Compositional Transfinite Semantics of While

Härmel Nestra

Institute of Computer Science

University of Tartu

e-mail: harmel.nestra@ut.ee

Motivation: Semantic Anomaly of Program Slicing

Program slicing

Program slicing is program transformation where parts of program are left out so that the computation of the interesting variables at interesting program points would not be affected.

- Applications in debugging and elsewhere in software engineering.

Example 1

Slicing w.r.t. variable `sum` at the end point:

```
n := input() ;                    n := input() ;
i := 0 ;                          i := 0 ;
sum := 0 ;                        sum := 0 ;
prod := 1 ;
while i < n do                    while i < n do
(                          →      (
  i := i + 1 ;                      i := i + 1 ;
  sum := sum + i ;                  sum := sum + i ;
  prod := prod * i
)                                 )
```

Algorithms

Classic algorithms for program slicing are based on control flow and data flow analysis.

- Relevant Sets (backward).
- Reaching Definitions (forward).

Example 2

Irrelevant loops can be sliced away:

```
n := input() ;                          n := input() ;
i := 0 ; sum := 0 ;                     i := 0 ; sum := 0 ;
while i < n do                          while i < n do
(                                       (
  i := i + 1 ;                            i := i + 1 ;
  sum := sum + i ;                        sum := sum + i ;
)                                  →    )
i := 0 ; prod := 1 ;
while not (i == n) do
(
  i := i + 1 ;
  prod := prod * i
)
```

Semantic anomaly

Problem: the sequence of values observed at some program point depends on the termination status of the loops.

- Undecidable.

Solutions: Transfinite Semantics vs Trajectories

Subject to change: Algorithms, definition, semantics

- Changing algorithms may keep slices too big and is unnecessary in practice.
- Changing the definition tend to allow too many subsets as slices.
- Change the semantics!

Transfinite semantics

In **transfinite semantics**, execution of programs can continue after infinite loops from some limit states.

- Loop bodies are run at most ω times during each execution of the loop.
- Semantic anomaly vanishes.
- Problem: How to define limit states?

Attempts

- Giacobazzi and Mastroeni 2003.
- Nestra 2004–2006: Without assuming structured control flow.
- Nestra 2007–2009: In the greatest fixpoint form.

Lack of natural properties such as compositionality.

Compositionality, substitutivity

In compositional semantics, the meaning of composed statements is expressed in terms of the semantics of their immediate constituents solely.

- Implies substitutivity: any substatement may be replaced with a semantically equivalent statement without changing the meaning of the whole statement.

Trajectory semantics

In trajectory semantics, the number of times a loop body is run during one execution of the loop is limited by a natural number, given as a parameter.

- Proposed by Danicic et al. (2010) for addressing semantic anomaly.
- Semantic anomaly vanishes since no loop is infinite.

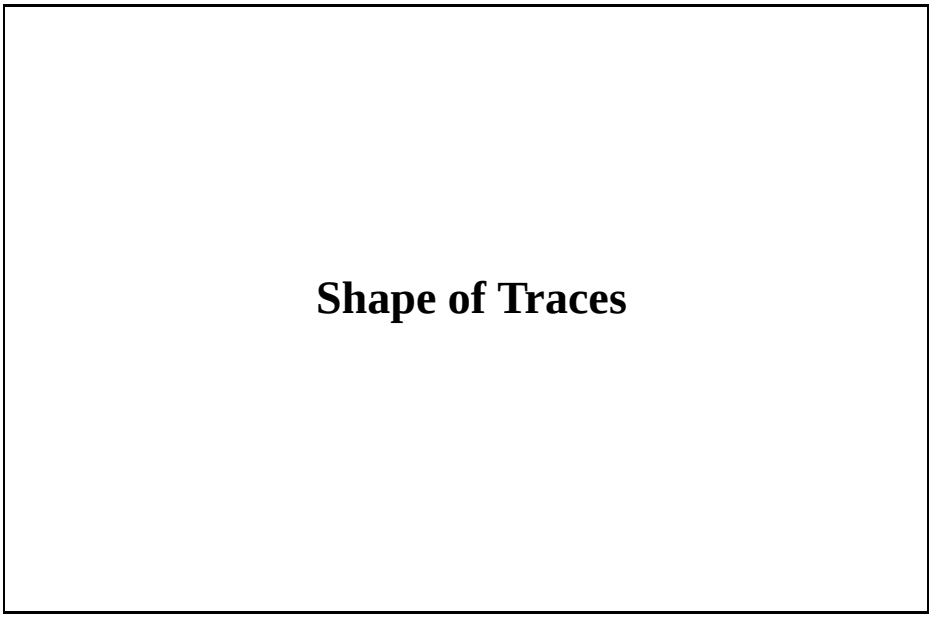
Relationship with standard semantics

- From transfinite semantics of a program, standard semantics can in principle be deduced directly by truncating the transfinite part.
- From trajectory semantics of an infinite loop, standard semantics must be collected from infinitely many finite beginnings.

Our Contribution

General characterization

- Compositional transfinite semantics w.r.t. which classic slicing algorithms are correct.
- Relationships with standard semantics and transfinite semantics in the form of greatest fixpoint.


Shape of Traces

Ordinal semantics

Intermediate states on execution traces are indexed by ordinal numbers $(0, 1, 2, \ldots, \omega, \omega + 1, \omega + 2, \ldots\ldots)$.

- The desired transfinite semantics cannot be represented in the form of least or greatest fixpoint that is standard in this area.
  - Greatest fixpoint would involve traces that include garbage after infinite loops. (Explained in our previous work (2007–2009).)
- Loop semantics can still be expressed naively via finite and infinite iterations.

Fractional semantics

Intermediate states on execution traces are indexed by rational numbers between 0 and 1.

- Traces develop into depth rather than into length.
- Each part of computation has its own interval of indices statically associated to it. (No space is left for garbage.)
- Introduced for expressing transfinite semantics in the standard fixpoint form.
- Studied by us previously (2006, 2007–2009).

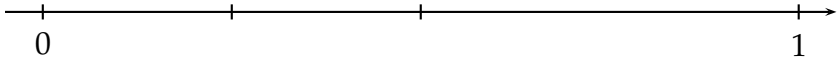
Fractional semantics: Example 1

Fractional trace of program

```
(z := x ; x := y) ; y := z
```

in the initial state $\left\{ \begin{array}{l} x \mapsto 1 \\ y \mapsto 2 \\ z \mapsto 0 \end{array} \right\}$ is

$$\left\{ \begin{array}{l} x \mapsto 1 \\ y \mapsto 2 \\ z \mapsto 0 \end{array} \right\} \quad \left\{ \begin{array}{l} x \mapsto 1 \\ y \mapsto 2 \\ z \mapsto 1 \end{array} \right\} \quad \left\{ \begin{array}{l} x \mapsto 2 \\ y \mapsto 2 \\ z \mapsto 1 \end{array} \right\} \quad \left\{ \begin{array}{l} x \mapsto 2 \\ y \mapsto 1 \\ z \mapsto 1 \end{array} \right\}$$



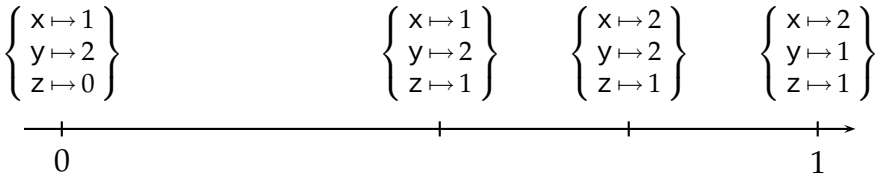
0 1

Fractional semantics: Example 2

Fractional trace of program

$$\texttt{z := x ; (x := y ; y := z)}$$

in the initial state $\left\{ \begin{array}{l} \mathsf{x} \mapsto 1 \\ \mathsf{y} \mapsto 2 \\ \mathsf{z} \mapsto 0 \end{array} \right\}$ is



Limit States

Limit state restriction

Limit state t where the computation falls after infinite computation $(s_i : i \in \mathbb{N})$ must satisfy the following:

Let $s_{k_1}, s_{k_2}, \ldots$ be all states observed while passing through the loop condition test point. Then

$$\lim(s_{k_i} : i \in \mathbb{N}) \dot{\sqsubseteq} t$$

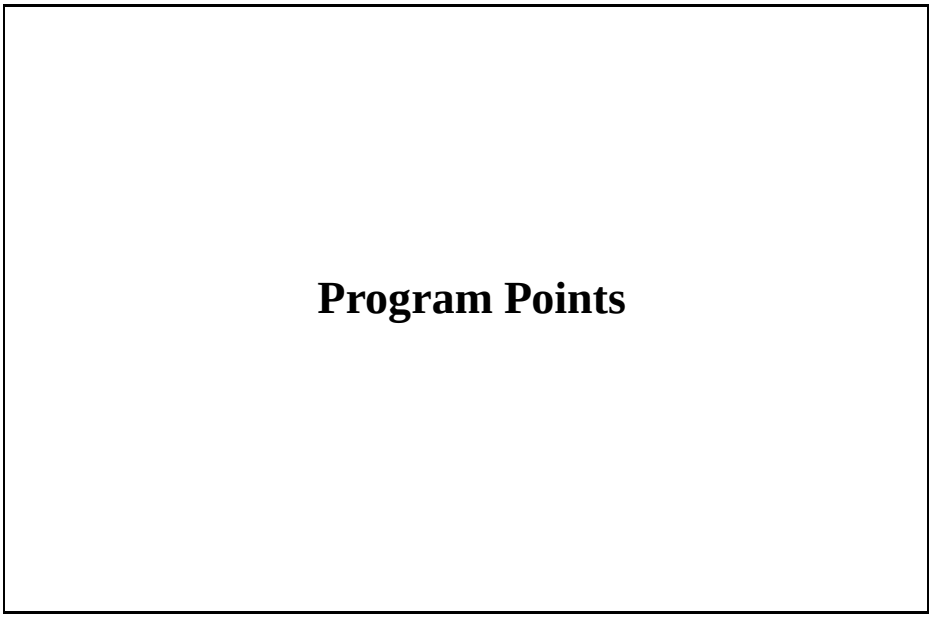
where:

- $\sqsubseteq$ is flat order on values, $\dot{\sqsubseteq}$ is obtained by pointwise lifting, and
- $\lim(v_i : i \in \mathbb{N}) = \left\{ \begin{array}{ll} u & \text{if } \exists n \in \mathbb{N}\, \forall i \geq n\, (v_i = u) \\ \bot & \text{otherwise} \end{array} \right\}$.

Recognition of states that influence the limit

How to recognize states observed at the loop condition test point?

- In the iteration form, we just take the first state of each iteration. (Fractional shape of traces not needed.)
- In the fixpoint form (fractional shape assumed), we may take states that are observed at indices $1 - \frac{1}{2^{2i}}$ for $i \in \mathbb{N}$.
- Otherwise, program points must be traced in semantics (makes description of semantics more complicated).


Program Points

Correpondence of program points and slice points

How can the correspondence be established and traced?

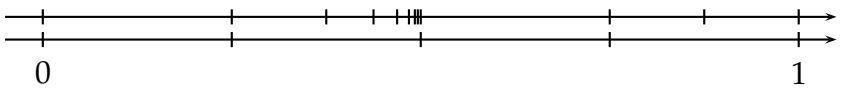
- In ordinal semantics: Program points must be traced explicitly;
- In fractional semantics: Trivial injection of indices does the job!
  - Assumes that statements are replaced with **`skip`** rather than removed. (Standard alternative.)

Correspondence of program points: Example

Slicing w.r.t. the value of `x` after the last assignment to `x`:

```
(                                    (
  y := 1 ;                             y := 1 ;
  while true do x := x + 1             skip
) ;                                  ) ;
(                                  → (
  x := y ;                             x := y ;
  (y := 2 ; z := 3)                    skip
)                                    )
```

The index sets for these programs are the following:



Technical Part

Types

Val	set of all values
$State = Var \to Val$	set of variable evaluations
$Conf = Stmt \times State$	set of configurations
$Elem_\kappa \supseteq State$	set of elements of semantic objects
$Base_\kappa$	set of all semantic objects (traces)
$Sem_\kappa = \wp(Base_\kappa)$	set of all meanings of programs
$s_\kappa \in Stmt \to Sem_\kappa$	semantics of statements

Structure of semantics

$$
\begin{aligned}
&\mathbf{s}_\kappa(\texttt{skip}) \\
&\qquad = \mathrm{axm}^\sharp_\kappa\,\{s \to s : s \in \mathit{State}\} \\
&\mathbf{s}_\kappa(X \ \texttt{:=}\ E) \\
&\qquad = \mathrm{axm}^\sharp_\kappa\,\{s \to s[X \mapsto \mathbf{e}(E)(s)] : s \in \mathit{State}\} \\
&\mathbf{s}_\kappa(T_1 \ \texttt{;}\ T_2) \\
&\qquad = \mathrm{rul}^\sharp_\kappa(\mathbf{s}_\kappa(T_1) \times \mathbf{s}_\kappa(T_2)) \\
&\mathbf{s}_\kappa(\texttt{if}\ E\ \texttt{then}\ T_1\ \texttt{else}\ T_2) \\
&\qquad = \mathrm{rul}^\sharp_\kappa(\mathrm{iftrue}_\kappa(E) \times \mathbf{s}_\kappa(T_1)) \cup \mathrm{rul}^\sharp_\kappa(\mathrm{iffalse}_\kappa(E) \times \mathbf{s}_\kappa(T_2)) \\
&\mathbf{s}_\kappa(\texttt{while}\ E\ \texttt{do}\ T) \\
&\qquad = \bigcup_{o \in \mathbb{O}^\omega} \mathrm{iter}_\kappa(E, \mathbf{s}_\kappa(T))(o)
\end{aligned}
$$

Conclusion

Conclusion about transfinite semantics

It is too early to write off transfinite semantics!

- It is closerly related to standard semantics than other approaches proposed for program slicing theory.

Conclusion about fractional semantics

Fractional semantics is useful!

- Enables transfinite semantics in the form of greatest fixpoint.
- Without having to trace program points, enables recognition of significant states on loop execution traces.
- Without having to trace program points, provides means for rigorously dealing with program point correspondence in the original program and its slice.
- In comparison with tree semantics:
  - Fractional semantics shows the order of steps explicitly;
  - Fractional semantics simplifies some proofs.