COMS22202: 2015/16

Language Engineering

Dr Oliver Ray
(csxor@Bristol.ac.uk)

Department of Computer Science
University of Bristol

Tuesday 15th March, 2016

Fixpoints in Haskell

Haskell syntax allows non-terminating/undefined or “divergent” definitions:

- `x=x`
- `y=1/0`

Haskell semantics assigns such expressions a special “undefined” or bottom value (which is implicitly added as a least element of every type):

- `⊥`

A Haskell function is said to be strict if it is divergent whenever it is applied to a divergent argument :

- `(*2)` is strict as `(*2) ⊥ = ⊥*2 = ⊥`
- `(const 2)` is not `(const 2) ⊥ = 2`

So, strictly speaking, `⊥` is a fixpoint (and hence the least fixpoint) of any strict function!

Fixpoints in Haskell

To compute least fixpoints of functions we can use the following definition:

- `fix f = f (fix f)`

As can be seen by verifying that

- `fix (*2)` = `(*2)(fix (*2))` = `(*2)(*2)(fix (*2))` = ... = ⊥
- `fix (const 2) = (const 2)(fix (const 2)) = 2`

A semantically equivalent but procedurally more efficient definition (which allows circular data structures) is

- `fix f = let r = fix f in f r`

As can be seen by comparing execution times on

- `(fix (2:))!!(10^8) = ... = 2`

Which takes the one hundred millionth element of an infinite string of 2's. The first `fix` definition is linear time while the second is constant time?

Fixpoints in Haskell

We can often use fixpoints to define recursive functions (e.g. factorials):

- `fac = fix (\f n -> if n==0 then 1 else n * f (n-1))`

Equivalently (by simply making the anonymous functions explicit):

- `fac = fix fg`
- `fg f = g where g n = (if n==0 then 1 else n * f (n-1))`

`n.b. fg f n = (if n==0 then 1 else n * f (n-1))`

Where

- `f,g :: Int->Int`
- `fg  :: (Int->Int)->(Int->Int)`

As can be seen by verifying that

- `fac 0 = (fix fg) 0 = fg (fix fg) 0 = 1`
- `fac 1 = (fix fg) 1 = fg (fix fg) 1 = 1 * (fix fg) 0 = 1`
- `fac 2 = (fix fg) 2 = fg (fix fg) 2 = 2 * (fix fg) 1 = 2`
- `fac 3 = (fix fg) 3 = fg (fix fg) 3 = 3 * (fix fg) 2 = 6`
- …

Fixpoints in Haskell

Any fixpoint f of F must compute factorials as this would imply

- `f n = (if n == 0 then 1 else n * f (n-1))`

Equivalently F can be seen as a function that takes an approximant to the factorial function and returns a better approximation.

For example, we could take id as an initial approximation. This gets two outputs correct (1! and 2!) .

But (F id) is a better approximation as it gets three correct (0!, 2! and 3!).

Carrying on (F^n id) gets n+2 values correct.

The Kleene fixpoint theorem can be used to show that the least fixpoint is equivalent to ($F^\omega \perp$)

Further Reading

- https://www.reddit.com/r/haskell/comments/186bi1/introducing_fixedpoints_via_haskell/
- http://stackoverflow.com/questions/4787421/how-do-i-use-fix-and-how-does-it-work
- https://en.wikibooks.org/wiki/Haskell/Fix_and_recursion

Soundness and Completeness Results

We can prove that our axiomatic semantics in some sense “agrees with” our denotational semantics

Soundness: if we can derive {P} S {Q} then if P holds in some state s and it is the case that [[S]] s = s’ then Q holds in state s’

Completeness: vice versa

Soundness: p183

for all P,Q and S

if

$\vdash \{ P(x_1 \ldots x_n) \}\ S\ \{ Q(x_1 \ldots x_n) \}$

then

$\models \forall_{s,s' \in State} . P((s\ x_1)\ldots(s\ x_n)) \wedge \mathcal{S}_{ds}[\![S]\!] = s' \rightarrow Q((s'\ x_1)\ldots(s'\ x_n))$

where

$x_1 \ldots x_n$ are all the variables in P and Q

(Relative) Completeness: p190

for all P,Q and S

if

$\models \forall_{s,s' \in State} . P((s\ x_1)\ldots(s\ x_n)) \wedge \mathcal{S}_{ds}[\![S]\!]\ s = s' \rightarrow Q((s'\ x_1)\ldots(s'\ x_n))$

then

$\vdash \{ P(x_1 \ldots x_n) \}\ S\ \{ Q(x_1 \ldots x_n) \}$

where

$x_1 \ldots x_n$ are all the variables in P and Q

Note: the assertions {true} skip {Q} and {true} S {false} should be derivable iff Q is valid and S is non-halting; which are both semi-decidable properties

This means there is no algorithmic way of identifying all (non-)instances of the consequence rule (and the loop rule in the total semantics)

But it can be shown the axiomatic semantics is complete providing that *the assertion language is sufficiently expressive* (a.k.a. relative completeness)

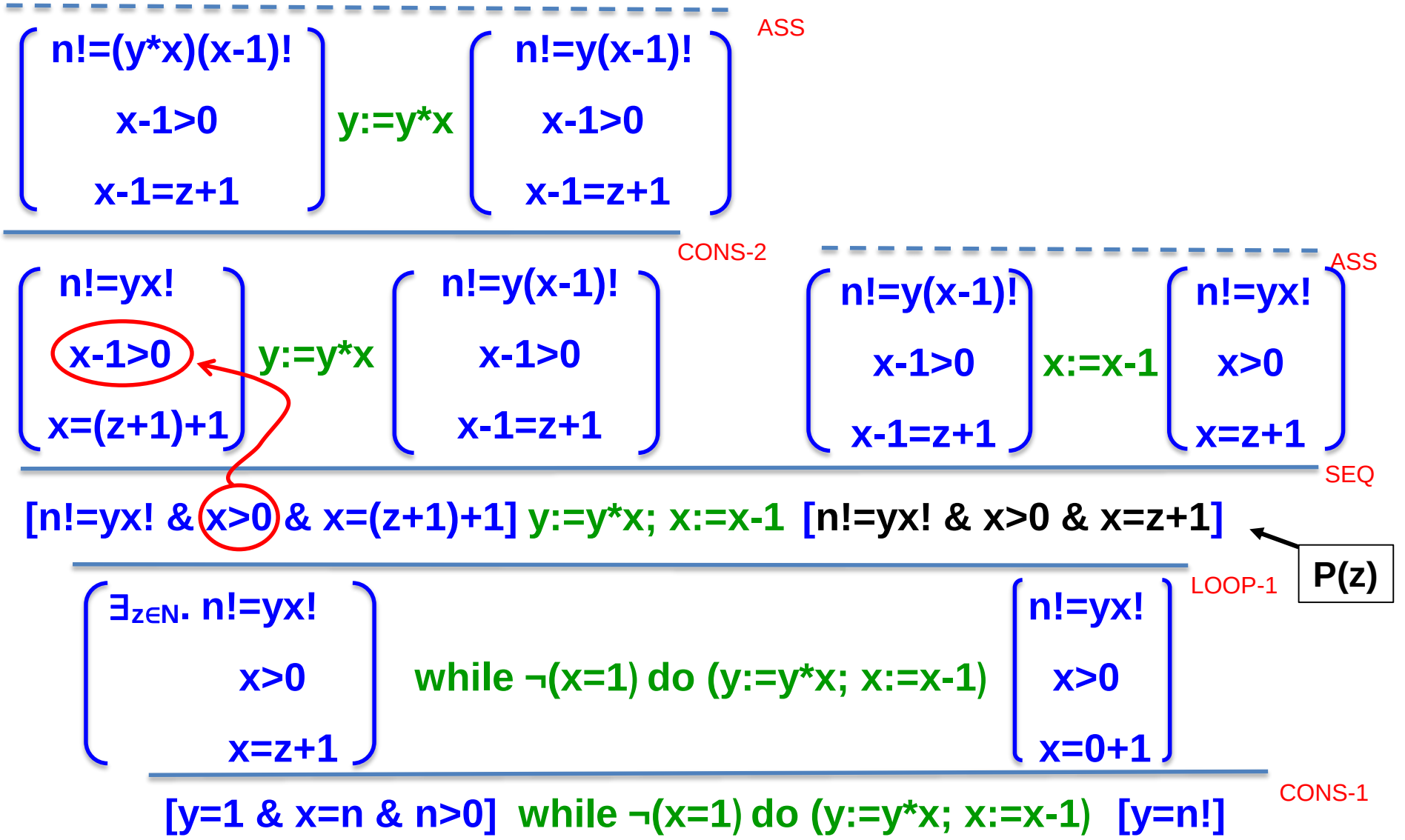
Another Result

we can also prove that if [P]S[Q] is derivable in the total semantics, then {P}S{Q} is derivable in the partial semantics (but the converse doesn't hold). This can be shown by induction on the shape of the inference trees:

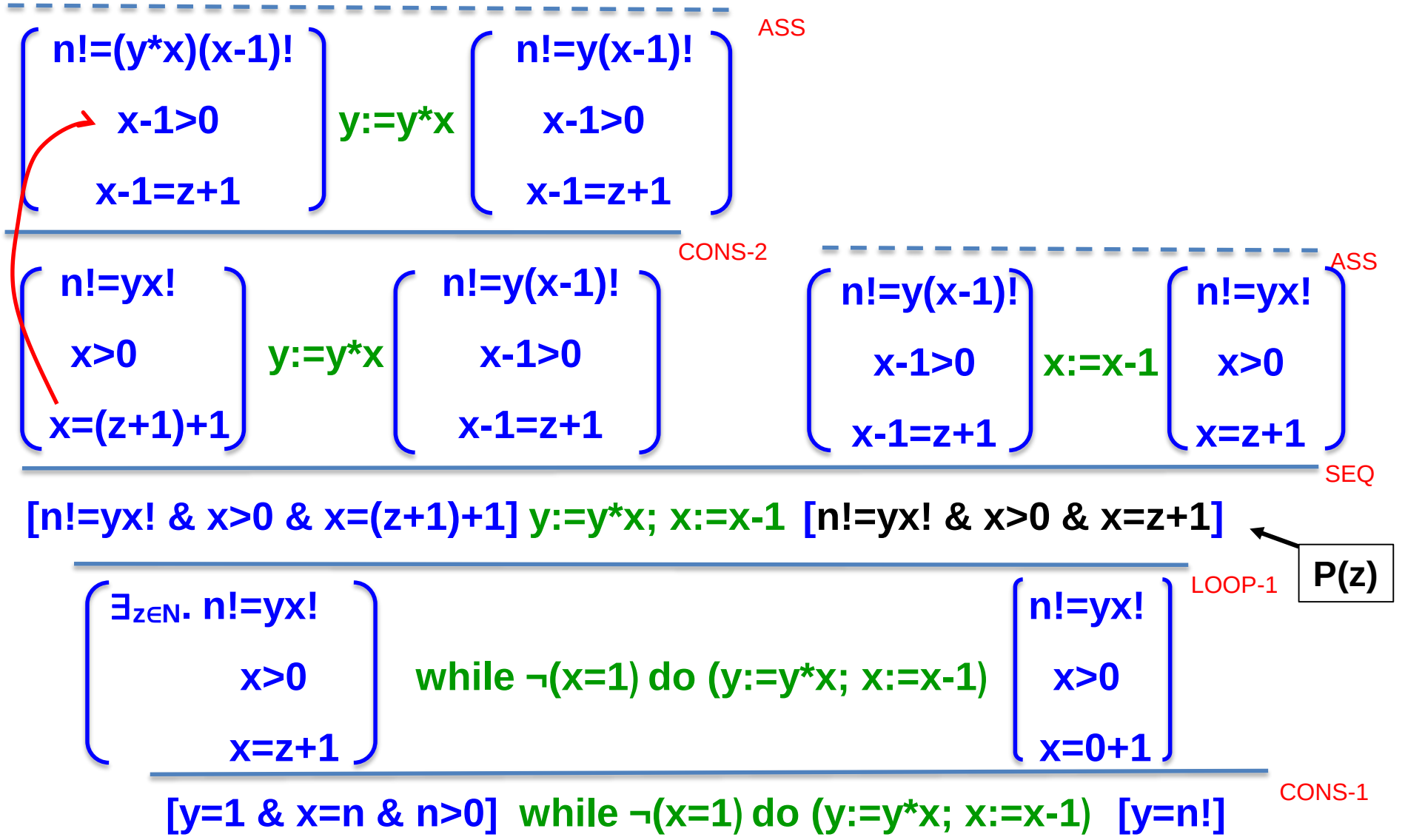
Base cases: the result holds if the derivation is an instance of the skip or ass axioms as these have the same pre- and post-conditions in both semantics.

Induction cases: the result holds for any instances if seq, cond and cons as these have the same pre- and post-conditions in both semantics. But the loop rule needs a bit of work. Suppose we find $[\exists z.P(z)]$ while b do S $[P(0)]$ on the basis that $[P(z+1)]$ S $[P(z)]$ and $\forall z \in N.P(z+1) \rightarrow b$ and $p(0) \rightarrow \neg b$. By the induction hypothesis we have $\{P(z+1)\}$ S $\{P(z)\}$. But if we set $P' = \exists z.P(z)$ then we can get $\{P' \wedge b\}$ S $\{P'\}$ from $\{P(z+1)\}$ S $\{P(z)\}$ using the consequence rule (i) as $P(z) \rightarrow P'$ trivially and (ii) $P' \wedge b \rightarrow P(z+1)$ trivially if $z>0$ and via a contradiction if $z=0$ - as P' would then entail $\neg b$. Now, from $\{P' \wedge b\}$ S $\{P'\}$ we can then get $\{P'\}$ S $\{P' \wedge \neg b\}$ or $\{\exists z.P(z)\}$ S $\{\exists z.P(z) \wedge \neg b\}$ by the loop rule. But now we can get $\{\exists z.P(z)\}$ S $\{P(0)\}$ by the consequence rule since $z=0$ is the only possibility that doesn't contradict the postcondition $\neg b$.

Error in last week's Slides



Corrected Proof (uses the fact z≥0)



Alternative Formulations of Loop Rule

What, if anything is wrong with the following loop rule definitions:

a)

$$\frac{[\ P(z+1)\ ]\ S\ [\ P(z)\ ]}{[\ \exists_{z\in N}.P(z)\ ]\ \text{while b do S}\ [\ P(0)\ ]} \quad \text{if} \quad \forall_{z\in N}.P(z+1) \rightarrow b \quad p(0) \rightarrow \neg b$$

b)

$$\frac{[\ P(z+1)\ \wedge b]\ S\ [\ P(z)\ ]}{[\ \exists_{z\in N}.P(z)\ ]\ \text{while b do S}\ [\ P(0)\ \wedge \neg b]}$$

c)

$$\frac{[\ P]\ \text{if b then (S; while b do S) else skip}\ [\ Q]}{[\ P]\ \text{while b do S}\ [\ Q]}$$

Alternative Formulations of Loop Rule

What, if anything is wrong with the following loop rule definitions:

a)

$$\frac{[\ P(z+1)\ ]\ S\ [\ P(z)\ ]}{[\ \exists_{z \in N}.P(z)\ ]\ \textbf{while b do S}\ [\ P(0)\ ]} \quad \textbf{if} \quad \forall_{z \in N}.P(z+1) \rightarrow b \quad p(0) \rightarrow \neg b$$

Nothing is wrong with this! This is definition given in the lecture slides. It can be shown that the axiomatic semantics which includes this rule is sound and complete with respect to the denotational semantics (Section 6.4 of the text book give some proof with respect to operational semantics, which is itself sound and complete with respect to the denotational semantics).

Alternative Formulations of Loop Rule

What, if anything is wrong with the following loop rule definitions:

b)

$$\frac{[\ P(z+1) \wedge b\ ]\ S\ [\ P(z)\ ]}{[\ \exists_{z \in N}.P(z)\ ]\ \textbf{while b do S}\ [\ P(0) \wedge \neg b\ ]}$$

Although this this rule looks a lot like the partial semantics loop rule, it makes the semantics unsound because we can use it to derive incorrect assertions such as the following

$$\frac{\dfrac{}{[\text{true}]\ \textbf{skip}\ [\ \text{true}\ ]}}{\dfrac{[\ \text{true} \wedge \text{true}\ ]\ \textbf{skip}\ [\ \text{true}\ ]}{[\ \exists_{z \in N}.\text{true}\ ]\ \textbf{while true do skip}\ [\ \text{true} \wedge \neg \text{true}\ ]}}$$

$$[\text{true}]\ \textbf{while true do skip}\ [\text{true}]$$

Alternative Formulations of Loop Rule

What, if anything is wrong with the following loop rule definitions:

c)

[P] if b then (S; while b do S) else skip [Q]
[P] while b do S [Q]

Unfortunately, using this as the loop rule would make the semantics incomplete because we cannot use it any proof since it would lead to an infinite regress.