A Fixedpoint Approach to (Co)Inductive and (Co)Datatype Definitions*

Lawrence C. Paulson
lcp@cl.cam.ac.uk
Computer Laboratory, University of Cambridge, England

28 August 1998

Abstract

This paper presents a fixedpoint approach to inductive definitions. Instead of using a syntactic test such as "strictly positive," the approach lets definitions involve any operators that have been proved monotone. It is conceptually simple, which has allowed the easy implementation of mutual recursion and iterated definitions. It also handles coinductive definitions: simply replace the least fixedpoint by a greatest fixedpoint.

The method has been implemented in two of Isabelle's logics, ZF set theory and higher-order logic. It should be applicable to any logic in which the Knaster-Tarski theorem can be proved. Examples include lists of n elements, the accessible part of a relation and the set of primitive recursive functions. One example of a coinductive definition is bisimulations for lazy lists. Recursive datatypes are examined in detail, as well as one example of a **codatatype**: lazy lists.

The Isabelle package has been applied in several large case studies, including two proofs of the Church-Rosser theorem and a coinductive proof of semantic consistency. The package can be trusted because it proves theorems from definitions, instead of asserting desired properties as axioms.

*J. Grundy and S. Thompson made detailed comments. Mads Tofte and the referees were also helpful. The research was funded by the SERC grants GR/G53279, GR/H40570 and by the ESPRIT Project 6453 "Types".

Contents

1 Introduction

Several theorem provers provide commands for formalizing recursive data structures, like lists and trees. Robin Milner implemented one of the first of these, for Edinburgh LCF [16]. Given a description of the desired data structure, Milner's package formulated appropriate definitions and proved the characteristic theorems. Similar is Melham's recursive type package for the Cambridge HOL system [15]. Such data structures are called **datatypes** below, by analogy with datatype declarations in Standard ML. Some logics take datatypes as primitive; consider Boyer and Moore's shell principle [4] and the Coq type theory [22].

A datatype is but one example of an **inductive definition**. Such a definition [2] specifies the least set R **closed under** given rules: applying a rule to elements of R yields a result within R. Inductive definitions have many applications. The collection of theorems in a logic is inductively defined. A structural operational semantics [13] is an inductive definition of a reduction or evaluation relation on programs. A few theorem provers provide commands for formalizing inductive definitions; these include Coq [22] and again the HOL system [5].

The dual notion is that of a **coinductive definition**. Such a definition specifies the greatest set R **consistent with** given rules: every element of R can be seen as arising by applying a rule to elements of R. Important examples include using bisimulation relations to formalize equivalence of processes [17] or lazy functional programs [1]. Other examples include lazy lists and other infinite data structures; these are called **codatatypes** below.

Not all inductive definitions are meaningful. **Monotone** inductive definitions are a large, well-behaved class. Monotonicity can be enforced by syntactic conditions such as "strictly positive," but this could lead to monotone definitions being rejected on the grounds of their syntactic form. More flexible is to formalize monotonicity within the logic and allow users to prove it.

This paper describes a package based on a fixedpoint approach. Least fixedpoints yield inductive definitions; greatest fixedpoints yield coinductive definitions. Most of the discussion below applies equally to inductive and coinductive definitions, and most of the code is shared.

The package supports mutual recursion and infinitely-branching datatypes and codatatypes. It allows use of any operators that have been proved monotone, thus accepting all provably monotone inductive definitions, including iterated definitions.

The package has been implemented in Isabelle [29, 25] using ZF set theory [24, 27]; part of it has since been ported to Isabelle/HOL (higher-order logic). The recursion equations are specified as introduction rules for the mutually recursive sets. The package transforms these rules into a mapping over sets, and attempts to prove that the mapping is monotonic and well-typed. If successful, the package makes fixedpoint definitions and proves the introduction, elimination and (co)induction rules. Users invoke the package by making simple declarations in Isabelle theory files.

Most datatype packages equip the new datatype with some means of expressing recursive functions. This is the main omission from my package. Its fixedpoint operators define only recursive sets. The Isabelle/ZF theory provides well-founded recursion [27], which is harder to use than structural recursion but considerably more general. Slind [34] has written a package to automate the definition of well-founded recursive functions in Isabelle/HOL.

Outline. Section 2 introduces the least and greatest fixedpoint operators. Section 3 discusses the form of introduction rules, mutual recursion and other points common to inductive and coinductive definitions. Section 4 discusses induction and coinduction rules separately. Section 5 presents several examples, including a coinductive definition. Section 6 describes datatype definitions. Section 7 presents related work. Section 8 draws brief conclusions. The appendices are simple user's manuals for this Isabelle package.

Most of the definitions and theorems shown below have been generated by the package. I have renamed some variables to improve readability.

2 Fixedpoint operators

In set theory, the least and greatest fixedpoint operators are defined as follows:

$$\begin{aligned} \texttt{lfp}(D,h) &\equiv \bigcap\{X \subseteq D \,.\, h(X) \subseteq X\} \\ \texttt{gfp}(D,h) &\equiv \bigcup\{X \subseteq D \,.\, X \subseteq h(X)\} \end{aligned}$$

Let D be a set. Say that h is **bounded by** D if $h(D) \subseteq D$, and **monotone below** D if $h(A) \subseteq h(B)$ for all A and B such that $A \subseteq B \subseteq D$. If h is bounded by D and monotone then both operators yield fixedpoints:

$$\begin{aligned} \texttt{lfp}(D,h) &= h(\texttt{lfp}(D,h)) \\ \texttt{gfp}(D,h) &= h(\texttt{gfp}(D,h)) \end{aligned}$$

These equations are instances of the Knaster-Tarski theorem, which states that every monotonic function over a complete lattice has a fixedpoint [6]. It is obvious from their definitions that `lfp` must be the least fixedpoint, and `gfp` the greatest.

This fixedpoint theory is simple. The Knaster-Tarski theorem is easy to prove. Showing monotonicity of h is trivial, in typical cases. We must also exhibit a bounding set D for h. Frequently this is trivial, as when a set of theorems is (co)inductively defined over some previously existing set of formulæ. Isabelle/ZF provides suitable bounding sets for infinitely-branching (co)datatype definitions; see §6.1. Bounding sets are also called **domains**.

The powerset operator is monotone, but by Cantor's theorem there is no set A such that $A = \mathcal{P}(A)$. We cannot put $A = \texttt{lfp}(D, \mathcal{P})$ because there is no suitable domain D. But §5.5 demonstrates that $\mathcal{P}$ is still useful in inductive definitions.

3 Elements of an inductive or coinductive definition

Consider a (co)inductive definition of the sets $R_1, \ldots, R_n$, in mutual recursion. They will be constructed from domains $D_1, \ldots, D_n$, respectively. The construction yields not $R_i \subseteq D_i$ but $R_i \subseteq D_1 + \cdots + D_n$, where R_i is contained in the image of D_i under an injection. Reasons for this are discussed elsewhere [27, §4.5].

The definition may involve arbitrary parameters $\vec{p} = p_1, \ldots, p_k$. Each recursive set then has the form $R_i(\vec{p})$. The parameters must be identical every time they occur within a definition. This would appear to be a serious restriction compared with other systems such as Coq [22]. For instance, we cannot define the lists of n elements as the set $\mathtt{listn}(A, n)$ using rules where the parameter n varies. Section 5.2 describes how to express this set using the inductive definition package.

To avoid clutter below, the recursive sets are shown as simply R_i instead of $R_i(\vec{p})$.

3.1 The form of the introduction rules

The body of the definition consists of the desired introduction rules. The conclusion of each rule must have the form $t \in R_i$, where t is any term. Premises typically have the same form, but they can have the more general form $t \in M(R_i)$ or express arbitrary side-conditions.

The premise $t \in M(R_i)$ is permitted if M is a monotonic operator on sets, satisfying the rule

$$\frac{A \subseteq B}{M(A) \subseteq M(B)}$$

The user must supply the package with monotonicity rules for all such premises.

The ability to introduce new monotone operators makes the approach flexible. A suitable choice of M and t can express a lot. The powerset operator $\mathcal{P}$ is monotone, and the premise $t \in \mathcal{P}(R)$ expresses $t \subseteq R$; see §5.5 for an example. The *list of* operator is monotone, as is easily proved by induction. The premise $t \in \mathtt{list}(R)$ avoids having to encode the effect of $\mathtt{list}(R)$ using mutual recursion; see §5.6 and also my earlier paper [27, §4.4].

Introduction rules may also contain **side-conditions**. These are premises consisting of arbitrary formulæ not mentioning the recursive sets. Side-conditions typically involve type-checking. One example is the premise $a \in A$ in the following rule from the definition of lists:

$$\frac{a \in A \quad l \in \mathtt{list}(A)}{\mathtt{Cons}(a, l) \in \mathtt{list}(A)}$$

3.2 The fixedpoint definitions

The package translates the list of desired introduction rules into a fixedpoint definition. Consider, as a running example, the finite powerset operator $\texttt{Fin}(A)$: the set of all finite subsets of A. It can be defined as the least set closed under the rules

$$\emptyset \in \texttt{Fin}(A) \qquad \frac{a \in A \quad b \in \texttt{Fin}(A)}{\{a\} \cup b \in \texttt{Fin}(A)}$$

The domain in a (co)inductive definition must be some existing set closed under the rules. A suitable domain for $\texttt{Fin}(A)$ is $\mathcal{P}(A)$, the set of all subsets of A. The package generates the definition

$$\begin{aligned}\texttt{Fin}(A) \equiv \texttt{lfp}(\mathcal{P}(A),\quad \lambda X\,.\,\{z \in \mathcal{P}(A).\, z = \emptyset \lor \\ (\exists a\, b\,.\, z = \{a\} \cup b \land a \in A \land b \in X)\})\end{aligned}$$

The contribution of each rule to the definition of $\texttt{Fin}(A)$ should be obvious. A coinductive definition is similar but uses `gfp` instead of `lfp`.

The package must prove that the fixedpoint operator is applied to a monotonic function. If the introduction rules have the form described above, and if the package is supplied a monotonicity theorem for every $t \in M(R_i)$ premise, then this proof is trivial.[1]

The package returns its result as an ML structure, which consists of named components; we may regard it as a record. The result structure contains the definitions of the recursive sets as a theorem list called `defs`. It also contains some theorems; `dom_subset` is an inclusion such as $\texttt{Fin}(A) \subseteq \mathcal{P}(A)$, while `bnd_mono` asserts that the fixedpoint definition is monotonic.

Internally the package uses the theorem `unfold`, a fixedpoint equation such as

$$\begin{aligned}\texttt{Fin}(A) = \{z \in \mathcal{P}(A).\, z = \emptyset \lor \\ (\exists a\, b\,.\, z = \{a\} \cup b \land a \in A \land b \in \texttt{Fin}(A))\}\end{aligned}$$

In order to save space, this theorem is not exported.

3.3 Mutual recursion

In a mutually recursive definition, the domain of the fixedpoint construction is the disjoint sum of the domain D_i of each R_i, for $i = 1, \ldots, n$. The package uses the injections of the binary disjoint sum, typically `Inl` and `Inr`, to express injections $h_{1n}, \ldots, h_{nn}$ for the n-ary disjoint sum $D_1 + \cdots + D_n$.

[1] Due to the presence of logical connectives in the fixedpoint's body, the monotonicity proof requires some unusual rules. These state that the connectives $\land$, $\lor$ and $\exists$ preserve monotonicity with respect to the partial ordering on unary predicates given by $P \sqsubseteq Q$ if and only if $\forall x.P(x) \to Q(x)$.

As discussed elsewhere [27, §4.5], Isabelle/ZF defines the operator `Part` to support mutual recursion. The set $\texttt{Part}(A, h)$ contains those elements of A having the form $h(z)$:

$$\texttt{Part}(A, h) \equiv \{x \in A \,.\, \exists z \,.\, x = h(z)\}.$$

For mutually recursive sets $R_1, \ldots, R_n$ with $n > 1$, the package makes $n + 1$ definitions. The first defines a set R using a fixedpoint operator. The remaining n definitions have the form

$$R_i \equiv \texttt{Part}(R, h_{in}), \qquad i = 1, \ldots, n.$$

It follows that $R = R_1 \cup \cdots \cup R_n$, where the R_i are pairwise disjoint.

3.4 Proving the introduction rules

The user supplies the package with the desired form of the introduction rules. Once it has derived the theorem `unfold`, it attempts to prove those rules. From the user's point of view, this is the trickiest stage; the proofs often fail. The task is to show that the domain $D_1 + \cdots + D_n$ of the combined set $R_1 \cup \cdots \cup R_n$ is closed under all the introduction rules. This essentially involves replacing each R_i by $D_1 + \cdots + D_n$ in each of the introduction rules and attempting to prove the result.

Consider the $\texttt{Fin}(A)$ example. After substituting $\mathcal{P}(A)$ for $\texttt{Fin}(A)$ in the rules, the package must prove

$$\emptyset \in \mathcal{P}(A) \qquad \frac{a \in A \quad b \in \mathcal{P}(A)}{\{a\} \cup b \in \mathcal{P}(A)}$$

Such proofs can be regarded as type-checking the definition.[2] The user supplies the package with type-checking rules to apply. Usually these are general purpose rules from the ZF theory. They could however be rules specifically proved for a particular inductive definition; sometimes this is the easiest way to get the definition through!

The result structure contains the introduction rules as the theorem list `intrs`.

3.5 The case analysis rule

The elimination rule, called `elim`, performs case analysis. It is a simple consequence of `unfold`. There is one case for each introduction rule. If $x \in \texttt{Fin}(A)$ then either $x = \emptyset$ or else $x = \{a\} \cup b$ for some $a \in A$ and $b \in \texttt{Fin}(A)$. Formally,

[2] The Isabelle/HOL version does not require these proofs, as HOL has implicit type-checking.

the elimination rule for $\texttt{Fin}(A)$ is written

$$\frac{x \in \texttt{Fin}(A) \qquad \begin{matrix}[x = \emptyset] \\ \vdots \\ Q\end{matrix} \qquad \begin{matrix}[x = \{a\} \cup b \quad a \in A \quad b \in \texttt{Fin}(A)]_{a,b} \\ \vdots \\ Q\end{matrix}}{Q}$$

The subscripted variables a and b above the third premise are eigenvariables, subject to the usual "not free in ..." proviso.

4 Induction and coinduction rules

Here we must consider inductive and coinductive definitions separately. For an inductive definition, the package returns an induction rule derived directly from the properties of least fixedpoints, as well as a modified rule for mutual recursion. For a coinductive definition, the package returns a basic coinduction rule.

4.1 The basic induction rule

The basic rule, called `induct`, is appropriate in most situations. For inductive definitions, it is strong rule induction [5]; for datatype definitions (see below), it is just structural induction.

The induction rule for an inductively defined set R has the form described below. For the time being, assume that R's domain is not a Cartesian product; inductively defined relations are treated slightly differently.

The major premise is $x \in R$. There is a minor premise for each introduction rule:

- If the introduction rule concludes $t \in R_i$, then the minor premise is $P(t)$.
- The minor premise's eigenvariables are precisely the introduction rule's free variables that are not parameters of R. For instance, the eigenvariables in the $\texttt{Fin}(A)$ rule below are a and b, but not A.
- If the introduction rule has a premise $t \in R_i$, then the minor premise discharges the assumption $t \in R_i$ and the induction hypothesis $P(t)$. If the introduction rule has a premise $t \in M(R_i)$ then the minor premise discharges the single assumption

  $$t \in M(\{z \in R_i \,.\, P(z)\}).$$

  Because M is monotonic, this assumption implies $t \in M(R_i)$. The occurrence of P gives the effect of an induction hypothesis, which may be exploited by appealing to properties of M.

The induction rule for $\mathtt{Fin}(A)$ resembles the elimination rule shown above, but includes an induction hypothesis:

$$\frac{x \in \mathtt{Fin}(A) \quad P(\emptyset) \quad \begin{array}{c}[a \in A \quad b \in \mathtt{Fin}(A) \quad P(b)]_{a,b} \\ \vdots \\ P(\{a\} \cup b)\end{array}}{P(x)}$$

Stronger induction rules often suggest themselves. We can derive a rule for $\mathtt{Fin}(A)$ whose third premise discharges the extra assumption $a \not\in b$. The package provides rules for mutual induction and inductive relations. The Isabelle/ZF theory also supports well-founded induction and recursion over datatypes, by reasoning about the **rank** of a set [27, §3.4].

4.2 Modified induction rules

If the domain of R is a Cartesian product $A_1 \times \cdots \times A_m$ (however nested), then the corresponding predicate P_i takes m arguments. The major premise becomes $\langle z_1, \ldots, z_m \rangle \in R$ instead of $x \in R$; the conclusion becomes $P(z_1, \ldots, z_m)$. This simplifies reasoning about inductively defined relations, eliminating the need to express properties of $z_1, \ldots, z_m$ as properties of the tuple $\langle z_1, \ldots, z_m \rangle$. Occasionally it may require you to split up the induction variable using `SigmaE` and `dom_subset`, especially if the constant `split` appears in the rule.

The mutual induction rule is called `mutual_induct`. It differs from the basic rule in two respects:

- Instead of a single predicate P, it uses n predicates $P_1, \ldots, P_n$: one for each recursive set.

- There is no major premise such as $x \in R_i$. Instead, the conclusion refers to all the recursive sets:

  $$(\forall z \,.\, z \in R_1 \to P_1(z)) \wedge \cdots \wedge (\forall z \,.\, z \in R_n \to P_n(z))$$

  Proving the premises establishes $P_i(z)$ for $z \in R_i$ and $i = 1, \ldots, n$.

If the domain of some R_i is a Cartesian product, then the mutual induction rule is modified accordingly. The predicates are made to take m separate arguments instead of a tuple, and the quantification in the conclusion is over the separate variables $z_1, \ldots, z_m$.

4.3 Coinduction

A coinductive definition yields a primitive coinduction rule, with no refinements such as those for the induction rules. (Experience may suggest refinements later.)

Consider the codatatype of lazy lists as an example. For suitable definitions of LNil and LCons, lazy lists may be defined as the greatest set consistent with the rules

$$\texttt{LNil} \in \texttt{llist}(A) \qquad \frac{a \in A \quad l \in \texttt{llist}(A)}{\texttt{LCons}(a, l) \in \texttt{llist}(A)}\ (-)$$

The $(-)$ tag stresses that this is a coinductive definition. A suitable domain for $\texttt{llist}(A)$ is $\texttt{quniv}(A)$; this set is closed under the variant forms of sum and product that are used to represent non-well-founded data structures (see §6.1).

The package derives an unfold theorem similar to that for $\texttt{Fin}(A)$. Then it proves the theorem coinduct, which expresses that $\texttt{llist}(A)$ is the greatest solution to this equation contained in $\texttt{quniv}(A)$:

$$\frac{x \in X \quad X \subseteq \texttt{quniv}(A) \quad \begin{matrix} [z \in X]_z \\ \vdots \\ z = \texttt{LNil} \lor (\exists a\, l.\, z = \texttt{LCons}(a, l) \land a \in A \land \\ l \in X \cup \texttt{llist}(A)) \end{matrix}}{x \in \texttt{llist}(A)}$$

This rule complements the introduction rules; it provides a means of showing $x \in \texttt{llist}(A)$ when x is infinite. For instance, if $x = \texttt{LCons}(0, x)$ then applying the rule with $X = \{x\}$ proves $x \in \texttt{llist}(\texttt{nat})$. (Here nat is the set of natural numbers.)

Having $X \cup \texttt{llist}(A)$ instead of simply X in the third premise above represents a slight strengthening of the greatest fixedpoint property. I discuss several forms of coinduction rules elsewhere [28].

The clumsy form of the third premise makes the rule hard to use, especially in large definitions. Probably a constant should be declared to abbreviate the large disjunction, and rules derived to allow proving the separate disjuncts.

5 Examples of inductive and coinductive definitions

This section presents several examples from the literature: the finite powerset operator, lists of n elements, bisimulations on lazy lists, the well-founded part of a relation, and the primitive recursive functions.

5.1 The finite powerset operator

This operator has been discussed extensively above. Here is the corresponding invocation in an Isabelle theory file. Note that $\texttt{cons}(a, b)$ abbreviates $\{a\} \cup b$ in

Isabelle/ZF.

```
Finite = Arith +
consts      Fin :: i=>i
inductive
  domains   "Fin(A)" <= "Pow(A)"
  intrs
    emptyI  "0 : Fin(A)"
    consI   "[| a: A;  b: Fin(A) |] ==> cons(a,b) : Fin(A)"
  type_intrs "[empty_subsetI, cons_subsetI, PowI]"
  type_elims "[make_elim PowD]"
end
```

Theory `Finite` extends the parent theory `Arith` by declaring the unary function symbol `Fin`, which is defined inductively. Its domain is specified as $\mathcal{P}(A)$, where A is the parameter appearing in the introduction rules. For type-checking, we supply two introduction rules:

$$\emptyset \subseteq A \qquad \frac{a \in C \quad B \subseteq C}{\{a\} \cup B \subseteq C}$$

A further introduction rule and an elimination rule express both directions of the equivalence $A \in \mathcal{P}(B) \leftrightarrow A \subseteq B$. Type-checking involves mostly introduction rules.

Like all Isabelle theory files, this one yields a structure containing the new theory as an ML value. Structure `Finite` also has a substructure, called `Fin`. After declaring `open Finite;` we can refer to the $\texttt{Fin}(A)$ introduction rules as the list `Fin.intrs` or individually as `Fin.emptyI` and `Fin.consI`. The induction rule is `Fin.induct`.

5.2 Lists of n elements

This has become a standard example of an inductive definition. Following Paulin-Mohring [22], we could attempt to define a new datatype $\texttt{listn}(A, n)$, for lists of length n, as an n-indexed family of sets. But her introduction rules

$$\texttt{Niln} \in \texttt{listn}(A, 0) \qquad \frac{n \in \texttt{nat} \quad a \in A \quad l \in \texttt{listn}(A, n)}{\texttt{Consn}(n, a, l) \in \texttt{listn}(A, \texttt{succ}(n))}$$

are not acceptable to the inductive definition package: `listn` occurs with three different parameter lists in the definition.

The Isabelle version of this example suggests a general treatment of varying parameters. It uses the existing datatype definition of $\texttt{list}(A)$, with constructors `Nil` and `Cons`, and incorporates the parameter n into the inductive set itself. It defines $\texttt{listn}(A)$ as a relation consisting of pairs $\langle n, l\rangle$ such that $n \in \texttt{nat}$ and $l \in \texttt{list}(A)$ and l has length n. In fact, $\texttt{listn}(A)$ is the converse of the length function on $\texttt{list}(A)$. The Isabelle/ZF introduction rules are

$$\langle 0, \texttt{Nil}\rangle \in \texttt{listn}(A) \qquad \frac{a \in A \quad \langle n, l\rangle \in \texttt{listn}(A)}{\langle \texttt{succ}(n), \texttt{Cons}(a,l)\rangle \in \texttt{listn}(A)}$$

The Isabelle theory file takes, as parent, the theory `List` of lists. We declare the constant `listn` and supply an inductive definition, specifying the domain as $\texttt{nat} \times \texttt{list}(A)$:

```
ListN = List +
consts  listn :: i=>i
inductive
  domains   "listn(A)" <= "nat*list(A)"
  intrs
    NilI  "<0,Nil>: listn(A)"
    ConsI "[| a: A;  <n,l>: listn(A) |] ==> <succ(n), Cons(a,l)>: listn(A)"
  type_intrs "nat_typechecks @ list.intrs"
end
```

The type-checking rules include those for 0, `succ`, `Nil` and `Cons`. Because $\texttt{listn}(A)$ is a set of pairs, type-checking requires the equivalence $\langle a, b\rangle \in A\times B \leftrightarrow a \in A \wedge b \in B$. The package always includes the rules for ordered pairs.

The package returns introduction, elimination and induction rules for `listn`. The basic induction rule, `listn.induct`, is

$$\frac{\langle z_1, z_2\rangle \in \texttt{listn}(A) \quad P(0, \texttt{Nil}) \quad \begin{array}{c}[a \in A \quad \langle n, l\rangle \in \texttt{listn}(A) \quad P(n,l)]_{a,l,n} \\ \vdots \\ P(\texttt{succ}(n), \texttt{Cons}(a,l))\end{array}}{P(z_1, z_2)}$$

This rule lets the induction formula to be a binary property of pairs, $P(n, l)$. It is now a simple matter to prove theorems about $\texttt{listn}(A)$, such as

$$\forall l \in \texttt{list}(A)\,.\, \langle \texttt{length}(l),\ l\rangle \in \texttt{listn}(A)$$

$$\texttt{listn}(A)``\{n\} = \{l \in \texttt{list}(A)\,.\, \texttt{length}(l) = n\}$$

This latter result — here $r``X$ denotes the image of X under r — asserts that the inductive definition agrees with the obvious notion of n-element list.

A "list of n elements" really is a list, namely an element of $\texttt{list}(A)$. It is subject to list operators such as append (concatenation). For example, a trivial induction on $\langle m, l\rangle \in \texttt{listn}(A)$ yields

$$\frac{\langle m, l\rangle \in \texttt{listn}(A) \quad \langle m', l'\rangle \in \texttt{listn}(A)}{\langle m + m',\ l@l'\rangle \in \texttt{listn}(A)}$$

where $+$ denotes addition on the natural numbers and @ denotes append.

5.3 Rule inversion: the function mk_cases

The elimination rule, listn.elim, is cumbersome:

$$\frac{x \in \texttt{listn}(A) \qquad \begin{matrix}[x = \langle 0, \texttt{Nil}\rangle] \\ \vdots \\ Q\end{matrix} \qquad \begin{matrix}\left[\begin{matrix} x = \langle \texttt{succ}(n), \texttt{Cons}(a, l)\rangle \\ a \in A \\ \langle n, l\rangle \in \texttt{listn}(A)\end{matrix}\right]_{a,l,n} \\ \vdots \\ Q\end{matrix}}{Q}$$

The ML function listn.mk_cases generates simplified instances of this rule. It works by freeness reasoning on the list constructors: $\texttt{Cons}(a, l)$ is injective in its two arguments and differs from Nil. If x is $\langle i, \texttt{Nil}\rangle$ or $\langle i, \texttt{Cons}(a, l)\rangle$ then listn.mk_cases deduces the corresponding form of i; this is called rule inversion. Here is a sample session:

```
listn.mk_cases list.con_defs "<i,Nil> : listn(A)";
  "[| <?i, []> : listn(?A); ?i = 0 ==> ?Q |] ==> ?Q" : thm
listn.mk_cases list.con_defs "<i,Cons(a,l)> : listn(A)";
  "[| <?i, Cons(?a, ?l)> : listn(?A);
      !!n. [| ?a : ?A; <n, ?l> : listn(?A); ?i = succ(n) |] ==> ?Q
   |] ==> ?Q" : thm
```

Each of these rules has only two premises. In conventional notation, the second rule is

$$\frac{\langle i, \texttt{Cons}(a, l)\rangle \in \texttt{listn}(A) \qquad \begin{matrix}\left[\begin{matrix} a \in A \\ \langle n, l\rangle \in \texttt{listn}(A) \\ i = \texttt{succ}(n)\end{matrix}\right]_{n} \\ \vdots \\ Q\end{matrix}}{Q}$$

The package also has built-in rules for freeness reasoning about 0 and succ. So if x is $\langle 0, l\rangle$ or $\langle \texttt{succ}(i), l\rangle$, then listn.mk_cases can deduce the corresponding form of l.

The function mk_cases is also useful with datatype definitions. The instance from the definition of lists, namely list.mk_cases, can prove that $\texttt{Cons}(a, l) \in \texttt{list}(A)$ implies $a \in A$ and $l \in \texttt{list}(A)$:

$$\frac{\texttt{Cons}(a, l) \in \texttt{list}(A) \qquad \begin{matrix}[a \in A \quad l \in \texttt{list}(A)] \\ \vdots \\ Q\end{matrix}}{Q}$$

A typical use of mk_cases concerns inductive definitions of evaluation relations. Then rule inversion yields case analysis on possible evaluations. For example, Isabelle/ZF includes a short proof of the diamond property for parallel contraction on combinators. Ole Rasmussen used `mk_cases` extensively in his development of the theory of residuals [32].

5.4 A coinductive definition: bisimulations on lazy lists

This example anticipates the definition of the codatatype $\texttt{llist}(A)$, which consists of finite and infinite lists over A. Its constructors are `LNil` and `LCons`, satisfying the introduction rules shown in §4.3. Because $\texttt{llist}(A)$ is defined as a greatest fixedpoint and uses the variant pairing and injection operators, it contains non-well-founded elements such as solutions to $\texttt{LCons}(a, l) = l$.

The next step in the development of lazy lists is to define a coinduction principle for proving equalities. This is done by showing that the equality relation on lazy lists is the greatest fixedpoint of some monotonic operation. The usual approach [31] is to define some notion of bisimulation for lazy lists, define equivalence to be the greatest bisimulation, and finally to prove that two lazy lists are equivalent if and only if they are equal. The coinduction rule for equivalence then yields a coinduction principle for equalities.

A binary relation R on lazy lists is a **bisimulation** provided $R \subseteq R^+$, where R^+ is the relation

$$\{\langle \texttt{LNil}, \texttt{LNil}\rangle\} \cup \{\langle \texttt{LCons}(a, l), \texttt{LCons}(a, l')\rangle \;.\; a \in A \land \langle l, l'\rangle \in R\}.$$

A pair of lazy lists are **equivalent** if they belong to some bisimulation. Equivalence can be coinductively defined as the greatest fixedpoint for the introduction rules

$$\langle \texttt{LNil}, \texttt{LNil}\rangle \in \texttt{lleq}(A) \qquad \frac{a \in A \quad \langle l, l'\rangle \in \texttt{lleq}(A)}{\langle \texttt{LCons}(a, l), \texttt{LCons}(a, l')\rangle \in \texttt{lleq}(A)}\ (-)$$

To make this coinductive definition, the theory file includes (after the declaration of $\texttt{llist}(A)$) the following lines:

```
consts    lleq :: i=>i
coinductive
  domains "lleq(A)" <= "llist(A) * llist(A)"
  intrs
    LNil  "<LNil,LNil> : lleq(A)"
    LCons "[| a:A; <l,l'>: lleq(A) |] ==> <LCons(a,l),LCons(a,l')>: lleq(A)"
  type_intrs  "llist.intrs"
```

The domain of $\texttt{lleq}(A)$ is $\texttt{llist}(A) \times \texttt{llist}(A)$. The type-checking rules include the introduction rules for $\texttt{llist}(A)$, whose declaration is discussed below (§6.3).

The package returns the introduction rules and the elimination rule, as usual. But instead of induction rules, it returns a coinduction rule. The rule is too big to display in the usual notation; its conclusion is $x \in \texttt{lleq}(A)$ and its premises are $x \in X$, $X \subseteq \texttt{llist}(A) \times \texttt{llist}(A)$ and

$$\begin{array}{c} [z \in X]_z \\ \vdots \\ z = \langle \texttt{LNil}, \texttt{LNil} \rangle \lor (\exists a\, l\, l' \,.\; z = \langle \texttt{LCons}(a,l), \texttt{LCons}(a,l') \rangle \land a \in A \land \\ \langle l, l' \rangle \in X \cup \texttt{lleq}(A)) \end{array}$$

Thus if $x \in X$, where X is a bisimulation contained in the domain of $\texttt{lleq}(A)$, then $x \in \texttt{lleq}(A)$. It is easy to show that $\texttt{lleq}(A)$ is reflexive: the equality relation is a bisimulation. And $\texttt{lleq}(A)$ is symmetric: its converse is a bisimulation. But showing that $\texttt{lleq}(A)$ coincides with the equality relation takes some work.

5.5 The accessible part of a relation

Let $\prec$ be a binary relation on D; in short, $(\prec) \subseteq D \times D$. The **accessible** or **well-founded** part of $\prec$, written $\texttt{acc}(\prec)$, is essentially that subset of D for which $\prec$ admits no infinite decreasing chains [2]. Formally, $\texttt{acc}(\prec)$ is inductively defined to be the least set that contains a if it contains all $\prec$-predecessors of a, for $a \in D$. Thus we need an introduction rule of the form

$$\frac{\forall y \,.\; y \prec a \to y \in \texttt{acc}(\prec)}{a \in \texttt{acc}(\prec)}$$

Paulin-Mohring treats this example in Coq [22], but it causes difficulties for other systems. Its premise is not acceptable to the inductive definition package of the Cambridge HOL system [5]. It is also unacceptable to the Isabelle package (recall §3.1), but fortunately can be transformed into the acceptable form $t \in M(R)$.

The powerset operator is monotonic, and $t \in \mathcal{P}(R)$ is equivalent to $t \subseteq R$. This in turn is equivalent to $\forall y \in t \,.\; y \in R$. To express $\forall y \,.\; y \prec a \to y \in \texttt{acc}(\prec)$ we need only find a term t such that $y \in t$ if and only if $y \prec a$. A suitable t is the inverse image of $\{a\}$ under $\prec$.

The definition below follows this approach. Here r is $\prec$ and $\texttt{field}(r)$ refers to D, the domain of $\texttt{acc}(r)$. (The field of a relation is the union of its domain and range.) Finally $r^{-}``\{a\}$ denotes the inverse image of $\{a\}$ under r. We supply the theorem Pow_mono, which asserts that $\mathcal{P}$ is monotonic.

```
consts    acc :: i=>i
inductive
  domains "acc(r)" <= "field(r)"
  intrs
    vimage  "[| r-``{a}: Pow(acc(r)); a: field(r) |] ==> a: acc(r)"
  monos     "[Pow_mono]"
```

The Isabelle theory proceeds to prove facts about $\texttt{acc}(\prec)$. For instance, $\prec$ is well-founded if and only if its field is contained in $\texttt{acc}(\prec)$.

As mentioned in §4.1, a premise of the form $t \in M(R)$ gives rise to an unusual induction hypothesis. Let us examine the induction rule, `acc.induct`:

$$\frac{x \in \texttt{acc}(r) \qquad \begin{array}{c}\left[\begin{array}{c} r^{-}``\{a\} \in \mathcal{P}(\{z \in \texttt{acc}(r) \,.\, P(z)\}) \\ a \in \texttt{field}(r) \end{array}\right]_a \\ \vdots \\ P(a)\end{array}}{P(x)}$$

The strange induction hypothesis is equivalent to $\forall y \,.\, \langle y, a\rangle \in r \to y \in \texttt{acc}(r) \wedge P(y)$. Therefore the rule expresses well-founded induction on the accessible part of $\prec$.

The use of inverse image is not essential. The Isabelle package can accept introduction rules with arbitrary premises of the form $\forall \vec{y} \,.\, P(\vec{y}) \to f(\vec{y}) \in R$. The premise can be expressed equivalently as

$$\{z \in D \,.\, P(\vec{y}) \wedge z = f(\vec{y})\} \in \mathcal{P}(R)$$

provided $f(\vec{y}) \in D$ for all $\vec{y}$ such that $P(\vec{y})$. The following section demonstrates another use of the premise $t \in M(R)$, where $M = \texttt{list}$.

5.6 The primitive recursive functions

The primitive recursive functions are traditionally defined inductively, as a subset of the functions over the natural numbers. One difficulty is that functions of all arities are taken together, but this is easily circumvented by regarding them as functions on lists. Another difficulty, the notion of composition, is less easily circumvented.

Here is a more precise definition. Letting $\vec{x}$ abbreviate $x_0, \ldots, x_{n-1}$, we can write lists such as $[\vec{x}]$, $[y+1, \vec{x}]$, etc. A function is **primitive recursive** if it belongs to the least set of functions in $\texttt{list(nat)} \to \texttt{nat}$ containing

- The **successor** function SC, such that $\texttt{SC}[y, \vec{x}] = y + 1$.
- All **constant** functions $\texttt{CONST}(k)$, such that $\texttt{CONST}(k)[\vec{x}] = k$.
- All **projection** functions $\texttt{PROJ}(i)$, such that $\texttt{PROJ}(i)[\vec{x}] = x_i$ if $0 \le i < n$.
- All **compositions** $\texttt{COMP}(g, [f_0, \ldots, f_{m-1}])$, where g and $f_0, \ldots, f_{m-1}$ are primitive recursive, such that

  $$\texttt{COMP}(g, [f_0, \ldots, f_{m-1}])[\vec{x}] = g[f_0[\vec{x}], \ldots, f_{m-1}[\vec{x}]].$$

- All **recursions** $\texttt{PREC}(f, g)$, where f and g are primitive recursive, such that

  $$\begin{aligned} \texttt{PREC}(f, g)[0, \vec{x}] &= f[\vec{x}] \\ \texttt{PREC}(f, g)[y+1, \vec{x}] &= g[\texttt{PREC}(f, g)[y, \vec{x}],\ y,\ \vec{x}]. \end{aligned}$$

```
Primrec = List +
consts
  primrec :: i
  SC      :: i
  ⋮
defs
  SC_def    "SC == lam l:list(nat).list_case(0, %x xs.succ(x), l)"
  ⋮
inductive
  domains "primrec" <= "list(nat)->nat"
  intrs
    SC       "SC : primrec"
    CONST    "k: nat ==> CONST(k) : primrec"
    PROJ     "i: nat ==> PROJ(i) : primrec"
    COMP     "[| g: primrec; fs: list(primrec) |] ==> COMP(g,fs): primrec"
    PREC     "[| f: primrec; g: primrec |] ==> PREC(f,g): primrec"
  monos      "[list_mono]"
  con_defs   "[SC_def,CONST_def,PROJ_def,COMP_def,PREC_def]"
  type_intrs "nat_typechecks @ list.intrs @
              [lam_type, list_case_type, drop_type, map_type,
               apply_type, rec_type]"
end
```

Figure 1: Inductive definition of the primitive recursive functions

Composition is awkward because it combines not two functions, as is usual, but $m+1$ functions. In her proof that Ackermann's function is not primitive recursive, Nora Szasz was unable to formalize this definition directly [35]. So she generalized primitive recursion to tuple-valued functions. This modified the inductive definition such that each operation on primitive recursive functions combined just two functions.

Szasz was using ALF, but Coq and HOL would also have problems accepting this definition. Isabelle's package accepts it easily since $[f_0, \ldots, f_{m-1}]$ is a list of primitive recursive functions and `list` is monotonic. There are five introduction rules, one for each of the five forms of primitive recursive function. Let us examine the one for `COMP`:

$$\frac{g \in \texttt{primrec} \quad fs \in \texttt{list}(\texttt{primrec})}{\texttt{COMP}(g, fs) \in \texttt{primrec}}$$

The induction rule for `primrec` has one case for each introduction rule. Due to the use of `list` as a monotone operator, the composition case has an unusual induction hypothesis:

$$\begin{array}{c} [g \in \texttt{primrec} \quad fs \in \texttt{list}(\{z \in \texttt{primrec} \,.\, P(z)\})]_{fs,g} \\ \vdots \\ P(\texttt{COMP}(g, fs)) \end{array}$$

The hypothesis states that fs is a list of primitive recursive functions, each satisfying the induction formula. Proving the COMP case typically requires structural induction on lists, yielding two subcases: either $fs = \texttt{Nil}$ or else $fs = \texttt{Cons}(f, fs')$, where $f \in \texttt{primrec}$, $P(f)$, and fs' is another list of primitive recursive functions satisfying P.

Figure 1 presents the theory file. Theory Primrec defines the constants SC, CONST, etc. These are not constructors of a new datatype, but functions over lists of numbers. Their definitions, most of which are omitted, consist of routine list programming. In Isabelle/ZF, the primitive recursive functions are defined as a subset of the function set $\texttt{list(nat)} \to \texttt{nat}$.

The Isabelle theory goes on to formalize Ackermann's function and prove that it is not primitive recursive, using the induction rule primrec.induct. The proof follows Szasz's excellent account.

6 Datatypes and codatatypes

A (co)datatype definition is a (co)inductive definition with automatically defined constructors and a case analysis operator. The package proves that the case operator inverts the constructors and can prove freeness theorems involving any pair of constructors.

6.1 Constructors and their domain

A (co)inductive definition selects a subset of an existing set; a (co)datatype definition creates a new set. The package reduces the latter to the former. Isabelle/ZF supplies sets having strong closure properties to serve as domains for (co)inductive definitions.

Isabelle/ZF defines the Cartesian product $A \times B$, containing ordered pairs $\langle a, b \rangle$; it also defines the disjoint sum $A + B$, containing injections $\texttt{Inl}(a) \equiv \langle 0, a \rangle$ and $\texttt{Inr}(b) \equiv \langle 1, b \rangle$. For use below, define the m-tuple $\langle x_1, \ldots, x_m \rangle$ to be the empty set $\emptyset$ if $m = 0$, simply x_1 if $m = 1$ and $\langle x_1, \langle x_2, \ldots, x_m \rangle \rangle$ if $m \geq 2$.

A datatype constructor $\texttt{Con}(x_1, \ldots, x_m)$ is defined to be $h(\langle x_1, \ldots, x_m \rangle)$, where h is composed of Inl and Inr. In a mutually recursive definition, all constructors for the set R_i have the outer form h_{in}, where h_{in} is the injection described in §3.3. Further nested injections ensure that the constructors for R_i are pairwise distinct.

Isabelle/ZF defines the set $\texttt{univ}(A)$, which contains A and furthermore contains $\langle a, b \rangle$, $\texttt{Inl}(a)$ and $\texttt{Inr}(b)$ for a, $b \in \texttt{univ}(A)$. In a typical datatype definition with set parameters A_1, …, A_k, a suitable domain for all the recursive sets is $\texttt{univ}(A_1 \cup \cdots \cup A_k)$. This solves the problem for datatypes [27, §4.2].

The standard pairs and injections can only yield well-founded constructions. This eases the (manual!) definition of recursive functions over datatypes. But they are unsuitable for codatatypes, which typically contain non-well-founded objects.

To support codatatypes, Isabelle/ZF defines a variant notion of ordered pair, written $\langle a; b\rangle$. It also defines the corresponding variant notion of Cartesian product $A \otimes B$, variant injections $\mathtt{QInl}(a)$ and $\mathtt{QInr}(b)$ and variant disjoint sum $A \oplus B$. Finally it defines the set $\mathtt{quniv}(A)$, which contains A and furthermore contains $\langle a; b\rangle$, $\mathtt{QInl}(a)$ and $\mathtt{QInr}(b)$ for a, $b \in \mathtt{quniv}(A)$. In a typical codatatype definition with set parameters A_1, …, A_k, a suitable domain is $\mathtt{quniv}(A_1 \cup \cdots \cup A_k)$. Details are published elsewhere [26].

6.2 The case analysis operator

The (co)datatype package automatically defines a case analysis operator, called R_`case`. A mutually recursive definition still has only one operator, whose name combines those of the recursive sets: it is called R_1_…_R_n_`case`. The case operator is analogous to those for products and sums.

Datatype definitions employ standard products and sums, whose operators are `split` and `case` and satisfy the equations

$$\begin{aligned}
\mathtt{split}(f, \langle x, y\rangle) &= f(x, y) \\
\mathtt{case}(f, g, \mathtt{Inl}(x)) &= f(x) \\
\mathtt{case}(f, g, \mathtt{Inr}(y)) &= g(y)
\end{aligned}$$

Suppose the datatype has k constructors $\mathtt{Con}_1$, …, $\mathtt{Con}_k$. Then its case operator takes $k + 1$ arguments and satisfies an equation for each constructor:

$$R\_\mathrm{case}(f_1, \ldots, f_k, \mathtt{Con}_i(\vec{x})) = f_i(\vec{x}), \qquad i = 1, \ldots, k$$

The case operator's definition takes advantage of Isabelle's representation of syntax in the typed λ-calculus; it could readily be adapted to a theorem prover for higher-order logic. If f and g have meta-type $i \Rightarrow i$ then so do $\mathtt{split}(f)$ and $\mathtt{case}(f, g)$. This works because `split` and `case` operate on their last argument. They are easily combined to make complex case analysis operators. For example, $\mathtt{case}(f, \mathtt{case}(g, h))$ performs case analysis for $A + (B + C)$; let us verify one of the three equations:

$$\mathtt{case}(f, \mathtt{case}(g, h), \mathtt{Inr}(\mathtt{Inl}(b))) = \mathtt{case}(g, h, \mathtt{Inl}(b)) = g(b)$$

Codatatype definitions are treated in precisely the same way. They express case operators using those for the variant products and sums, namely `qsplit` and `qcase`.

To see how constructors and the case analysis operator are defined, let us examine some examples. Further details are available elsewhere [27].

6.3 Example: lists and lazy lists

Here is a declaration of the datatype of lists, as it might appear in a theory file:

```
consts  list :: i=>i
datatype "list(A)" = Nil | Cons ("a:A", "l: list(A)")
```

And here is a declaration of the codatatype of lazy lists:

```
consts  llist :: i=>i
codatatype "llist(A)" = LNil | LCons ("a: A", "l: llist(A)")
```

Each form of list has two constructors, one for the empty list and one for adding an element to a list. Each takes a parameter, defining the set of lists over a given set A. Each is automatically given the appropriate domain: $\mathtt{univ}(A)$ for $\mathtt{list}(A)$ and $\mathtt{quniv}(A)$ for $\mathtt{llist}(A)$. The default can be overridden.

Since $\mathtt{list}(A)$ is a datatype, it enjoys a structural induction rule, `list.induct`:

$$\frac{x \in \mathtt{list}(A) \quad P(\mathtt{Nil}) \quad \begin{array}{c}[a \in A \quad l \in \mathtt{list}(A) \quad P(l)]_{a,l} \\ \vdots \\ P(\mathtt{Cons}(a,l))\end{array}}{P(x)}$$

Induction and freeness yield the law $l \neq \mathtt{Cons}(a,l)$. To strengthen this, Isabelle/ZF defines the rank of a set and proves that the standard pairs and injections have greater rank than their components. An immediate consequence, which justifies structural recursion on lists [27, §4.3], is

$$\mathtt{rank}(l) < \mathtt{rank}(\mathtt{Cons}(a,l)).$$

But $\mathtt{llist}(A)$ is a codatatype and has no induction rule. Instead it has the coinduction rule shown in §4.3. Since variant pairs and injections are monotonic and need not have greater rank than their components, fixedpoint operators can create cyclic constructions. For example, the definition

$$\mathtt{lconst}(a) \equiv \mathtt{lfp}(\mathtt{univ}(a), \lambda l \,.\, \mathtt{LCons}(a,l))$$

yields $\mathtt{lconst}(a) = \mathtt{LCons}(a, \mathtt{lconst}(a))$.

It may be instructive to examine the definitions of the constructors and case operator for $\mathtt{list}(A)$. The definitions for $\mathtt{llist}(A)$ are similar. The list constructors are defined as follows:

$$\begin{array}{rcl} \mathtt{Nil} & \equiv & \mathtt{Inl}(\emptyset) \\ \mathtt{Cons}(a,l) & \equiv & \mathtt{Inr}(\langle a,l\rangle) \end{array}$$

The operator `list_case` performs case analysis on these two alternatives:

$$\mathtt{list\_case}(c,h) \equiv \mathtt{case}(\lambda u \,.\, c, \mathtt{split}(h))$$

Let us verify the two equations:

$$
\begin{aligned}
\texttt{list\_case}(c, h, \texttt{Nil}) &= \texttt{case}(\lambda u \,.\, c, \texttt{split}(h), \texttt{Inl}(\emptyset)) \\
&= (\lambda u \,.\, c)(\emptyset) \\
&= c \\
\texttt{list\_case}(c, h, \texttt{Cons}(x, y)) &= \texttt{case}(\lambda u \,.\, c, \texttt{split}(h), \texttt{Inr}(\langle x, y\rangle)) \\
&= \texttt{split}(h, \langle x, y\rangle) \\
&= h(x, y)
\end{aligned}
$$

6.4 Example: mutual recursion

In mutually recursive trees and forests [27, §4.5], trees have the one constructor `Tcons`, while forests have the two constructors `Fnil` and `Fcons`:

```
consts  tree, forest, tree_forest    :: i=>i
datatype "tree(A)"   = Tcons ("a: A",  "f: forest(A)")
and      "forest(A)" = Fnil  |  Fcons ("t: tree(A)",  "f: forest(A)")
```

The three introduction rules define the mutual recursion. The distinguishing feature of this example is its two induction rules.

The basic induction rule is called `tree_forest.induct`:

$$
\frac{x \in \texttt{tree\_forest}(A) \qquad \begin{array}{c}\left[\begin{array}{l} a \in A \\ f \in \texttt{forest}(A) \\ P(f) \end{array}\right]_{a,f} \\ \vdots \\ P(\texttt{Tcons}(a, f))\end{array} \qquad P(\texttt{Fnil}) \qquad \begin{array}{c}\left[\begin{array}{l} t \in \texttt{tree}(A) \\ P(t) \\ f \in \texttt{forest}(A) \\ P(f) \end{array}\right]_{t,f} \\ \vdots \\ P(\texttt{Fcons}(t, f))\end{array}}{P(x)}
$$

This rule establishes a single predicate for $\texttt{tree\_forest}(A)$, the union of the recursive sets. Although such reasoning is sometimes useful [27, §4.5], a proper mutual induction rule should establish separate predicates for $\texttt{tree}(A)$ and $\texttt{forest}(A)$. The package calls this rule `tree_forest.mutual_induct`. Observe the usage of P and Q in the induction hypotheses:

$$
\frac{\begin{array}{c}\left[\begin{array}{l} a \in A \\ f \in \texttt{forest}(A) \\ Q(f) \end{array}\right]_{a,f} \\ \vdots \\ P(\texttt{Tcons}(a, f))\end{array} \qquad Q(\texttt{Fnil}) \qquad \begin{array}{c}\left[\begin{array}{l} t \in \texttt{tree}(A) \\ P(t) \\ f \in \texttt{forest}(A) \\ Q(f) \end{array}\right]_{t,f} \\ \vdots \\ Q(\texttt{Fcons}(t, f))\end{array}}{(\forall z \,.\, z \in \texttt{tree}(A) \rightarrow P(z)) \land (\forall z \,.\, z \in \texttt{forest}(A) \rightarrow Q(z))}
$$

Elsewhere I describe how to define mutually recursive functions over trees and forests [27, §4.5].

Both forest constructors have the form $\mathtt{Inr}(\cdots)$, while the tree constructor has the form $\mathtt{Inl}(\cdots)$. This pattern would hold regardless of how many tree or forest constructors there were.

$$\begin{array}{rcl} \mathtt{Tcons}(a,l) & \equiv & \mathtt{Inl}(\langle a,l\rangle) \\ \mathtt{Fnil} & \equiv & \mathtt{Inr}(\mathtt{Inl}(\emptyset)) \\ \mathtt{Fcons}(a,l) & \equiv & \mathtt{Inr}(\mathtt{Inr}(\langle a,l\rangle)) \end{array}$$

There is only one case operator; it works on the union of the trees and forests:

$$\mathtt{tree\_forest\_case}(f,c,g) \equiv \mathtt{case}(\mathtt{split}(f),\ \mathtt{case}(\lambda u\,.\,c, \mathtt{split}(g)))$$

6.5 Example: a four-constructor datatype

A bigger datatype will illustrate some efficiency refinements. It has four constructors $\mathtt{Con}_0$, ..., $\mathtt{Con}_3$, with the corresponding arities.

```
consts    data :: [i,i] => i
datatype  "data(A,B)" = Con0
                      | Con1 ("a: A")
                      | Con2 ("a: A", "b: B")
                      | Con3 ("a: A", "b: B", "d: data(A,B)")
```

Because this datatype has two set parameters, A and B, the package automatically supplies $\mathtt{univ}(A \cup B)$ as its domain. The structural induction rule has four minor premises, one per constructor, and only the last has an induction hypothesis. (Details are left to the reader.)

The constructors are defined by the equations

$$\begin{array}{rcl} \mathtt{Con}_0 & \equiv & \mathtt{Inl}(\mathtt{Inl}(\emptyset)) \\ \mathtt{Con}_1(a) & \equiv & \mathtt{Inl}(\mathtt{Inr}(a)) \\ \mathtt{Con}_2(a,b) & \equiv & \mathtt{Inr}(\mathtt{Inl}(\langle a,b\rangle)) \\ \mathtt{Con}_3(a,b,c) & \equiv & \mathtt{Inr}(\mathtt{Inr}(\langle a,b,c\rangle)). \end{array}$$

The case analysis operator is

$$\begin{array}{l} \mathtt{data\_case}(f_0,f_1,f_2,f_3) \equiv \mathtt{case}(\mathtt{case}(\lambda u\,.\,f_0,\ f_1), \\ \qquad\qquad\qquad \mathtt{case}(\mathtt{split}(f_2),\ \mathtt{split}(\lambda v\,.\,\mathtt{split}(f_3(v))))) \end{array}$$

This may look cryptic, but the case equations are trivial to verify.

In the constructor definitions, the injections are balanced. A more naive approach is to define $\mathtt{Con}_3(a,b,c)$ as $\mathtt{Inr}(\mathtt{Inr}(\mathtt{Inr}(\langle a,b,c\rangle)))$; instead, each constructor has two injections. The difference here is small. But the ZF examples include a 60-element enumeration type, where each constructor has 5 or 6 injections. The naive approach would require 1 to 59 injections; the definitions would be quadratic in size. It is like the advantage of binary notation over unary.

The result structure contains the case operator and constructor definitions as the theorem list `con_defs`. It contains the case equations, such as

$$\texttt{data\_case}(f_0, f_1, f_2, f_3, \texttt{Con}_3(a, b, c)) = f_3(a, b, c),$$

as the theorem list `case_eqns`. There is one equation per constructor.

6.6 Proving freeness theorems

There are two kinds of freeness theorems:

- **injectiveness** theorems, such as

$$\texttt{Con}_2(a, b) = \texttt{Con}_2(a', b') \leftrightarrow a = a' \land b = b'$$

- **distinctness** theorems, such as

$$\texttt{Con}_1(a) \neq \texttt{Con}_2(a', b')$$

Since the number of such theorems is quadratic in the number of constructors, the package does not attempt to prove them all. Instead it returns tools for proving desired theorems — either manually or during simplification or classical reasoning.

The theorem list `free_iffs` enables the simplifier to perform freeness reasoning. This works by incremental unfolding of constructors that appear in equations. The theorem list contains logical equivalences such as

$$\begin{aligned}
\texttt{Con}_0 = c &\quad\leftrightarrow\quad c = \texttt{Inl}(\texttt{Inl}(\emptyset)) \\
\texttt{Con}_1(a) = c &\quad\leftrightarrow\quad c = \texttt{Inl}(\texttt{Inr}(a)) \\
&\quad\vdots \\
\texttt{Inl}(a) = \texttt{Inl}(b) &\quad\leftrightarrow\quad a = b \\
\texttt{Inl}(a) = \texttt{Inr}(b) &\quad\leftrightarrow\quad \texttt{False} \\
\langle a, b\rangle = \langle a', b'\rangle &\quad\leftrightarrow\quad a = a' \land b = b'
\end{aligned}$$

For example, these rewrite $\texttt{Con}_1(a) = \texttt{Con}_1(b)$ to $a = b$ in four steps.

The theorem list `free_SEs` enables the classical reasoner to perform similar replacements. It consists of elimination rules to replace $\texttt{Con}_0 = c$ by $c = \texttt{Inl}(\texttt{Inl}(\emptyset))$ and so forth, in the assumptions.

Such incremental unfolding combines freeness reasoning with other proof steps. It has the unfortunate side-effect of unfolding definitions of constructors in contexts such as $\exists x \,.\, \texttt{Con}_1(a) = x$, where they should be left alone. Calling the Isabelle tactic `fold_tac con_defs` restores the defined constants.

7 Related work

The use of least fixedpoints to express inductive definitions seems obvious. Why, then, has this technique so seldom been implemented?

Most automated logics can only express inductive definitions by asserting axioms. Little would be left of Boyer and Moore's logic [4] if their shell principle were removed. With ALF the situation is more complex; earlier versions of Martin-Löf's type theory could (using wellordering types) express datatype definitions, but the version underlying ALF requires new rules for each definition [7]. With Coq the situation is subtler still; its underlying Calculus of Constructions can express inductive definitions [14], but cannot quite handle datatype definitions [22]. It seems that researchers tried hard to circumvent these problems before finally extending the Calculus with rule schemes for strictly positive operators. Recently Giménez has extended the Calculus of Constructions with inductive and coinductive types [11], with mechanized support in Coq.

Higher-order logic can express inductive definitions through quantification over unary predicates. The following formula expresses that i belongs to the least set containing 0 and closed under `succ`:

$$\forall P\,.\,P(0) \land (\forall x\,.\,P(x) \to P(\mathtt{succ}(x))) \to P(i)$$

This technique can be used to prove the Knaster-Tarski theorem, which (in its general form) is little used in the Cambridge HOL system. Melham [15] describes the development. The natural numbers are defined as shown above, but lists are defined as functions over the natural numbers. Unlabelled trees are defined using Gödel numbering; a labelled tree consists of an unlabelled tree paired with a list of labels. Melham's datatype package expresses the user's datatypes in terms of labelled trees. It has been highly successful, but a fixedpoint approach might have yielded greater power with less effort.

Elsa Gunter [12] reports an ongoing project to generalize the Cambridge HOL system with mutual recursion and infinitely-branching trees. She retains many features of Melham's approach.

Melham's inductive definition package [5] also uses quantification over predicates. But instead of formalizing the notion of monotone function, it requires definitions to consist of finitary rules, a syntactic form that excludes many monotone inductive definitions.

PVS [21] is another proof assistant based on higher-order logic. It supports both inductive definitions and datatypes, apparently by asserting axioms. Datatypes may not be iterated in general, but may use recursion over the built-in `list` type.

The earliest use of least fixedpoints is probably Robin Milner's. Brian Monahan extended this package considerably [19], as did I in unpublished work.[3] LCF

[3] The datatype package described in my LCF book [23] does *not* make definitions, but merely asserts axioms.

is a first-order logic of domain theory; the relevant fixedpoint theorem is not Knaster-Tarski but concerns fixedpoints of continuous functions over domains. LCF is too weak to express recursive predicates. The Isabelle package might be the first to be based on the Knaster-Tarski theorem.

8 Conclusions and future work

Higher-order logic and set theory are both powerful enough to express inductive definitions. A growing number of theorem provers implement one of these [9, 33]. The easiest sort of inductive definition package to write is one that asserts new axioms, not one that makes definitions and proves theorems about them. But asserting axioms could introduce unsoundness.

The fixedpoint approach makes it fairly easy to implement a package for (co)inductive definitions that does not assert axioms. It is efficient: it processes most definitions in seconds and even a 60-constructor datatype requires only a few minutes. It is also simple: The first working version took under a week to code, consisting of under 1100 lines (35K bytes) of Standard ML.

In set theory, care is needed to ensure that the inductive definition yields a set (rather than a proper class). This problem is inherent to set theory, whether or not the Knaster-Tarski theorem is employed. We must exhibit a bounding set (called a domain above). For inductive definitions, this is often trivial. For datatype definitions, I have had to formalize much set theory. To justify infinitely-branching datatype definitions, I have had to develop a theory of cardinal arithmetic [30], such as the theorem that if κ is an infinite cardinal and $|X(\alpha)| \leq \kappa$ for all $\alpha < \kappa$ then $|\bigcup_{\alpha<\kappa} X(\alpha)| \leq \kappa$. The need for such efforts is not a drawback of the fixedpoint approach, for the alternative is to take such definitions on faith.

Care is also needed to ensure that the greatest fixedpoint really yields a coinductive definition. In set theory, standard pairs admit only well-founded constructions. Aczel's anti-foundation axiom [3] could be used to get non-well-founded objects, but it does not seem easy to mechanize. Isabelle/ZF instead uses a variant notion of ordered pairing, which can be generalized to a variant notion of function. Elsewhere I have proved that this simple approach works (yielding final coalgebras) for a broad class of definitions [26].

Several large studies make heavy use of inductive definitions. Lötzbeyer and Sandner have formalized two chapters of a semantics book [37], proving the equivalence between the operational and denotational semantics of a simple imperative language. A single theory file contains three datatype definitions (of arithmetic expressions, boolean expressions and commands) and three inductive definitions (the corresponding operational rules). Using different techniques, Nipkow [20] and Rasmussen [32] have both proved the Church-Rosser theorem; inductive definitions specify several reduction relations on λ-terms. Recently, I have applied inductive definitions to the analysis of cryptographic protocols [29].

To demonstrate coinductive definitions, Frost [10] has proved the consistency of the dynamic and static semantics for a small functional language. The example is due to Milner and Tofte [18]. It concerns an extended correspondence relation, which is defined coinductively. A codatatype definition specifies values and value environments in mutual recursion. Non-well-founded values represent recursive functions. Value environments are variant functions from variables into values. This one key definition uses most of the package's novel features.

The approach is not restricted to set theory. It should be suitable for any logic that has some notion of set and the Knaster-Tarski theorem. I have ported the (co)inductive definition package from Isabelle/ZF to Isabelle/HOL (higher-order logic). Völker [36] is investigating how to port the (co)datatype package. HOL represents sets by unary predicates; defining the corresponding types may cause complications.

References

[1] Abramsky, S., The lazy lambda calculus, In *Research Topics in Functional Programming*, D. A. Turner, Ed. Addison-Wesley, 1977, pp. 65–116

[2] Aczel, P., An introduction to inductive definitions, In *Handbook of Mathematical Logic*, J. Barwise, Ed. North-Holland, 1977, pp. 739–782

[3] Aczel, P., *Non-Well-Founded Sets*, CSLI, 1988

[4] Boyer, R. S., Moore, J. S., *A Computational Logic*, Academic Press, 1979

[5] Camilleri, J., Melham, T. F., Reasoning with inductively defined relations in the HOL theorem prover, Tech. Rep. 265, Comp. Lab., Univ. Cambridge, Aug. 1992

[6] Davey, B. A., Priestley, H. A., *Introduction to Lattices and Order*, Cambridge Univ. Press, 1990

[7] Dybjer, P., Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics, In *Logical Frameworks*, G. Huet G. Plotkin, Eds. Cambridge Univ. Press, 1991, pp. 280–306

[8] Dybjer, P., Nordström, B., Smith, J., Eds., *Types for Proofs and Programs: International Workshop TYPES '94*, LNCS 996. Springer, 1995

[9] Farmer, W. M., Guttman, J. D., Thayer, F. J., IMPS: An interactive mathematical proof system, *J. Auto. Reas.* **11**, 2 (1993), 213–248

[10] Frost, J., A case study of co-induction in Isabelle, Tech. Rep. 359, Comp. Lab., Univ. Cambridge, Feb. 1995

[11] Giménez, E., Codifying guarded definitions with recursive schemes, In Dybjer et al. [8], pp. 39–59

[12] Gunter, E. L., A broader class of trees for recursive type definitions for HOL, In *Higher Order Logic Theorem Proving and Its Applications: HUG '93* (Published 1994), J. Joyce C. Seger, Eds., LNCS 780, Springer, pp. 141–154

[13] Hennessy, M., *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*, Wiley, 1990

[14] Huet, G., Induction principles formalized in the Calculus of Constructions, In *Programming of Future Generation Computers* (1988), K. Fuchi M. Nivat, Eds., Elsevier, pp. 205–216

[15] Melham, T. F., Automating recursive type definitions in higher order logic, In *Current Trends in Hardware Verification and Automated Theorem Proving*, G. Birtwistle P. A. Subrahmanyam, Eds. Springer, 1989, pp. 341–386

[16] Milner, R., How to derive inductions in LCF, note, Dept. Comp. Sci., Univ. Edinburgh, 1980

[17] Milner, R., *Communication and Concurrency*, Prentice-Hall, 1989

[18] Milner, R., Tofte, M., Co-induction in relational semantics, *Theoretical Comput. Sci.* **87** (1991), 209–220

[19] Monahan, B. Q., *Data Type Proofs using Edinburgh LCF*, PhD thesis, University of Edinburgh, 1984

[20] Nipkow, T., More Church-Rosser proofs (in Isabelle/HOL), In *Automated Deduction — CADE-13 International Conference* (1996), M. McRobbie J. K. Slaney, Eds., LNAI 1104, Springer, pp. 733–747

[21] Owre, S., Shankar, N., Rushby, J. M., *The PVS specification language*, Computer Science Laboratory, SRI International, Menlo Park, CA, Apr. 1993, Beta release

[22] Paulin-Mohring, C., Inductive definitions in the system Coq: Rules and properties, In *Typed Lambda Calculi and Applications* (1993), M. Bezem J. Groote, Eds., LNCS 664, Springer, pp. 328–345

[23] Paulson, L. C., *Logic and Computation: Interactive proof with Cambridge LCF*, Cambridge Univ. Press, 1987

[24] Paulson, L. C., Set theory for verification: I. From foundations to functions, *J. Auto. Reas.* **11**, 3 (1993), 353–389

[25] Paulson, L. C., *Isabelle: A Generic Theorem Prover*, Springer, 1994, LNCS 828

[26] Paulson, L. C., A concrete final coalgebra theorem for ZF set theory, In Dybjer et al. [8], pp. 120–139

[27] Paulson, L. C., Set theory for verification: II. Induction and recursion, *J. Auto. Reas.* **15**, 2 (1995), 167–215

[28] Paulson, L. C., Mechanizing coinduction and corecursion in higher-order logic, *J. Logic and Comput.* **7**, 2 (Mar. 1997), 175–204

[29] Paulson, L. C., Tool support for logics of programs, In *Mathematical Methods in Program Development: Summer School Marktoberdorf 1996*, M. Broy, Ed., NATO ASI Series F. Springer, Published 1997, pp. 461–498

[30] Paulson, L. C., Grąbczewski, K., Mechanizing set theory: Cardinal arithmetic and the axiom of choice, *J. Auto. Reas.* **17**, 3 (Dec. 1996), 291–323

[31] Pitts, A. M., A co-induction principle for recursively defined domains, *Theoretical Comput. Sci.* **124** (1994), 195–219

[32] Rasmussen, O., The Church-Rosser theorem in Isabelle: A proof porting experiment, Tech. Rep. 364, Computer Laboratory, University of Cambridge, May 1995

[33] Saaltink, M., Kromodimoeljo, S., Pase, B., Craigen, D., Meisels, I., An EVES data abstraction example, In *FME '93: Industrial-Strength Formal Methods* (1993), J. C. P. Woodcock P. G. Larsen, Eds., LNCS 670, Springer, pp. 578–596

[34] Slind, K., Function definition in higher-order logic, In *Theorem Proving in Higher Order Logics: TPHOLs '96* (1996), J. von Wright, J. Grundy, J. Harrison, Eds., LNCS 1125

[35] Szasz, N., A machine checked proof that Ackermann's function is not primitive recursive, In *Logical Environments*, G. Huet G. Plotkin, Eds. Cambridge Univ. Press, 1993, pp. 317–338

[36] Völker, N., On the representation of datatypes in Isabelle/HOL, In *Proceedings of the First Isabelle Users Workshop* (Sept. 1995), L. C. Paulson, Ed., Technical Report 379, Comp. Lab., Univ. Cambridge, pp. 206–218

[37] Winskel, G., *The Formal Semantics of Programming Languages*, MIT Press, 1993

A Inductive and coinductive definitions: users guide

A theory file may contain any number of inductive and coinductive definitions. They may be intermixed with other declarations; in particular, the (co)inductive sets **must** be declared separately as constants, and may have mixfix syntax or be subject to syntax translations.

The syntax is rather complicated. Please consult the examples above and the theory files on the ZF source directory.

Each (co)inductive definition adds definitions to the theory and also proves some theorems. Each definition creates an ML structure, which is a substructure of the main theory structure.

Inductive and datatype definitions can take up considerable storage. The introduction rules are replicated in slightly different forms as fixedpoint definitions, elimination rules and induction rules. Lötzbeyer and Sandner's six definitions occupy over 600K in total. Defining the 60-constructor datatype requires nearly 560K.

A.1 The result structure

Many of the result structure's components have been discussed in §3; others are self-explanatory.

`thy` is the new theory containing the recursive sets.

`defs` is the list of definitions of the recursive sets.

`bnd_mono` is a monotonicity theorem for the fixedpoint operator.

`dom_subset` is a theorem stating inclusion in the domain.

`intrs` is the list of introduction rules, now proved as theorems, for the recursive sets. The rules are also available individually, using the names given them in the theory file.

`elim` is the elimination rule.

`mk_cases` is a function to create simplified instances of `elim`, using freeness reasoning on some underlying datatype.

For an inductive definition, the result structure contains two induction rules, `induct` and `mutual_induct`. (To save storage, the latter rule is just `True` unless more than one set is being defined.) For a coinductive definition, it contains the rule `coinduct`.

Figure 2 summarizes the two result signatures, specifying the types of all these components.

```
sig
val thy          : theory
val defs         : thm list
val bnd_mono     : thm
val dom_subset   : thm
val intrs        : thm list
val elim         : thm
val mk_cases     : thm list -> string -> thm
(Inductive definitions only)
val induct       : thm
val mutual_induct: thm
(Coinductive definitions only)
val coinduct    : thm
end
```

Figure 2: The result of a (co)inductive definition

A.2 The syntax of a (co)inductive definition

An inductive definition has the form

```
inductive
  domains     domain declarations
  intrs       introduction rules
  monos       monotonicity theorems
  con_defs    constructor definitions
  type_intrs  introduction rules for type-checking
  type_elims  elimination rules for type-checking
```

A coinductive definition is identical, but starts with the keyword `coinductive`.

The `monos`, `con_defs`, `type_intrs` and `type_elims` sections are optional. If present, each is specified as a string, which must be a valid ML expression of type `thm list`. It is simply inserted into the `.thy.ML` file; if it is ill-formed, it will trigger ML error messages. You can then inspect the file on your directory.

domain declarations consist of one or more items of the form *string* `<=` *string*, associating each recursive set with its domain.

introduction rules specify one or more introduction rules in the form *ident string*, where the identifier gives the name of the rule in the result structure.

monotonicity theorems are required for each operator applied to a recursive set in the introduction rules. There **must** be a theorem of the form $A \subseteq B \Longrightarrow M(A) \subseteq M(B)$, for each premise $t \in M(R_i)$ in an introduction rule!

constructor definitions contain definitions of constants appearing in the introduction rules. The (co)datatype package supplies the constructors' definitions here. Most (co)inductive definitions omit this section; one exception is the primitive recursive functions example (§5.6).

type_intrs consists of introduction rules for type-checking the definition, as discussed in §3. They are applied using depth-first search; you can trace the proof by setting

`trace_DEPTH_FIRST := true`.

type_elims consists of elimination rules for type-checking the definition. They are presumed to be safe and are applied as much as possible, prior to the `type_intrs` search.

The package has a few notable restrictions:

- The theory must separately declare the recursive sets as constants.
- The names of the recursive sets must be identifiers, not infix operators.
- Side-conditions must not be conjunctions. However, an introduction rule may contain any number of side-conditions.
- Side-conditions of the form $x = t$, where the variable x does not occur in t, will be substituted through the rule `mutual_induct`.

Isabelle/HOL uses a simplified syntax for inductive definitions, thanks to type-checking. There are no `domains`, `type_intrs` or `type_elims` parts.

B Datatype and codatatype definitions: users guide

This section explains how to include (co)datatype declarations in a theory file. Please include `Datatype` as a parent theory; this makes available the definitions of `univ` and `quniv`.

B.1 The result structure

The result structure extends that of (co)inductive definitions (Figure 2) with several additional items:

```
val con_defs  : thm list
val case_eqns : thm list
val free_iffs : thm list
val free_SEs  : thm list
val mk_free   : string -> thm
```

Most of these have been discussed in §6. Here is a summary:

`con_defs` is a list of definitions: the case operator followed by the constructors. This theorem list can be supplied to `mk_cases`, for example.

`case_eqns` is a list of equations, stating that the case operator inverts each constructor.

`free_iffs` is a list of logical equivalences to perform freeness reasoning by rewriting. A typical application has the form

```
by (asm_simp_tac (ZF_ss addsimps free_iffs) 1);
```

`free_SEs` is a list of safe elimination rules to perform freeness reasoning. It can be supplied to `eresolve_tac` or to the classical reasoner:

```
by (fast_tac (ZF_cs addSEs free_SEs) 1);
```

`mk_free` is a function to prove freeness properties, specified as strings. The theorems can be expressed in various forms, such as logical equivalences or elimination rules.

The result structure also inherits everything from the underlying (co)inductive definition, such as the introduction rules, elimination rule, and (co)induction rule.

B.2 The syntax of a (co)datatype definition

A datatype definition has the form

```
datatype <= domain
  datatype declaration and  datatype declaration and ...
  monos         monotonicity theorems
  type_intrs    introduction rules for type-checking
  type_elims    elimination rules for type-checking
```

A codatatype definition is identical save that it starts with the keyword `codatatype`.

The `monos`, `type_intrs` and `type_elims` sections are optional. They are treated like their counterparts in a (co)inductive definition, as described above. The package supplements your type-checking rules (if any) with additional ones that should cope with any finitely-branching (co)datatype definition.

domain specifies a single domain to use for all the mutually recursive (co)datatypes. If it (and the preceeding `<=`) are omitted, the package supplies a domain automatically. Suppose the definition involves the set parameters $A_1, \ldots, A_k$. Then $\texttt{univ}(A_1 \cup \cdots \cup A_k)$ is used for a datatype definition and $\texttt{quniv}(A_1 \cup \cdots \cup A_k)$ is used for a codatatype definition.

These choices should work for all finitely-branching (co)datatype definitions. For examples of infinitely-branching datatypes, see file `ZF/ex/Brouwer.thy`.

datatype declaration has the form

string = *constructor* | *constructor* | ...

The *string* is the datatype, say `"list(A)"`. Each *constructor* has the form

name (*premise* , *premise* , ...) *mixfix*

The *name* specifies a new constructor while the *premises* its typing conditions. The optional *mixfix* phrase may give the constructor infix, for example.

Mutually recursive *datatype declarations* are separated by the keyword `and`.

Isabelle/HOL's datatype definition package is (as of this writing) entirely different from Isabelle/ZF's. The syntax is different, and instead of making an inductive definition it asserts axioms.

Note. In the definitions of the constructors, the right-hand sides may overlap. For instance, the datatype of combinators has constructors defined by

$$\begin{aligned} \mathtt{K} &\equiv \mathtt{Inl}(\emptyset) \\ \mathtt{S} &\equiv \mathtt{Inr}(\mathtt{Inl}(\emptyset)) \\ p\#q &\equiv \mathtt{Inr}(\mathtt{Inl}(\langle p, q\rangle)) \end{aligned}$$

Unlike in previous versions of Isabelle, `fold_tac` now ensures that the longest right-hand sides are folded first.