlaste tähelepanu üle 20 aasta tagasi, kui mõisteti, et see tehnika on kasulik programmide silumisel. Katsetega tehti isegi kindlaks, et vilunud programmeerijad konstrueerivad programmivigade otsimisel oma peas sliitse. Hiljem on slitseerimisele leitud muidki rakendusi tarkvaratehnikas.

Algoritmide leidmine programmide automaatseks slitseerimiseks on üldjoontes lihtne (keeruliseks võivad asja teha konkreetse programmeerimiskeele keerukamad erisused), kuid juba ammu märgati, et nende algoritmide korrektsuse tõestamine takerdub nn semantilise anomaalia taha, mis ilmneb juhul, kui originaalprogramm töötab lõpmatult, kuid tänu lõpmatu tsükli väljaslitseerimisele lõpetab sliits töö lõpliku ajaga ja jõuab seetõttu koodis kaugemale kui originaalprogramm. Tegemist on põhimõttelise probleemiga, kuna teatavasti on tsükli termineeruvuse kindlakstegemine algoritmiliselt mittelahenduv.

On välja pakutud mitmeid lahendusvariante semantilisest anomaaliast lahtisaamiseks. Üks neist on transfiniitsete semantikate kasutamine. Transfiniitne semantika on semantika, mille järgi programmi lõpmatu töö järel töötab programm mingist kindlaksmääratud piirseisundist edasi, täites sellele tsüklile oma koodis järgnevaid käske. Programmi täitmise mudel selles semantikas on algseisundist sõltuv transfiniitne seisundite jada.

Käesoleva doktoritöö üheks tulemuseks on väljaarendatud transfiniitsete semantikate matemaatiline alusteooria. Uuritakse lähemalt transfiniitset iteratsiooni, mis on erijuht üldisest tuntud transfiniitsest rekursioonist. Defineeritakse mitu varianti transfiniitsest koorekursioonist, mis on analoogid tuntud listikoorekursioonile, uuritakse seoseid nende vahel ja seoseid transfiniitse iteratsiooni ja transfiniitse koorekursiooni vahel. See osa on autori poolt publitseeritud artiklites [9, 10] (siin ja edaspidi antakse kirjandusviited doktoritöö allikanimekirja järgi lk 109).

Doktoritöö peamiseks tulemuseks on kahe andmevooanalüüsil põhineva slitseerimisalgoritmi korrektsuse tõestamine teatavate transfiniitsete semantikate suhtes. Selleks on välja arendatud eraldi matemaatiline alusteooria, milles neid tulemusi sõnastada ja tõestada. See osa on osaliselt publitseeritud artiklis [10], ülejäänud osas aga veel ilmumata (protsess on käimas).

Lähenemise üldideeks on vaade slitseerimisele kui kahest etapist koosnevale prot-

sessile. Esimesel etapil asendatakse elementaarlaused, mis tuleks välja slitseerida, tühjade lausetega, jättes nii juhtvoograafi isomorfismi täpsuseni puutumata. Teisel etapil eemaldatakse esimesel sissetoodud laused, jättes puutumata andmevoo. Esimese etapi teisendus on töös vaadeldud üldisemalt, kus asenduslaused pole tingimata tühjad, vaid võivad olla suvalised sellised laused, mis ei kirjuta üle rohkem muutujaid kui asendatav lause. Mõlema etapi semantiline korrektsus on analüüsitud eraldi. Esimene teisendus võib termineeruvast programmist teha mittetermineeruva, mis näitab, et standardsete (st mitte transfiniitsete) semantikate suhtes programmide slitseerimise korrektsuse seda laadi tõestamine ei lähe läbi.

Lisaks on doktoritöös lühidalt käsitletud ka mõnda põhitemaatikaga seonduvat küsimust. Näiteks on tõestatud nn vähima sliitsi ülesande mittelahenduvus transfiniitsete semantikate suhtes (standardse semantika suhtes on vastav tulemus tuntud). See osa on publitseeritud artiklis [10]. Samuti esitatakse näidete peal sissejuhatus murdsemantikatesse, mis põhineb autori värskeimal publikatsioonil [11]. Autori poolt sisse toodud murdsemantika mõiste on transfiniitse semantika mõiste üldistus, mis lubab arvutusprotsessi mudelis ka tagurpidi lõpmatuid jadasid — transfiniitse jada puhul on lõpmatus alati ettepoole suunatud. Kui lõpmatu sügavusega rekursiooni transfiniitse semantika defineerimine on problemaatiline seetõttu, et loomulikul viisil pole võimalik anda piirseisundeid, siis murdsemantika võimaldab rekursiooni tähendust anda nii, et lõpmata sügavast rekursioonist tullakse välja nagu lõplikustki tasehaaval. Murdsemantikas võib rekursiivse programmi tähendus olla fraktaalstruktuur. Siiski on murdsemantika veel nii vähe uuritud, et pole selge, millises ulatuses ta üldse saab olla rekursiivsetel programmidel rakendatav.

CURRICULUM VITAE

Name	HÄRMEL NESTRA
Position	Researcher of Institute of Computer Science, University of Tartu, Estonia
Date of birth	19. 03. 1974
Education	1992–1996, University of Tartu, faculty of mathematics
	1996–1998, University of Tartu, master studies in computer science
	1998–2004, University of Tartu, doctoral studies in computer science
Languages	Estonian (the native language), English; a little Russian, German, Finnish
Academical degrees	1998, MSc in Computer Science, thesis *Polytypic Functional Programming: from Categorical Groundwork to Practice*
Professional experience	1997 University of Tartu, Institute of Computer Science, programmer
	2002–2003 University of Tartu, Institute of Computer Science, assistant
	2003 University of Trier (Germany), Computer Science Departement, co-worker on research
	2003– . . . University of Tartu, Institute of Computer Science, researcher

Training

March 1998, the 3rd CIDEC Winter School, Palmse, Estonia
March 1999, the 4th CIDEC Winter School, Palmse, Estonia
August 2000, the 12th ESSLLI, Birmingham, England
March 2001, the 6th CIDEC Winter School, Palmse, Estonia
August 2001, the 13th ESSLLI, Helsinki, Finland
March 2002, the 7th CIDEC Winter School, Palmse, Estonia
March 2004, the 9th CIDEC Winter School, Palmse, Estonia
August 2004, the 5th AFP Summer School, Tartu, Estonia
March 2005, the 10th CIDEC Winter School, Palmse, Estonia
March 2006, the 11th CIDEC Winter School, Palmse, Estonia

CURRICULUM VITAE

Nimi	HÄRMEL NESTRA
Amet	Tartu Ülikooli arvutiteaduse instituudi teadur
Sünnikuupäev	19. 03. 1974
Haridustee	1992–1996, Tartu Ülikool, matemaatikateaduskond
	1996–1998, Tartu Ülikool, magistriõpe informaatika erialal
	1998–2004, Tartu Ülikool, doktoriõpe informaatika erialal
Keeleoskus	eesti (emakeelena), inglise; veidi vene, saksa, soome
Akadeemilised kraadid	1998, MSc informaatika erialal, magistritöö *Polütüüpne funktsionaalne programmeerimine: kategooriateoreetilistest alustest praktikasse*
Erialane töökogemus	1997 Tartu Ülikool, arvutiteaduse instituut, programmeerija
	2002–2003 Tartu Ülikool, arvutiteaduse instituut, assistent
	2003 Trieri Ülikool (Saksamaa), arvutiteaduse teaduskond, teaduslik kaastöötaja
	2003– . . . Tartu Ülikool, arvutiteaduse instituut, teadur
Erialane enesetäiendus	märts 1998, 3. CIDECi talvekool, Palmse, Eesti
	märts 1999, 4. CIDECi talvekool, Palmse, Eesti
	august 2000, 12. ESSLLI, Birmingham, Inglismaa
	märts 2001, 6. CIDECi talvekool, Palmse, Eesti
	august 2001, 13. ESSLLI, Helsinki, Soome
	märts 2002, 7. CIDECi talvekool, Palmse, Eesti
	märts 2004, 9. CIDECi talvekool, Palmse, Eesti
	august 2004, 5. AFP suvekool, Tartu, Eesti
	märts 2005, 10. CIDECi talvekool, Palmse, Eesti
	märts 2006, 11. CIDECi talvekool, Palmse, Eesti

LIST OF ORIGINAL PUBLICATIONS

[1] Nestra, H.: Handling Substitution without Induction. In Piliére, C. (ed.): *Proceedings of the ESSLLI-2000 Student Session*. University of Birmingham (2000) 178–188

[2] Nestra, H.: A Framework for Studying Substitution. In Gyimóthy, T. (ed.): *Seventh Symposium on Programming Languages and Software Tools*. University of Szeged (2001) 168–182

[3] Nestra, H.: A Framework for Studying Substitution. Acta Cybernetica **15** (2002) 633–652

[4] Nestra, H.: Transfinite Corecursion. In Pettersson, P. and Yi, W. (eds.): *Proceedings of the 16th Nordic Workshop on Programming Theory*. Uppsala University Technical report 2004-041 (2004) 30–32

[5] Nestra, H.: Transfinite Semantics in Program Slicing. In Vene, V., Meriste, M. (eds.): *Proceedings of the Ninth Symposium on Programming Languages and Software Tools*. University of Tartu (2005) 126–140

[6] Nestra, H.: Transfinite Corecursion. Nordic Journal of Computing **12**(2) (2005) 133–156

[7] Nestra, H.: Transfinite Semantics in Program Slicing. Proceedings of the Estonian Academy of Sciences. Engineering **11**(4) (2005) 313–328

[8] Nestra, H.: Fractional Semantics. In Johnson, M., Vene, V. (eds.): *Proceedings of AMAST 2006*. Lecture Notes in Computer Science **4019** (2006) 278–292