Chapter 4
Inductive Predicates

Yves Bertot and Lawrence C. Paulson

Second readers: Christine Paulin-Mohring and Andrei Popescu

In the process of describing systems and their properties, users of proof assistants introduce not only recursive datatypes, as described in Chapter **??**, but also sets constructed by a recursive process. These sets are often represented by their characteristic predicates, and we use the terms interchangeably. Such sets or predicates are said to be *inductively defined* and are found throughout mathematics and computer science.

The concept of inductive definition is inherently constructive, and it is equally important to intuitionistic and classical logic. For that reason, inductive predicates can be defined in nearly all proof assistants. Nevertheless, there are some differences in approach. Below we present some representative examples of inductive definitions, outline the theory, discuss some pragmatic issues concerned with implementations and finally look at a few applications.

4.1 Introduction and Motivating Examples

It is hard to think of a better introduction than Peter Aczel's in the *Handbook of Mathematical Logic*, published in 1977:

> Inductive definitions of sets are often informally presented by giving some rules for generating elements of the set and then adding that an object is to be in the set only if it has been generated according to the rules. An equivalent formulation is to characterise the set as the smallest set closed under the rules [1, p. 740].

So an inductive definition declares a set to be *closed* under the given rules, and moreover, to be the *minimal* such set. Minimality is itself the foundation of proof by

Yves Bertot
Université Côte d'Azur, Inria, France, e-mail: Yves.Bertot@inria.fr

Lawrence C. Paulson
University of Cambridge Computer Laboratory, UK, e-mail: lp15@cam.ac.uk

induction: any set S that is closed under the given rules describes a superset of the minimal set. But we shall need to be precise about what we mean by a "rule" and examine the conditions the given rules must satisfy in order to yield a meaningful inductive definition.

Because the newly defined set appears among the sets used as inputs for some of the rules, proving that an element is a member may involve a repetitive process. In this sense, there is a similarity between inductive definitions and recursive functions as described in Chapter **??**. However, membership in a set defined using a recursive function is essentially decidable, while inductive definitions can be used to define undecidable sets. For instance, the set of theorems of first-order logic, the set of terminating programs in a simple Turing complete language, or the reduction relation on λ-terms can be defined as inductive predicates. This added expressive power justifies a specific treatment. Our examples will be more prosaic.

4.1.1 Well-Formed Parenthesized Expressions

Consider finite strings of characters where there are two special characters: "(" (opening parenthesis) and ")" (closing parenthesis). We might define a string of characters to be well formed if every opening parenthesis has a corresponding closing parenthesis occurring to its right in the string and every closing parenthesis corresponds to a unique opening parenthesis. Although correct, this definition of well-formed expressions would not help us understand how parentheses add structure to strings. A more useful definition makes it clear that parentheses are meant to enclose some content:

Definition 4.1. The set of *well-formed* parenthesized strings is the least set satisfying the following rules:

1. strings containing no parentheses are well formed;
2. the concatenation of two well-formed strings is well formed;
3. if a string s is well formed, then the string "(s)" is well formed.

Rules such as these that make up an inductive definition are called *introduction rules*. The terminology comes from natural deduction, where an introduction rule states a sufficient condition for concluding (or "introducing") a certain sort of claim.

The rules above list three separate sufficient conditions for concluding that a string is well formed. It is then possible to show that "$a(b)c$" is a well formed string:

- a, b, and c are well formed (by Rule 1);
- so (b) is well formed, by Rule 3;
- $(b)c$ is well formed, using two previous facts and Rule 2;
- $a(b)c$ is well formed, using two previous facts and Rule 2.

We wrote that inductive definitions suggest repetition. As an illustration, we can see that the string composed of n opening parentheses followed by n closing parentheses

$$\overbrace{(\cdots(}^{n}\ \overbrace{)\cdots)}^{n}$$

is well parenthesized and the justification is provided by n repeated applications of Rule 3 to Rule 1.

Dually, to prove that every element of the inductively defined set satisfies a given property, we use induction. Let EQ stand for the property that a string has equal numbers of left and right parentheses. We can prove EQ for every well-formed string by induction:

1. Strings containing no parentheses satisfy EQ;
2. if two strings both satisfy EQ, so does their concatenation;
3. if a string s satisfies EQ, then so the string "(s)".

Since EQ satisfies all three introduction rules, but the well-formed strings are the minimal such set, all well-formed strings satisfy EQ. We have just proved a theorem *by induction on the definition* of a well-formed string.

If we reflect on this proof, we see that proof by induction follows a *divide-and-conquer* approach: we check separately for each rule that the elements they introduce in the set satisfy the target property. We reason that each element of the inductively defined set must have been introduced by one of the rules. Moreover, two of the three cases rely on assumptions—*induction hypotheses*—stating that the property we seek to establish already holds for the precursor elements in the inductive construction.

Just as introduction rules provide a means of showing that some elements belong to the inductively defined set, induction provides a means of showing that some do not. For instance, "(" is not well formed: it does not satisfy EQ. In the terminology of natural deduction, induction is an elimination rule.

4.1.2 Even Numbers

The property of "being an even number" can be defined as a repetitive process. Formally, this is an inductive definition over the natural numbers, which must already have been defined as a type.

Definition 4.2. The predicate even is defined as the least predicate satisfying the following rules:

1. 0 is even;

2. if x is even, then $\mathsf{succ}(\mathsf{succ}(x))$ is even.

Starting from Rule 1, successive applications of Rule 2 make it possible to prove that 2, 4, 6, and so on are even.

As in Section 4.1.1, we can use minimality to prove properties of this predicate—e.g., $\neg\mathsf{even}(1)$. Consider the predicate P_1 defined by $P_1(n) \leftrightarrow n \neq 1$.

1. Since $0 \neq 1$, clearly $P_1(0)$ holds.
2. Since $\mathsf{succ}(\mathsf{succ}(x)) \neq 1$, we also see that $P_1\,(\mathsf{succ}(\mathsf{succ}(x)))$. In fact, we could have assumed $P_1(x)$ (as an induction hypothesis), but $\mathsf{succ}(\mathsf{succ}(x))$ never equals 1.

We can conclude that $P_1(x)$ satisfies the introduction rules. Thus, by minimality, the predicate even is included in $P_1(x)$; symbolically, $\mathsf{even}(x) \to x \neq 1$. So $\mathsf{even}(1)$ does not hold.

Reflecting on this proof, we see that it is performed by analyzing the introduction rules and confirming that neither of them can be used to prove $\mathsf{even}(1)$. This idea to look at the introduction rules and ask which of them could be used to prove the result is an essential part of proof by induction.

In general, it is a good idea to prove stronger properties, such as $x = 0 \pmod 2$:

1. Trivially, $0 = 0 \pmod 2$ holds.
2. If $x = 0 \pmod 2$ then $\mathsf{succ}(\mathsf{succ}(x)) = 0 \pmod 2$.

We can then conclude that $\mathsf{even}(x) \to x = 0 \pmod 2$. This stronger statement can then be used to prove $\neg\mathsf{even}(1)$. A strengthened induction formula is often necessary for the inductive proof to work at all. An example is $\neg\mathsf{even}(3)$, since $n \neq 3$ is not closed under the second rule.

We shall see in Section 4.3 that inductive predicates also provide systematic tools which make it possible to show that whenever a number is not even, the successor of its successor is also not even.

4.1.3 Natural and Integer Real Numbers

This time we assume the existence of a type $\mathbb{R}$ of real numbers. Actually, we only need to know that two numbers 0 and 1 are in $\mathbb{R}$ and that there exists a binary operation $+$ in $\mathbb{R}$. We wish to define the subset of $\mathbb{R}$ that contains exactly the natural numbers. This can be done as follows:

Definition 4.3. The *natural real numbers* are defined inductively by the following rules:

1. 0 is natural;
2. if x is natural, then $x+1$ is natural.

With this definition, it is easy to show that any positive integer real number satisfies the predicate "natural". Note that the very same definition but using the type $\mathbb{N}$ of natural numbers would be true for the whole type and therefore trivial.

This definition of the natural numbers is similar to the one given in Chapter **??** for the *type* of natural numbers. But there is a subtle point connected with the foundations of mathematics. If we say "the natural numbers are the smallest set containing 0 and closed under successor", then we are conjuring them up out of nothing, or more precisely, making ontological assumptions about the existence of suitable representations of zero and successors. With inductive predicates we always assume that our domain has already been shown to exist; we use induction to specify subsets of that domain. In this case, we assume the type of real numbers to exist, and we are defining a predicate on this type: natural.

An alternative definition of natural number could refer to addition:

Definition 4.4. Natural real numbers (take 2)

1. 0 is natural;
2. 1 is natural;
3. if two numbers are natural, then their sum is also natural.

Call the second predicate natural2. Using induction, we can prove that the two definitions are equivalent: showing that natural2 satisfies the introduction rules of natural proves $\mathsf{natural}(x) \to \mathsf{natural2}(x)$, and analogously for the converse.

Let us do the first implication: natural2 satisfies the introduction rules of natural.

1. 0 satisfies natural2 (as expressed by the first introduction rule of natural2);
2. if x satisfies natural2, then so does $x+1$ (by the second and third rules of natural2).

The opposite implication, $\mathsf{natural2}(x) \to \mathsf{natural}(x)$, is more difficult because it involves addition. We define an auxiliary predicate,

$$\mathsf{ADDN}(x) = (\forall y.\ \mathsf{natural}(y) \to \mathsf{natural}(x+y))$$

Note that $\mathsf{ADDN}(x) \to \mathsf{natural}(x)$, since we can put $y = 0$ and have $\mathsf{natural}(0)$. Then it is enough to prove that ADDN satisfies the introduction rules of natural2:

1. $\mathsf{ADDN}(0)$: obvious because y and $0+y$ are the same number.
2. $\mathsf{ADDN}(1)$ is $\mathsf{natural}(y) \to \mathsf{natural}(1+y)$, which holds by the second rule of natural.
3. We need to show that $\mathsf{ADDN}(x)$ and $\mathsf{ADDN}(y)$ imply $\mathsf{ADDN}(x+y)$. Given z such that $\mathsf{natural}(z)$, we obtain $\mathsf{natural}(y+z)$ by $\mathsf{ADDN}(y)$ and $\mathsf{natural}(x+y+z)$ by $\mathsf{ADDN}(x)$. Hence $\mathsf{ADDN}(x+y)$.

As a result, ADDN is a superset of natural2. So $\mathsf{natural2}(x) \to \mathsf{ADDN}(x)$ and hence $\mathsf{natural2}(x) \to \mathsf{natural}(x)$.

This proof illustrates again that in order to prove a simple fact by induction, it is often practical to prove a seemingly *stronger* statement. In this case, ADDN looks

a stronger statement than natural, although after proving the equivalence between natural and natural2, it turns out that ADDN is equivalent to both.

A benefit of having multiple inductive definitions of one concept is that they yield multiple points of view. This happens in programming language semantics, where small-step operational semantics compete with big-step operational semantics (both of them defined inductively). Each has an advantage: small-step semantics to describe the execution of concrete programs and big-step semantics to reason about language metatheory.

In the case of natural and natural2, the former is handy to establish a correspondence with conventional proofs by induction on natural numbers (as in Peano arithmetic), while the latter is practical for divide-and-conquer approaches concerning large natural numbers. A proof that 2^{10} satisfies natural using only introduction rules will have 2^{10} instances of the second introduction rule for that inductive predicate and one instance of the first rule, while a proof that 2^{10} satisfies natural2 will contain only 10 instances of the third introduction rule for that inductive predicate and one instance of the second rule.

Similarly, these definitions of natural numbers suggest several ways to define the set of integer real numbers. Here is one possibility:

Definition 4.5. The *integer real numbers* are defined inductively by

1. 1 is integer;
2. if x and y are integers then $x - y$ is integer

This works rather ingeniously. First, note that 0 satisfies the predicate, since $0 = 1 - 1$. Through this, we can reach -1 as $0 - 1$. Then 2 as $1 - (-1)$, then -2, 3, ...

4.1.4 The Transitive Closure of a Relation

A relation $\prec$ on a set A is a two-argument predicate with both arguments in A. It can also be viewed as a subset of $A \times A$. There are many ways to define the transitive closure $\prec^+$ of $\prec$ using introduction rules. This is the most natural definition:

Definition 4.6. The transitive closure of the relation $\prec$ is the smallest subset of $A \times A$ that satisfies the following rules:

1. if $x \prec y$ holds, then $x \prec^+ y$ also holds;
2. if $x \prec^+ y$ and $y \prec^+ z$ hold, then $x \prec^+ z$ holds.

So $\prec^+$ is the smallest transitive relation extending $\prec$. But we could break the symmetry in the second rule, for example as follows:

Definition 4.7. The transitive closure of the relation $\prec$ is the smallest subset of $A \times A$ that satisfies the following rules:

1. if $x \prec y$ holds, then $x \prec^+ y$ also holds;
2. if $x \prec y$ and $y \prec^+ z$ hold, then $x \prec^+ z$ holds.

This second definition suggests an operational interpretation. That is, it suggests a way to search for a proof that two elements are connected by $\prec^+$: first check whether they are directly connected by $\prec$, otherwise search for an intermediate element that is connected by $\prec$ to the first and then connected by $\prec^+$to the second. The asymmetry between $x \prec y$ and $y \prec^+ z$ makes it mathematically less appealing, but it might work better with automatic tools. As in Section 4.1.3, it is easy to prove by induction that the two definitions of transitive closure are equivalent.

Readers acquainted with the Prolog language will note that many Prolog programs define inductive properties. Each introduction rule of an inductive predicate corresponds to a clause in a Prolog program. A Prolog programmer would normally write a program for computing the transitive closure of a relation according to Definition 4.7, not Definition 4.6, as the latter is more prone to looping. Some proof assistants can be programmed to execute the Prolog proof strategy. But then alternative proof strategies could also be programmed, and all proof assistants offer the option of applying rules manually.

4.1.5 Programming Language Semantics

Programming languages, even low-level ones, can be formalized in proof assistants. The first step is usually to describe the language's *abstract syntax* as an inductive type. Then various aspects of the programming language—type checking, static semantics, runtime execution—are naturally described as inductive predicates, where each inference rule corresponds to one of the inductive predicate's introduction rule, as in the formal definition of Standard ML by Milner and his colleagues [14]. Decidable aspects, such as the type discipline, can sometimes be described using recursive functions as introduced in Chapter **??**. However, for undecidable aspects, such as the dynamic semantics of Turing-complete programming languages, inductive predicates are ideal.

The operational semantics of a programming language is generally formalized as a transition relation involving program *configurations*: fragments of program text (as abstract syntax) possibly coupled with a store, environment, and other state components.

- With a *small-step semantics*, the transition relation relates configurations to successor configurations. Describing program execution in individual steps is particularly suitable for capturing nondeterminism and nonterminating processes.
- With a *big-step semantics*, the transition relation relates configurations to terminal outcomes, a style yielding more compact definitions.

Either way, language properties such as type preservation can then be proved by induction. Bertot [4] (using Coq) and Nipkow and Klein [15] (using Isabelle/HOL) both present substantial examples involving the semantics of imperative languages in various forms, such as small-step, big-step and compilation to a stack machine.

In Section 4.3 below, the semantics of a tiny functional programming language is defined and some properties proved.

4.1.6 The Accessible Part of a Relation

Well-founded relations are well known in computer science as a way of reasoning about termination. A relation $\prec$ on a set A is said to be *well founded* if and only if there are no infinitely decreasing chains $x_0 \succ \cdots \succ x_n \succ \cdots$ in A. The termination of an iterative or recursive process can be shown by exhibiting a quantity that decreases under such a relation at every step.

The accessible part of $\prec$, written $\mathsf{Acc}_\prec$, is the subset of A containing all elements for which $\prec$ is well founded [1]. Then we can say that $\prec$ is well founded if $A = \mathsf{Acc}_\prec$. This formal treatment of well-foundedness provides a way to treat the termination of partial recursive functions, and it has an extremely concise inductive definition.

Definition 4.8. Given a binary relation $\prec$ on a type A, the *accessible part* of $\prec$ is defined by the rule that if $x \in \mathsf{Acc}_\prec$ for all $x \prec a$, then $a \in \mathsf{Acc}_\prec$.

For the newcomer, this is certainly a difficult definition to understand: it seems to loop without defining anything, and the rule can have infinitely many premises. To make sense of it, let us look at an example.

Consider the relation $<$ on the natural numbers. The number 0 has no predecessor by this relation, so according to the definition of the accessible part, $0 \in \mathsf{Acc}_<$ holds: the condition is vacuously true. Next, the number 1 has only one predecessor, namely 0. This predecessor satisfies $\mathsf{Acc}_<$ and hence $1 \in \mathsf{Acc}_<$. It similarly follows that every natural number is in the accessible part of $<$ and so $<$ is well founded.

In some cases, an element of the type may have an infinity of antecedents. A simple example is the lexicographic ordering on pairs of natural numbers:

$$(a',b') \prec (a,b) \leftrightarrow a' < a \vee [a' = a \wedge b' < b]$$

In this case, each pair $(0,n)$ has n predecessors and is accessible and the pair $(1,0)$ has infinitely many predecessors (any pair of the form $(0,n)$ is a predecessor), but all of them are accessible and so therefore is $(1,0)$. Finiteness is not necessary for inductive predicates to hold.

On the other hand, consider the relation $\leq$ on natural numbers: its accessible part is empty. It is obvious that for $a \in \mathsf{Acc}_\leq$ to hold, we need again $a \in \mathsf{Acc}_\leq$ to hold. Formally, observe that the introduction rule holds when the empty set is substituted: if $x \in \emptyset$ for all $x \leq a$, then $a \in \emptyset$. (This holds because $a \leq a$ holds and under the current assumption this implies $a \in \emptyset$.) By minimality, $\mathsf{Acc} \subseteq \emptyset$ and thus is empty.

Finally, well-founded relations can be defined as relations for which every element of the domain is accessible.

Well-founded relations play an important role in higher-order proof assistants. Instead of the highly restrictive form of recursion frequently provided by default, we can define recursive functions in a style that closely resembles programming in conventional functional programming languages. The classical definition of a well-founded relation—as one that lacks infinitely decreasing chains—is problematical in the context of constructive logic because it involves a double negation. Using $\mathsf{Acc}_{\prec}$ eliminates this issue.

4.1.7 Finite Enumerated Sets, Singleton Sets, and Equality

For any finite collection of values x_1, ..., x_n in the same type T, the finite set $\{x_1,\ldots,x_n\}$ is the least set S satisfying the rules $x_1 \in S$, ..., $x_n \in S$. The minimality statement expresses that if we wish to establish that some property holds for any y in the finite set $\{x_1,\ldots,x_n\}$, we only need to check that it holds for each x_i.

If we restrict this reasoning to just one value x, then we can use membership in the singleton $\{x\}$ to express equality. In other words, $y \in \{x\}$ is just a way to write $x = y$. The induction principle associated with this inductive predicate repeats what was said before: if $x = y$ and we want to prove $P(y)$, we only need to check that $P(x)$ holds. Thus the introduction rule for the singleton set is

$$x = x$$

and the associated induction principle is

$$x = y \to P(x) \to P(y)$$

In practical terms, the induction principle describes what happens when rewriting with an equality theorem.

Equality can therefore be *defined* as an inductive predicate. This is actually the approach taken in most proof assistants based on dependent type theory, such as Agda, Coq, and Lean.

4.1.8 Criteria for a Meaningful Inductive Definition

In proof assistants, inductive definitions must be *monotonic*: each of the introduction rules is required to be monotonic from a set-theoretic point of view.

Consider the following meaningless "definition".

Definition 4.9. The silly predicate
if $\neg\mathsf{silly}(a)$, then $\mathsf{silly}(a)$.

If this predicate were definable inductively, then we would expect it to be meaningful. But every possibility leads to a contradiction.

- If $\mathsf{silly}(0)$ holds, then it must have been proved using the sole introduction rule (here we use the case analysis on the introduction rules). This entails that $\neg\mathsf{silly}(0)$ holds. So $\mathsf{silly}(0)$ and $\neg\mathsf{silly}(0)$ hold simultaneously, contradiction.
- Since $\mathsf{silly}(0)$ leads to a contradiction, $\neg\mathsf{silly}(0)$ must hold. But then $\mathsf{silly}(0)$ also holds, by the introduction rule.

Thus, it's clear that the definition of silly is nonsensical.

Each introduction rule of an inductive definition gives criteria for *introducing* elements to the set S being defined. Each rule says that some particular element may be added to the set provided that certain other elements (possibly infinitely many) already belong to S. This specifies an iterative process for building up S, and it converges, though possibly only after transfinitely many iterations [3]. Rules may impose restrictions, but never of the form $a \notin S$ for the same set S. Such a constraint could prevent the process from converging: adding new elements could remove others, thereby adding others, etc. And of course the conclusion of an introduction rule cannot have the form $a \notin S$.

The concept of an inductive definition can be defined formally, as we shall see below, and this yields precise criteria for introduction rules to be meaningful: they must be monotonic. Inductive definition facilities provided in a proof assistant may enforce a stronger criterion that is easier to recognize because it is syntactic: the newly defined predicate can only appear *positively* in the premises of the introduction rules. Essentially the same restriction appears in the previous chapter: when defining a type inductively, the type itself cannot appear recursively on the left side of the function arrow.

4.2 Inductive Definitions: From Theory to Implementation

Now we have seen what inductive definitions can do, but how can we provide them to users? In some proof assistants, particularly those based on classical higher-order logic, one can implement inductive definitions by formalizing their underlying theory. This direct approach can be expected to accept the largest class of definitions since there is no intermediary formalism. For some type theories, it is better—especially if we are concerned about the computational content of proofs—to include an inductive definition principle as part of the type theory [16]. Code for inductive definitions must then be added to the proof assistant's kernel, and the type theory's metatheory gives us a precise basis for judging whether this implementation is correct.

Either way, the proof assistant must provide an interface to the theory. Formalizing an inductive definition will then involve specifying the predicate and its arguments, and enumerating all the introduction rules. The proof assistant must check that the introduction rules are (semantically) monotonic or (syntactically) positive, as described above, to ensure that the inductive definition makes sense. The proof assistant should also provide tools for reasoning about the defined predicates: induction principles themselves, obviously, but there is more.

4.2.1 The Theory of Inductive Definitions

Aczel [1] considers sets of rules of the form $X \to x$, where X is the set of premises and x is the conclusion. Supposing now that Φ is a set of rules, he says that a given set A is Φ-*closed* if each rule in Φ whose premises are in A also has its conclusion in A. Finally he specifies $I(\Phi)$, the set inductively defined by Φ, to be

$$I(\Phi) = \bigcap \{A \mid A \text{ is } \Phi\text{-closed}\}$$

Clearly, this definition is too abstract and too general: there can be infinitely many rules, and a rule can have infinitely many premises. An implementation must provide a specification language that can capture these possibilities finitely. All of our examples featured rules involving variables, each expressing an infinite set of instances. Our definition of $\mathsf{Acc}_{\prec}$ had a universally quantified premise, standing for possibly infinitely many specific premises.

Barwise [3] approaches inductive definitions more abstractly in terms of monotonic maps over a set A. A function ϕ is *monotonic* provided

$$R \subseteq S \to \phi(R) \subseteq \phi(S)$$

And every such function has a least fixed point,

$$\mathsf{lfp}(\phi) = \bigcap \{A \mid \phi(A) \subseteq A\}$$

This is a special case of the Knaster–Tarski fixed point theorem for complete lattices. We can also express $\mathsf{lfp}(\phi)$ as a union, defining

$$I^{\alpha}_{\phi} = \phi\Big(\bigcup_{\beta<\alpha} I^{\beta}_{\phi}\Big) \tag{4.1}$$

$$\mathsf{lfp}(\phi) = \bigcup_{\alpha \in \mathsf{ON}} I^{\alpha}_{\phi} \tag{4.2}$$

ON denotes the class of all ordinals. Equation (4.1) is a definition by transfinite recursion, but a union over the natural numbers suffices for simple (finitely branching) inductive definitions. The inductive definition expresses the iterative construction of a set. Starting with the empty set, elements are repeatedly added using the function ϕ, with the process terminating at some transfinite level. The strictly positive nature of the process should now be clear: elements are added, never removed.

The two approaches (sets of rules and monotonic maps) are easily shown to be equivalent, as suggested by the similarity between the definitions of $I(\Phi)$ and $\mathsf{lfp}(\phi)$. Given a rule set Φ on a set A, Aczel [1] defines a monotonic function ϕ on subsets $Y \subseteq A$ by

$$\phi(Y) = \{x \in A \mid X \subseteq Y \text{ for some rule } X \to x \text{ in } \Phi\}. \tag{4.3}$$

Monotonicity is obvious: if $Y \subseteq Z$, then $X \subseteq Y$ implies $X \subseteq Z$, so $\phi(Y) \subseteq \phi(Z)$. Conversely, given the function ϕ, the set Φ of rules consists of all $X \to x$ such that $x \in \phi(X)$.

The essence of all definition packages for inductive sets or predicates is to restrict the form of introduction rules in such a way that monotonic functions from some type into itself can be defined, and then to define each new inductive predicate as the least fixed point of that function. Incidentally, the *greatest* fixed point yields *coinductive* definitions, which are described in Chapter **??**.

4.2.2 From Introduction Rules to Monotonic Maps

Each introduction rule describes a principle for inserting new elements into the inductively defined set, say, I. So each rule must be a statement whose conclusion has the form $a \in I$. The premises of the rule may refer to other existing elements of I. As we have seen, a set of rules in the sense of Aczel naturally gives rise to a monotonic map. This fact continues to hold if we allow variables in rules to abbreviate infinite sets of rules in the obvious way. It will be instructive to revisit our previous examples to consider the monotonic maps we get. The accessible part of a relation, $\mathsf{Acc}_{\prec}$, is particularly interesting because its rule contains a quantifier.

From the formula (4.3) for ϕ, we can see that each rule of the inductive definition makes a separate contribution. Given the functions $\phi_1, \ldots, \phi_k$ associated with the introduction rules $r_1, \ldots, r_k$ for an inductive definition, we can construct the union of all these functions in a single set function:

$$\phi(S) = \phi_1(S) \cup \cdots \cup \phi_k(S)$$

In the case of the inductive predicate $\mathsf{even}(x)$, we get

$$\phi_{\mathsf{even}}(S) = \{0\} \cup \{\mathsf{succ}(\mathsf{succ}(x)) \mid x \in S\} \tag{4.4}$$

This function is obviously monotonic; that is, if $R \subseteq S$, then we are sure that $\phi_{\mathsf{even}}(R) \subseteq \phi_{\mathsf{even}}(S)$. The subset of $\mathbb{N}$ where even holds is exactly the least fixed point of ϕ_{even}, by definition. As remarked in (4.2), the least fixed point can be expressed as the union of iterations of a monotonic function. Here the union is simply $\bigcup_k \{\mathsf{succ}^{2k}(0)\}$.

Consider now the transitive closure (Definition 4.6). The first introduction rule for $\prec^+$ simply includes a copy of $\prec$. The second introduction rule has the following

shape (in set notation):

$$\forall x\, y\, z.\ (x,y) \in S \to (y,z) \in S \to (x,z) \in S$$

The associated map for a set S is

$$\{(x,z) \mid \exists y.\ (x,y) \in S \wedge (y,z) \in S\}$$

and this is nothing but the relational composition $S \circ S$. The final result is again obviously monotonic:

$$\phi_{\prec^+}(S) = (\prec) \cup (S \circ S)$$

Now recall the alternative definition of transitive closure (Definition 4.7): if $x \prec y$ and $y \prec^+ z$ hold, then $x \prec^+ z$ holds. Then

$$\phi_{\prec^+}(S) = (\prec) \cup (\prec \circ S)$$

Again regarding the least fixed point as a union according to (4.2), we can see that the union is over the number of $\prec$-hops in the transitive closure.

As a final illustration, recall that the inductive definition of $\mathsf{Acc}_\prec$ has a single introduction rule, with a universally quantified premise:

$$(\forall x \prec a.\ \mathsf{Acc}_\prec(x)) \to Acc_\prec(a)$$

The set function that is generated for this definition is $\phi_{\mathsf{Acc}_\prec}(S) = \{a \mid \forall x \prec a.\ x \in S\}$.

4.2.3 A Generalization: Set-Valued Functions in Introduction Rules

Proof assistants typically allow the introduction rules for I to have premises that are universally quantified, having the general form $\forall y.\ P(y) \to y \in I$, as in $\mathsf{Acc}_\prec$ above. This possibility can be generalized as follows: to allow premises that refer to $f(I)$, where f is any monotonic set-valued function. Allowing a variety of choices for f opens up a number of new possibilities.

Let $\mathsf{Pow}(S)$ denote the powerset of S, the set of all subsets of S. Now we have another way of expressing any universally quantified premise:

$$\{y \mid P(y)\} \in \mathsf{Pow}(I)$$

This works because

$$\{y \mid P(y)\} \in \mathsf{Pow}(I) \leftrightarrow \{y \mid P(y)\} \subseteq I \leftrightarrow (\forall y.\ P(y) \to y \in I)$$

So allowing the powerset operator in inductive definitions could be an alternative to allowing quantified premises.

Occasionally, other functions make sense within an inductive definition. Paulson has formalized the set of primitive recursive functions [17]. Recall that these include constant and projection functions, but the crucial case is function composition. These functions take multiple arguments and therefore composition combines some n-ary function g with a *list* of functions $[f_0,\ldots,f_{n-1}]$. The introduction rule takes the following form:

$$g \in \mathsf{primrec} \wedge fs \in \mathsf{List}(\mathsf{primrec}) \to \mathsf{COMP}(g,fs) \in \mathsf{primrec}$$

Here $\mathsf{List}(S)$ denotes the set of all lists whose elements belong to the set S. The rule neatly expresses that g must be primitive recursive and that fs must be a list of primitive recursive functions.[1]

Why is this inductive definition legitimate? Intuitively, it is positive: as we visualize the iterative construction of the set of primitive recursive functions, we see that the available lists of primitive recursive functions also grows. Formally, the function List is monotonic: if $R \subseteq S$, then $\mathsf{List}(R) \subseteq \mathsf{List}(S)$. It is probably relevant that the two functions $\mathsf{Pow}(S)$ and $\mathsf{List}(S)$ are similar, both creating a collection (of sets or lists, respectively) drawing elements from S.

Arbitrary functions cannot be allowed in an inductive definition, or it might not be meaningful. It is required that any function used in an inductive construction must be monotonic. Monotonicity is necessary both for an intuitive understanding of the definition and to derive its formal properties. Isabelle's implementation of inductive definitions allows functions to be used but requires the user to supply theorems sufficient to prove monotonicity. The induction principle and other necessary properties of the definition are then proved internally, and similarly across the HOL family of proof assistants.

In proof assistants based on dependent type theory such as Coq, Agda, or Lean, the question of monotonic type constructors like List is treated once and for all at the level of inductive type definitions, because inductive predicates are only a special instance of inductive type definitions relying on dependent types. Nevertheless, these proof assistants do not provide a specific notation for List as an operator on sets, so that the introduction rule mentioned above would have to be written in the longer form

$$g \in \mathsf{primrec} \wedge (\forall f \in fs.\ f \in \mathsf{primrec}) \to \mathsf{COMP}(g,fs) \in \mathsf{primrec}$$

where the membership notation is used with two possible meanings: membership in a list and membership in the inductively defined set.

[1] The definition of COMP is irrelevant here, but note that it is actual function composition: primrec is a set of functions, each function from lists of natural numbers to natural numbers not a set of abstract syntax trees.

4.2.4 Well-Formed Introduction Rules

In this section, we revisit the monotonicity constraint and describe clear constraints on the form of introduction rules. When these constraints are satisfied, the monotonicity property is guaranteed.

Crucially, introduction rules always describe how the new predicate can be positively satisfied. This predicate is never negated. An introduction rule always has the form of a universally quantified implication, possibly nested:

$$P_1 \to (P_2 \to (\cdots (P_n \to Q) \cdots))$$

The conclusion, Q, must be an instance of the predicate being defined. If $n = 0$, the nested implication degenerates to an atomic formula asserting the predicate.

For the transitive closure $\prec^+$ of the relation $\prec$, the first introduction rule is

$$\forall x\, y.\ x \prec y \to x \prec^+ y$$

For the set of even numbers, the first introduction rule is simply

$$\mathsf{even}(0)$$

The inductive predicate is also allowed to appear in the left-hand side of implications (in the P_i), but this is carefully restricted. As a first rule of thumb, each antecedent of an implication can itself be a sequence of implications or a universally quantified formula, and the defined inductive predicate is only allowed to appear as the conclusion of this formula. This requirement is known as the *strict positivity constraint*.

For the transitive closure of a relation, the second introduction rule is

$$\forall x\, y\, z.\ x \prec^+ y \to y \prec^+ z \to x \prec^+ z$$

The introduction rule is composed of two nested implications, and there are thus two antecedents: $x \prec^+ y$ and $y \prec^+ z$. The conclusions of each of these two antecedents are instances of the defined inductive predicate $\prec^+$. Here it is easy, because these antecedents are atomic.

A more elaborate example is the accessible part of a relation. In this case, in a context where A and $\prec$ are already defined, the single introduction rule has the form

$$\forall a.\ (\forall x \prec a.\ \mathsf{Acc}_\prec(x)) \to Acc_\prec(a) \tag{4.5}$$

The introduction rule has one universally quantified implication and the antecedent is

$$\forall x \prec a.\ \mathsf{Acc}_\prec(x)$$

This is a universally quantified implication and the defined inductive predicate appears solely in its conclusion. The constraints are satisfied.

For silly, the introduction rule would be

$$\forall x.\ \neg \mathsf{silly}(x) \to \mathsf{silly}(x)$$

The leftmost instance of $\mathsf{silly}(x)$ is negated, this violates the strict positivity constraint.

Some negations are permissible. Recall our example of the well-formed parenthesized strings (Definition 4.1). The first introduction rule states that strings containing no parentheses are well formed. In this case, negation is used to describe a constraint on strings, but negation is not applied to the inductive predicate itself. Side conditions attached to an introduction rule can be arbitrary logical formulas provided they do not mention the inductive predicate.

Note that nested implications can equivalently be expressed using conjunctions, which is also acceptable:

$$(P_1 \wedge P_2 \wedge \cdots \wedge P_n) \to Q$$

For instance, the second introduction rule for transitive closure may also be written in the following fashion, which is logically equivalent to the one above:

$$\forall x\, y\, z.\ x \prec^+ y \wedge y \prec^+ z \to x \prec^+ z$$

These syntactic conditions should be easy to remember and conform reasonably well to the theory that we have seen already. There are slight differences for each implementation, especially concerning their treatment of monotonic operators.

4.2.5 The Abstract Induction Principle

The defined predicate corresponds to the *least* fixed point using ϕ, the set function associated with the inductive definition:

$$\phi(S) \subseteq S \to \mathsf{lfp}(\phi) \subseteq S \tag{4.6}$$

This fact immediately yields an abstract induction principle: to show that every element of $\mathsf{lfp}(\phi)$ belongs to S, it suffices to show $\phi(S) \subseteq S$. In Section 4.1, we saw that induction is instrumental to show what consequences can be drawn when the inductive predicate holds and to show that some elements do not satisfy the predicate.

In the case of ϕ_{even}, using equation (4.4) and replacing S by $\{x \mid P(x)\}$, principle (4.6) amounts to

$$\big(P(0) \wedge (\forall x.\ P(x) \to P(\mathsf{succ}(\mathsf{succ}(x))))\big) \to \big(\forall x.\ \mathsf{even}(x) \to P(x)\big)$$

The induction rule above is weaker than it could be. In the inductive step, we want to assume the existence of some x that not only satisfies the induction hypothesis $P(x)$ but is also a member of the inductively defined set. For the even numbers, it

should be this:

$$\big(P(0) \land (\forall x.\, \mathsf{even}(x) \land P(x) \to P(\mathsf{succ}(\mathsf{succ}(x))))\big) \to \big(\forall x.\, \mathsf{even}(x) \to P(x)\big)$$

One way to get this stronger form is to instantiate S differently: $\{x \in \mathsf{lfp}(\phi) \mid P(x)\}$. For the even numbers, the base case becomes $P(0) \land \mathsf{even}(0)$, which is obviously equivalent to $P(0)$. In the inductive step we need to show

$$\mathsf{even}(x) \land P(x) \to P(\mathsf{succ}(\mathsf{succ}(x))) \quad \text{and} \quad \mathsf{even}(x) \land P(x) \to \mathsf{even}(\mathsf{succ}(\mathsf{succ}(x)))$$

but the latter one is trivial by the introduction rules of even.

We can achieve these stronger induction principles more abstractly by noting that by monotonicity,

$$\phi(S \cap \mathsf{lfp}(\phi)) \subseteq \phi(\mathsf{lfp}(\phi)) = \mathsf{lfp}(\phi) \text{ for all } S$$

Therefore, if $\phi(S \cap \mathsf{lfp}(\phi)) \subseteq S$, then $\phi(S \cap \mathsf{lfp}(\phi)) \subseteq S \cap \mathsf{lfp}(\phi)$, and by (4.6),

$$\mathsf{lfp}(\phi) \subseteq S \cap \mathsf{lfp}(\phi) \subseteq S$$

We have just proved an induction principle that is stronger than (4.6):

$$\phi(S \cap \mathsf{lfp}(\phi)) \subseteq S \to \mathsf{lfp}(\phi) \subseteq S \tag{4.7}$$

This stronger form is used as the foundation for the induction rule provided by most proof assistants. For those based on higher-order logic, it is straightforwardly derived from primitive definitions. For proof assistants based on dependent type theory, this induction principle is directly implemented using the recursor of an inductive type, as will be described in Section 4.2.7.

4.2.6 Inductive Predicates as a Higher-Order Logic Construction

Higher-order logic has no built-in support for inductive definitions. Instead, they can be expressed as the intersection of all sets closed under a collection of introduction rules. It suffices to define

$$I(x) = (\forall P.\, \mathrm{Closed}[P] \to P(x))$$

where Closed$[P]$ is a formula expressing that P is closed under all of the introduction rules. As an illustration, the even numbers could be defined like this:

$$\mathsf{ieven}(x) = \big(\forall P.\, P(0) \land (\forall n.\, P(n) \to P(\mathsf{succ}(\mathsf{succ}(n)))) \to P(x)\big)$$

(We use a different symbol because this is a different definition.) When Φ is a set of monotonic introduction rules, the intersection of all predicates closed under Φ de-

fines the corresponding inductive predicate $I(\Phi)$. As a consequence of monotonicity, $I(\Phi)$ will also be closed under Φ and the induction principle will be provable. This technique is the formal equivalent of the fixed point theory outlined in Sections 4.2.1 and 4.2.5 above.

When predicates are described as type families, as in type theory, defining a new predicate by quantifying over all predicates requires a system supporting *impredicativity*, such as Coq or Lean. Impredicativity is the idea of a self-referential expression: consider the formula $\forall P.\ P$, which asserts that every formula is true, including itself. (The formula is actually false.) Impredicativity is not allowed in Agda.

This capability to define inductive predicates by a direct encoding of intersection was also exploited in the early days of theorem provers based on type theory (supporting impredicativity). Now, inductive predicates rely mostly on inductive types, as described in Section 4.2.7. Proof assistants based on higher-order logic do encode inductive definitions as intersections, hiding the details from users by preprogrammed procedures.

As a side note, non-monotonic introduction rules are accepted by this approach of defining sets by intersection. The intersection is always defined, although it does not necessarily possess useful properties. When applied to the silly definition of Section 4.1.8, any predicate that satisfies the sole introduction rule is true over the entire underlying type. As a result, the intersection is also the entire type. On the other hand, this definition does not let us express that any element of the silly set has been obtained by the sole introduction rule. This gives the uncomfortable situation where every element of the original type is inside the inductively defined set, but none of them is *explained* by any of the introduction rules. The intersection is degenerate and induction here is meaningless.

4.2.7 Inductive Predicates as Inductive Types in Dependent Type Theory

In dependent type theory, the propositions-as-types interpretation is the main guiding principle, as explained in Section **??**. There is a striking similarity between inductive predicates and inductive types, so that proof assistants based on dependent type theory actually rely on the same machinery to handle both concepts.

First, we recall the situation with inductive types. When talking about the inductive type of natural numbers, there is only one type and many different elements, but each element is obtained by taking the base constructor 0 or applying the other constructor succ to an existing number. Thus, the constructor succ is viewed as a function, in this case a function with a simple non-dependent type : $\mathbb{N} \to \mathbb{N}$.

We now turn our attention to inductive predicates. The evolution from simple inductive types to inductive predicates relies on two extensions that are made possible by dependent types. First, we choose to describe a *family* of types. This was already alluded to in Section **??** when considering types Fin n (a type with n elements) and Vec_α n. In the case of the even inductive predicate, this means that the notion of even number is described by a collection of types even 0, even 1, even 2, and so on, which are all different. Some of these types are inhabited and some are not. So even actually is a function of type $\mathbb{N} \to$ Type. Second, the introduction rules are functions with a dependent type. More precisely, the introduction rule stating that succ (succ x) is even if x is even has the following type:

$$\mathsf{evenAdd2s} : \forall x : \mathbb{N}, \mathsf{even}\ x \to \mathsf{even}\ (\mathsf{succ}\ (\mathsf{succ}\ x))$$

It is a function that takes two arguments: the first one is a natural number; the second one is an element of type even x. According to the propositions-as-types interpretation, the second argument is a proof of even x. The value that this function returns is a proof of even (succ (succ x)). We have the same ingredients as when constructing elements of type $\mathbb{N}$: to construct an element of one of the types even k, we may need to apply the introduction rule to an element of one of the types even l. The introduction rules for the inductive predicate are thus interpretations of the constructors for an inductive family of types.

The other constructor for even is a constant, and it provides an element only in one member of the type family: even 0. In other words, it provides a proof of the proposition that 0 is even:

$$\mathsf{even0} : \mathsf{even}\ 0$$

The next simplest proof using the two constructors is as follows:

$$\mathsf{evenAdd2s}\ 0\ \mathsf{even0}$$

It is a proof that 2 is even. The next one is a proof that 4 is even.

$$\mathsf{evenAdd2s}\ 2\ (\mathsf{evenAdd2s}\ 0\ \mathsf{even0})$$

The nature of the introduction rules is such that not every natural numbers n enjoys that even n is inhabited. There is no way to construct an element of even (succ 0).

Intuitively, the inductive definition even describes a family of datatypes that can be interpreted as proof trees describing how some numbers can be shown to be even using only the introduction rules, in the same manner that the inductive type $\mathbb{N}$ describes how numbers can be represented as trees built using only succ and 0. It is important that some members of the type family are empty, which can be interpreted as false propositions, while others are nonempty, which can be interpreted as true propositions.

Exploiting this uniformity between inductive types and inductive predicates, introduction rules are usually called *constructors*.

Reasoning by induction on inductive datatypes applies to both situations. When considering an inductive predicate, this is interpreted as rule induction, when considering a datatype, this is interpreted as structural induction.

As explained in Chapter **??**, the induction principle associated with the type of natural numbers is given by recursion on natural numbers, fine-tuned to construct dependently typed functions. If we similarly use recursion for the datatype even, we proceed as follows:

1. Consider a predicate on trees of the even family. It needs to have an extra argument for the index of the family; we will write it as $P : \forall n.\ \mathsf{even}(n) \to \mathsf{Type}$.
2. We use the recursor to define a function of type $\forall n.\ \forall t.\ P\ n\ t$.
3. For the base case, we need to provide a value of type $P\ 0\ \mathsf{even0}$.
4. For the recursive case, we are looking at an element obtained by evenAdd2s, applied on some natural number k and some element t of even k. We must provide a function that can use k, t, and the result of a recursive call on t to construct a value of type $P\ (\mathsf{succ}\ (\mathsf{succ}\ k))(\mathsf{evenAdd2s}\ k\ t)$. Note that the result of the recursive call will have type $P\ k\ t$. This function will thus have the type

$$\forall k.\ \forall t : \mathsf{even}\ k.\ P\ k\ t \to P\ (\mathsf{succ}\ (\mathsf{succ}\ k))\ (\mathsf{evenAdd2s}\ k\ t)$$

Thus, the type of the recursor for dependent programming, which name even_rec for this chapter, has the following shape:

$$\begin{array}{ll} \mathsf{even\_rec}: & \forall P : \forall k.\ \mathsf{even}\ k \to \mathsf{Type}. \\ & P\ 0\ \mathsf{even0} \to \\ & (\forall k.\ \forall t_k : \mathsf{even}\ k.\ P\ k\ t_k \to P\ (\mathsf{succ}\ (\mathsf{succ}\ k))(\mathsf{evenAdd2s}\ k\ t_k)) \\ & \to \\ & \forall n.\ \forall t_k.\ P\ n\ t_k \end{array}$$

This is a strong induction principle, where the shape of proofs can be used to express more algorithmic content, as can be illustrated in Section 4.5.2. A simpler induction principle can be obtained by specializing P to a predicate that does not use the value of terms in type even. In that case, some universal quantifications become nondependent products and can be written as plain implications:

$$P\ 0 \to (\forall k.\ \mathsf{even}\ k \to P\ k \to P\ (\mathsf{succ}\ (\mathsf{succ}\ k))) \to \forall n.\ \mathsf{even}\ n \to P\ n$$

We recognize here the induction principle that was described earlier and is used in virtually all proof assistants (except maybe Agda, as we explain below). The idea of using a predicate P that does not observe the argument of type even n obeys the concept of *proof irrelevance*: only the existence of proofs matters, not their specific shape.

In the Coq system, there are two ways of defining inductive predicates:

1. The type family can be declared as a member of the Prop type of types. Then the simplified induction principle is generated automatically and named with the suffix _ind added to the type name.

2. The type family can be declared as a member of the Type type of types. Then it is considered to be a datatype, and the stronger induction principle is generated. Different proofs of the same statement will indeed be distinct (no proof irrelevance), in the same way that the members of the type of natural numbers are distinct.

In the Agda system, no induction principle is generated when an inductive type is created. Nevertheless, proofs by rule induction are as frequent in this system as in the other proof assistants. Proofs by induction simply are recursive programs, where the patterns to be covered in the recursive definitions are all the constructors of the inductive type family, and the recursive calls provide the induction hypotheses [21].

4.3 Rule Inversion

Detective novels typically involve a Holmes or Poirot reconstructing a long sequence of events from a few clues. They reason back from the clue itself (say, a lipstick-smeared cigarette) to the circumstances it suggests (a particular suspect had been present). In detective novels such reasoning is speculative, but with an inductive definition we have a formal specification of the exact circumstances consistent with a particular situation. For a trivial example: if we know that $\mathsf{succ}(\mathsf{succ}(x))$ is even, then we also know that x is even through a process called *rule inversion*, and here we exactly invert one of the rules of Definition 4.2.

Rule inversion is effectively case analysis over the introduction rules of an inductive definition. It is based on the idea that an instance of the inductive predicate can hold only if one of the introduction rules made it hold. When the instance is abstract—simply the inductive predicate applied to variables—the inversion statement is a disjunction of the conditions for the respective rules. When the instance is more specific, inversion can yield concise, strong conclusions.

Let us illustrate this with our running example on the even predicate. The following logical statement captures rule inversion in the most general case:

$$\mathsf{even}(x) \leftrightarrow x = 0 \vee \exists y.\, x = \mathsf{succ}(\mathsf{succ}(y)) \wedge \mathsf{even}(y) \tag{4.8}$$

A more concise rule can be generated in the case where the argument to even is an instance of succ. The left-hand side of the disjunction can be discarded as an application of the injectivity and disjointness properties seen in Chapter **??** [2], and the right-hand side can be simplified by the injectivity of succ:

[2] This disjointness property states that the images of datatype constructors are disjoint subsets of the datatype, it comes hand-in-hand with the fact that these constructors are injective.

$$\mathsf{even}(\mathsf{succ}(x)) \leftrightarrow \exists y.\ x = \mathsf{succ}(y) \wedge \mathsf{even}(y)$$

The same kind of treatment can be performed when the argument is 1, and one then obtains $\neg\mathsf{even}(1)$ or equivalently

$$\mathsf{even}(1) \leftrightarrow \bot$$

Each proof assistant handles rule inversion in its own way. In Agda and Lean, inversion is provided directly at the level of case analysis on an inductive type. Users do not need to provide the cases that can automatically be shown to be impossible, and the notation () can be used when none of the introduction rules is possible.

A proof that 1 is not even will look as follows in Agda. The use of () to express that no rule applies makes the proof very concise.

```
evenNot1 : even 1 → Empty
evenNot1 ()
```

During interactive proofs, users actually write incomplete proofs, and the pattern-matching mechanism may not always have enough information to show that some cases are impossible. The following example is a slightly different presentation of the same problem, where the system has already inferred that the constructor even0 could not have been used to obtain $\mathsf{even}(\mathsf{succ}(x))$, but the second constructor has not been ruled out. In the example displayed here, we use the notation conventions of Agda, so that the constructor corresponding to the successor of a natural number is actually written `suc` (see also Section 4.4).

```
evenNot1Alt : ∀x → even(suc x) → x ≡ 0 → Empty
evenNot1Alt _ (evenAdd2s y h1) h2 = { }0
```

In this code fragment, we follow the Agda convention that $\equiv$ is used to represent the equality predicate between two objects. This text is accepted as an *incomplete* proof by Agda, and { }0 represents a hole in the proof. The proof can be completed by performing a case analysis on the hypothesis h_2, which in this context should be a proof of suc y $\equiv$ 0, where y— is the first argument of constructor evenAdd2s. Equality is itself described using an inductive predicate (Section 4.1.7) and there is only one available case, which is reflexivity. This case is impossible, by the disjointness property. In all, the proof is written as follows: either the user writes this text or the case analysis on h_2 modifies the text accordingly.

```
evenNot1Alt : ∀x → even(suc x) → x ≡ 0 → Empty
evenNot1Alt _ (evenAdd2s y h1) ()
```

Coq provides a tactic called `inversion` which takes as argument an hypothesis name, denoting an instance of an inductive predicate. This tactic generates as many goals as there are applicable introduction rules to obtain this hypothesis. This tactic is used in interactive proof mode. A separate command called `Derive`

`inversion` will generate the proof term that is used by the `inversion` tactic, so that this proof term can be shared between many uses of the tactic.

Using the tactic, a proof that 1 is not even will look as follows in Coq:

```
Lemma evenNot1 : ~ even 1.
Proof. intros h; inversion h. Qed.
```

For the more progressive example `evenNot1Alt`, the proof is written as follows:

```
Lemma evenNot1Alt x : even (S x) -> x <> 0.
Proof.
intros h; inversion h.
discriminate.
Qed.
```

After the inversion step, the goal becomes `S n <> 0` and the disjointness property is used. This is done with the `discriminate` tactic.

In the HOL4 theorem prover, each inductive definition generates a master inversion theorem whose name is coined by adding the suffix _cases to the type name and this inversion theorem contains the equivalence to a disjunction mentioned earlier. For instance, for the definition of even, the `even_cases` theorem has exactly the statement given in equation 4.8.

This theorem actually expresses more than just the inversion property, because it is an equivalence. In automatic proof processes, repeated rewriting with this theorem loops forever, and therefore it needs to be used with care, but specialized instances to terms of the form 0 or succ are well behaved and mix well with automatic proof tools.

In Isabelle/HOL theorems are more systematically generated, so that they can be included in databases for automatic proof.

inductive_simps *evenNot1: "even(1)"*

The analysis is performed and produces a statement expressing that *even(1)* can be rewritten to *False*. This combines nicely with other automatic simplifications.

The power of generating all inversion rules as statements for rewriting is illustrated in the following example, a typical application to prove that an evaluation or reduction relation yields a unique result.

4.3.1 Extended Example: A Tiny Functional Language

Our example—done in Isabelle/HOL—is a tiny functional programming language operating on the Booleans and the natural numbers. It has conditional expressions and equality tests. The language admits nonsensical terms such as *Succ T*, but they do no harm here.

```
datatype exp = T | F | Zero | Succ exp
           | IF exp exp exp
           | EQ exp exp
```

We consider a basic small-step operational semantics [15, S7.3]. The conditional expression `IF p x y` reduces to `x` or `y` provided `p` has already been reduced to `T` or `F`; otherwise, it reduces to `IF q x y` provided `p` reduces to some `q`. For `Succ x` the only possibility is to reduce `x`, but for `EQ x y` there are six possibilities: to return `T` or `F` immediately or—see the last three rules—further to reduce its arguments. No order is imposed on which argument to reduce first or even to ensure that one evaluation finishes before the other starts. If both operands are `Succ(Succ Zero)` then `EQ` reduces to `T`, but the evaluation could take zero, one or two steps.

```
inductive Eval :: "exp ⇒ exp ⇒ bool" (infix "⇛" 50) where
    IF_T: "IF T x y ⇛ x"
  | IF_F: "IF F x y ⇛ y"
  | IF_Eval: "p ⇛ q ⟹ IF p x y ⇛ IF q x y"
  | Succ_Eval: "x ⇛ y ⟹ Succ x ⇛ Succ y"
  | EQ_same:  "EQ x x ⇛ T"
  | EQ_S0:    "EQ (Succ x) Zero ⇛ F"
  | EQ_0S:    "EQ Zero (Succ y) ⇛ F"
  | EQ_SS:    "EQ (Succ x) (Succ y) ⇛ EQ x y"
  | EQ_Eval1: "x ⇛ z ⟹ EQ x y ⇛ EQ z y"
  | EQ_Eval2: "y ⇛ z ⟹ EQ x y ⇛ EQ x z"
```

The question therefore arises: does this nondeterministic evaluation process always yield a unique result? Can we prove this?

```
proposition
  assumes "x ⇛ y" "x ⇛ z" shows "y = z"
```

No. One counterexample is `x = EQ (Succ T) (Succ T)`, which can reduce to either `EQ T T` or `T`.[3] The weaker conclusion `"∃u. y ⇛ u ∧ z ⇛ u"` still fails. To state the uniqueness property correctly, we must define *multi-step evaluation*, a form of transitive closure:

```
inductive EvalStar :: "exp ⇒ exp ⇒ bool" (infix "⇛⋆" 50) where
    Id:   "x ⇛⋆ x"
  | Step: "x ⇛ y ⟹ y ⇛⋆ z ⟹ x ⇛⋆ z"
```

Then we must prove simple lemmas about multi-step evaluation:

```
lemma Succ_EvalStar:
  assumes "x ⇛⋆ y" shows "Succ x ⇛⋆ Succ y"
```

In addition to the one above, we need similar lemmas for `IF` and `EQ`. Each is a one-line induction on $\Rightarrow\!\!\!\!\Rightarrow^\star$. The uniqueness result needs multi-step evaluation in the conclusion:

```
proposition
  assumes "x ⇛ y" "x ⇛ z"  shows "∃u. y ⇛⋆ u ∧ z ⇛⋆ u"
```

This *confluence* property is proved straightforwardly, by induction on `"x ⇛ y"`. The 10 resulting subgoals are proved—in most cases automatically—with the help of rule inversion on `"x ⇛ z"` for 10 separate instances of `x`. So let us examine how rule inversion works on $\Rightarrow\!\!\!\!\Rightarrow$.

[3] This counterexample is found automatically by Isabelle's Nitpick subsystem [6].

4.3.2 Rule Inversion for the Evaluation Predicate

Consider the formula `T ⇛ z`. The operational semantics above does not provide any rule for reducing `T`, so `T ⇛ z` is simply false. By the same reasoning, `F ⇛ z` and `0 ⇛ z` are also false. In Isabelle, these instances of rule inversion are generated by the following commands; the resulting theorems (e.g., `T_simp`) can be supplied to the simplifier to ensure that `T ⇛ z` is automatically rewritten to `False`.

inductive_simps `T_simp: "T ⇛ z"`
inductive_simps `F_simp: "F ⇛ z"`
inductive_simps `Zero_simp: "Zero ⇛ z"`

Now consider the formula `Succ x ⇛ z`. The operational semantics specifies a single rule for `Succ x`, namely that there must be a reduction within `x`.

inductive_simps `Succ_simp: "Succ x ⇛ z"`

This automatically generates a theorem called `Succ_simp`:

```
(Succ x ⇛ z) ⟷ (∃y. z = Succ y ∧ x ⇛ y)
```

The case of the conditional expression is particularly instructive. For `IF T x y` there is only one possibility, namely `IF T x y ⇛ x`. It is tempting to use rule inversion with this precise pattern, but as it turns out, treating `IF` in the general case is just as powerful:

inductive_simps `IF_Eval_simp: "IF p x y ⇛ z"`

This yields the following theorem:

```
(IF p x y ⇛ z) ⟷ (p = T ∧ z = x) ∨ (p = F ∧ z = y) ∨
                 (∃q. z = IF q x y ∧ p ⇛ q))
```

Using this to simplify `IF T x y ⇛ z` quickly rewrites the formula to `z = x`: the simplifier will rewrite the other two disjuncts to `False` because both `T = F` and `T ⇛ q` rewrite to `False`. With the latter, we are using our first example of rule inversion above.

There remains the case of `EQ`. The inductive definition includes six separate rules for `EQ`, and so rule inversion yields a disjunction to cover those six possibilities. It is ugly but nevertheless useful.

inductive_simps `EQ_simp: "EQ x y ⇛ z"`

The alert reader may have noticed that we are not using the most specific patterns. We have even used the general pattern for `EQ` covering six out of the ten total rules. Why not simply do rule inversion for the most general pattern of all, namely `x ⇛ y`?

The answer is that the resulting theorem would be useless in simplification: it would rewrite formulas of the form `x ⇛ y` to other formulas of that form and would therefore loop. In the case of `EQ`, none of the six disjuncts involve the original pattern, namely `EQ x y`.

The power of rule inversion comes from applying it repeatedly to simplify any assertion involving an inductively defined predicate. Rewriting can execute lengthy chains of reasoning using just a few proof tactics.

4.4 Illustrating Definitions and Tools

In this section, we show how a simple inductive definition is encoded in a variety of theorem provers. The inductive definition that we choose is the definition of even numbers with two constructors. This definition defines a subset of the natural numbers. The type of natural numbers is actually called `nat` or `Nat` in all systems we are illustrating, the constructors have different names: the successor constructor is called `succ` in Isabelle/HOL, HOL4 or Lean, `suc` in Agda, and `S` in Coq. To provide examples that are easily copied and pasted for experimentation with the various systems, we chose to adhere to the convention of each proof assistant instead of uniformizing our examples to use the same type and constructor names for natural numbers.

4.4.1 Agda

In Agda, defining an inductive predicate is a special case of defining an inductive datatype. The specificity comes from the fact that one is actually defining an inductive family of types, using dependent types.

```
data even : Nat -> Set where
  even0 : even zero
  evenAdd2s : forall x -> even x -> even (suc (suc x))
```

When writing `Nat -> Set` on the first line, we express that this is a family of types indexed by natural numbers.

This definition introduces `even`, `even0`, and `evenAdd2s`, the type, and the two introduction rules. The induction principle is not generated as a separate theorem. It will never be needed: inductive proofs and inversion proofs simply rely on the possibility to define recursive functions by cases on this datatype.

Below are the proofs that 1 is not even and that every even number is the double of another number. (We introduce an ad hoc datatype `double x` to express that concept, and the notation `\==` is used to represent equality.)

```
evenNot1 : even 1 -> Empty
evenNot1 ()

data double (x : Nat) : Set where
  Cdouble : forall (y : Nat) ->
                x \== 2 * y -> double x

evendouble : forall x -> even x -> double x
evendouble .0 even0 = Cdouble 0 refl
evendouble ._ (evenAdd2s x p) with evendouble x p
... | Cdouble y py =
```

```
Cdouble (suc y) (doublestep x y py)
```

In this proof, we assume `doublestep` to be a proof of

$$\forall x\, y.\ x = 2 * y \to \mathsf{succ}(\mathsf{succ}(x)) = 2 \times \mathsf{succ}(y)$$

4.4.2 Coq

In Coq, as in Agda, inductive predicates simply are inductive families of types. However, Coq makes a distinction between types intended to serve as data and types intended to serve as propositions. This distinction is made in the final type of the definition—here, `Prop`:

```
Inductive even : nat -> Prop :=
  even0 : even 0
| evenAdd2s : forall x, even x -> even (S (S x)).
```

This definition introduces `even`, `even0`, `evenAdd2s`, and an induction principle named `even_ind`. This principle is used by the `induction` and `elim` tactics. (Implementing proofs by induction as recursive functions is possible in Coq, as it is in Agda or Lean, but it not is idiomatic.)

Here are proofs that 1 is not even and that every even number is the double of another number. We use Coq's built-in existential quantifier.

```
Lemma evenNot1 : ~ even 1.
Proof. Intros abs; inversion abs. Qed.

Lemma evendouble x : even x -> exists y, x = 2 * y.
Proof.
induction 1 as [ | x xeven IH].
- exists 0; easy.
- destruct IH as [y py]; exists (S y).
  now apply doublestep.
Qed.
```

This is a goal directed proof. After the induction step, there are two goals:

1. the trivial $\exists y.\ 0 = 2 \times y$;
2. the induction step, namely $\exists y.\ S\,(S\,x) = 2 \times y$.

In the second goal, the hypothesis `IH` is the existential statement $\exists y.\ x = 2 \times y$. The step `destruct IH` transforms it into a variable `y` and a proof about `y`. We can then say which value will serve as witness for the existential statement, and here also we assume that the necessary lemma `doublestep` has been proved beforehand (see Section 4.4.1).

4.4.3 HOL4

In the HOL4 theorem prover, each definition of an inductive predicate runs an ML program that generates an internal definition of the predicate as an intersection (as in Section 4.2.6). It automatically derives the key properties in the form of HOL4 theorems:

1. the conjunction of all the introduction rules, saved under the name *I*_rules, where *I* is the name of the inductive predicate;
2. the minimality principle, saved under the name *I*_ind;
3. an equivalence expressing the inversion principle, named *I*_cases;
4. the strong induction principle, as described in Section 4.2.5, named *I*_strongind.

Thus, individual theorems for each of the introduction rules are not created.

In HOL4, inductive predicates are essentially different from inductive types. However, the high-level tactic for proof by induction makes it possible to refer to a premise that is an inductive predicate by giving the name of the predicate. The inversion theorem (with suffix `_cases`) can be used automatically by the main tactics, but must be restricted to a single use to avoid infinite looping.

```
Inductive even:
  even 0 /\
  !n. even n ==> even (SUC (SUC n))

Theorem evenNot1[simp]:
   ~ (even 1)
Proof
  simp[Once even_cases]
QED

Theorem evendouble[simp]:
  !n. even n ==> ?y. n = 2 * y
Proof
  Induct_on `even` >> rw[] >> exists_tac ``SUC y`` >>
  simp[]
QED
```

In this proof, we do not rely on an auxiliary theorem, but the final arithmetic statement is handled by the `simp` tactic, which includes a fair amount of automation.

4.4.4 Isabelle/HOL

Inductive definitions in Isabelle/HOL in some ways resemble those of HOL4: an internal definition of the predicate as an intersection is generated, along with the-

orems corresponding to the key properties. Other noteworthy features include the rule inversion mechanism already shown.

The definition of even, including the optional rule labels, is as follows:

```
inductive even where
    even0:     "even 0"
  | evenAdd2s: "even n ⟹ even (Suc (Suc n))"
```

The proof that 1 is not even is immediate by case analysis.

```
lemma "¬ even 1"
  using even.cases by auto
```

The proof that every even number is the double of another number goes by induction followed by a call to a Presburger arithmetic decision procedure, which takes care of the quantifier reasoning.

```
lemma "even n ⟹ ∃y. n = 2 * y"
  by (induction rule: even.induct; presburger)
```

4.5 Inductive Predicates for Reasoning about Recursive Functions

Most proofs assistants offer ways to define recursive functions by (1) providing primitive recursion over inductive types and (2) allowing functions to be defined by well-founded recursion. These tools are well-suited to describe total or terminating recursive functions.

Relying on inductive predicates to describe recursive function makes it possible to open new possibilities, sometimes by simply providing more efficient tools to reason about recursive function that would be definable by other means, sometimes by making it possible to describe recursive functions where termination is not guaranteed for all arguments, or simply not even known to be provable.

A partial recursive function f of type $A \to B$ can be modeled by an inductive predicate on the type $A \times B$ or by a two argument predicate on A and B. Moreover, f's domain of definition—the set of inputs in A where f is guaranteed to terminate—can also be defined inductively.

For instance, the pathological function $f^\dagger$ that would be defined by the equation $f^\dagger(x) = f^\dagger(x) + 1$ can be studied thanks to this approach: it can be described, but its domain of definition is empty, as will be illustrated in Section 4.5.3.

4.5.1 Inductive Definition of the Function Graph

A simple example of recursion is a function f that takes an input x and performs some test $t(x)$, where t has type $A \to \texttt{bool}$. If $t(x)$ is true then $f(x)$ returns $v(x)$

and otherwise $g(x, f(h(x)))$. Written more mathematically, this has the following shape:[4]

$$f(x) = \text{if } t(x) \text{ then } v(x) \text{ else } g(x, f(h(x))) \tag{4.9}$$

Such a function could be modeled as an inductive relation by the predicate f_p on $A \times B$ that is the minimal satisfying the following rules:

- if $t(x) = \top$ then $f_\mathrm{p}(x, v(x))$ for all x;
- if $t(x) = \bot$ and $f_\mathrm{p}(h(x), r)$ then $f_\mathrm{p}(x, g(x, r))$ for all x and r.

The predicate f_p describes the graph of the function f, viewed as a subset $A \times B$. This can be generalized to a function that performs several tests, with more than one recursive call, using pattern matching on an arbitrary datatype, etc. There could be any number of introduction rules, each with many occurrences of the f_p predicate.

The domain of definition of f, written D_f, can be defined as the set of values x such that $f_\mathrm{p}(x, v)$ holds for some v. The value $f(x)$, when it exists, is the unique value v such that $f_\mathrm{p}(x, v)$ holds. Most required properties of f's domain and value can be proved by induction on f_p. Because this inductive presentation describes a relation instead of a function, the uniqueness of $f(x)$ needs to be proved separately. The proof is usually straightforward because the introduction rules for f_p are mutually exclusive, each having a regular form corresponding to a different combination of tests.

This inductive presentation of recursive functions is also useful even for total recursive functions that can already be defined in the proof assistant, either by primitive recursion on a datatype or by well-founded recursion. Using the induction principle associated with f_p instead of recursion on the input datatypes can dramatically reduce the number of cases needed in proofs concerning f.

To illustrate this, let us consider the function f_3 that takes as input three boolean values, and returns true exactly when the three boolean values are `true`. The definition we just stated can be written as a pattern matching construct as follows:

```
f3(a,b,c) =
   match (a,b,c) with
   | true, true, true => true
   | (_, _, _) => false
   end.
```

At the abstract level, there are only two cases in this definition and the corresponding inductive relation is the smallest relation f_{3p} satisfying the two introduction rules:

- $f_{3p}((\top, \top, \top), \top)$;
- for any tuple t, if $t \neq (\top, \top, \top)$ then $f_{3p}(t, \bot)$.

However, the obvious reasoning by induction on the inputs will consider the boolean values one by one, leading to at least four cases. The difference becomes crucial

[4] In programming we would ask whether $t(x)$ terminated or not, but in logic it is not an issue.

when considering large functions, for instance in compiler development or programming language research.

In practice, reasoning by induction using the inductive predicate f_{p} is called *functional induction*. Two usage patterns arise, depending on whether the function is provably total or not.

When the function being defined is total, this is usually proved by relying on structural recursion or well-founded induction. Extra support is provided by generating automatically the inductive predicate at the time of defining the function [2,20] and providing enough high-level tools that the users do not need to know about the existence of the inductive predicate: the induction principle is rewritten so that every instance of $f_{\mathrm{p}}(x,y)$ is encoded by the corresponding replacement of y by $f(x)$.

For the simple function f of equation (4.9), the functional induction principle has the following shape:

$$\begin{array}{l}\forall P.\ (\forall x.\ t(x) = \top \to P(x,(v(x))) \wedge \\ \quad (\forall x.\ t(x) = \bot \wedge P(h(x), f(h(x))) \to P(x, g(x, f(h(x)))) \to \\ \quad \forall x.\ P(x, f(x))\end{array}$$

In Isabelle/HOL, the function definition tool relies on this technique both to provide the functional induction principle and to admit partial functions, even in the presence of nested recursion [12]. For a function that is not total, the above induction principle is invalid, because the property that is sought cannot be guaranteed outside the domain of definition. The induction principle then needs to be expressed in the following manner.

$$\begin{array}{l}\forall P.\ (\forall x.\ t(x) = \top \to P(x,(v(x))) \wedge \\ \quad (\forall x.\ t(x) = \bot \wedge P(h(x), f(h(x))) \wedge h(x) \in D_f \to P(x, g(x, f(h(x)))) \to \\ \quad \forall x.\ x \in D_f \to P(x, f(x))\end{array}$$

This induction principle can be used to reason about the value returned by the function for all inputs in the domain of definition D_f, which is itself defined from the graph predicate f_{p}. However, a direct inductive definition is also sometimes convenient.

In particular, this makes it possible to prove that the function behaves as prescribed by the initial definition. For the function given by equation (4.9), this gives

$$\forall x.\ x \in D_f \to f(x) = \text{if } t(x) \text{ then } v(x) \text{ else } g(x, f(h(x)))$$

4.5.2 Inductive Definition of the Function Domain

Instead of defining inductively the graph of a recursive function, we can restrict our attention to the domain of the function only. In this case, we associate with a function of type $A \to B$ a predicate on A. Considering the same abstract function f

as in the previous section, given by equation (4.9), we will define the domain of f as the minimal set D_f satisfying the following introduction rules:

- if $t(x)$ is true, then $D_f(x)$ holds;
- if $t(x)$ is false and $D_f(h(x))$ holds, then $D_f(x)$ holds.

It is obvious that f is guaranteed to terminate for every element of $D_f(x)$. Actually, the initial proponent of using this approach advocates that D_f should be a datatype (in dependent type theory), and the function f should be defined as a *total* recursive function on this datatype [7].

When $D_f(x)$ is considered a datatype, its elements are trees that describe the recursive behavior of the function f. Because of the use of dependent types, these trees already contain the information and computation of successive calls to t and h. Thus, the computation of $f(x)$ from a tree in D_f only contains successive computations of g and v. If f is total, this description can be complemented by a function producing an element in $D_f(x)$ for every x in A. Then the computation of f is staged in two parts:

1. computing all calls to t and h, yielding an element in $D_f(x)$;
2. performing all computations of g and ultimately v.

This approach can be used for any proof assistant with dependent types, such as Agda, Coq, and Lean.

For a second example illustrating the use of the strong induction principle introduced in Section 4.2.7, let us consider the recursive function satisfying the following computation rules, for a given value v in type A and g of a suitable type:

- $f_e\ 0 = v$
- $f_e(\mathsf{succ}(\mathsf{succ}(x)) = g(x, f_e(x))$

These computation rules only specify the value for even numbers, so the domain of f_e is inductively described by the even predicate. This may be convenient for developers who want to avoid having to describe what happens for odd numbers.

The computation behavior can be described by the following application of the recursor for dependent programming:

$$f_{e,dep} = \mathsf{even\_rec}\ (\lambda x\ p, A)\ v\ (\lambda x, p\, r, g(x, r)$$

In Agda, the recursor would not be used, but the function $f_{e,dep}$ would be defined recursively as a recursive function. Whatever the approach, the $f_{e,dep}$ that is obtained is a *two-argument function*, where the second argument is a proof that the second argument is in the domain (here, a proof that the first argument is even).

The approach of defining the domain of definition and then the function is staged in the sense that it requires the domain of the function to be described completely before the function can be defined. This is ill-suited when the description of the domain involves the values of some recursive calls, as in nested recursion. For dependent type theory, Bove and Capretta proposed in 2001 to rely on **simultaneous** inductive and recursive definitions [9] but this idea has not been adopted. In later work, Bove follows an approach based on the graph description [8].

In Coq, we have the choice to put the type of D_f in Type or in Prop. If the first choice is taken, then the function f can be defined as above. If the second choice is taken, then the computation of the proof in $D_f(x)$ corresponds to information that is not available to produce data. The second part of the computation has to redo the computations of t and h and the information stored in D_f can only be used to show that recursive calls stay in the domain of definition [5, sect. 15.4].

In formalisms equipped with Hilbert's choice operator ε, such as HOL4 and Isabelle/HOL, the recursive function can be defined to be indeterminate outside its domain. The operator can be used in combination with the graph presentation:

$$f(x) = \varepsilon(\lambda y.f_{\mathrm{p}}(x,y))$$

This is the approach used internally by Krauss's function definition mechanism [12] in Isabelle/HOL, which automatically shows that the relation f_{p} is the graph of a partial function.[5]

Another approach, advocated in [5, sect. 15.3], relies on iterating the definition of f. We first associate with f a higher-order function $F : (A \to B) \to (A \to B)$ of which f is intended to be the fixed point. For the example of equation (4.9), F is defined mathematically by

$$F(f',x) = \text{if } t(x) \text{ then } v(x) \text{ else } g(x,f'(h(x)))$$

We can show the following property by induction on the domain predicate:

$$\forall x.\, D_f(x) \to \exists w.\, \exists k.\, \forall f'.\, w = F^k(f',x)$$

When execution terminates, the value $f(x)$ (namely, w above) is reached after a certain number (namely, k) of recursive calls. The function f' is irrelevant: it is forgotten when computation terminates.

The function f can then be defined in two different ways, depending on whether the proof assistant is based on higher-order logic or on dependent type theory. In higher-order logic, the function f can be defined by

$$f(x) = \varepsilon(\lambda w.\exists k.\, \forall f'.\, w = F^k(f',x))$$

In dependent type theory, the function f can be defined by structural recursion over the proof of $D_f(x)$. It is easy to show that f is a partial fixedpoint of F:

$$\forall x.\, D_f(x) \to f(x) = F(f,x)$$

This last equation, known as the *fixed point equation*, is crucial to be able to reason about the recursive computations.

When one wishes to prove that some element e belongs to the domain of definitions, for a concrete e, this can be done by repeated application of the introduction rules for the domain predicate D_f. An alternative solution exists by relying on an-

[5] Krauss built upon an earlier PhD. thesis, by Slind [19].

other presentation of the higher-order function F. The idea is to imitate domain theory [18], adding an element in the output type to model termination. In practice this can be done by using an option type. For the simple function of equation (4.9), the higher-order function has the following type and value:

$$\begin{aligned} F &: (A \to B_\perp) \to A \to B_\perp \\ F_\perp(f', x) &= \text{if } t(x) \text{ then} \\ &\quad \mathsf{Some}(v(x)) \\ &\quad \text{else} \\ &\quad \text{match } f'(h(x)) \text{ with} \\ &\quad\quad \mid \mathsf{Some}(r) \Rightarrow g(x, r) \\ &\quad\quad \mid \mathsf{None} \Rightarrow \mathsf{None} \\ &\quad \text{end} \end{aligned}$$

The function $F_\perp$ performs the same computations as F when recursion terminates, but it detects nontermination in recursive calls and propagates it in the result.

It is easy to prove the following statement:

$$\forall x.\ \exists k.\ F_\perp(\lambda x.\mathsf{None}, x) \neq \mathsf{None} \leftrightarrow x \in D_f$$

In practice, checking that $F_\perp^n(\lambda x.\mathsf{None}, e) = \mathsf{Some}(v)$ for a well-chosen n is a simple computational way to check that e belongs to the domain and to compute the value v of $f(e)$. This is particularly useful in proof assistants based on dependent type theory, where function execution can be more concise and efficient than other proof procedures.

4.5.3 Handling of pathological cases

It is interesting to see how the inductive predicate presentation of recursive functions works for the pathological function $f^\dagger$ defined by $f^\dagger(x) = f^\dagger(x) + 1$. The corresponding inductive predicate is defined as the inductive predicate $f_{\mathrm{p}}^\dagger$ with the following introduction rule:

- if $f_{\mathrm{p}}^\dagger(x, y)$ holds, then $f_{\mathrm{p}}^\dagger(x, y+1)$ holds.

This inductive predicate definition is well formed and $f_{\mathrm{p}}^\dagger(x, y)$ can be used for further reasoning. A simple proof by induction on the predicate $f_{\mathrm{p}}^\dagger$ shows that the empty set is a superset of $f_{\mathrm{p}}^\dagger$. The introduction of $f^\dagger$ can proceed: the function exists, but it is nowhere defined. Since the defining equation for $f^\dagger$ is only guaranteed to be satisfied inside the domain of definition, this is consistent with the laws of logic and arithmetic.

If we look solely at the inductive definition of the domain $D_{f^\dagger}$, we see that this domain is the inductive predicate given by the following only introduction rule:

- if $D_{f^\dagger}(x)$ holds, then $D_{f^\dagger}(x)$ holds.

A very simple proof by induction shows that this domain of definition is empty.

4.6 The Limitations of Inductive predicates

It may seem that inductive predicates are strictly more powerful than recursive functions, but what really matters is how natural the description of a concept is. Coming back to the notion of even natural numbers, there is a very natural presentation of the concept using a function that returns a boolean value:

- 0 is even
- $\mathsf{even}\ (\mathsf{succ}\ x) = \neg(\mathsf{even}\ x)$

This presentation is not suitable for an inductive definition, because of the use of negation, which leads to a non-positive occurrence of even. However for some purposes, this definition is simpler to use than the one based on an inductive definition. For instance, showing that 1, 3, or 5 are even only requires direct application of the computations rules instead of lemmas obtained through inversion techniques.

A more advanced example is the concept of *reducibility* found in Girard, Lafont, and Taylor's proof of strong normalization for the typed λ-calculus [10]. This can easily be defined recursively (over the structure of simple types), but it also involves non-positive occurences of the defined predicate. The technique described in Section 4.5 is still applicable, but the presentation as a recursive function is simply more natural.

The use of recursive functions instead of inductive predicates is particularly useful in proof assistants based on dependent type theory, where proofs about recursive definitions are typically more concise than those involving inductive predicates. For instance, Coq's Mathematical Components library [11, 13] advocates recursive definitions of boolean-valued functions over inductive definitions whenever possible.

Acknowledgements Thanks to Jasmin Blanchette for a detailed reading and commentary.

References

1. Aczel, P.: An introduction to inductive definitions. In: J. Barwise (ed.) Handbook of Mathematical Logic, pp. 739–782. North-Holland (1977) 1, 8, 11, 12
2. Barthe, G., Forest, J., Pichardie, D., Rusu, V.: Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In: Functional and Logic Programming (FLOPS'06). Fuji Susono, Japan (2006). URL https://hal.inria.fr/inria-00564237 31
3. Barwise, J.: Admissible Sets and Structures: An Approach to Definability Theory, chap. VI: Inductive Definitions, pp. 197–254. Springer (1975). URL https://projecteuclid.org/euclid.pl/1235418478 10, 11

4. Bertot, Y.: Theorem-proving support in programming language semantics. In: Y. Bertot, G. Huet, J.J. Lévy, G. Plotkin (eds.) From Semantics to Computer Science, Essays in Honour of Gilles Kahn, pp. 337–362. Cambridge University Press (2009) 7
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series. Springer (2004). DOI 10.1007/978-3-662-07964-5. URL https://doi.org/10.1007/978-3-662-07964-5 33
6. Blanchette, J.C.: Relational analysis of (co)inductive predicates, (co)algebraic datatypes, and (co)recursive functions. Software Quality Journal **21**(1), 101–126 (2013). DOI 10.1007/s11219-011-9148-5. URL https://doi.org/10.1007/s11219-011-9148-5 24
7. Bove, A.: General recursion in type theory. In: H. Geuvers, F. Wiedijk (eds.) Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers, *Lecture Notes in Computer Science*, vol. 2646, pp. 39–58. Springer (2002). DOI 10.1007/3-540-39185-1_3. URL https://doi.org/10.1007/3-540-39185-1_3 32
8. Bove, A.: Another look at function domains. Electronic Notes in Theoretical Computer Science **249**, 61 – 74 (2009). DOI https://doi.org/10.1016/j.entcs.2009.07.084. URL http://www.sciencedirect.com/science/article/pii/S1571066109003065. Proceedings of the 25th Conference on Mathematical Foundations of Programming Semantics (MFPS 2009) 32
9. Bove, A., Capretta, V.: Nested general recursion and partiality in type theory. In: R.J. Boulton, P.B. Jackson (eds.) Theorem Proving in Higher Order Logics, 14th International Conference, TPHOLs 2001, Edinburgh, Scotland, UK, September 3-6, 2001, Proceedings, *Lecture Notes in Computer Science*, vol. 2152, pp. 121–135. Springer (2001). DOI 10.1007/3-540-44755-5_10. URL https://doi.org/10.1007/3-540-44755-5_10 32
10. Girard, J.Y., Taylor, P., Lafont, Y.: Proofs and Types, *Cambridge Tracts in Theoretical Computer Science*, vol. 7. Cambridge University Press, Cambridge (1989) 35
11. Gonthier, G., Mahboubi, A.: An introduction to small scale reflection in coq. J. Formaliz. Reason. **3**(2), 95–152 (2010). DOI 10.6092/issn.1972-5787/1979. URL https://doi.org/10.6092/issn.1972-5787/1979 35
12. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. J. Autom. Reasoning **44**(4), 303–336 (2010). DOI 10.1007/s10817-009-9157-2. URL https://doi.org/10.1007/s10817-009-9157-2 31, 33
13. Mahboubi, A., Tassi, E.: Mathematical Components. Zenodo (2021). DOI 10.5281/zenodo.4457887. URL https://doi.org/10.5281/zenodo.4457887 35
14. Milner, R., Tofte, M., Harper, R.: Definition of standard ML. MIT Press (1990) 7
15. Nipkow, T., Klein, G.: Concrete Semantics: With Isabelle/HOL. Springer (2014). DOI 10.1007/978-3-319-10542-0. URL https://doi.org/10.1007/978-3-319-10542-0 7, 24
16. Paulin-Mohring, C.: Inductive definitions in the system Coq - rules and properties. In: M. Bezem, J.F. Groote (eds.) Typed Lambda Calculi and Applications, International Conference, TLCA '93, LNCS 664, pp. 328–345. Springer (1993). DOI 10.1007/BFb0037116. URL https://doi.org/10.1007/BFb0037116 10
17. Paulson, L.C.: A fixedpoint approach to (co)inductive and (co)datatype definitions. In: G.D. Plotkin, C. Stirling, M. Tofte (eds.) Proof, Language, and Interaction, Essays in Honour of Robin Milner, pp. 187–212. The MIT Press (2000) 14
18. Scott, D.S.: Data types as lattices. SIAM J. Comput. **5**(3), 522–587 (1976). DOI 10.1137/0205037. URL https://doi.org/10.1137/0205037 34
19. Slind, K.: Reasoning about terminating functional programs. Ph.D. thesis, Technical University Munich, Germany (1999). URL http://tumb1.biblio.tu-muenchen.de/publ/diss/in/1999/slind.ps 33

20. Sozeau, M., Mangin, C.: Equations reloaded: high-level dependently-typed functional programming and proving in Coq. Proc. ACM Program. Lang. **3**(ICFP), 86:1–86:29 (2019). DOI 10.1145/3341690. URL https://doi.org/10.1145/3341690 31
21. Wadler, P., Kokke, W., Siek, J.G.: Programming Language Foundations in Agda (2020). Available at http://plfa.inf.ed.ac.uk/20.07/ 21