Fractional Semantics

Härmel Nestra

Institute of Computer Science

University of Tartu

e-mail: harmel.nestra@ut.ee

Outline

- Kinds of semantics generally
- Motivation of using non-standard semantics: program slicing
- Old solution: transfinite semantics
- New solution: fractional semantics
- Conclusion

Semantics generally

Essence

Semantics of a programming language gives a meaning to every correct syntactic object of the language.

Trace semantics

Trace semantics is a semantics in which the meaning of a program/statement/declaration is an execution trace depending on the values of some initial parameters.

- Sets of possible execution traces in the case of non-deterministic semantics.

Standard semantics

Standard semantics is a semantics in which the meaning of syntactic objects closely reflects the corresponding real world processes desirably occurring in the implementations of the language.

- The level of abstraction may vary.
- In standard trace semantics, components of traces are normally indexed with natural numbers.

Strict vs lazy semantics

- In **strict semantics**, executing an erroneous part of code leads to the same error as when executing this part separately.
- **Lazy semantics** has facilities to overcome parts of code producing an error when executed separately.

Program slicing

An example

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

$\longrightarrow$

```
sum := 0;

i := 0;
while i < n do (
  i := i + 1;
  sum := sum + i;

);
```

Criterion: $\{(\varepsilon, \texttt{sum})\}$, where ε denotes the end point.

An attempt to define

Let P be a program and $\mathcal{X}$ the set of all variables in it.

A **slicing criterion** is a set $C \subseteq V(\mathrm{cfg}\, P) \times \mathcal{X}$.

A **slice of P w.r.t. C** is any program Q where

- all statements occur also in P in the same order,
- for every $(p, Z) \in C$, program Q computes the same sequence of values as program P for program point p and variable Z.

Semantic anomaly

If the original program loops then we might have slices which execute the program points in the criterion more times than the original program.

For example, for criterion $\{(\varepsilon, \{\texttt{x}\})\}$ where ε is the end vertex, we get

```
a := 1;
x := 0
```

$\longrightarrow$

```
x := 0
```

```
while true do ;
x := 0
```

$\longrightarrow$

```
x := 0
```

So-called semantic anomaly.

Observations

- The definition refers to a trace semantics...
- Implicitly, the standard semantics was assumed...
- Using a lazier non-standard semantics would help!

Transfinite semantics

Essence

Transfinite semantics is a semantics according to which computation may continue after an infinite number of steps from some limit state determined somehow by the infinite computation performed.

Traces

In transfinite trace semantics, the execution traces are transfinite sequences.

- The states in the transfinite list are normally indexed with ordinal numbers.

$$s_0, s_1, s_2, \ldots\ldots s_\omega, s_{\omega+1}, \ldots s_{\omega\cdot 2}, s_{\omega\cdot 2+1}, \ldots\ldots s_{\omega^2}, s_{\omega^2+1}, \ldots$$

- The body of every loop is repeated at most ω times during one execution.

The old example

```
while true do ;
x := 0
```

$\longrightarrow$

```
x := 0
```

In transfinite semantics, the semantic anomaly does not arise.

Infinitely deep recursion

Consider declaration

```
proc p() is call p().
```

Where to jump from the infinite black hole?

Observations

- It would be natural to unload infinitely deep recursion level by level starting from infinity...
- Every part of a transfinite computation must have a first element...
- We need a more general kind of semantics!

Fractional semantics

Essence

Fractional semantics is a trace semantics where trace components correspond to rational numbers.

- Transfinite trace semantics can be expressed as fractional.
- Rational numbers enable also infinite decreasing sequences.
- Intervals of rationals are statically associated with code fragments.

Example: assignments and composition

The execution trace of

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

at state

$$(\texttt{x} \mapsto 1, \texttt{y} \mapsto 2, \texttt{z} \mapsto 0)$$

is

$0 \mapsto \langle (\texttt{x} \mapsto 1, \texttt{y} \mapsto 2, \texttt{z} \mapsto 0) \mid [\texttt{z := x} \,;\, (\texttt{x := y} \,;\, \texttt{y := z})] \rangle,$
$\frac{1}{2} \mapsto \langle (\texttt{x} \mapsto 1, \texttt{y} \mapsto 2, \texttt{z} \mapsto 1) \mid [\texttt{x := y} \,;\, \texttt{y := z}] \rangle,$
$\frac{3}{4} \mapsto \langle (\texttt{x} \mapsto 2, \texttt{y} \mapsto 2, \texttt{z} \mapsto 1) \mid [\texttt{y := z}] \rangle,$
$1 \mapsto \langle (\texttt{x} \mapsto 2, \texttt{y} \mapsto 1, \texttt{z} \mapsto 1) \mid [\varepsilon] \rangle.$

Example: loop

If the second assignment is replaced with

$$W = \texttt{while z > 0 do z := z - 1}$$

then, on the same initial state, the execution trace is as follows:

$0 \mapsto \langle (\texttt{x} \mapsto 1, \texttt{y} \mapsto 2, \texttt{z} \mapsto 0) \mid [\texttt{z := x} \,;\, (W \,;\, \texttt{y := z})] \rangle,$
$\frac{1}{2} \mapsto \langle (\texttt{x} \mapsto 1, \texttt{y} \mapsto 2, \texttt{z} \mapsto 1) \mid [W \,;\, \texttt{y := z}] \rangle,$
$\frac{5}{8} \mapsto \langle (\texttt{x} \mapsto 1, \texttt{y} \mapsto 2, \texttt{z} \mapsto 1) \mid [(\texttt{z := z - 1} \,;\, W) \,;\, \texttt{y := z}] \rangle,$
$\frac{11}{16} \mapsto \langle (\texttt{x} \mapsto 1, \texttt{y} \mapsto 2, \texttt{z} \mapsto 0) \mid [W \,;\, \texttt{y := z}] \rangle,$
$\frac{3}{4} \mapsto \langle (\texttt{x} \mapsto 1, \texttt{y} \mapsto 2, \texttt{z} \mapsto 0) \mid [\texttt{y := z}] \rangle,$
$1 \mapsto \langle (\texttt{x} \mapsto 1, \texttt{y} \mapsto 0, \texttt{z} \mapsto 0) \mid [\varepsilon] \rangle.$

Comments

- Placing of runs of other parts of the code on the trace remained unchanged.
- This would be the case even if the initial state was changed.
- Accommodating transfinite traces goes as easily.
  - Every countable set of ordinals can be order-preservingly mapped into any non-trivial interval of rationals.

Examples: infinite loops

- The components of the execution trace of

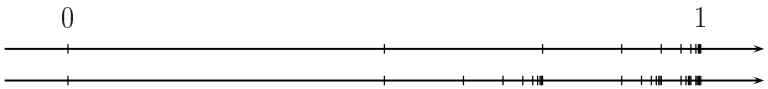
  while true **do** ε

  are numbered by ordinals $0, 1, 2, 3, \ldots$ and ω in transfinite semantics and by $0, \frac{1}{2}, \frac{3}{4}, \frac{7}{8}, \ldots$ and 1 in lazy fractional semantics.

- The components of double infinite loop

  while true **do while** true **do** ε

  are indexed by ordinals from 0 to ω^2 in transfinite semantics and as shown below in lazy fractional semantics.

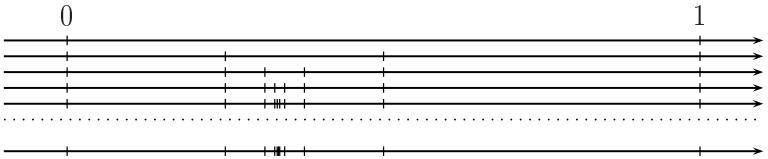


Example: simple infinite recursion

Define procedure p by declaration

proc p() **is call** p().

Its lazy fractional semantics results in the following domain construction process and final domain:



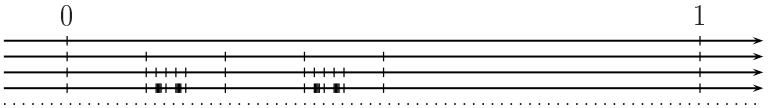
– Two infinite sequences — one ascending and another descending — are both converging to $\frac{1}{3}$.

Example: a more complicated recursion

Define procedure q by declaration

proc q() **is** (**call** q() ; **call** q())

Its lazy fractional semantics results in the following domain construction process:



Comments

- Each step adds twice more points than the previous since the number of calls doubles every level.
- The fixpoint domain forms a fractal structure.
  - A rational number between 0 and 1 belongs to it iff its octal representation is finite and each its digit after octal point is either 1 or 3 except for the last one which can be also 2 or 4.
  - The set of all possible limits of converging sequences of rationals in this set is uncountable.

Lazy fractional semantics

It is partially possible to compute the lazy fractional semantics in lazy functional language Haskell.

- The definition of the semantics in Haskell directly follows the mathematical specification.
  - No tricks!

Conclusion

Conclusion

- Fractional semantics...
  - provides a natural framework for accommodating transfinite executions;
  - removes the principal obstacle of handling recursion using transfinite semantics;
  - shows that transfinite computations in the case of while-loop are analogues to fractal computations in the case of recursion;
  - associates parts of code statically with index intervals.
- However, the work in the case of recursion is still in progress.
  - Currently, I do not know how to define lazy fractional semantics for some important cases.