

Fresh identifiers

Andrei Popescu Thomas Bauereiss

May 14, 2024

Abstract

This entry defines a type class with an operator returning a fresh identifier, given a set of already used identifiers and a preferred identifier. The entry provides a default instantiation for any infinite type, as well as executable instantiations for natural numbers and strings.

Contents

1	The type class <i>fresh</i>	1
2	Fresh identifier generation for natural numbers	2
3	Fresh identifier generation for strings	3
3.1	A partial order on strings	3
3.2	Incrementing a string	4
3.3	The fresh-identifier operator	5
3.4	Lifting to string literals	6
4	Fresh identifier generation for infinite types	8

1 The type class *fresh*

```
theory Fresh
  imports Main
begin
```

A type in this class comes with a mechanism to generate fresh items. The fresh operator takes a list of items to be avoided, xs , and a preferred element to be generated, x .

It is required that implementations of fresh for specific types produce x if possible (i.e., if not in xs).

While not required, it is also expected that, if x is not possible, then implementation produces an element that is as close to x as possible, given a notion of distance.

```

class fresh =
  fixes fresh :: 'a set  $\Rightarrow$  'a  $\Rightarrow$  'a
  assumes fresh-notIn:  $\bigwedge xs x. finite\ xs \implies fresh\ xs\ x \notin xs$ 
  and fresh-eq:  $\bigwedge xs x. x \notin xs \implies fresh\ xs\ x = x$ 

```

The type class *fresh* is essentially the same as the type class *infinite* but with an emphasis on fresh item generation.

```

class infinite =
  assumes infinite-UNIV:  $\neg finite\ (UNIV :: 'a\ set)$ 

```

We can subclass *fresh* to *infinite* since the latter has no associated operators (in particular, no additional operators w.r.t. the former).

```

subclass (in fresh) infinite
  apply (standard)
  using finite-list local.fresh-notIn by auto

```

end

2 Fresh identifier generation for natural numbers

```

theory Fresh-Nat
  imports Fresh
begin

```

Assuming $x \leq y$, *fresh2* $xs\ x\ y$ returns an element outside the interval (x, y) that is fresh for xs and closest to this interval, favoring smaller elements:

```

function fresh2 :: nat set  $\Rightarrow$  nat  $\Rightarrow$  nat  $\Rightarrow$  nat where
fresh2 xs x y =
  (if  $x \notin xs \vee infinite\ xs$  then  $x$  else
   if  $y \notin xs$  then  $y$  else
   fresh2 xs (x-1) (y+1))
by auto

```

```

termination
  apply (relation measure ( $\lambda(xs,x,y). (Max\ xs) + 1 - y$ ))
  by (simp-all add: Suc-diff-le)

```

```

lemma fresh2-notIn: finite xs  $\implies fresh2\ xs\ x\ y \notin xs$ 
  by (induct xs x y rule: fresh2.induct) auto

```

```

lemma fresh2-eq:  $x \notin xs \implies fresh2\ xs\ x\ y = x$ 
  by auto

```

```

declare fresh2.simps[simp del]

```

```

instantiation nat :: fresh
begin

```

fresh $xs\ x\ y$ returns an element that is fresh for xs and closest to x , favoring smaller elements:

definition *fresh-nat* :: *nat set* \Rightarrow *nat* \Rightarrow *nat* **where**
fresh-nat *xs* *x* \equiv *fresh2* *xs* *x* *x*

instance by *standard* (use *fresh2-notIn* *fresh2-eq* **in** \langle *auto simp add: fresh-nat-def* \rangle)

end

Code generation

lemma *fresh2-list*[*code*]:
fresh2 (*set xs*) *x* *y* =
 (if *x* \notin *set xs* then *x* else
 if *y* \notin *set xs* then *y* else
fresh2 (*set xs*) (*x*-1) (*y*+1))
by (*auto simp: fresh2.simps*)

Some tests:

value [*fresh* {}] (1::*nat*),
fresh {3,5,2,4} 3]

end

3 Fresh identifier generation for strings

theory *Fresh-String*
imports *Fresh*
begin

3.1 A partial order on strings

The first criterion is the length, and the second the encoding of last character.

definition *ordst* :: *string* \Rightarrow *string* \Rightarrow *bool* **where**
ordst *X* *Y* \equiv
 (*length* *X* \leq *length* *Y* \wedge *X* \neq [] \wedge *Y* \neq [] \wedge *of-char* (*last* *X*) $<$ (*of-char*(*last* *Y*))
 :: *nat*))
 \vee (*length* *X* $<$ *length* *Y*)

definition *ordstNS* :: *string* \Rightarrow *string* \Rightarrow *bool* **where**
ordstNS *X* *Y* \equiv *X* = *Y* \vee *ordst* *X* *Y*

lemma *ordst-antirefl*: \neg *ordst* *X* *X*
by(*auto simp add: ordst-def*)

lemma *ordst-trans*:

assumes *As1*: *ordst* *X* *Y* **and** *As2*: *ordst* *Y* *Z*

shows *ordst* *X* *Z*

proof(*cases* (*length* *X* $<$ *length* *Y*) \vee (*length* *Y* $<$ *length* *Z*))
assume (*length* *X* $<$ *length* *Y*) \vee (*length* *Y* $<$ *length* *Z*)

```

moreover
{assume  $length\ X < length\ Y$ 
 moreover have  $length\ Y \leq length\ Z$ 
 using  $As2\ ordst-def$  by force
 ultimately have  $length\ X < length\ Z$  by force
 hence  $?thesis$  using  $ordst-def$  by force}
moreover
{assume  $length\ Y < length\ Z$ 
 moreover have  $length\ X \leq length\ Y$ 
 using  $As1\ ordst-def$  by force
 ultimately have  $length\ X < length\ Z$  by force
 hence  $?thesis$  using  $ordst-def$  by force}
ultimately show  $?thesis$  by force
next
assume  $\neg (length\ X < length\ Y \vee length\ Y < length\ Z)$ 
hence  $Ft: \neg length\ X < length\ Y \wedge \neg length\ Y < length\ Z$  by force
hence  $(of-char(last\ X) :: nat) < of-char(last\ Y) \wedge$ 
 $(of-char(last\ Y) :: nat) < of-char(last\ Z) \wedge$ 
 $length\ X \leq length\ Y \wedge length\ Y \leq length\ Z$ 
using  $As1\ As2\ ordst-def$  by force
hence  $(of-char(last\ X) :: nat) < of-char(last\ Z) \wedge$ 
 $length\ X \leq length\ Z$  by force
moreover have  $X \neq [] \wedge Z \neq []$ 
using  $As1\ As2\ Ft\ ordst-def$  by force
ultimately show  $?thesis$  using  $ordst-def[of\ X\ Z]$  by force
qed

```

```

lemma  $ordstNS-refl: ordstNS\ X\ X$ 
by ( $simp\ add: ordstNS-def$ )

```

```

lemma  $ordstNS-trans:$ 
 $ordstNS\ X\ Y \implies ordstNS\ Y\ Z \implies ordstNS\ X\ Z$ 
by ( $metis\ ordstNS-def\ ordst-trans$ )

```

```

lemma  $ordst-ordstNS-trans:$ 
 $ordst\ X\ Y \implies ordstNS\ Y\ Z \implies ordst\ X\ Z$ 
by ( $metis\ ordstNS-def\ ordst-trans$ )

```

```

lemma  $ordstNS-ordst-trans:$ 
 $ordstNS\ X\ Y \implies ordst\ Y\ Z \implies ordst\ X\ Z$ 
by ( $metis\ ordstNS-def\ ordst-trans$ )

```

3.2 Incrementing a string

If the last character is \geq 'a' and $<$ 'z', then $upChar$ increments this last character; otherwise $upChar$ appends an 'a'.

```

fun  $upChar :: string \Rightarrow string$  where
 $upChar\ Y =$ 
 $(if\ (Y \neq [] \wedge of-char(last\ Y) \geq (97 :: nat) \wedge$ 

```

```

      of-char(last Y) < (122 :: nat))
    then (butlast Y) @
      [char-of(of-char(last Y) + (1 :: nat))]
    else Y @ "a"
  )

```

lemma *upChar-ordst*: *ordst Y (upChar Y)*

proof –

```

{assume ¬(Y ≠ [] ∧ of-char(last Y) ≥ (97 :: nat)
        ∧ of-char(last Y) < (122 :: nat))

```

hence *upChar Y = Y @ "a"* **by force**

hence *?thesis* **using** *ordst-def* **by force**

```

}

```

moreover

```

{assume As: Y ≠ [] ∧ of-char(last Y) ≥ (97 :: nat)
        ∧ of-char(last Y) < (122 :: nat)

```

hence *Ft*: *upChar Y = (butlast Y) @*

```

  [char-of(of-char(last Y) + (1 :: nat))]

```

by force

hence *Ft'*: *last (upChar Y) = char-of(of-char(last Y) + (1 :: nat))*

by force

```

hence of-char(last (upChar Y)) mod (256 :: nat) =
  (of-char(last Y) + 1) mod 256

```

by force

moreover

```

have of-char(last(upChar Y)) < (256 :: nat) ∧
  of-char(last Y) + 1 < (256 :: nat)

```

using *As Ft'* **by force**

ultimately

have *of-char (last Y) < (of-char (last(upChar Y)) :: nat)* **by force**

moreover

from *Ft* **have** *length Y ≤ length (upChar Y)* **by force**

ultimately have *?thesis* **using** *ordst-def* **by force**}

ultimately show *?thesis* **by force**

qed

3.3 The fresh-identifier operator

fresh Xs Y changes *Y* as little as possible so that it becomes disjoint from all strings in *Xs*.

function *fresh-string* :: *string set* ⇒ *string* ⇒ *string*

where

Up: *Y ∈ Xs* ⇒ *finite Xs* ⇒ *fresh-string Xs Y = fresh-string (Xs - {Y}) (upChar Y)*

|

Fresh: *Y ∉ Xs* ∨ *infinite Xs* ⇒ *fresh-string Xs Y = Y*

by auto

termination

apply(*relation measure* (λ(*Xs*, *Y*). *card Xs*), *simp-all*)

by (metis card-gt-0-iff diff-Suc-less empty-iff)

lemma *fresh-string-ordstNS*: ordstNS Y (fresh-string Xs Y)
proof (induction Xs Y rule: fresh-string.induct[case-names Up Fresh])
 case (Up Y Xs)
 hence ordst Y (fresh-string (Xs - {Y}) (upChar Y))
 using upChar-ordst[of Y] ordst-ordstNS-trans by force
 hence ordstNS Y (fresh-string (Xs - {Y}) (upChar Y))
 using ordstNS-def by auto
 thus ?case
 using Up.hyps by auto
next
 case (Fresh Y Xs)
 then show ?case
 by (auto intro: ordstNS-refl)
qed

lemma *fresh-string-set*: finite Xs \implies fresh-string Xs Y \notin Xs
proof (induction Xs Y rule: fresh-string.induct[case-names Up Fresh])
 case (Up Y Xs)
 show ?case
 proof
 assume fresh-string Xs Y \in Xs
 then have fresh-string (Xs - {Y}) (upChar Y) \in Xs
 using Up.hyps by force
 then have fresh-string (Xs - {Y}) (upChar Y) = Y
 using Up.IH ⟨finite Xs⟩ by blast
 moreover have ordst Y (fresh-string (Xs - {Y}) (upChar Y))
 using upChar-ordst[of Y] fresh-string-ordstNS ordst-ordstNS-trans by auto
 ultimately show False
 using ordst-antirefl by auto
 qed
qed auto

Code generation:

lemma *fresh-string-if*:
 fresh-string Xs Y = (
 if Y \in Xs \wedge finite Xs then fresh-string (Xs - {Y}) (upChar Y)
 else Y)
 by simp

lemmas *fresh-string-list*[code] = fresh-string-if[**where** Xs = set Xs **for** Xs, *simplified*]

Some tests:

value [fresh-string {} "Abc",
 fresh-string {"X", "Abc"} "Abd",
 fresh-string {"X", "Y"} "Y",
 fresh-string {"X", "Yaa", "Ya", "Yaa"} "Ya",

```

fresh-string {"X", "Yaa", "Yz", "Yza"} "Yz",
fresh-string {"X", "Y", "Yab", "Y"} "Y"

```

Here we do locale interpretation rather than class instantiation, since *string* is a type synonym for *char list*.

```

interpretation fresh-string: fresh where fresh = fresh-string
by standard (use fresh-string-set in auto)

```

3.4 Lifting to string literals

```

abbreviation is-ascii str ≡ (∀ c ∈ set str. ¬digit7 c)

```

```

lemma map-ascii-of-idem:
  is-ascii str ⇒ map String.ascii-of str = str
by (induction str) (auto simp: String.ascii-of-idem)

```

```

lemma is-ascii-butlast:
  is-ascii str ⇒ is-ascii (butlast str)
by (auto dest: in-set-butlastD)

```

```

lemma ascii-char-of:
  fixes c :: nat
  assumes c < 128
  shows ¬digit7 (char-of c)
  using assms
  by (auto simp: char-of-def bit-iff-odd)

```

```

lemmas ascii-of-char-of-idem = ascii-char-of[THEN String.ascii-of-idem]

```

```

lemma is-ascii-upChar:
  is-ascii str ⇒ is-ascii (upChar str)
by (auto simp: ascii-char-of is-ascii-butlast)

```

```

lemma is-ascii-fresh-string:
  is-ascii Y ⇒ is-ascii (fresh-string Xs Y)
proof (induction Xs Y rule: fresh-string.induct[case-names Up Fresh])
  case (Up Y Xs)
  show ?case
    using Up.IH[OF is-ascii-upChar[OF <is-ascii Y>]] Up.hyps
    by auto
qed auto

```

For string literals we can properly instantiate the class.

```

instantiation String.literal :: fresh
begin

```

```

context
  includes literal.lifting
begin

```

```

lift-definition fresh-literal :: String.literal set  $\Rightarrow$  String.literal  $\Rightarrow$  String.literal
  is fresh-string
  using is-ascii-fresh-string
  by blast

```

```

instance by (standard; transfer) (use fresh-string-set in auto)

```

```

end

```

```

end

```

Code generation:

```

context
  includes literal.lifting
begin

```

```

lift-definition upChar-literal :: String.literal  $\Rightarrow$  String.literal is upChar
  using is-ascii-upChar
  by blast

```

```

lemma upChar-literal-upChar[code]:
  upChar-literal s = String.implode (upChar (String.explode s))
  by transfer (auto simp: map-ascii-of-idem is-ascii-butlast ascii-of-char-of-idem)

```

```

lemma fresh-literal-if:
  fresh xs y = (if y  $\in$  xs  $\wedge$  finite xs then fresh (xs - {y}) (upChar-literal y) else y)
  by transfer (intro fresh-string-if)

```

```

lemmas fresh-literal-list[code] = fresh-literal-if[where xs = set xs for xs, simplified]

```

```

end

```

Some tests:

```

value [fresh {} (STR "Abc"),
  fresh {STR "X", STR "Abc"} (STR "Abd"),
  fresh {STR "X", STR "Y"} (STR "Y"),
  fresh {STR "X", STR "Yaa", STR "Ya", STR "Yaa"} (STR "Ya"),
  fresh {STR "X", STR "Yaa", STR "Yz", STR "Yza"} (STR "Yz"),
  fresh {STR "X", STR "Y", STR "Yab", STR "Y"} (STR "Y")]

```

```

end

```

4 Fresh identifier generation for infinite types

```

theory Fresh-Infinite
  imports Fresh

```


begin

This is a default fresh operator for infinite types for which more specific (smarter) alternatives are not (yet) available.

definition (*in infinite*) *fresh* :: 'a set \Rightarrow 'a \Rightarrow 'a **where**
fresh xs x \equiv *if* $x \notin xs \vee$ *infinite xs then* x *else* (*SOME y. y* \notin *xs*)

sublocale *infinite* < *fresh* **where** *fresh* = *fresh*

apply *standard*

subgoal unfolding *fresh-def*

by (*metis ex-new-if-finite local.infinite-UNIV someI-ex*)

subgoal unfolding *fresh-def* **by** *simp* .

end