1 Domain package examples

Domain constructors are strict by default.

domain d1 = d1a | d1b "d1" "d1"

lemma "d1b·⊥·y = ⊥" by simp

Constructors can be made lazy using the lazy keyword.

domain d2 = d2a | d2b (lazy "d2")

lemma "d2b·x ≠ ⊥" by simp

Strict and lazy arguments may be mixed arbitrarily.
domain \( d_3 = d_{3a} \mid d_{3b} \) (lazy "d2") "d2"

lemma "\( P (d_{3b} \cdot x \cdot y = \bot) \iff P (y = \bot) \)" by simp

Selectors can be used with strict or lazy constructor arguments.

domain \( d_4 = d_{4a} \mid d_{4b} \) (lazy \( d_{4b, \text{left}} :: "d2" \)) (\( d_{4b, \text{right}} :: "d2" \))

lemma "\( y \neq \bot \implies d_{4b, \text{left}} \cdot (d_{4b} \cdot x \cdot y) = x \)" by simp

Mixfix declarations can be given for data constructors.

domain \( d_5 = d_{5a} \mid d_{5b} \) (lazy "d5") "d5" (infixl "#:#" 70)

lemma "\( d_{5a} \neq x :#: y :#: z \)" by simp

Mixfix declarations can also be given for type constructors.

domain \( ('a, 'b) \text{lazypair} \) (infixl "*::" 25) = lpair (lazy \( \text{lfst :: 'a} \)) (lazy \( \text{lsnd :: 'b} \)) (infixl "*::" 75)

lemma "\( \forall p::('a :*: 'b). p \sqsubseteq \text{lfst} :: p :*: \text{lsnd} :: p \)"
by (rule allI, case_tac p, simp_all)

Non-recursive constructor arguments can have arbitrary types.

domain \( ('a, 'b) d_6 = d_6 \) "int lift" "'a \oplus 'b u" (lazy "('a :*: 'b) \times ('b \rightarrow 'a)")

Indirect recursion is allowed for sums, products, lifting, and the continuous function space. However, the domain package does not generate an induction rule in terms of the constructors.

domain \( 'a d_7 = d_{7a} \) "'a d_7 \oplus int lift" | \( d_{7b} \) "'a \odot 'a d_7" | \( d_{7c} \) (lazy "'a d_7 \rightarrow 'a")

— Indirect recursion detected, skipping proofs of (co)induction rules

Note that \( d_7.induct \) is absent.

Indirect recursion is also allowed using previously-defined datatypes.

domain \( 'a \text{slist} = \text{SNil} \mid \text{SCons 'a 'a \text{slist}} \)

domain \( 'a \text{ stree} = \text{STip} \mid \text{SBranch "'a stree slist} \)

Mutually-recursive datatypes can be defined using the \textit{and} keyword.

domain \( d_8 = d_{8a} \mid d_{8b} \) "d8" \text{ and } \( d_9 = d_{9a} \mid d_{9b} \) (lazy "d9")

Non-regular recursion is not allowed.

Mutually-recursive datatypes must have all the same type arguments, not necessarily in the same order.

domain \( ('a, 'b) \text{ list1} = \text{Nil1} \mid \text{Cons1 'a "('b, 'a) list2} \)
(b, 'a) list2 = Nil2 | Cons2 'b "('a, 'b) list1"

Induction rules for flat datatypes have no admissibility side-condition.

\textbf{domain} 'a flattree = Tip | Branch "'a flattree" "'a flattree"

\textbf{lemma} "[P ⊥; P Tip; \land x y. [x ≠ ⊥; y ≠ ⊥; P x; P y] \implies P (Branch x y)]
\implies P x"
by \textbf{(rule flattree.induct)} — no admissibility requirement

Trivial datatypes will produce a warning message.

\textbf{domain} triv = Triv triv triv
— domain Domain_ex.triv is empty!

\textbf{lemma} "(x::triv) = ⊥" by (induct x, simp_all)

Lazy constructor arguments may have unpointed types.

\textbf{domain} natlist = nnil | ncons (lazy "nat discr") natlist

Class constraints may be given for type parameters on the LHS.

\textbf{domain} ('a::predomain) box = Box (lazy 'a)

\textbf{domain} ('a::countable) stream = snil | scons (lazy "'a discr") "'a stream"

1.1 Generated constants and theorems

\textbf{domain} 'a tree = Leaf (lazy 'a) | Node (left :: "'a tree") (right :: "'a tree")

\textbf{lemmas} tree_abs_bottom_iff =
iso.abs_bottom_iff [OF iso.intro [OF tree.abs_iso tree.rep_iso]]

Rules about isomorphism

\textbf{term} tree_rep
\textbf{term} tree_abs
\textbf{thm} tree.rep_iso
\textbf{thm} tree.abs_iso
\textbf{thm} tree.iso_rews

Rules about constructors

\textbf{term} Leaf
\textbf{term} Node
\textbf{thm} Leaf_def Node_def
\textbf{thm} tree.nchotomy
\textbf{thm} tree.exhaust
\textbf{thm} tree.compacts
\textbf{thm} tree.con_rews
\textbf{thm} tree.dist_les
\textbf{thm} tree.dist_eqs
thm tree.inverts
thm tree.injects

Rules about case combinator
term tree_case
thm tree.tree_case_def
thm tree.case_rews

Rules about selectors
term left
term right
thm tree.sel_rews

Rules about discriminators
term is_Leaf
term is_Node
thm tree.dis_rews

Rules about monadic pattern match combinators
term match_Leaf
term match_Node
thm tree.match_rews

Rules about take function
term tree.take
thm tree.take_def
thm tree.take_0
thm tree.take_Suc
thm tree.take_rews
thm tree.chain_take
thm tree.take_take
thm tree.deflation_take
thm tree.take_below
thm tree.take lemma
thm tree.lub_take
thm tree.reach
thm tree.finite_induct

Rules about finiteness predicate
term tree.finite
thm tree.finite_def
thm tree.finite

Rules about bisimulation predicate
term tree.bisim
thm tree.bisim_def
thm tree.coinduct
Induction rule

thm tree.induct

1.2 Known bugs

Declaring a mixfix with spaces causes some strange parse errors.

end

2 Fixrec package examples

theory Fixrec_ex
imports HOLCF
begin

2.1 Basic fixrec examples

Fixrec patterns can mention any constructor defined by the domain package, as well as any of the following built-in constructors: Pair, spair, sinl, sinr, up, ONE, TT, FF.

Typical usage is with lazy constructors.

fixrec down :: "'a u → 'a"
where "down · (up · x) = x"

With strict constructors, rewrite rules may require side conditions.

fixrec from_sinl :: "'a ⊕ 'b → 'a"
where "x ≠ ⊥ ⇒ from_sinl · (sinl · x) = x"

Lifting can turn a strict constructor into a lazy one.

fixrec from_sinl_up :: "'a ⊕ 'b ⊕ 'a u → 'a"
where "from_sinl_up · (sinl · (up · x)) = x"

Fixrec also works with the HOL pair constructor.

fixrec down2 :: "'a u × 'b u → 'a × 'b"
where "down2 · (up · x, up · y) = (x, y)"

2.2 Examples using fixrec_simp

A type of lazy lists.

domain 'a llist = lNil | lCons (lazy 'a) (lazy "'a llist")

A zip function for lazy lists.

Notice that the patterns are not exhaustive.

fixrec
lzip :: "'a llist → 'b llist → ('a × 'b) llist"
where
"lzip·(lCons·x·xs)·(lCons·y·ys) = lCons·(x, y)·(lzip·xs·ys)"
| "lzip·lNil·lNil = lNil"

fixrec_simp is useful for producing strictness theorems.

Note that pattern matching is done in left-to-right order.

lemma lzip_stricts [simp]:
"lzip·⊥·ys = ⊥"
"lzip·lNil·⊥ = ⊥"
"lzip·(lCons·x·xs)·⊥ = ⊥"
by fixrec_simp+

fixrec_simp can also produce rules for missing cases.

lemma lzip.Undefs [simp]:
"lzip·lNil·(lCons·y·ys) = ⊥"
"lzip·(lCons·x·xs)·lNil = ⊥"
by fixrec_simp+

2.3 Pattern matching with bottoms

As an alternative to using fixrec_simp, it is also possible to use bottom as a constructor pattern. When using a bottom pattern, the right-hand-side must also be bottom; otherwise, fixrec will not be able to prove the equation.

fixrec
from_sinr_up :: "'a ⊕ 'b ⊥ → 'b"
where
"from_sinr_up·⊥ = ⊥"
| "from_sinr_up·(sinr·(up·x)) = x"

If the function is already strict in that argument, then the bottom pattern does not change the meaning of the function. For example, in the definition of from_sinr_up, the first equation is actually redundant, and could have been proven separately by fixrec_simp.

A bottom pattern can also be used to make a function strict in a certain argument, similar to a bang-pattern in Haskell.

fixrec
seq :: "'a → 'b → 'b"
where
"seq·⊥·y = ⊥"
| "x ≠ ⊥ ⇒ seq·x·y = y"

2.4 Skipping proofs of rewrite rules

Another zip function for lazy lists.
Notice that this version has overlapping patterns. The second equation cannot be proved as a theorem because it only applies when the first pattern fails.

\begin{verbatim}
fixrec
  lzip2 :: 'a llist → 'b llist → ('a × 'b) llist
where
  "lzip2 (lCons·x·xs)·(lCons·y·ys) = lCons·(x, y)·(lzip2·xs·ys)"
| (unchecked) "lzip2·xs·ys = 1Nil"
\end{verbatim}

Usually fixrec tries to prove all equations as theorems. The "unchecked" option overrides this behavior, so fixrec does not attempt to prove that particular equation.

Simp rules can be generated later using fixrec simp.

\begin{verbatim}
lemma lzip2_simps [simp]:
  "lzip2·(lCons·x·xs)·1Nil = 1Nil"
  "lzip2·1Nil·(lCons·y·ys) = 1Nil"
  "lzip2·1Nil·1Nil = 1Nil"
by fixrec simp+

lemma lzip2_stricts [simp]:
  "lzip2·⊥·ys = ⊥"
  "lzip2·(lCons·x·xs)·⊥ = ⊥"
by fixrec simp+
\end{verbatim}

### 2.5 Mutual recursion with fixrec

Tree and forest types.

\begin{verbatim}
domain 'a tree = Leaf (lazy 'a) | Branch (lazy "'a forest")
and 'a forest = Empty | Trees (lazy "'a tree") "'a forest"
\end{verbatim}

To define mutually recursive functions, give multiple type signatures separated by the keyword and.

\begin{verbatim}
fixrec
  map_tree :: "('a → 'b) → ('a tree → 'b tree)"
and
  map_forest :: "('a → 'b) → ('a forest → 'b forest)"
where
  "map_tree·f·(Leaf·x) = Leaf·(f·x)"
| "map_tree·f·(Branch·ts) = Branch·(map_forest·f·ts)"
| "map_forest·f·Empty = Empty"
| "ts ≠ ⊥ ⇒ map_forest·f·(Trees·t·ts) = Trees·(map_tree·f·t)·(map_forest·f·ts)"

lemma map_tree_strict [simp]: "map_tree·f·⊥ = ⊥"
by fixrec simp
\end{verbatim}
lemma map_forest_strict [simp]: "\(\text{map} \cdot f \cdot \bot = \bot\)"
by fixrec_simp

2.6 Looping simp rules

The defining equations of a fixrec definition are declared as simp rules by
default. In some cases, especially for constants with no arguments or functions
with variable patterns, the defining equations may cause the simplifier
to loop. In these cases it will be necessary to use a \([\text{simp del}]\) declaration.

fixrec
\(\text{repeat} :: 'a \rightarrow 'a \text{llist}\)
where
\([\text{simp del}]\): "\(\text{repeat} \cdot x = \text{lCons} \cdot x \cdot (\text{repeat} \cdot x)\)"

We can derive other non-looping simp rules for \(\text{repeat}\) by using the \(\text{subst}\)
method with the \(\text{repeat.simps}\) rule.

lemma repeat_simps [simp]:
"\(\text{repeat} \cdot x \neq \bot\)"
"\(\text{repeat} \cdot x \neq \text{Nil}\)"
"\(\text{repeat} \cdot x = \text{lCons} \cdot y \cdot \text{ys} \iff x = y \land \text{repeat} \cdot x = \text{ys}\)"
by (subst repeat.simps, simp)+

lemma llist_case_repeat [simp]:
"\(\text{llist_case} \cdot z \cdot f \cdot (\text{repeat} \cdot x) = f \cdot x \cdot (\text{repeat} \cdot x)\)"
by (subst repeat.simps, simp)

For mutually-recursive constants, looping might only occur if all equations
are in the simpset at the same time. In such cases it may only be necessary
to declare \([\text{simp del}]\) on one equation.

fixrec
\(\text{inf} \_ \text{tree} :: \"\text{'a} \text{\ tree}\" \text{ and } \text{inf} \_ \text{forest} :: \"\text{'a} \text{\ forest}\"
where
\([\text{simp del}]\): "\(\text{inf} \_ \text{tree} = \text{Branch} \cdot \text{inf} \_ \text{forest}\"
| "\(\text{inf} \_ \text{forest} = \text{Trees} \cdot \text{inf} \_ \text{tree} \cdot (\text{Trees} \cdot \text{inf} \_ \text{tree} \cdot \text{Empty})\)"

2.7 Using fixrec inside locales

locale test =
fixes foo :: \"\text{'a} \rightarrow 'a\"
assumes foo_strict: "\(\text{foo} \cdot \bot = \bot\)"
begin

fixrec
\(\text{bar} :: \"\text{'a} \text{\ u} \rightarrow 'a\"
where
"\(\text{bar} \cdot (\text{up} \cdot x) = \text{foo} \cdot x\)"

lemma bar_strict: "\(\text{bar} \cdot \bot = \bot\)"
by fixrec_simp

end

end

3 Definitional domain package

theory New_Domain
imports HOLCF
begin

UPDATE: The definitional back-end is now the default mode of the domain package. This file should be merged with Domain_ex.thy.

Provided that domain is the default sort, the new_domain package should work with any type definition supported by the old domain package.

domain 'a llist = LNil | LCons (lazy 'a) (lazy "'a llist")

The difference is that the new domain package is completely definitional, and does not generate any axioms. The following type and constant definitions are not produced by the old domain package.

thm type_definition_lлист
thm lлист_abs_def lлист_rep_def

The new domain package also adds support for indirect recursion with user-defined datatypes. This definition of a tree datatype uses indirect recursion through the lazy list type constructor.

domain 'a ltree = Leaf (lazy 'a) | Branch (lazy "'a ltree lлист")

For indirect-recursive definitions, the domain package is not able to generate a high-level induction rule. (It produces a warning message instead.) The low-level reach lemma (now proved as a theorem, no longer generated as an axiom) can be used to derive other induction rules.

thm ltree.reach

The definition of the take function uses map functions associated with each type constructor involved in the definition. A map function for the lazy list type has been generated by the new domain package.

thm ltree.take_rews
thm lлист_map_def

lemma ltree.induct:
  fixes P :: "'a ltree ⇒ bool"
  assumes adm: "adm P"
  assumes bot: "P ⊥"
assumes Leaf: "∀x. P (Leaf x)"
assumes Branch: "∀f l. ∀x. P (f x) ⟹ P (Branch (l list_map f l))"
shows "P x"

proof -
  have "P (∐ i. l tree_take i x)"
  using adm
  proof (rule admD)
    fix i
    show "P (l tree_take i x)"
    proof (induct i arbitrary: x)
      case (0 x)
      show "P (l tree_take 0 x)" by (simp add: bot)
    next
      case (Suc n x)
      show "P (l tree_take (Suc n) x)"
      apply (cases x)
      apply (simp add: bot)
      apply (simp add: Leaf)
      apply (simp add: Branch Suc)
      done
    qed
    qed (simp add: l tree.chain_take)
  thus ?thesis
  by (simp add: l tree.reach)
  qed

end