The Journal of Logic and Algebraic Programming 78 (2009) 573–592

Contents lists available at ScienceDirect

The Journal of Logic and Algebraic Programming

journal homepage: www.elsevier.com/locate/jlap

Transfinite semantics in the form of greatest fixpoint

Härmel Nestra

Institute of Computer Science, University of Tartu, J. Liivi 2, 50409 Tartu, Estonia

ARTICLE INFO

Article history:
Available online 5 April 2009

Keywords:
Transfinite semantics
Tree semantics
Fractional semantics
Greatest fixpoint

ABSTRACT

Transfinite semantics is a semantics according to which program executions can continue working after an infinite number of steps. Such a view of programs can be useful in the theory of program transformations.

So far, transfinite semantics have been successfully defined for iterative loops. This paper provides an exhaustive definition for semantics that enable also infinitely deep recursion.

The definition is actually a parametric schema that defines a family of different transfinite semantics. As standard semantics also match the same schema, our framework describes both standard and transfinite semantics in a uniform way.

All semantics are expressed as greatest fixpoints of monotone operators on some complete lattices. It turns out that, for transfinite semantics, the corresponding lattice operators are cocontinuous. According to Kleene's theorem, this shows that transfinite semantics can be expressed as a limit of iteration which is not transfinite.

1. Introduction

1.1. Motivation and context

It is sometimes useful to imagine program runs as if they were able to overcome non-termination. According to this view, a computation that falls into an infinite loop or infinitely deep recursion continues after completing the infinite subcomputation. This has applications in the theory of program transformations such as slicing.

Program slicing is a program transformation technique where the aim is to omit statements from a given program in such a way that executing the remaining program (so-called *slice*) would compute some data of our special interest exactly the same way as the original program. The interesting data is specified in the form of a list of variables that are coupled with the program points at which their values are important. This list is called *slicing criterion*; for each couple in the criterion, the sequence of values obtained by the variable at that program point during the whole execution of the original program is required to coincide with the sequence of values obtained by the same variable at the corresponding program point of the slice during its execution.

This transformation is used in several branches of software engineering. A well-known application is in debugging: when one discovers a wrong value of a variable at some point of execution, one may slice the program w.r.t. the criterion containing this variable coupled with this program point; then the error must lay inside the slice which is hopefully a much smaller program than the original one and finding the error from a smaller program is easier. This application was one of the earliest discovered and studied (see [14]).

Classically, slices are computed via data-flow analysis (for details, see [1] or [13]). When a loop has no influence on the values of the interesting variables via data flow, such slicing algorithms do not keep it. But infinite loops can have another kind of influence: they may prevent the program from reaching some assignments. If an infinite loop is sliced away, the

E-mail address: harmel.nestra@ut.ee

1567-8326/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jlap.2009.03.001

resulting program reaches farther in the code than the original program and thus may do assignments to the interesting variables that the original program never does.

In order to treat this as a correct behaviour, we can say that the original program also makes these assignments but this happens after an infinite number of steps. Program semantics that follow this view are usually called transfinite. The idea of using transfinite semantics in program slicing theory was first proposed by Cousot [2]. This approach was followed later in the work of Giacobazzi and Mastroeni [4] and ours [11,10].

An alternative approach that can be used is to consider this kind of simplification as an incorrect slicing and require that the slicers kept all possibly infinite loops that lexically precede an assignment to an interesting variable. However, when slicing, the smaller the slice is the better we have done, and keeping unnecessary loops is countermining. In the debugging case described above, slicing that is purely based on data-flow analysis is satisfactory because if a wrong value really occurs then this happens after a finite run already and the slice therefore finds the wrong value in the same way.

The word 'transfinite' actually fails to characterize the whole variety of imaginary computation processes that arise in our approach. In the case where only iterative loops can be non-terminating, the sequence of execution steps is really transfinite in the sense that the steps can be enumerated by ordinal numbers in their execution order. For unloading an infinitely deep call-stack level-by-level, such an enumeration is impossible since no infinite decreasing sequences of ordinals exist.

In [10], we showed that the sequence of steps in this case is more like a fractal structure. We proposed the idea of enumerating steps with rational numbers and called such semantics *fractional*. However, we failed to give a complete definition of fractional semantics in the presence of recursion.

1.2. About the paper

In this paper, we provide an exhaustive definition schema for semantics that enable returns from infinitely deep recursion level-by-level. The semantics are expressed in terms of greatest fixpoints of monotone operators on complete lattices of set-valued functions. At the statement level for example, these functions take statements to sets of semantic objects that somehow describe the execution of the statement.

Our framework actually defines a large family of semantics in a uniform way. There is one definition schema for all semantics and it refers to a small number of underlying sets and mappings as parameters. The uniform parametric representation is somewhat similar to that in our earlier work [10]. However, the semantics specification in [10] does not include any order relation, thus fixpoints were specified in ad-hoc manners while monotone operators together with their least and greatest fixpoints provide a standard framework for semantics.

The purpose of doing the work parametrically for many semantics is to emphasize the closeness of transfinite semantics to well-known semantics. We complete the definition schema for nine concrete semantics by providing values of parameters. The semantics are classified by two attributes: by the size and by the shape. By the size, a semantics can be either finite, standard or transfinite. By the shape, we consider ordinal trace semantics, fractional trace semantics and tree semantics.

Not all of these semantics are satisfactory. As already explained, transfinite ordinal trace semantics is not what one desires. Although the definition via a greatest fixpoint is correct, it leads to empty semantics in the case of infinitely deep recursion. Even for statements without recursion, the transfinite ordinal trace semantics that is given by a greatest fixpoint can return too large trace sets that contain besides the desired traces also arbitrary other traces that include the desired traces as segments. We provide an example on this in Section 5.3. (Semantics that assign chaotic behaviour to programs are sometimes called demonic (for instance, in [3]). This term seems to be chosen in order to characterize the aspect that everything bad is possible.)

Therefore, there is a need for fractional semantics where the index space is statically distributed between parts of program and no space is left for garbage. Alternatively, one can use tree semantics where executions are built up in tree form similarly to proof trees of natural semantics.

Concerning the size attribute, finite semantics can be obtained from standard semantics by omitting all infinite semantic objects. From the transfinite point of view, standard semantics is obtained from finite semantics by adding infinite semantic objects whose parts after the first infinity are truncated.

Finite semantics is nothing exotic; for example, classical natural semantics include only finite trees.

In our work, it turns out that the details for standard semantics are more complicated than those of transfinite semantics. Due to this, one can even roughly specify transfinite semantics as those obtained by replacing the least fixpoint (that is commonly used for presenting semantics of programming languages) with the greatest fixpoint and using either fractional or tree shape.

Lastly, we prove that the monotone operators whose greatest fixpoints are taken for our non-standard semantics are cocontinuous. This means (by Kleene's theorem) that the semantics can be achieved by iteration which is not transfinite, even if the semantics is transfinite. Thanks to the parametric uniform framework, proofs are mostly obtained simultaneously for all semantics.

In our approach, the semantics are not a priori deterministic. This is presumably not a big drawback since non-determinism is always introduced after non-termination.

1.3. Examples

A toy example illustrating transfinite semantics in program slicing is the following. Suppose the code fragment (**while** `true` **do** `x := x + 1`) ; `x := 1` has to be sliced w.r.t. its final point and variable `x`. That means, a desirably shorter code fragment that computes the same sequence of values for `x` at the corresponding (i.e. final) program point is looked for. The assignment in the loop body can never influence the final value of `x` since `x` is yet updated after the loop; hence slicing algorithms typically remove the loop and get the result `x := 1`. Here, `x` gets value 1. The execution trace of the original code fragment consists of an infinite succession of states where the value of `x` increases continuously, followed by states $\{\mathtt{x} \mapsto \top\}, \{\mathtt{x} \mapsto 1\}$ in transfinite semantics ($\top$ means that no concrete value can be associated to the variable). Hence `x` gets value 1 by the end of the execution in transfinite semantics but not in standard semantics. If the latter is assumed, a slicer should keep the loop in order to follow the notion of slice.

The example is extremely simple and impractical but one can easily generalize it and put it into a larger context (e.g., into the body of a bigger loop) to capture real examples. As termination is undecidable, no algorithm can ever decide for all input programs whether a loop may be sliced away or not within standard semantics.

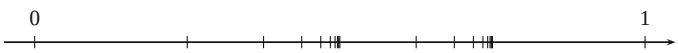
Fractional traces were introduced by us in [10]. The execution steps of computations are indexed by rational numbers from the interval [0; 1].

For example, the execution trace of the swap program `z := x` ; (`x := y` ; `y := z`) at the initial state $(\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2, \mathtt{z} \mapsto 0)$ is

$$\begin{aligned}
0 &\mapsto (\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2, \mathtt{z} \mapsto 0),\\
\frac{1}{2} &\mapsto (\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2, \mathtt{z} \mapsto 1),\\
\frac{3}{4} &\mapsto (\mathtt{x} \mapsto 2, \mathtt{y} \mapsto 2, \mathtt{z} \mapsto 1),\\
1 &\mapsto (\mathtt{x} \mapsto 2, \mathtt{y} \mapsto 1, \mathtt{z} \mapsto 1).
\end{aligned}$$

As the program is a composition of two statements, the interval [0; 1] is first divided into two equal pieces; and as the second statement is itself a composition of two statements, the second half $\left[\frac{1}{2}; 1\right]$ is also divided into two equal pieces. The assignments `z := x`, `x := y` and `y := z` are therefore run within intervals $\left[0; \frac{1}{2}\right]$, $\left[\frac{1}{2}; \frac{3}{4}\right]$ and $\left[\frac{3}{4}; 1\right]$, respectively. This is so independently of the initial state.

If the semantics is transfinite, the reservation of intervals would remain the same even if the three assignments were replaced with arbitrary three statements S_1, S_2, S_3. For example, if $S_1 = S_2 =$ **while** `true` **do** `x := x + 1` and $S_3 =$ `x := 1` then the index set of an execution trace of statement S_1 ; (S_2 ; S_3) is depicted in the following figure.



In tree form, the execution of the swap program is

$$\dfrac{\dfrac{\{\mathtt{x}\mapsto 1, \mathtt{y}\mapsto 2, \mathtt{z}\mapsto 0\} \to \{\mathtt{x}\mapsto 1, \mathtt{y}\mapsto 2, \mathtt{z}\mapsto 1\}}{\dfrac{\{\mathtt{x}\mapsto 1, \mathtt{y}\mapsto 2, \mathtt{z}\mapsto 1\} \to \{\mathtt{x}\mapsto 2, \mathtt{y}\mapsto 2, \mathtt{z}\mapsto 1\}} \qquad \dfrac{\{\mathtt{x}\mapsto 2, \mathtt{y}\mapsto 2, \mathtt{z}\mapsto 1\} \to \{\mathtt{x}\mapsto 2, \mathtt{y}\mapsto 1, \mathtt{z}\mapsto 1\}}{\{\mathtt{x}\mapsto 1, \mathtt{y}\mapsto 2, \mathtt{z}\mapsto 1\} \to \{\mathtt{x}\mapsto 2, \mathtt{y}\mapsto 1, \mathtt{z}\mapsto 1\}}}{\{\mathtt{x}\mapsto 1, \mathtt{y}\mapsto 2, \mathtt{z}\mapsto 0\} \to \{\mathtt{x}\mapsto 2, \mathtt{y}\mapsto 1, \mathtt{z}\mapsto 1\}}$$

where the same initial state as above was chosen. The transfinite tree semantics of statement (**while** `true` **do** `x := x + 1`) ; `x := 1` at the initial state $\{\mathtt{x} \mapsto 0\}$ may be of the form

$$\dfrac{t_0 \qquad \dfrac{}{\{\mathtt{x} \mapsto \top\} \to \{\mathtt{x} \mapsto 1\}}}{\{\mathtt{x} \mapsto 0\} \to \{\mathtt{x} \mapsto 1\}}$$

where t_i is equal to

$$\dfrac{\dfrac{}{\{x \mapsto i\} \to \{x \mapsto i\}} \qquad \dfrac{\dfrac{}{\{x \mapsto i\} \to \{x \mapsto i+1\}} \quad t_{i+1}}{\{x \mapsto i\} \to \{x \mapsto \top\}}}{\{x \mapsto i\} \to \{x \mapsto \top\}}\,.$$

In order to achieve uniformity, our trees have branches corresponding to conditional tests (in natural semantics, tests are not reflected in the tree structure). Such branches consist of one node where the state does not change (the left branch in tree t_i is an instance).

Using the special value $\top$ is more or less a matter of choice here. In the greatest fixpoint semantics, one can naturally use non-determinism at this point (Section 6 provides some further discussion on this).

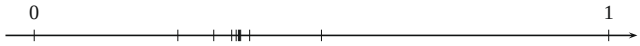
Fractional traces share both tree and trace properties: they reflect branching of the deduction tree while keeping the execution order evident. Fractional semantics forgets information about the depth of the nodes in trees. For example, trees

$$\dfrac{}{s \to t} \quad \text{and} \quad \dfrac{\dfrac{}{s \to t}}{s \to t}$$

have the same fractional counterpart $\{0 \mapsto s, 1 \mapsto t\}$. Thus fractional traces form an intermediate representation between trees and usual traces.

One may notice that our trees contain only state pairs while, in natural semantics, proof trees include also code fragments. Similarly, trace semantics are often expressed together with an additional "rest of code" component occurring in states. This is so, for instance, in the textbook structural operational semantics of [12] and also in the fractional trace semantics of [10]. We have omitted code parameters in this paper in order to make the framework simpler. They can be added if needed.

For a toy example involving infinitely deep recursion, assume that the body of the definition of procedure `p` is `x := x + 1 ; call p ()` and consider the code fragment `call p () ; x := 1`. In principle, this example is the same as the first example in this subsection where iteration is replaced with recursion. Hence it should be treated similarly. In the fractional semantics of this paper, the index set of the traces of this program can be depicted as follows:



There is one accumulation point $\left(\text{namely } \dfrac{5}{14}\right)$. Enumeration of states by ordinals in the order in which they occur is impossible as this point is infinitely approximated from both sides.

1.4. Related work

The idea of using transfinite semantics in program slicing was proposed by Cousot in [2] and in the later version [3] of the same work and developed further by Giacobazzi and Mastroeni [4].

Both mentioned works are deeply engaged on abstraction hierarchies of semantics. Cousot's work shows that many well-known fixpoint semantics can be put into an abstraction hierarchy where every semantics can be obtained from each one immediately preceding it via an abstraction mapping satisfying special requirements (inter alia, having a Galois adjoint). Such a hierarchy is called Cousot hierarchy in the literature now. The work by Giacobazzi and Mastroeni adds transfinite semantics to the hierarchy. An abstraction hierarchy involving nine semantics shows up also in this paper; we do not impose any special requirements on abstraction functions but it is likely that a big part of our hierarchy actually can be built up as a Cousot hierarchy.

Parts of the hierarchies in [2–4] resemble our two-dimensional classification of semantics but, contrastingly to our approach, their systems of semantics are not given parametrically.

During the last decade, several authors have argued in favour of generalizations of natural semantics that include also infinite trees. They enable to describe terminating and non-terminating computations uniformly like trace semantics but in tree form. Such semantics, usually called coinductive, are achieved via greatest fixpoints like ours.

For example, Glesner [5] uses coinductive natural semantics for proving correctness of translators and type safety. She however leaves transfinite parts of executions out of consideration, observing only what she calls "effective parts" of the trees. In our terms, the effective parts are left-finite trees. Contrastingly to [5] where these trees are achieved via truncation, a semantics like this (standard tree-based semantics) is defined directly in the form of greatest fixpoint in our framework.

With similar reasons, Leroy [7] and Leroy and Grall [8] propose a coinductive big-step semantics for a functional language (the lambda-calculus with constants). Interpreted along the lines of this paper, proof trees of "coevaluations" of [7,8] give rise to transfinite traces, but this aspect is never observed or exploited.

The non-determinism coming forth from coinductivity is restricted in no way in [7,8], the result term of an endless reduction sequence can be arbitrary. As explained in Section 6 of this paper, some restrictions must be imposed on the

Syntactic categories:

Var — the set of all variables
$Proc$ — the set of all procedure identifiers
$Expr$ — the set of all expressions
$Stmt$ — the set of all statements
$Decl$ — the set of all procedure declarations
Mod — the set of all modules

Grammar:

$$
\begin{aligned}
Stmt &\to Var \ \texttt{:=}\ Expr \\
&\mid \texttt{call}\ Proc(Expr,\ \ldots,\ Expr) \\
&\mid Stmt\ ;\ Stmt \\
&\mid \texttt{if}\ Expr\ \texttt{then}\ Stmt\ \texttt{else}\ Stmt \\
&\mid \texttt{while}\ Expr\ \texttt{do}\ Stmt \\
Decl &\to \texttt{proc}\ Proc(Var,\ \ldots,\ Var)\ \texttt{is}\ Stmt \\
Mod &\to Decl^{*}
\end{aligned}
$$

Fig. 1. Abstract syntax of **Proc**.

Semantics	Finite	Standard	Transfinite
Ordinal trace	$\overrightarrow{+}$	$\overrightarrow{\omega}$	$\overrightarrow{\propto}$
Fractional trace	$\widetilde{+}$	$\widetilde{\omega}$	$\widetilde{\propto}$
Tree	$\widehat{+}$	$\widehat{\omega}$	$\widehat{\propto}$

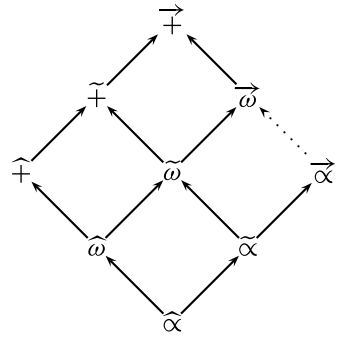


Fig. 2. The kinds of semantics, their notation and hierarchy.

state occurring after an endless computation in order to make our transfinite semantics useful for applications. It seems that modifying the coevaluation semantics similarly would also make them more widely applicable. Including infinite terms (expressing infinite data structures in lazy functional languages) and imposing rules for finding the limit terms of endless reduction sequences (for example, the infinite list consisting of zeros could be the limit of a sequence of finite lists of unboundedly growing length all consisting of zeros) would bring coevaluation semantics together with the approach of Kennaway et al. [6].

2. Syntax of the language

We are going to work on a simple language with procedures. The syntax of the language is presented in Fig. 1. We call this language **Proc** although it has several small divergences from the language that was used in our earlier work [10] and that was also called **Proc**.

One divergence from the earlier paper is that, for keeping fractional semantics and tree semantics simpler, we omitted empty statements here. This means that every execution makes at least one computation step. The other divergence is concerning procedures: here, we distinguish between declarations and modules (modules are possibly empty finite lists of declarations) while, in [10], any sequence of procedure definitions was called a declaration. None of the divergences is very conceptual.

As in [10], the inner structure of expressions is not important and hence not explained in the grammar.

3. Semantics of statements

3.1. Classification of semantics

Fig. 2 presents the classification of semantics explained in Section 1. Finite, standard and transfinite size are denoted by $+$, ω and $\propto$, respectively; ordinal trace, fractional trace and tree shape are denoted by $\overrightarrow{\cdot}$, $\widetilde{\cdot}$ and $\widehat{\cdot}$, respectively. Notation of each semantics is obtained by composing the notations of classes where the semantics belongs to.

Fig. 2 also presents a hierarchy of these semantics that is based on an abstraction relationship. A semantics being more abstract than another means that it can be obtained from the other by forgetting some details (or, technically, by applying a surjective mapping). The more abstract a semantics is, the higher it is situated in the picture. Arrows lead from more concrete to more abstract semantics. The arrow from transfinite ordinal trace semantics to standard trace semantics is dotted since it is valid for statements only (as explained in Section 1, transfinite ordinal semantics does not work for procedures).

$$
\begin{array}{lll}
Val & & \text{— the set of all values} \\
State & = Var \to Val & \text{— the set of variable evaluations} \\
Base_\kappa & & \text{— the base set of semantic objects} \\
Env_\kappa & = Proc \to Expr^* \to \wp(Base_\kappa) & \text{— the set of procedure environments}
\end{array}
$$

Fig. 3. Semantic domains.

This hierachy resembles the Cousot hierarchy of semantics [2–4] but, in this paper, we are not concerned in the question whether or to what extent it really is a Cousot hierarchy. This is the subject of future work.

3.2. Uniform framework

Figs. 3 and 4 present our semantics definition schema; κ denotes the kind of semantics (it can be any of the nine).

Sets and operators whose meaning is not specified are mostly defined later for each semantics separately. However, we forsake precise definitions of the set Val of values and the semantics e of expressions. For e, assume for simplicity that it always returns a normal value. If desired, a treatment of runtime errors can be added to the theory along the lines of our paper [10]. In addition, we use an order relation on Val which is equality on normal values but treats $\top$ (if $Val \ni \top$) as greater than any normal value. The role of $\top$ as possible limit value of infinite sequences is discussed in Section 6; it is safe to forget about $\top$ before that point.

Denote by $\wp(X)$ the powerset of X, and by X^* the set of all finite vectors with components from X. Furthermore, X^+ is X^* without the empty vector. Let $\mathbb{T} \subset Val$ be the set of truth values. Composition of functions is written like $f\ ;\ g$ (the left-hand function is applied first).

For each κ, the semantics s_κ of statements, depending on procedure environment e, is defined as the greatest fixpoint of some operator $\boldsymbol{f}_\kappa(e)$. As a semantics basically maps statements to sets of semantic objects, the operator $\boldsymbol{f}_\kappa(e)$ is a transformation of such mappings. The order on the mappings that is assumed when talking about the greatest fixpoint is the pointwise lift of the set inclusion order.

Roughly, the operator $\boldsymbol{f}_\kappa(e)$ adds one syntax-driven level to the evaluation of the argument statement. More precisely, the result of the transformation on any mapping is a mapping which decomposes its argument statement, applies the original mapping to each component, and finally fuses the resulting components together. Both the decomposition and the fusion are syntax-driven.

We handle the decomposition and fusion stages separately. Thereby, the decomposition is the same for all kinds of semantics while the fusion depends on both semantics and environment. In terms of category theory, the decomposition δ is a $\mathcal{C}$-coalgebra of syntactic objects and fusions $\varphi_\kappa(e)$ are $\mathcal{C}$-algebras of sets of semantic objects for an endofunctor $\mathcal{C}$ on the category of sets and functions. The task of $\mathcal{C}$ is to distinguish between different syntactic constructs of statements.

Note that the decomposition does not necessarily find the syntactic subobjects of its argument. The idea of decomposition is to provide the syntactic objects in terms of which the semantics of the original statement has to be expressed. In the case of loop, the result of decomposition is even longer than the original statement since the meaning of **while** *E* **do** *S* is unravelled using the semantics of the expression *E* and the complex statement *S* ; **while** *E* **do** *S*.

Both functor $\mathcal{C}$ and decomposition δ are defined completely in Fig. 4, so semantics s_κ becomes defined when φ_κ does. In order to achieve complete definition of φ_κ for some fixed κ, it suffices to define operators ini_κ, fin_κ, elem_κ, comp_κ for that κ. For these four operators, Fig. 4 provides only signature.

The names of operators axm and rul are suggested by tree semantics where the semantic objects are composed by axioms and rules. However, their counterparts can be observed for every semantics since, from a general point of view, every semantics is built up in a similar recurrent way.

3.3. Details for concrete semantics

Here, we complete the definitions of all semantics by defining $Base_\kappa$, as well as ini_κ, fin_κ, elem_κ and comp_κ, for all κ. Yet establishing the existence of the greatest fixpoint of $\boldsymbol{f}_\kappa(e)$ for all κ is left to Section 5.

For ordinal trace semantics, the base domain should contain execution traces of various lengths where the states are enumerated by indices 0, 1, 2, etc. Depending on size, the indices can be limited differently. In the transfinite case, the limit is a sufficiently large transfinite ordinal $\propto$. Assume that $\propto$ is a power of ω (as shown in [9], taking $\propto = \omega^\omega$ ensures that all programs without calls can be executed to the end). As natural numbers are small ordinal numbers, the indices are ordinals also for finite and standard semantics.

For arbitrary ordinal o, denote the set of all ordinals less than o by $\mathbb{O}_o$ and the set of all ordinals not exceeding o by $\mathbb{O}^o$, i.e.,

$$\mathbb{O}_o = \{\pi : \pi < o\}, \quad \mathbb{O}^o = \{\pi : \pi \leqslant o\} = \mathbb{O}_o \cup \{o\} = \mathbb{O}_{o+1}.$$

$$e \in Expr \to State \to Val \text{ — the semantics of expressions}$$

$$\begin{aligned} \mathrm{ini}_\kappa &\in Base_\kappa \to State && \text{— initial state in the given semantic object}\\ \mathrm{fin}_\kappa &\in Base_\kappa \to State \cup \{\bot\} && \text{— final state in the given semantic object}\end{aligned}$$

$$\begin{aligned} \mathrm{sound}_\kappa &\in (Base_\kappa)^+ \to \mathbb{T} && \text{— composability of semantic objects}\\ \mathrm{sound}_\kappa(u_1,\ldots,u_l) &\iff \forall i = 1,\ldots,l-1\left(\mathrm{fin}_\kappa u_i = \mathrm{ini}_\kappa u_{i+1} \vee \mathrm{fin}_\kappa u_i = \bot\right)\end{aligned}$$

$$\begin{aligned} \mathrm{elem}_\kappa &\in State^2 \to Base_\kappa && \text{— elementary semantic objects}\\ \mathrm{comp}_\kappa &\in \left\{\boldsymbol{u} : \boldsymbol{u} \in (Base_\kappa)^+,\ \mathrm{sound}_\kappa \boldsymbol{u}\right\} \to Base_\kappa && \text{— composition of semantic objects}\end{aligned}$$

$$\begin{aligned} \mathrm{axm}_\kappa &\in \wp\left(State^2\right) \to \wp(Base_\kappa) && \mathrm{axm}_\kappa(P) = \left\{\mathrm{elem}_\kappa(s,t) : (s,t) \in P\right\}\\ \mathrm{rul}_\kappa &\in (\wp(Base_\kappa))^+ \to \wp(Base_\kappa) && \mathrm{rul}_\kappa(U_1,\ldots,U_l) = \left\{\mathrm{comp}_\kappa \boldsymbol{u} : \boldsymbol{u} \in U_1 \times \ldots \times U_l,\ \mathrm{sound}_\kappa \boldsymbol{u}\right\}\end{aligned}$$

$$\begin{aligned} \mathrm{iftrue}_\kappa &\in Expr \to \wp(Base_\kappa) && \mathrm{iftrue}_\kappa(E) = \mathrm{axm}_\kappa\left\{(s,s) : s \in State,\ e(E)(s) \geq \mathrm{tt}\right\}\\ \mathrm{iffalse}_\kappa &\in Expr \to \wp(Base_\kappa) && \mathrm{iffalse}_\kappa(E) = \mathrm{axm}_\kappa\left\{(s,s) : s \in State,\ e(E)(s) \geq \mathrm{ff}\right\}\end{aligned}$$

$$\begin{aligned} \mathcal{C}(A) &= \overbrace{Var \times Expr}^{\text{assignment}} + \overbrace{Proc \times Expr^*}^{\text{call}} + \overbrace{A \times A}^{\text{composition}} + \overbrace{Expr \times A \times A}^{\text{conditional}} + \overbrace{Expr \times A}^{\text{loop}}\\ \mathcal{C}(f) &= \mathrm{id}_{Var\times Expr} + \mathrm{id}_{Proc\times Expr^*} + f \times f + \mathrm{id}_{Expr} \times f \times f + \mathrm{id}_{Expr} \times f\end{aligned}$$

$$\delta \in Stmt \to \mathcal{C}(Stmt) \text{ — decomposition}$$

$$\begin{aligned} \delta(X \texttt{ := } E) &= (X,E)\\ \delta(\texttt{call } P(E_1,\ \ldots,\ E_l)) &= (P,(E_1,\ldots,E_l))\\ \delta(S_1 \ ;\ S_2) &= (S_1,S_2)\\ \delta(\texttt{if } E \texttt{ then } S_1 \texttt{ else } S_2) &= (E,S_1,S_2)\\ \delta(\texttt{while } E \texttt{ do } S) &= (E,(S \ ;\ \texttt{while } E \texttt{ do } S))\end{aligned}$$

$$\varphi_\kappa \in Env_\kappa \to \mathcal{C}(\wp(Base_\kappa)) \to \wp(Base_\kappa) \text{ — fusion}$$

$$\begin{aligned} \varphi_\kappa(e)(X,E) &= \mathrm{axm}_\kappa\left\{(s,s[X \mapsto e(E)(s)]) : s \in State\right\}\\ \varphi_\kappa(e)(P,(E_1,\ldots,E_l)) &= \mathrm{rul}_\kappa(e(P)(E_1,\ldots,E_l))\\ \varphi_\kappa(e)(U_1,U_2) &= \mathrm{rul}_\kappa(U_1,U_2)\\ \varphi_\kappa(e)(E,U_1,U_2) &= \mathrm{rul}_\kappa(\mathrm{iftrue}_\kappa(E),U_1) \cup \mathrm{rul}_\kappa(\mathrm{iffalse}_\kappa(E),U_2)\\ \varphi_\kappa(e)(E,U) &= \mathrm{rul}_\kappa(\mathrm{iftrue}_\kappa(E),U) \cup \mathrm{rul}_\kappa(\mathrm{iffalse}_\kappa(E))\end{aligned}$$

$$\begin{aligned} \boldsymbol{f}_\kappa &\in Env_\kappa \to (Stmt \to \wp(Base_\kappa)) \to (Stmt \to \wp(Base_\kappa)) && \boldsymbol{f}_\kappa(e)(z) = \delta\ ;\ \mathcal{C}(z)\ ;\ \varphi_\kappa(e)\\ s_\kappa &\in Stmt \to Env_\kappa \to \wp(Base_\kappa) && s_\kappa(S)(e) = \mathrm{gfp}(\boldsymbol{f}_\kappa(e))(S)\end{aligned}$$

Fig. 4. Common skeleton to all semantics of statements.

Then the base sets of finite, standard and transfinite ordinal trace semantics are defined like shown in Fig. 5 as sets of functions from indices to states. We will denote the state on trace l corresponding to ordinal o by l_o.

We say that a trace l *ends* if l has last element (in the transfinite cases, this is not equivalent to finiteness). Equivalently, a trace ends iff its index set is of the form $\mathbb{O}^o$ for some o. All transfinite traces are required to end since the core idea of transfinite semantics is the possibility of getting out from all loops. Hence the only endless traces are the infinite traces of standard semantics.

The following introduces the length of a trace. This notion is reasonable only in the case of ordinal traces (or other traces that can be reinterpreted as ordinal traces).

Definition 1. For any trace $l \in \mathbb{O}_o \to State$, the *length* of l is the least upper bound of $\mathbb{O}_o$. Denote the length of l by $|l|$.

By definition, $|l| = o$ whenever $l \in \mathbb{O}^o \to State = \mathbb{O}_{o+1} \to State$. This way, the length of a trace is the number of *steps* on it; for traces l that end, the number of states is $|l| + 1$.

In the definitions of ini, fin, elem and comp in Fig. 5, the exact kind is omitted because the definitions are common for all three kinds. Since the composition of semantic object lists is actually used for lists containing either one or two elements, the operation is specified for these cases only. If the first list ends then this operation coincides with "brazing concatenation"

$$Base_{\overrightarrow{+}} = \bigcup_{\substack{o \\ o<\omega}} (\mathbb{O}^o \to State)$$

$$Base_{\overrightarrow{\omega}} = \bigcup_{\substack{o \\ 0<o\leqslant\omega}} (\mathbb{O}_o \to State) = Base_{\overrightarrow{+}} \cup (\mathbb{O}_\omega \to State)$$

$$Base_{\overrightarrow{\alpha}} = \bigcup_{\substack{o \\ o<\alpha}} (\mathbb{O}^o \to State)$$

$$\mathrm{ini}_{\vec{\cdot}}(l) = l_0 \qquad \mathrm{fin}_{\vec{\cdot}}(l) = \left\{ \begin{array}{ll} l_{|l|} & \text{if } l \text{ ends} \\ \bot & \text{otherwise} \end{array} \right\}$$

$$\mathrm{elem}_{\vec{\cdot}}(s,t) = l \in \mathbb{O}^1 \to State \text{ such that } l_0 = s, l_1 = t$$

$$\mathrm{comp}_{\vec{\cdot}}(l) = l$$

$$\mathrm{comp}_{\vec{\cdot}}(l,m) = \left\{ \begin{array}{l} n \in \mathbb{O}^{|l|+|m|} \text{ such that } n_o = \left\{ \begin{array}{ll} l_o & \text{if } o < |l| \\ m_{o-|l|} & \text{otherwise} \end{array} \right\} \quad \text{if } l \text{ ends} \\ l \qquad\qquad\qquad\qquad\qquad\qquad\quad \text{otherwise} \end{array} \right\}$$

Fig. 5. Definition of the base sets and operators for ordinal trace semantics.

of [10], i.e., concatenation where the last element of the first list and the first element of the second list are fused together into one element. As $\mathrm{comp}_{\vec{\cdot}}$ is invoked only if the argument vector satisfies $\mathrm{sound}_{\vec{\cdot}}$ (see Fig. 4), it is guaranteed that these two elements always equal and one copy can be simply ignored. The assumption that α is a power of ω ensures that $\mathbb{O}_\alpha$ is closed w.r.t. binary addition, hence concatenation involves no overflows.

For introducing base sets for tree semantics and fractional semantics, we use cyclic definitions; these self-references must be resolved coinductively (i.e., anything is in whenever one cannot falsify this using the definition a finite number of times).

Definition 2. A *tree* t over set X has form $\dfrac{u_1 \cdots u_l}{x}$ where $x \in X$ and $u_1, \ldots, u_l$ ($l \geqslant 0$) are trees. Thereby, x is called the *root* of t and denoted $\mathrm{root}\ t$.

The notation $\dfrac{u_1 \cdots u_l}{x}$ means a purely formal construct resembling deduction trees where $u_1, \ldots, u_l$ stand at place of the premises and x stands at place of the conclusion.

Trees can be finite, as well as infinite. We need also an intermediate class of trees.

Definition 3. A tree $\dfrac{u_1 \cdots u_l}{x}$ is *left-finite* iff either $l = 0$ or each of $u_1, \ldots, u_{l-1}$ is finite and u_l is left-finite.

Every finite tree is clearly left-finite. In any left-finite tree, only the rightmost branch can be infinite. For example, the transfinite tree semantics of the statement (**while** `true` **do** `x := x + 1`) `;` `x := 1` that was presented in Section 1.3 is not left-finite since its left immediate subtree is infinite. However, its subtrees that were denoted t_i there are left-finite since the only infinite branch of them is the rightmost one.

Note that a tree in the form of an infinite chain c where $c = \dfrac{c}{x}$ for some x is also left-finite by definition. It is likely that such trees have no practical value and could be left out of all domains, together with all trees containing such subtrees.

In our framework, trees contain elements of the form $s \to t$ or $s \to \bot$ where $s, t \in State$; call them *transitions*. As the idea is to do an impression of deduction trees, not all trees of transitions are welcome. Restrictions along the lines of Glesner [5] must be imposed. We call the appropriate trees consistent.

Definition 4. A tree $\dfrac{u_1 \cdots u_l}{x}$ is *consistent* iff either $l = 0$ and x is of the form $s_0 \to s_1$ for $s_1 \neq \bot$, or $l > 0$ and each u_i is consistent and there exist $s_0, \ldots, s_l \in State \cup \{\bot\}$ such that the following holds:

(1) $x = s_0 \to s_l$;
(2) $\mathrm{root}\ u_i = s_{i-1} \to s_i$ for each $i = 1, \ldots, l$;
(3) $s_i = \bot$ only if $i = l$.

The definition implies that, in order to contain $\bot$, a consistent tree must be infinite.

The base sets of tree semantics together with underlying operators are defined in Fig. 6. Again, composition is defined for argument vectors containing one or two components only. Note that composition entails adding one level to the tree even

$$
\begin{aligned}
Base_{\hat{+}} &\ - \text{ the set of all finite consistent trees}\\
Base_{\hat{\omega}} &\ - \text{ the set of all left-finite consistent trees}\\
Base_{\hat{\propto}} &\ - \text{ the set of all consistent trees that do not contain } \bot
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{ini}_{\hat{\cdot}}(t) &= s_1 \text{ such that } \mathrm{root}\ t = s_1 \to s_2 \text{ for some } s_2 \in State\\
\mathrm{fin}_{\hat{\cdot}}(t) &= s_2 \text{ such that } \mathrm{root}\ t = s_1 \to s_2 \text{ for some } s_1 \in State
\end{aligned}
$$

$$
\mathrm{elem}_{\hat{\cdot}}(s,t) = \frac{}{s \to t}
$$

$$
\mathrm{comp}_{\hat{\cdot}}(t) = \frac{t}{\mathrm{root}\ t}
$$

$$
\mathrm{comp}_{\hat{\cdot}}(u,v) = \left\{ \begin{array}{ll} \dfrac{u \qquad v}{\mathrm{ini}_{\hat{\cdot}}\, u \to \mathrm{fin}_{\hat{\cdot}}\, v} & \text{if } \mathrm{fin}_{\hat{\cdot}}\, u \neq \bot \\ \dfrac{u}{\mathrm{root}\ u} & \text{otherwise} \end{array} \right\}
$$

Fig. 6. Definition of the base sets and operators for tree semantics.

in the case of one-element argument vector. It is easy to check that $Base_{\hat{+}}$, $Base_{\hat{\omega}}$ and $Base_{\hat{\propto}}$ are all closed w.r.t. $\mathrm{elem}_{\hat{\cdot}}$ and $\mathrm{comp}_{\hat{\cdot}}$.

For defining fractional trace semantics, it is first necessary to specify the sets of rational numbers that can be used for indexing. They must reflect the tree structure so that branching in a subtree means division of the corresponding segment of rationals into equal pieces. For example, $\left\{0, \frac{1}{2}, 1\right\}$ and $\left\{0, \frac{1}{4}, \frac{1}{2}, 1\right\}$ are allowed but $\left\{0, \frac{1}{3}, 1\right\}$ is not. We call the appropriate sets tree-like. Also notions that correspond to finiteness and left-finiteness in the case of trees are important. Denote by $[a; b]$ the set of rational numbers x such that $a \leqslant x \leqslant b$.

Definition 5. A set Z is *tree-like* iff $\{0,1\} \subseteq Z \subseteq [0;1]$ and either $Z = \{0,1\}$ or there exists an integer $n > 1$ such that, for each $i = 0, \ldots, n-1$, the linear projection of $Z \cap \left[\frac{i}{n}; \frac{i+1}{n}\right]$ to $[0;1]$ is tree-like. (The linear mapping making the projection is $\lambda a.\, na - i$, hence $\frac{i}{n}$ is mapped to 0 and $\frac{i+1}{n}$ is mapped to 1.)

Call a tree-like set *left-finite* iff either it is finite or its only accumulation point is 1.

Fig. 7 gives the base sets and the underlying operators of fractional trace semantics. The fractional traces are functions from tree-like sets of rational numbers to states. Composition of two functions, among which the first does not end with $\bot$, is found by compressing the first function to the first half of $[0;1]$ and the second function to the second half of the same interval, using linear mappings. This way, the middle point $\frac{1}{2}$ gets its state from both argument functions. As $\mathrm{comp}_{\tilde{\cdot}}$ is invoked only for argument vectors satisfying $\mathrm{sound}_{\tilde{\cdot}}$, the colliding states are equal and the result is a function again. Again, it is easy to check that $Base_{\tilde{+}}$, $Base_{\tilde{\omega}}$ and $Base_{\tilde{\propto}}$ are all closed w.r.t. $\mathrm{elem}_{\tilde{\cdot}}$ and $\mathrm{comp}_{\tilde{\cdot}}$.

Finally, note that, if necessary, it is possible to make further restrictions to the semantic objects forming the sets $Base_{\kappa}$. When doing this, one only has to ensure that $Base_{\kappa}$ remains closed w.r.t. elem_{κ} and comp_{κ}.

$$
\begin{aligned}
Base_{\tilde{+}} &= \{Z : Z \text{ is a finite tree-like set}\} \to State\\
Base_{\tilde{\omega}} &= (\{Z : Z \text{ is a left-finite tree-like set}\} \to State) \cap\\
&\quad \{f : \forall a\, (f(a) = \bot \iff a = 1 \ \wedge\ \mathrm{dom} f \text{ is infinite})\}\\
Base_{\tilde{\propto}} &= \{Z : Z \text{ is a tree-like set}\} \to State
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{ini}_{\tilde{\cdot}}(f) &= f(0)\\
\mathrm{fin}_{\tilde{\cdot}}(f) &= f(1)
\end{aligned}
$$

$$
\mathrm{elem}_{\tilde{\cdot}}(s,t) = \{0 \mapsto s, 1 \mapsto t\}
$$

$$
\begin{aligned}
\mathrm{comp}_{\tilde{\cdot}}(f) &= f\\
\mathrm{comp}_{\tilde{\cdot}}(g,h) &= \left\{ \begin{array}{ll} (((\lambda a.\, 2a)\ ;\ g) \cup ((\lambda a.\, 2a-1)\ ;\ h)) & \text{if } \mathrm{fin}_{\tilde{\cdot}}\, g \neq \bot \\ g & \text{otherwise} \end{array} \right\}
\end{aligned}
$$

Fig. 7. Definition of the base sets and operators for fractional trace semantics.

Further restrictions are needed if we want to keep the values of variables during transfinite computations under control; currently, they are just undetermined after each infinite loop. This is discussed further in Section 6.

3.4. Properties of semantics

Additionally to the common structure of all semantics given in Fig. 4, one can observe some common properties that cannot be deduced from the common framework. They are formulated in Propositions 6 and 7.

Proposition 6 refines the nature of the operators involved in the framework: namely, every elementary semantic object built upon two states has the same states as the initial and final ones; the composition of semantic objects always shares the initial state with the first term in the composition; analogously for the final state and the last term except if a preceding term ends with non-state; and the terms in the composition that follow a term ending with non-state do not matter.

Proposition 6. *For all kinds κ of semantics that are represented in Fig. 2, the following holds*:

(i) $\mathrm{ini}_\kappa(\mathrm{elem}_\kappa(s,t)) = s$ *for all* $s,t \in State$;
(ii) $\mathrm{fin}_\kappa(\mathrm{elem}_\kappa(s,t)) = t$ *for all* $s,t \in State$;
(iii) $\mathrm{ini}_\kappa(\mathrm{comp}_\kappa(u_1,\ldots,u_l)) = \mathrm{ini}_\kappa u_1$ *for* $l = 1,2$ *and all* $u_1,\ldots,u_l \in Base_\kappa$ *such that* $\mathrm{sound}_\kappa(u_1,\ldots,u_l)$;
(iv) $\mathrm{fin}_\kappa(\mathrm{comp}_\kappa(u_1,\ldots,u_l)) = \mathrm{fin}_\kappa u_l$ *for* $l = 1,2$ *and all* $u_1,\ldots,u_l \in Base_\kappa$ *such that* $\mathrm{sound}_\kappa(u_1,\ldots,u_l)$ *and none of* $\mathrm{fin}_\kappa u_1,\ldots,\mathrm{fin}_\kappa u_{l-1}$ *is* $\bot$;
(v) *for all* $u,v,w \in Base_\kappa$, *if* $\mathrm{sound}_\kappa(u,v)$ *and* $\mathrm{sound}_\kappa(u,w)$ *and* $\mathrm{fin}_\kappa u = \bot$ *then* $\mathrm{comp}_\kappa(u,v) = \mathrm{comp}_\kappa(u,w)$.

Proof. Straightforward by specifications in Figs. 5–7. □

The cocontinuity of semantic operators (discussed shortly in Section 1 and proven in Section 5) holds thanks to the injectivity of mappings elem_κ and comp_κ.

Proposition 7.

(i) *For all kinds κ of semantics occurring in Fig. 2,* elem_κ *is one-to-one*.
(ii) *For all kinds κ of non-standard semantics occurring in Fig. 2,* comp_κ *is one-to-one*.

Proof

(i) A direct consequence of Proposition 6(i)–(ii).
(ii) A straightforward case study by the specifications in Fig. 5–7. First note that fin_κ never returns $\bot$ for non-standard semantics. Hence the first case is chosen in each definition of binary comp_κ. In all these cases, the result contains both operands entirely. □

4. Semantics of procedures

Assume that the code is syntactically correct. This means basically that all calls to procedures have the right number of arguments and every called procedure has exactly one declaration.

The definition of semantics of procedures is given uniformly for all kinds of semantics using the same underlying sets and operators already defined for semantics of statements.

We first define the semantics of single declarations in Fig. 8. It is a pair that associates a meaning to the procedure declared; this meaning depends on a given environment parameter. The type of the object associated to the procedure is such that the semantics of a declaration can be used as elements of environments (an environment can be treated as a set of such pairs).

The rhs of the definition of declaration semantics is written like a computer program in order to make the long expression maximally readable. The components of the resulting pair are separated by the $\mapsto$ sign for emphasizing the intent to use the pair as a building block of a function. The second component of the pair is a mapping whose argument is a possible vector of actual parameters found in a call statement. The mapping returns a set of semantic objects obtained by wrapping entry and return actions around the semantics of the body of the procedure ($s_\kappa(S)(e)$). The entry action ($\mathrm{axm}_\kappa\{(s, s[X_1 \mapsto e(E_1)(s), \ldots, X_l \mapsto e(E_l)(s)]) : s \in State\}$) changes the values of formal variables according to the actual parameters. The return action ($\mathrm{axm}_\kappa\{(s,t) : s,t \in State, \forall Y \neq X_1,\ldots,X_l\,(s(Y) = t(Y))\}$) leaves the values of the formal parameters undetermined but keeps the other variables unchanged. Finally, set meet with $\{t : t \in Base_\kappa,\ \forall i\,(\mathrm{ini}_\kappa(t)(X_i) = \mathrm{fin}_\kappa(t)(X_i))\}$ takes care of the formal parameters getting back their values they had before the call.

The meaning of single declarations does not involve semantic cycles. If the body of the procedure calls the same procedure, its meaning is taken from the environment parameter. The semantics of recursion is given at module level, see Fig. 9. The definition schema of module semantics m_κ is similar to the schema in Fig. 4: we again use a decomposition coalgebra, a functor and fusion algebras. Like in the case of statements, the fusion algebras depend on an additional environment parameter. A superscript $^{\mathrm{m}}$ is used for distinguishing the elements of module semantics from that of statement semantics.

$$d_\kappa \in Decl \to Env_\kappa \to Proc \times (Expr^* \to \wp(Base_\kappa))$$

$$\begin{array}{l} d_\kappa(\mathbf{proc}\ P(X_1, \ldots, X_l)\ \mathbf{is}\ S)(e) \\ \quad = (\\ \qquad P \mapsto \\ \qquad \lambda(E_1, \ldots, E_l). \\ \qquad\quad \mathrm{rul}_\kappa \\ \qquad\quad (\\ \qquad\qquad \mathrm{axm}_\kappa \left\{(s, s[X_1 \mapsto e(E_1)(s), \ldots, X_l \mapsto e(E_l)(s)]) : s \in State\right\}, \\ \qquad\qquad \mathrm{rul}_\kappa \\ \qquad\qquad (\\ \qquad\qquad\quad s_\kappa(S)(e), \\ \qquad\qquad\quad \mathrm{axm}_\kappa \left\{(s,t) : s, t \in State,\ \forall Y \neq X_1, \ldots, X_l\, (s(Y) = t(Y))\right\} \\ \qquad\qquad) \\ \qquad\quad) \\ \qquad\quad \cap \left\{t : t \in Base_\kappa,\ \forall i \left(\mathrm{ini}_\kappa(t)(X_i) = \mathrm{fin}_\kappa(t)(X_i)\right)\right\} \\ \quad) \end{array}$$

Fig. 8. Semantics of procedure declarations.

$$\mathcal{C}^{\mathrm{m}}(A) = A \times \wp(Decl) \qquad \mathcal{C}^{\mathrm{m}}(f) = f \times \mathrm{id}_{\wp(Decl)}$$

$$\delta^{\mathrm{m}} \in Mod \to \mathcal{C}^{\mathrm{m}}(Mod) \qquad \delta^{\mathrm{m}}(D_1\ ;\ \ldots\ ;\ D_m) = (D_1\ ;\ \ldots\ ;\ D_m, \{D_1, \ldots, D_m\})$$

$$\varphi^{\mathrm{m}}_\kappa \in Env_\kappa \to \mathcal{C}^{\mathrm{m}}(Env_\kappa) \to Env_\kappa \qquad \varphi^{\mathrm{m}}_\kappa(e)(e', \mathcal{D}) = e\left[d_\kappa(D)(e') : D \in \mathcal{D}\right]$$

$$\boldsymbol{f}^{\mathrm{m}}_\kappa \in Env_\kappa \to (Mod \to Env_\kappa) \to (Mod \to Env_\kappa) \qquad \boldsymbol{f}^{\mathrm{m}}_\kappa(e)(z) = \delta^{\mathrm{m}}\ ;\ \mathcal{C}^{\mathrm{m}}(z)\ ;\ \varphi^{\mathrm{m}}_\kappa(e)$$

$$m_\kappa \in Mod \to Env_\kappa \to Env_\kappa \qquad m_\kappa(M)(e) = \mathrm{gfp}(\boldsymbol{f}^{\mathrm{m}}_\kappa(e))(M)$$

Fig. 9. Semantics of modules.

The aim is to define the semantics of any module as a transformation of environments of Env_κ which updates the values of the procedures declared in the module according to their declaration while keeping the values of other procedures unchanged. As the semantics of a module must be described in terms of the semantics of its declarations and also in terms of itself (because they can be recursive), the decomposition δ^{m} repeats its argument, as well as gives the set of its declarations. A fusion $\varphi^{\mathrm{m}}_\kappa$, given an environment e and a pair $(e', \mathcal{D})$ of a new environment and a set of declarations, returns the update of e with the semantics of all procedures declared in $\mathcal{D}$ provided by e' (the expression $e\left[d_\kappa(D)(e') : D \in \mathcal{D}\right]$ denotes the update of e with all correspondences that occur in the set $\left\{d_\kappa(D)(e') : D \in \mathcal{D}\right\}$); hence precisely the procedures that are declared in $\mathcal{D}$ are updated).

Now, the operator $\boldsymbol{f}^{\mathrm{m}}_\kappa$ and the module semantics m_κ are defined similarly to $\boldsymbol{f}_\kappa$ and s_κ, respectively. The order on $Mod \to Env_\kappa$ is defined by repeated lifting of the inclusion order on $\wp(Base_\kappa)$.

5. Correctness of the definitions

Here, we remove the last uncertainties concerning the definitions of the semantics by proving the existence of the greatest fixpoints that were referred to by the definitions. Furthermore, for six non-standard semantics, we prove that the greatest fixpoints can be expressed as limits of non-transfinite sequences.

By Tarski's well-known theorem, an operator on a complete lattice has a greatest (as well as least) fixpoint whenever it is monotone. The domains of our operators $\boldsymbol{f}_\kappa(e)$ and $\boldsymbol{f}^{\mathrm{m}}_\kappa(e)$ are complete lattices since the order relations were defined via lifting of powerset lattices. So for the existence of greatest fixpoints, it suffices to prove monotonicity.

According to Kleene's theorem, for expressing a greatest fixpoint as the limit of a sequence, it suffices to establish cocontinuity of the corresponding operator. (An operator is called *continuous* iff it preserves lubs of non-empty chains; *cocontinuity* is the dual notion.)

Proving these two things is the content of Sections 5.1 and 5.2. In the case of statements, we prove continuity instead of monotonicity; this is fine since continuity (as well as cocontinuity) implies monotonicity. We prefer to prove the stronger property when possible since this sometimes enables to reuse fragments of proofs. In the case of procedures, continuity does not hold but, again to be able to unify two similar proof fragments, we state monotonicity in the form of preserving glbs of finite non-empty chains.

5.1. Semantics of statements

We obtain the desired results as corollaries from a list of small lemmas. Note that continuity is stated for all semantics matching the schema in Fig. 4, irrespectively of whether they occur in the table in Fig. 2. Cocontinuity assumes properties of our concrete semantics, it holds only for six non-standard semantics under consideration.

The first two lemmas establish continuity and cocontinuity of axm_κ and rul_κ. For the latter, we assume the set inclusion order being lifted componentwise to lists of sets.

Lemma 8. *For all semantics defined by the schema in Fig. 4:*
(i) *the operator* axm_κ *preserves all lubs*;
(ii) *the operator* rul_κ *preserves lubs of chains*.

Proof. (i) Let $(P_i : i \in I)$ be any family of sets of pairs of states. For any $t \in Base_\kappa$,

$$\begin{aligned} t \in \mathrm{axm}_\kappa\left(\bigcup_{i\in I} P_i\right) &\iff \exists p \in \bigcup_{i\in I} P_i\,(t = \mathrm{elem}_\kappa p)\\ &\iff \exists i \in I\,\exists p \in P_i\,(t = \mathrm{elem}_\kappa p)\\ &\iff \exists i \in I\,(t \in \mathrm{axm}_\kappa(P_i))\\ &\iff t \in \bigcup_{i\in I} \mathrm{axm}_\kappa(P_i). \end{aligned}$$

(ii) Let $(L_i : i \in I)$ be a chain of lists of sets of semantic objects. This implies that the lists have equal length l and, for arbitrary two of them, either all corresponding components are in relation $\subseteq$ or they are in relation $\supseteq$. Let $L_i = (U_{i,1}, \ldots, U_{i,l})$; then

$$\begin{aligned} & t \in \mathrm{rul}_\kappa \bigvee_{i\in I} L_i\\ &\iff t \in \mathrm{rul}_\kappa\left(\bigcup_{i\in I} U_{i,1}, \ldots, \bigcup_{i\in I} U_{i,l}\right)\\ &\iff \exists u_1 \in \bigcup_{i\in I} U_{i,1}, \ldots, u_l \in \bigcup_{i\in I} U_{i,l}\,\big(\mathrm{sound}_\kappa(u_1, \ldots, u_l) \ \wedge\ t = \mathrm{comp}_\kappa(u_1, \ldots, u_l)\big)\\ &\iff \exists i \in I\,\exists u_1 \in U_{i,1}, \ldots, u_l \in U_{i,l}\,\big(\mathrm{sound}_\kappa(u_1, \ldots, u_l) \ \wedge\ t = \mathrm{comp}_\kappa(u_1, \ldots, u_l)\big)\\ &\iff \exists i \in I\,\big(t \in \mathrm{rul}_\kappa(U_{i,1}, \ldots, U_{i,l})\big)\\ &\iff t \in \bigcup_{i\in I} \mathrm{rul}_\kappa L_i, \end{aligned}$$

where the 'only if' part of the third 'iff' (bringing a common $i \in I$ to the front) holds by the assumption that L is a chain. □

Lemma 9. *Let* $\kappa \in \left\{\overrightarrow{\mp}, \widetilde{\mp}, \widehat{\mp}, \overrightarrow{\propto}, \widetilde{\propto}, \widehat{\propto}\right\}$. *Then*:
(i) *the operator* axm_κ *preserves glbs of non-empty families*;
(ii) *the operator* rul_κ *preserves glbs of non-empty chains*.

Proof. (i) Let $(P_i : i \in I)$ be any non-empty family of sets of pairs of states. For any $t \in Base_\kappa$,

$$\begin{aligned} t \in \mathrm{axm}_\kappa\left(\bigcap_{i\in I} P_i\right) &\iff \exists p \in \bigcap_{i\in I} P_i\,(t = \mathrm{elem}_\kappa p)\\ &\iff \exists p \in State^2\,\forall i \in I\,(p \in P_i \ \wedge\ t = \mathrm{elem}_\kappa p)\\ &\iff \forall i \in I\,\exists p \in State^2\,(p \in P_i \ \wedge\ t = \mathrm{elem}_\kappa p)\\ &\iff \forall i \in I\,(t \in \mathrm{axm}_\kappa(P_i))\\ &\iff t \in \bigcap_{i\in I} \mathrm{axm}_\kappa(P_i), \end{aligned}$$

where the 'if' part of the third 'iff' (interchanging the quantifiers) holds by injectivity of elem_κ (Proposition 7).

(ii) Let $(L_i : i \in I)$ be a non-empty chain of lists of sets of semantic objects. This implies that lists have equal length l; let $L_i = (U_{i,1}, \ldots, U_{i,l})$. We get

$$\begin{aligned}
& t \in \mathrm{rul}_\kappa \bigwedge_{i\in I} L_i \\
& \iff\ t \in \mathrm{rul}_\kappa \left(\bigcap_{i\in I} U_{i,1}, \ldots, \bigcap_{i\in I} U_{i,l}\right) \\
& \iff\ \exists \boldsymbol{u} \in \left(\bigcap_{i\in I} U_{i,1}\right) \times \ldots \times \left(\bigcap_{i\in I} U_{i,l}\right) (\mathrm{sound}_\kappa \boldsymbol{u}\ \wedge\ t = \mathrm{comp}_\kappa \boldsymbol{u}) \\
& \iff\ \exists \boldsymbol{u} \in Base_\kappa^l\ \forall i \in I\, (\boldsymbol{u} \in U_{i,1} \times \ldots \times U_{i,l}\ \wedge\ \mathrm{sound}_\kappa \boldsymbol{u}\ \wedge\ t = \mathrm{comp}_\kappa \boldsymbol{u}) \\
& \iff\ \forall i \in I\, \exists \boldsymbol{u} \in Base_\kappa^l\, (\boldsymbol{u} \in U_{i,1} \times \ldots \times U_{i,l}\ \wedge\ \mathrm{sound}_\kappa \boldsymbol{u}\ \wedge\ t = \mathrm{comp}_\kappa \boldsymbol{u}) \\
& \iff\ \forall i \in I\, (t \in \mathrm{rul}_\kappa\, (U_{i,1}, \ldots, U_{i,l})) \\
& \iff\ t \in \bigcap_{i\in I} \mathrm{rul}_\kappa L_i,
\end{aligned}$$

where the 'if' part of the fourth 'iff' (interchanging the quantifiers) holds by injectivity of comp_κ (Proposition 7). □

Assume the inclusion order on $\wp(Base_\kappa)$ being standardly lifted to $\mathcal{C}(\wp(Base_\kappa))$. (Note that $\mathcal{C}(\wp(Base_\kappa))$ does not become a lattice since the elements in different summand sets are incomparable.)

Lemma 10. *For all semantics defined by the schema in Fig. 4 and for every $e \in Env_\kappa$, the operator $\varphi_\kappa(e)$ preserves lubs of non-empty chains.*

Proof. A chain in $\mathcal{C}(\wp(Base_\kappa))$ can consist of elements of one of five different types correspondingly to the summand of the categorical sum in the definition of $\mathcal{C}$. Consider all cases.

Assignment. The lifted order in this summand is discrete, thus any non-empty chain contains just one element of the form (X, E) and we get

$$\varphi_\kappa(e)\left(\bigvee_{i\in I}(X,E)\right) = \varphi_\kappa(e)(X,E) = \bigcup_{i\in I}\varphi_\kappa(e)(X,E).$$

Call. Similar to the assignment case.

Composition. The elements of our chain are of the form (U_i, V_i). Using Lemma 8, we get

$$\varphi_\kappa(e)\left(\bigvee_{i\in I}(U_i, V_i)\right) = \mathrm{rul}_\kappa\left(\bigvee_{i\in I}(U_i, V_i)\right) = \bigcup_{i\in I}\mathrm{rul}_\kappa(U_i, V_i) = \bigcup_{i\in I}\varphi_\kappa(e)(U_i, V_i).$$

Conditional. The elements of our chain are of the form (E, U_i, V_i) where E does not change. Hence both the family of pairs $(\mathrm{iftrue}_\kappa(E), U_i)$ and the family of pairs $(\mathrm{iffalse}_\kappa(E), V_i)$ are chains. Using Lemma 8, we obtain

$$\begin{aligned}
\varphi_\kappa(e)\left(\bigvee_{i\in I}(E, U_i, V_i)\right) &= \varphi_\kappa(e)\left(E, \bigcup_{i\in I} U_i, \bigcup_{i\in I} V_i\right) \\
&= \mathrm{rul}_\kappa\left(\mathrm{iftrue}_\kappa(E), \bigcup_{i\in I} U_i\right) \cup \mathrm{rul}_\kappa\left(\mathrm{iffalse}_\kappa(E), \bigcup_{i\in I} V_i\right) \\
&= \bigcup_{i\in I}\mathrm{rul}_\kappa\left(\mathrm{iftrue}_\kappa(E), U_i\right) \cup \bigcup_{i\in I}\mathrm{rul}_\kappa\left(\mathrm{iffalse}_\kappa(E), V_i\right) \\
&= \bigcup_{i\in I}\left(\mathrm{rul}_\kappa\left(\mathrm{iftrue}_\kappa(E), U_i\right) \cup \mathrm{rul}_\kappa\left(\mathrm{iffalse}_\kappa(E), V_i\right)\right) \\
&= \bigcup_{i\in I}\varphi_\kappa(e)(E, U_i, V_i).
\end{aligned}$$

Loop. The elements of our chain are of the form (E, U_i) where E does not change. Noting again that the pairs $(\mathrm{iftrue}_\kappa(E), U_i)$ form a chain and using Lemma 8, we obtain

$$\begin{aligned}\varphi_\kappa(e)\left(\bigvee_{i\in I}(E,U_i)\right) &= \varphi_\kappa(e)\left(E,\bigcup_{i\in I}U_i\right)\\ &= \mathrm{rul}_\kappa\left(\mathrm{iftrue}_\kappa(E),\bigcup_{i\in I}U_i\right)\cup\mathrm{rul}_\kappa(\mathrm{iffalse}_\kappa(E))\\ &= \bigcup_{i\in I}\mathrm{rul}_\kappa\left(\mathrm{iftrue}_\kappa(E),U_i\right)\cup\mathrm{rul}_\kappa(\mathrm{iffalse}_\kappa(E))\\ &= \bigcup_{i\in I}\varphi_\kappa(e)(E,U_i).\quad\square\end{aligned}$$

Lemma 11. *For all* $\kappa\in\left\{\overrightarrow{+},\widetilde{+},\widehat{+},\overrightarrow{\alpha},\widetilde{\alpha},\widehat{\alpha}\right\}$ *and for all* $e\in Env_\kappa$, *the operator* $\varphi_\kappa(e)$ *preserves glbs of non-empty chains*.

Proof. Like in the proof of the previous lemma, consider five cases of the summand set where the members of our chain are taken from.

Assignment, call, composition are handled like the same cases of Lemma 10.

Conditional. The elements of our chain are of the form (E, U_i, V_i). Similarly to the corresponding case in Lemma 10, write

$$\begin{aligned}\varphi_\kappa(e)\left(\bigwedge_{i\in I}(E,U_i,V_i)\right) &= \varphi_\kappa(e)\left(E,\bigcap_{i\in I}U_i,\bigcap_{i\in I}V_i\right)\\ &= \mathrm{rul}_\kappa\left(\mathrm{iftrue}_\kappa(E),\bigcap_{i\in I}U_i\right)\cup\mathrm{rul}_\kappa\left(\mathrm{iffalse}_\kappa(E),\bigcap_{i\in I}V_i\right)\\ &= \bigcap_{i\in I}\mathrm{rul}_\kappa\left(\mathrm{iftrue}_\kappa(E),U_i\right)\cup\bigcap_{i\in I}\mathrm{rul}_\kappa(\mathrm{iffalse}_\kappa(E),V_i)\\ &= \bigcap_{i\in I}\left(\mathrm{rul}_\kappa(\mathrm{iftrue}_\kappa(E),U_i)\cup\mathrm{rul}_\kappa(\mathrm{iffalse}_\kappa(E),V_i)\right)\\ &= \bigcap_{i\in I}\varphi_\kappa(e)(E,U_i,V_i).\end{aligned}$$

Here, the third equality holds by Lemma 9. The fourth equality needs special attention. Downward inclusion is easy. For the other direction, suppose

$$t\in\bigcap_{i\in I}\left(\mathrm{rul}_\kappa(\mathrm{iftrue}_\kappa(E),U_i)\cup\mathrm{rul}_\kappa(\mathrm{iffalse}_\kappa(E),V_i)\right),$$

i.e., $t\in\mathrm{rul}_\kappa(\mathrm{iftrue}_\kappa(E),U_i)\cup\mathrm{rul}_\kappa(\mathrm{iffalse}_\kappa(E),V_i)$ for every $i\in I$. If $t\in\mathrm{rul}_\kappa(\mathrm{iftrue}_\kappa(E),U_i)$ for every $i\in I$ or $t\in\mathrm{rul}_\kappa(\mathrm{iffalse}_\kappa(E),V_i)$ for every $i\in I$ then we are done. Therefore consider the case where $t\notin\mathrm{rul}_\kappa(\mathrm{iffalse}_\kappa(E),V_{i_1})$ for some $i_1\in I$ and $t\notin\mathrm{rul}_\kappa(\mathrm{iftrue}_\kappa(E),U_{i_2})$ for some $i_2\in I$. But the pairs (U_i,V_i) form a chain and hence are all pairwise comparable; let i_* be such that $(U_{i_*},V_{i_*})=(U_{i_1},V_{i_1})\wedge(U_{i_2},V_{i_2})$. As rul_κ is monotone by Lemma 9 (as well as by Lemma 8), it turns out that $t\notin\mathrm{rul}_\kappa(\mathrm{iffalse}_\kappa(E),V_{i_*})$ and $t\notin\mathrm{rul}_\kappa(\mathrm{iftrue}_\kappa(E),U_{i_*})$, a contradiction.

Loop. The elements of our chain are of the form (E, U_i). Using Lemma 9, we get

$$\begin{aligned}\varphi_\kappa(e)\left(\bigwedge_{i\in I}(E,U_i)\right) &= \varphi_\kappa(e)\left(E,\bigcap_{i\in I}U_i\right)\\ &= \mathrm{rul}_\kappa\left(\mathrm{iftrue}_\kappa(E),\bigcap_{i\in I}U_i\right)\cup\mathrm{rul}_\kappa\ (\mathrm{iffalse}_\kappa(E))\\ &= \bigcap_{i\in I}\mathrm{rul}_\kappa(\mathrm{iftrue}_\kappa(E),U_i)\cup\mathrm{rul}_\kappa\ (\mathrm{iffalse}_\kappa(E))\\ &= \bigcap_{i\in I}(\mathrm{rul}_\kappa(\mathrm{iftrue}_\kappa(E),U_i)\cup\mathrm{rul}_\kappa(\mathrm{iffalse}_\kappa(E)))\\ &= \bigcap_{i\in I}\varphi_\kappa(e)(E,U_i),\end{aligned}$$

where the second last equality holds by distributivity between join and (non-empty) meet. $\square$

The following fact is standard since $\mathcal{C}$ is a polynomial functor. We omit the proof.

Lemma 12. *The part of functor $\mathcal{C}$ that works on functions preserves lubs and glbs of non-empty families.*

Now, it remains to tie all pieces together.

Theorem 13. *Let κ be a kind of semantics defined by the schema in Fig. 4 and let $e \in Env_\kappa$.*
(i) The operator $\boldsymbol{f}_\kappa(e)$ preserves lubs of non-empty chains.
(ii) If $\kappa \in \left\{\overrightarrow{+}, \widetilde{+}, \widehat{+}, \overrightarrow{\alpha}, \widetilde{\alpha}, \widehat{\alpha}\right\}$ then $\boldsymbol{f}_\kappa(e)$ preserves glbs of non-empty chains.

Proof . (i) Let $(z_i : i \in I)$ be a non-empty chain of mappings of type $Stmt \to \wp(Base_\kappa)$. In order to prove $\boldsymbol{f}_\kappa(e)\left(\bigvee_{i\in I} z_i\right) = \bigvee_{i\in I} \boldsymbol{f}_\kappa(e)(z_i)$, fix a statement S and prove the equality $\boldsymbol{f}_\kappa(e)\left(\bigvee_{i\in I} z_i\right)(S) = \bigcup_{i\in I} \boldsymbol{f}_\kappa(e)(z_i)(S)$. By Lemmas 12 and 10, indeed,

$$\begin{aligned}
\boldsymbol{f}_\kappa(e)\left(\bigvee_{i\in I} z_i\right)(S) &= \varphi_\kappa(e)\left(\mathcal{C}\left(\bigvee_{i\in I} z_i\right)(\delta(S))\right)\\
&= \varphi_\kappa(e)\left(\bigvee_{i\in I} \mathcal{C}(z_i)(\delta(S))\right)\\
&= \bigcup_{i\in I} \varphi_\kappa(e)(\mathcal{C}(z_i)(\delta(S)))\\
&= \bigcup_{i\in I} \boldsymbol{f}_\kappa(e)(z_i)(S).
\end{aligned}$$

(ii) Analogously, by using Lemmas 12 and 11. □

One may ask whether the assumption $\kappa \in \left\{\overrightarrow{+}, \widetilde{+}, \widehat{+}, \overrightarrow{\alpha}, \widetilde{\alpha}, \widehat{\alpha}\right\}$ is really needed. The following proposition shows that the answer is yes.

Proposition 14. *For $\kappa \in \{\overrightarrow{\omega}, \widetilde{\omega}, \widehat{\omega}\}$, the operators $\boldsymbol{f}_\kappa(e)$ do not preserve glbs of non-empty chains.*

Proof. Fix two statements S_1, S_2 arbitrarily. Choose an infinite semantic object t and a family $(v_i : i \in \mathbb{N})$ of semantic objects, all from $Base_\kappa$. Let $(z_i : i \in \mathbb{N})$ be a family of mappings of type $Stmt \to \wp(Base_\kappa)$ such that $z_i(S_1) = \{t\}$ and $z_i(S_2) = \{v_i, v_{i+1}, \dots\}$ for each i. Then

$$\bigcap_{i\in\mathbb{N}} z_i(S_1) = \{t\}, \qquad \bigcap_{i\in\mathbb{N}} z_i(S_2) = \varnothing.$$

Therefore,

$$\left(\boldsymbol{f}_\kappa(e)\left(\bigwedge_{i\in\mathbb{N}} z_i\right)\right)(S_1 \;;\; S_2) = \varphi_\kappa(e)\left(\mathcal{C}\left(\bigwedge_{i\in\mathbb{N}} z_i\right)(\delta(S_1 \;;\; S_2))\right) = \varphi_\kappa(e)(\{t\}, \varnothing) = \mathrm{rul}_\kappa(\{t\}, \varnothing) = \varnothing.$$

However,

$$\begin{aligned}
\bigcap_{i\in\mathbb{N}} \boldsymbol{f}_\kappa(e)(z_i)(S_1 \;;\; S_2) &= \bigcap_{i\in\mathbb{N}} \varphi_\kappa(e)(\mathcal{C}(z_i)(\delta(S_1 \;;\; S_2)))\\
&= \bigcap_{i\in\mathbb{N}} \varphi_\kappa(e)(\{t\}, \{v_i, v_{i+1}, \dots\})\\
&= \bigcap_{i\in\mathbb{N}} \mathrm{rul}_\kappa(\{t\}, \{v_i, v_{i+1}, \dots\})\\
&= \bigcap_{i\in\mathbb{N}} \{\mathrm{comp}_\kappa\, t\}\\
&= \{\mathrm{comp}_\kappa\, t\}. \qquad \square
\end{aligned}$$

5.2. Semantics of procedures

Again, we prove monotonicity (that is equivalent to preservation of glbs of finite non-empty chains) and cocontinuity results simultaneously via a series of lemmas.

Lemma 15. *Let κ be a kind of semantics matching the schema in Fig. 4.*
(i) *Then φ_κ preserves glbs of finite non-empty chains.*
(ii) *If $\kappa \in \left\{\overrightarrow{+}, \widetilde{+}, \widehat{+}, \overrightarrow{\alpha}, \widetilde{\alpha}, \widehat{\alpha}\right\}$ then φ_κ preserves glbs of all non-empty chains.*

Proof. Let $(e_i : i \in I)$ be any (finite for part (i)) non-empty chain of environments. We must prove that $\varphi_\kappa\left(\bigwedge_{i\in I} e_i\right)(x) = \bigcap_{i\in I} \varphi_\kappa(e_i)(x)$ for all $x \in \mathcal{C}(\wp(Base_\kappa))$. For all cases different from call, this is straightforward since the return value of φ_κ does not depend on the environment argument. For the call case,

$$\begin{aligned}\varphi_\kappa\left(\bigwedge_{i\in I} e_i\right)(P,(E_1,\ldots,E_l)) &= \mathrm{rul}_\kappa\left(\bigcap_{i\in I} e_i(P)(E_1,\ldots,E_l)\right)\\ &= \bigcap_{i\in I} \mathrm{rul}_\kappa\,(e_i(P)(E_1,\ldots,E_l)) = \bigcap_{i\in I} \varphi_\kappa(e_i)(P,(E_1,\ldots,E_l)),\end{aligned}$$

where the second equality (preservation of glb by rul_κ) follows from monotonicity of rul_κ (a consequence of Lemma 8) for part (i) and from Lemma 9 for part (ii). □

Lemma 16. *Let κ be a kind of semantics matching the schema in Fig. 4.*
(i) *Then $\boldsymbol{f}_\kappa$ preserves glbs of finite non-empty chains.*
(ii) *If $\kappa \in \left\{\overrightarrow{+}, \widetilde{+}, \widehat{+}, \overrightarrow{\alpha}, \widetilde{\alpha}, \widehat{\alpha}\right\}$ then $\boldsymbol{f}_\kappa$ preserves glbs of all non-empty chains.*

Proof. Let $(e_i : i \in I)$ be any (finite for part (i)) non-empty chain of environments. Using Lemma 15, we obtain for arbitrary semantics $z \in Stmt \to \wp(Base_\kappa)$ and statement S that

$$\boldsymbol{f}_\kappa\left(\bigwedge_{i\in I} e_i\right)(z)(S) = \varphi_\kappa\left(\bigwedge_{i\in I} e_i\right)(\mathcal{C}(z)(\delta(S))) = \bigcap_{i\in I} \varphi_\kappa(\mathcal{C}(z)(\delta(S))) = \bigcap_{i\in I} \boldsymbol{f}_\kappa(e_i)(z)(S). \qquad \square$$

Lemma 17. *Let κ be a kind of semantics matching the schema in Fig. 4 and let $S \in Stmt$.*
(i) *Then $s_\kappa(S)$ preserves glbs of finite non-empty chains.*
(ii) *If $\kappa \in \left\{\overrightarrow{+}, \widetilde{+}, \widehat{+}, \overrightarrow{\alpha}, \widetilde{\alpha}, \widehat{\alpha}\right\}$ then $s_\kappa(S)$ preserves glbs of all non-empty chains.*

Proof. (i) We show that $s_\kappa(S)$ is monotone. Let $e_1, e_2 \in Env_\kappa$ such that $e_1 \le e_2$. By Lemma 16, $\boldsymbol{f}_\kappa(e_1) \le \boldsymbol{f}_\kappa(e_2)$. Expanding gfp by Tarski's theorem, we get

$$\mathrm{gfp}(\boldsymbol{f}_\kappa(e_1)) = \bigvee_{\substack{z\in Stmt\to\wp(Base_\kappa)\\ z\le \boldsymbol{f}_\kappa(e_1)(z)}} z \le \bigvee_{\substack{z\in Stmt\to\wp(Base_\kappa)\\ z\le \boldsymbol{f}_\kappa(e_2)(z)}} z = \mathrm{gfp}(\boldsymbol{f}_\kappa(e_2)),$$

where relation $\le$ in the middle holds because every z that occurs as an argument of the join in the left occurs also as an argument of the join in the right. Consequently, $s_\kappa(S)(e_1) \le s_\kappa(S)(e_2)$.
(ii) Let $(e_i : i \in I)$ be a non-empty chain of environments. By Lemma 16, we get

$$s_\kappa(S)\left(\bigwedge_{i\in I} e_i\right) = \mathrm{gfp}\left(\boldsymbol{f}_\kappa\left(\bigwedge_{i\in I} e_i\right)\right)(S) = \mathrm{gfp}\left(\bigwedge_{i\in I} \boldsymbol{f}_\kappa(e_i)\right)(S).$$

Hence we would be done if gfp preserved glbs of non-empty chains. But here, gfp can be expanded by Kleene's theorem and it is well known that such fixpoint operator preserves glbs of non-empty chains of cocontinuous functions (see, for example, Chapter 8 of Winskel [15]; it is formulated dually for lubs and continuous functions there). It remains to use Theorem 13. □

Lemma 18. *Let κ be a kind of semantics matching the schemata in Figs. 4 and 8, and let $D \in Decl$.*
(i) *Then $d_\kappa(D)$ preserves glbs of finite non-empty chains.*
(ii) *If $\kappa \in \left\{\overrightarrow{+}, \widetilde{+}, \widehat{+}, \overrightarrow{\alpha}, \widetilde{\alpha}, \widehat{\alpha}\right\}$ then $d_\kappa(D)$ preserves glbs of all non-empty chains.*

Proof. The environment argument occurs in one place in the definition of d_κ — in $s_\kappa(S)(e)$. By Lemma 17, one can bring glb out from this. By Lemma 8, axm_κ and rul_κ are monotone. Therefore, the finite glb (for part (i)) can be brought out of all operators axm_κ and rul_κ. For part (ii), the same can be done by Lemma 9. As set meet with a constant is also cocontinuous, the glb can be moved to the front of the long expression and we have done. □

Lemma 19. *Let κ be a kind of semantics matching the schemata in Figs. 4, 8 and 9, and let $e \in Env_\kappa$.*
(i) *Then $\varphi^{\mathrm{m}}_\kappa(e)$ preserves glbs of finite non-empty chains.*
(ii) *If $\kappa \in \left\{\overrightarrow{+}, \widetilde{+}, \widehat{+}, \overrightarrow{\alpha}, \widetilde{\alpha}, \widehat{\alpha}\right\}$ then $\varphi^{\mathrm{m}}_\kappa(e)$ preserves glbs of all non-empty chains.*

Proof. Let $(e'_i, \mathcal{D})$, $i \in I$, be the elements of our chain. We write

$$\begin{aligned}\varphi^{\mathrm{m}}_\kappa(e)\left(\bigwedge_{i\in I}(e'_i,\mathcal{D})\right) &= \varphi^{\mathrm{m}}_\kappa(e)\left(\bigwedge_{i\in I}e'_i,\mathcal{D}\right) = e\left[d_\kappa(D)\left(\bigwedge_{i\in I}e'_i\right) : D\in\mathcal{D}\right]\\ &= \bigwedge_{i\in I}\left(e\left[d_\kappa(D)(e'_i) : D\in\mathcal{D}\right]\right) = \bigwedge_{i\in I}\varphi^{\mathrm{m}}_\kappa(e)(e'_i,\mathcal{D}),\end{aligned}$$

where only the second last equality is unclear. To prove it, apply both sides to some arguments P and v. If P is declared by no declaration in $\mathcal{D}$ then

$$e\left[d_\kappa(D)\left(\bigwedge_{i\in I}e'_i\right) : D\in\mathcal{D}\right](P)(v) = e(P)(v) = \bigcap_{i\in I}e(P)(v) = \bigcap_{i\in I}e\left[d_\kappa(D)(e'_i) : D\in\mathcal{D}\right](P)(v).$$

Consider the case where P is declared by declaration $D \in \mathcal{D}$. Interpreting the pairs $d_\kappa(D)(e')$ as partial mappings that are defined on P only and using Lemma 18, we obtain

$$\begin{aligned}e\left[d_\kappa(D)\left(\bigwedge_{i\in I}e'_i\right) : D\in\mathcal{D}\right](P)(v) &= d_\kappa(D)\left(\bigwedge_{i\in I}e'_i\right)(P)(v)\\ &= \left(\bigwedge_{i\in I}d_\kappa(D)(e'_i)\right)(P)(v)\\ &= \bigcap_{i\in I}d_\kappa(D)(e'_i)(P)(v)\\ &= \bigcap_{i\in I}e\left[d_\kappa(D)(e'_i) : D\in\mathcal{D}\right](P)(v).\ \square\end{aligned}$$

Lemma 20. *The part of functor $\mathcal{C}^{\mathrm{m}}$ that works on functions preserves lubs and glbs of non-empty families.*

Proof. Standard. □

Theorem 21. *Let κ be a kind of semantics matching the schemata in Figs. 4, 8 and 9, and let $e \in Env_\kappa$.*
(i) *Then $\boldsymbol{f}^{\mathrm{m}}_\kappa(e)$ preserves glbs of finite non-empty chains.*
(ii) *If $\kappa \in \left\{\overrightarrow{+}, \widetilde{+}, \widehat{+}, \overrightarrow{\alpha}, \widetilde{\alpha}, \widehat{\alpha}\right\}$ then $\boldsymbol{f}^{\mathrm{m}}_\kappa(e)$ preserves glbs of all non-empty chains.*

Proof. Similar to the proof of Theorem 13. □

5.3. Example

Cocontinuity of semantics means that it can be obtained as a limit of a step-by-step approximation process that is not transfinite. In order to illustrate this process, we show how the transfinite tree semantics of the example program $P =$ (**while** `true` **do** `x := x + 1`) ; `x := 1` which was observed in Section 1 takes shape. Similarly to that example, assume that states contain only variable `x` (this means, for instance, that $s[\mathtt{x} \mapsto v] = \{\mathtt{x} \mapsto v\}$ for any state s and value v).

Denote the sequence of approximations by $s_0, s_1, \ldots$. The process starts from $\top$; this means that $s_0(S)$ contains all trees in $Base_{\widehat{\alpha}}$, irrespectively of S.

As $s_{i+1} = \varphi_{\widehat{\alpha}}(e)(s_i)$ for each i (the environment e does not matter), we have

$$s_{i+1}(\texttt{x := 1}) = \varphi_{\widehat{\alpha}}(e)(\texttt{x}, 1) = \mathrm{axm}_{\widehat{\alpha}}\,\{(s, \{\texttt{x} \mapsto 1\}) : s \in State\} = \{\mathrm{elem}_{\widehat{\alpha}}(s, \{\texttt{x} \mapsto 1\}) : s \in State\},$$

i.e., starting from s_1, the semantics of `x := 1` contains precisely the one-vertex trees $\overline{s \to \{\texttt{x} \mapsto 1\}}$. Analogously, the semantics of `x := x + 1` for each s_{i+1} consists of trees $\overline{s \to \{\texttt{x} \mapsto s(\texttt{x}) + 1\}}$, i.e., the one-vertex trees where the value of `x` in the final state is one larger than that in the initial state.

Denote $W =$ **while** `true` **do** `x := x + 1`. Now

$$s_1(W \texttt{ ; x := 1}) = \varphi_{\widehat{\alpha}}(e)(s_0(W), s_0(\texttt{x := 1})) = \varphi_{\widehat{\alpha}}(e)(\top, \top) = \mathrm{rul}_{\widehat{\alpha}}(\top, \top)$$

consists of trees of the form $\dfrac{u \qquad v}{\mathrm{ini}_{\widehat{\alpha}} u \to \mathrm{fin}_{\widehat{\alpha}} v}$ such that $\mathrm{fin}_{\widehat{\alpha}} u = \mathrm{ini}_{\widehat{\alpha}} v$ (by condition $\mathrm{sound}_{\widehat{\alpha}}(u, v)$) and

$$s_{i+2}(W \texttt{ ; x := 1}) = \mathrm{rul}_{\widehat{\alpha}}(s_{i+1}(W), s_{i+1}(\texttt{x := 1}))$$

consists of trees of the form $\dfrac{u \qquad \overline{s' \to \{\texttt{x} \mapsto 1\}}}{s \to \{\texttt{x} \mapsto 1\}}$ where $u \in s_{i+1}(W)$ such that $\mathrm{ini}_{\widehat{\alpha}} = s$ and $\mathrm{fin}_{\widehat{\alpha}} u = s'$. The limit semantics $s_{\widehat{\alpha}}(W \texttt{ ; x := 1})$ therefore contains trees of the same form for which u belongs to all $s_{i+1}(W)$.

It remains to study the development of semantics of W. First,

$$\begin{aligned} s_1(W) &= \varphi_{\widehat{\alpha}}(e)(\texttt{true}, s_0(\texttt{x := x + 1 ; } W)) = \varphi_{\widehat{\alpha}}(e)(\texttt{true}, \top) \\ &= \mathrm{rul}_{\widehat{\alpha}}(\mathrm{iftrue}_{\widehat{\alpha}}(\texttt{true}), \top) \cup \mathrm{rul}_{\widehat{\alpha}}(\mathrm{iffalse}_{\widehat{\alpha}}(\texttt{true})) \\ &= \mathrm{rul}_{\widehat{\alpha}}(\mathrm{axm}_{\widehat{\alpha}}\,\{(s,s) : s \in State\}, \top) \cup \mathrm{rul}_{\widehat{\alpha}}(\mathrm{axm}_{\widehat{\alpha}}\varnothing) \\ &= \mathrm{rul}_{\widehat{\alpha}}(\{\overline{s \to s} : s \in State\}, \top), \end{aligned}$$

i.e., $s_1(W)$ consists of trees having the form $\dfrac{\overline{s \to s} \qquad u}{s \to s'}$ where $\mathrm{ini}_{\widehat{\alpha}} u = s$ and $\mathrm{fin}_{\widehat{\alpha}} u = s'$. Analogously to the analysis of $s_1(W \texttt{ ; x := 1})$, we can see that $s_1(\texttt{x := x + 1 ; } W) = \mathrm{rul}_{\widehat{\alpha}}(\top, \top)$, therefore

$$s_2(W) = \mathrm{rul}_{\widehat{\alpha}}(\{\overline{s \to s} : s \in State\}, s_1(\texttt{x := x + 1 ; } W))$$

consists of trees of the form

$$\dfrac{\dfrac{}{\overline{s \to s}} \qquad \dfrac{u \qquad v}{s \to s'}}{s \to s'},$$

where $\mathrm{ini}_{\widehat{\alpha}} u = s$, $\mathrm{fin}_{\widehat{\alpha}} u = \mathrm{ini}_{\widehat{\alpha}} v$ and $\mathrm{fin}_{\widehat{\alpha}} v = s'$. Continuing this way, each new level refines the nature of the trees belonging to the semantics. Fixing the initial state $s = \{\texttt{x} \mapsto 0\}$, the limit semantics contains the tree presented in the example in Section 1 but also other trees obtained from that one by replacing $\top$ with whatever constant.

In transfinite fractional trace semantics, the process and the result are similar but the transfinite nature of the trace is more evident. The infinite branch of the process we described for tree semantics goes within $\left[0; \dfrac{1}{2}\right]$ but there are also points $\dfrac{1}{2}$ and 1 involved already during the first steps.

Contrastingly, the result of the process for transfinite ordinal trace semantics is different. Suppose $\propto = \omega^\omega$ (the limit length of trace). Firstly, $s_0(W) = \top$ containing all traces of length less than $\propto$. Next we obtain $s_1(W) = \mathrm{rul}_{\overrightarrow{\alpha}}(\{(s,s) : s \in State\}, \top)$ containing all traces starting with a double state. Since $s_1(\texttt{x := x + 1 ; } W) = \mathrm{rul}_{\overrightarrow{\alpha}}(\top, \top) = \top$, we have $s_2(W) = s_1(W)$. Furthermore,

$$s_2(\texttt{x := x + 1 ; } W) = \mathrm{rul}_{\overrightarrow{\alpha}}(\{(s, \{\texttt{x} \mapsto s(\texttt{x}) + 1\}) : s \in State\}, s_1(W))$$

and thus $s_3(W) = \mathrm{rul}_{\overrightarrow{\alpha}}(\{(s,s) : s \in State\}, s_2(\texttt{x := x + 1 ; } W))$ contains all traces starting with double s followed by double $\{\texttt{x} \mapsto s(\texttt{x}) + 1\}$. This way, we see that the limit semantics of W contains all infinite traces whose first ω states are s, s, $\{\texttt{x} \mapsto s(\texttt{x}) + 1\}$, $\{\texttt{x} \mapsto s(\texttt{x}) + 1\}$, $\{\texttt{x} \mapsto s(\texttt{x}) + 2\}$, $\{\texttt{x} \mapsto s(\texttt{x}) + 2\}$, etc. As $\propto > \omega$, there is an undetermined computation

following the first ω steps (not only one undetermined state as in the tree and fractional trace semantics). For program W ; x := 1, the value of x can change to 1 after whatever amount of this garbage.

As the "demonic behaviour" occurs only after the first ω steps, the standard ordinal trace semantics is free of this and coincides with the usual agreement of what a program really does.

6. Issues of determinism

Determinism of semantics can be broken in two ways. Firstly, a program can have several possible behaviours in the same initial circumstances. Secondly, a program can have no behaviours at all.

Non-determinism of the first kind can be acceptable. Concerning the application in program slicing discussed in Section 1, multiplicity of behaviours does not exclude a semantics from being a reasonable setting provided non-determinism does not add unexpected data dependences.

Moreover, we can make the statement semantics deterministic by redefining $Base_\kappa$ in such a way that it contains precisely the semantic objects that reflect runs that meet the following two conditions:

1. If a variable has one value at one stage of execution and another value in some later moment, then between these two observations, there exist two consecutive on timeline states where the value of this variable is different.
2. Whenever a state directly follows an endless sequence of states where the value of a variable does not stabilize, the value of the variable in this state is $\top$.

In other words, the conditions say that the value of any variable after an endless computation is determined by the behaviour of the value of that variable during the computation, whereby it is $\top$ for non-stabilizing value sequences and otherwise equals the value to which the sequence stabilizes. With this, we make the nature of transfinite traces similar to those used in the work of Giacobazzi and Mastroeni [4].

As argued in [9], however, this restriction is not satisfying in program slicing. Instead, the sequences whose stabilization matters must be composed from the values observed at the head program point of the infinitely working loop rather than during the whole run. This change is not so easy to do in our semantics since one cannot detect the program points where the states occurring on the trace are observed. For that, one must add program points to semantic objects. For example, the trace semantics would be much more like structural operational semantics of [12], containing "rest of code" components which show the current program point. In principle, this can be done along the lines of our work [10].

It is not clear how to interpret the second kind of non-determinism. This is likely to be undesirable since it would mean that also the first ω steps of execution that are really performed remain outside the semantics.

Currently, we have no proof that our transfinite fractional and tree semantics of all programs provide at least one behaviour for every initial state.

7. Further work

The problems related to determinism (discussed in Section 6) are worth further investigation. Discovering conditions under which the transfinite fractional and tree semantics of modules, equipped with restrictions that enable them to use as the setting of program slicing, are deterministic would be interesting. Of course, it is important to study whether the semantics bring along programs with missing behaviour.

Another possible piece of further work would be investigating whether the hierarchy of semantics considered in this paper can be built up as a Cousot hierarchy (discussed in Section 3.1). Transfinite semantics in the Cousot hierarchy has already been a subject of work of Giacobazzi and Mastroeni [4]; but the way they built up their semantics is very different from that of us and, moreover, their work did not involve tree semantics or fractional semantics.

Acknowledgements

The work was partially supported by the Estonian Science Foundation under grant nos. 6713 and 7543.

The author thanks the anonymous reviewers who made a lot of valuable suggestions and comments.

References

[1] D.W. Binkley, K.B. Gallagher, Program slicing, Advances in Computers, 43 (1996) 1–50.
[2] P. Cousot, Constructive design of a hierarchy of semantics of a transition system by abstract interpretation, Electronic Notes in Theoretical Computer Science 6 (1997) 25.
[3] P. Cousot, Constructive design of a hierarchy of semantics of a transition system by abstract interpretation, Theoretical Computer Science 277 (2002) 47–103.
[4] R. Giacobazzi, I. Mastroeni, Non-standard semantics for program slicing, Higher-Order Symbolic Computation 16 (2003) 297–339.
[5] S. Glesner, A proof calculus for natural semantics based on greatest fixed point semantics, Electronic Notes in Theoretical Computer Science 132 (2005) 75–93.
[6] R. Kennaway, J.W. Klop, R. Sleep, F.-J. de Vries, Transfinite reductions in orthogonal term rewriting systems, Information and Computation 119 (1) (1995) 18–38.
[7] X. Leroy, Coinductive big-step operational semantics, In: P. Sestoft (Ed.), Proceedings of ESOP 2006, Lecture Notes in Computer Science, vol. 3924, 2006, pp. 54–68.

[8] X. Leroy, H. Grall, Coinductive big-step operational semantics, Information and Computation 207 (2) (2009) 284–304.
[9] H. Nestra, Transfinite semantics in program slicing, Proceedings of the Estonian Academy of Sciences: Engineering 11 (4) (2005) 313–328.
[10] H. Nestra, Fractional semantics, in: M. Johnson, V. Vene, (Eds.), Proceedings of AMAST 2006, Lecture Notes in Computer Science, vol. 4019, 2006, pp. 278–292.
[11] H. Nestra, Iteratively defined transfinite trace semantics and program slicing with respect to them, Ph.D. Thesis, University of Tartu, 2006, 119 pp.
[12] F. Nielson, H.R. Nielson, Semantics with Applications: An Appetizer, Springer, 2007.
[13] F. Tip, A survey of program slicing techniques, Journal of Programming Languages 3 (3) (1995) 121–181.
[14] M. Weiser, Programmers use slices when debugging, Communications of the ACM 25 (7) (1982) 446–452.
[15] G. Winskel, The Formal Semantics of Programming Languages, The MIT Press, 1993.