Isabelle/HOLCF Tutorial

April 15, 2020

Contents

1 Domain package examples **1**
1.1 Generated constants and theorems 3
1.2 Known bugs 5

2 Fixrec package examples **5**
2.1 Basic *fixrec* examples 5
2.2 Examples using *fixrec_simp* 5
2.3 Pattern matching with bottoms 6
2.4 Skipping proofs of rewrite rules 6
2.5 Mutual recursion with *fixrec* 7
2.6 Looping simp rules 8
2.7 Using *fixrec* inside locales 8

3 Definitional domain package **9**

1 Domain package examples

theory *Domain_ex*
imports *HOLCF*
begin

Domain constructors are strict by default.

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

lemma "*d1b*·⊥·*y* = ⊥" ⟨*proof*⟩

Constructors can be made lazy using the *lazy* keyword.

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

lemma "*d2b*·*x* ≠ ⊥" ⟨*proof*⟩

Strict and lazy arguments may be mixed arbitrarily.

domain *d3* = *d3a* | *d3b* (**lazy** "d2") "d2"

lemma "P (d3b·x·y = ⊥) ⟷ P (y = ⊥)" ⟨*proof*⟩

Selectors can be used with strict or lazy constructor arguments.

domain *d4* = *d4a* | *d4b* (**lazy** *d4b_left* :: "d2") (*d4b_right* :: "d2")

lemma "y ≠ ⊥ ⟹ d4b_left·(d4b·x·y) = x" ⟨*proof*⟩

Mixfix declarations can be given for data constructors.

domain *d5* = *d5a* | *d5b* (**lazy** "d5") "d5" (**infixl** ":#:" 70)

lemma "d5a ≠ x :#: y :#: z" ⟨*proof*⟩

Mixfix declarations can also be given for type constructors.

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

lemma "∀p::('a :*: 'b). p ⊑ lfst·p :*: lsnd·p"
⟨*proof*⟩

Non-recursive constructor arguments can have arbitrary types.

domain ('a, 'b) *d6* = *d6* "int lift" "'a ⊕ 'b u" (**lazy** "('a :*: 'b) × ('b → 'a)")

Indirect recusion 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 *d7* = *d7a* "'a d7 ⊕ int lift" | *d7b* "'a ⊗ 'a d7" | *d7c* (**lazy** "'a d7 → 'a")
  — Indirect recursion detected, skipping proofs of (co)induction rules

Note that *d7.induct* is absent.

Indirect recursion is also allowed using previously-defined datatypes.

domain 'a *slist* = *SNil* | *SCons* 'a "'a slist"

domain 'a *stree* = *STip* | *SBranch* "'a stree slist"

Mutually-recursive datatypes can be defined using the *and* keyword.

domain *d8* = *d8a* | *d8b* "d9" **and** *d9* = *d9a* | *d9b* (**lazy** "d8")

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) *list1* = *Nil1* | *Cons1* 'a "('b, 'a) list2"

and *('b, 'a) list2 = Nil2 | Cons2 'b "('a, 'b) list1"*

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

domain *'a flattree = Tip | Branch "'a flattree" "'a flattree"*

lemma "$[\![$P $\bot$; P Tip; $\bigwedge$x y. $[\![$x $\neq$ $\bot$; y $\neq$ $\bot$; P x; P y$]\!]$ $\Longrightarrow$ P (Branch$\cdot$x$\cdot$y)$]\!]$ $\Longrightarrow$ P x"
⟨*proof*⟩

Trivial datatypes will produce a warning message.

domain *triv = Triv triv triv*
— domain *Domain_ex.triv* is empty!

lemma "(x::triv) = $\bot$" ⟨*proof*⟩

Lazy constructor arguments may have unpointed types.

domain *natlist = nnil | ncons* (**lazy** *"nat discr") natlist*

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

domain *('a::predomain) box = Box* (**lazy** *'a)*

domain *('a::countable) stream = snil | scons* (**lazy** *"'a discr") "'a stream"*

1.1 Generated constants and theorems

domain *'a tree = Leaf* (**lazy** *'a) | Node (left :: "'a tree") (right :: "'a tree")*

lemmas *tree_abs_bottom_iff =*
iso.abs_bottom_iff [OF iso.intro [OF tree.abs_iso tree.rep_iso]]

Rules about ismorphism

term *tree_rep*
term *tree_abs*
thm *tree.rep_iso*
thm *tree.abs_iso*
thm *tree.iso_rews*

Rules about constructors

term *Leaf*
term *Node*
thm *Leaf_def Node_def*
thm *tree.nchotomy*
thm *tree.exhaust*
thm *tree.compacts*
thm *tree.con_rews*
thm *tree.dist_les*
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 u ⊕ 'b → '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)·⊥ = ⊥"
⟨proof⟩
```

fixrec_simp can also produce rules for missing cases.

```
lemma lzip_undefs [simp]:
  "lzip·lNil·(lCons·y·ys) = ⊥"
  "lzip·(lCons·x·xs)·lNil = ⊥"
⟨proof⟩
```

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.

```
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 = lNil"
```

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*.

```
lemma lzip2_simps [simp]:
  "lzip2·(lCons·x·xs)·lNil = lNil"
  "lzip2·lNil·(lCons·y·ys) = lNil"
  "lzip2·lNil·lNil = lNil"
⟨proof⟩

lemma lzip2_stricts [simp]:
  "lzip2·⊥·ys = ⊥"
  "lzip2·(lCons·x·xs)·⊥ = ⊥"
⟨proof⟩
```

2.5 Mutual recursion with *fixrec*

Tree and forest types.

```
domain 'a tree = Leaf (lazy 'a) | Branch (lazy "'a forest")
and    'a forest = Empty | Trees (lazy "'a tree") "'a forest"
```

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

```
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·⊥ = ⊥"
⟨proof⟩
```

lemma *map_forest_strict [simp]: "map_forest·f·⊥ = ⊥"*
⟨*proof*⟩

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 *[simp del]* declaration.

```
fixrec
  repeat :: "'a → 'a llist"
where
  [simp del]: "repeat·x = lCons·x·(repeat·x)"
```

We can derive other non-looping simp rules for *repeat* by using the *subst* method with the *repeat.simps* rule.

```
lemma repeat_simps [simp]:
  "repeat·x ≠ ⊥"
  "repeat·x ≠ lNil"
  "repeat·x = lCons·y·ys ⟷ x = y ∧ repeat·x = ys"
⟨proof⟩
```

```
lemma llist_case_repeat [simp]:
  "llist_case·z·f·(repeat·x) = f·x·(repeat·x)"
⟨proof⟩
```

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 *[simp del]* on one equation.

```
fixrec
  inf_tree :: "'a tree" and inf_forest :: "'a forest"
where
  [simp del]: "inf_tree = Branch·inf_forest"
| "inf_forest = Trees·inf_tree·(Trees·inf_tree·Empty)"
```

2.7 Using *fixrec* inside locales

```
locale test =
  fixes foo :: "'a → 'a"
  assumes foo_strict: "foo·⊥ = ⊥"
begin
```

```
fixrec
  bar :: "'a u → 'a"
where
  "bar·(up·x) = foo·x"
```

```
lemma bar_strict: "bar·⊥ = ⊥"
```

⟨*proof*⟩

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_llist*
thm *llist_abs_def llist_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 llist"*)

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 *llist_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·(llist_map·f·l))"
  shows "P x"
⟨proof⟩

end
```