Inductive Definitions via Inference Rules in ACL2*

Alessandro Coglio Grant Jurgensen
Kestrel Institute https://kestrel.edu

Certain predicates are best defined inductively as the smallest ones that satisfy some inference rules. Prime examples are typing judgments and evaluation reductions in programming language semantics, and logical assertions in formal deductive systems. While inference rules are readily expressible as logical implications, the 'smallest' requirement needs some machinery to be expressed in ACL2. This paper describes a tool to automate that machinery: given signatures for one or more predicates, and inference rules involving the predicates, the tool generates definitions of the predicates, theorems saying that they are the smallest ones that satisfy the rules, and facilities for reasoning about them.

1 Introduction

Certain predicates are best defined as the smallest ones that satisfy some inference rules. An *inference rule* is a logical implication

$$P_1 \wedge \cdots \wedge P_n \implies C$$

often written as

$$\frac{P_1 \quad \cdots \quad P_n}{C}\ \text{N}$$

where $P_1, \ldots, P_n$ are the premises (possibly none, i.e. $n = 0$), C is the conclusion, and N names the rule.

A simple but illustrative example is the reflexive and transitive closure R^* of a given binary relation R, which can be defined as the smallest relation that satisfies the following inference rules:

$$\frac{R(x,y)}{R^*(x,y)}\ \text{BASE} \qquad \frac{}{R^*(x,x)}\ \text{REFL} \qquad \frac{R^*(x,y) \quad R^*(y,z)}{R^*(x,z)}\ \text{TRANS}$$

These definitions are *inductive* because of the stated 'smallest' requirement: the defined predicate implies any other predicate satisfying the same inference rules. Equivalently, the predicate holds on exactly the values for which there is a finite derivation, i.e. a *proof tree*, using the inference rules. Under suitable conditions on the inference rules, the smallest predicate that satisfies them exists. This generalizes to multiple predicates, simultaneously defined by inference rules involving all of them.

Inductive definitions via inference rules are widely used in programming language semantics to define typing judgments, evaluation reductions, etc. They are also used in logic to formalize deductive systems. The predicates they define may not be computable; when they are computable, the formulation with inference rules is generally clearer and higher-level than a directly executable definition. This is what 'best defined', at the beginning of this section, refers to.

Since inference rules are logical implications, they are directly expressible in ACL2's first-order logic. But the 'smallest' requirement is not, because its direct expression is higher-order (see Section 2); it can be expressed, but with a more elaborate formulation.

*This work was partially supported by the Defense Advanced Research Projects Agency under award number HR0011259E006. The views, opinions and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

To appear in EPTCS.

© A. Coglio and G. Jurgensen
This work is licensed under the
Creative Commons Attribution License.

This paper describes an event macro tool, `definductive`, to define predicates inductively via inference rules in ACL2. It works by reifying the aforementioned proof trees as ACL2 objects and by defining the predicates in terms of existence of valid proof trees. It generates theorems to reason about the predicates, including the inference rules themselves. It generates validation theorems showing that any other predicates satisfying all the rules are implied by the defined ones, i.e. that the defined predicates are indeed the smallest ones. Besides their role in defining the predicates, the proof trees also provide support for inductive reasoning. The tool also generates an induction scheme that applies directly to the predicates instead of the proof trees. The tool is available in the ACL2 community books [22, `[books]/std/util/definductive.lisp`] and documented in the manual [21, `definductive`].

Although the approach of reifying proof trees for specific applications is not new (see Section 7), its automation for general predicates and inference rules appears to be new in ACL2. It also aligns ACL2 with existing higher-order provers, which have similar facilities (see Section 7).

Section 2 exemplifies how inductive definitions via inference rules can be expressed in higher-order logic, while Section 3 exemplifies how they can be expressed in ACL2. The automation realized by `definductive` is described in Section 4, followed by a description of reasoning techniques in Section 5. Section 6 illustrates a practical application of `definductive`. Related and future work are discussed in Section 7 and Section 8. Some conclusions are drawn in Section 9.

2 Mathematical Definition

Under suitable conditions, a collection of inference rules defines a monotone function on the set of possible predicates, whose least fixpoint is the definition of the predicate(s), i.e. the smallest one(s) satisfying the inference rules. A general treatment is in the literature, e.g. [1]; below is an exemplification.

Given a universe of values $\mathcal{U}$ and a binary relation $R \subseteq \mathcal{U} \times \mathcal{U}$, the three inference rules in Section 1 define a function $\Phi : \mathcal{P}(\mathcal{U} \times \mathcal{U}) \to \mathcal{P}(\mathcal{U} \times \mathcal{U})$, where $\mathcal{P}(A)$ is the powerset of a set A. The function Φ maps a relation $\rho \subseteq \mathcal{U} \times \mathcal{U}$ to the relation $\Phi(\rho) \subseteq \mathcal{U} \times \mathcal{U}$ defined as

$$\begin{aligned} \Phi(\rho) \;=\; & \{\langle x,y\rangle \mid \langle x,y\rangle \in R\} \\ & \cup\; \{\langle x,x\rangle \mid x \in \mathcal{U}\} \\ & \cup\; \{\langle x,z\rangle \mid \exists\, y \in \mathcal{U} \,.\, \langle x,y\rangle \in \rho \;\wedge\; \langle y,z\rangle \in \rho\} \end{aligned}$$

where we use a set membership notation for the relations, instead of the function application notation in the inference rules, to highlight the treatment of the relations as sets. The three sets in the union above are mechanically derived from the three inference rules, using the variable ρ for the relation R^* being defined. A relation $\rho \subseteq \mathcal{U} \times \mathcal{U}$ satisfies the rules exactly when $\rho \supseteq \Phi(\rho)$.

The function Φ is monotone, i.e. $\rho_1 \subseteq \rho_2 \Longrightarrow \Phi(\rho_1) \subseteq \Phi(\rho_2)$. Thus, by the Knaster–Tarski theorem, it has a least fixpoint, i.e. the reflexive and transitive closure of R, which can be written as

$$R^* \;=\; \iota\, \rho \,.\, (\Phi(\rho) = \rho \;\wedge\; (\forall\, \rho' \,.\, \Phi(\rho') = \rho' \Longrightarrow \rho \subseteq \rho'))$$

which uses a definite description to say that R^* is a fixpoint and that any other fixpoint is a superset; equivalently, it could be defined as

$$R^* \;=\; \iota\, \rho \,.\, (\rho \supseteq \Phi(\rho) \;\wedge\; (\forall\, \rho' \,.\, \rho' \supseteq \Phi(\rho') \Longrightarrow \rho \subseteq \rho'))$$

i.e. R^* satisfies the rules and any other predicate satisfying the rules is a superset.

3 Definition in ACL2

Given a binary relation

```
(defstub r (* *) => *) ; uninterpreted, for abstraction
```

its reflexive and transitive closure `r*` can be systematically defined as follows in ACL2.

Although Φ from Section 2 cannot be directly defined in first-order logic, proof trees of (instances of) inference rules can, e.g. using the FTY algebraic data types [21, `fty`], as a sum type of the form[1]

```
(fty::deftagsum r*-proof ; proof trees for r*
  (:base (x y))
  (:refl (x))
  (:trans (x y z (premise1-proof r*-proof) (premise2-proof r*-proof))))
```

where each summand corresponds to an inference rule in Section 1. Each summand value represents an instance of the corresponding inference rule, as (i) a value for each free variable in the rule and (ii) a sub-proof for each R^* premise:

- The BASE rule has free variables x and y, and no R^* premise. Thus, the `:base` summand has untyped fields `x` and `y` for the values of those variables.
- The REFL rule has a free variable x, and no premise. Thus, the `:refl` summand has an untyped field `x` for the value of that variable.
- The TRANS rule has free variables x, y, and z, and two R^* premises. Thus, the `:trans` summand has untyped fields `x`, `y`, and `z` for the values of those variables, and `r*-proof`-typed fields for proof trees for the premises.

Since the form of each inference rule is known, the free variable values alone are sufficient to determine the instance of the rule,[2] as reified below.

The validity of `proof` as a proof tree for the assertion `(r* a b)` is formalized as

```
(defun r*-proof-validp (proof a b) ; proof proves (r* a b)
  (declare (xargs :guard (r*-proofp proof)))
  (r*-proof-case
   proof
   :base (and (r proof.x proof.y)
              (equal a proof.x)
              (equal b proof.y))
   :refl (and (equal a proof.x)
              (equal b proof.x))
   :trans (and (equal a proof.x)
               (equal b proof.z)
               (r*-proof-validp proof.premise1-proof proof.x proof.y)
               (r*-proof-validp proof.premise2-proof proof.y proof.z))))
```

where `r*-proof-case` is a case analysis macro generated by `fty::deftagsum` that binds the variable `proof.`f to each field f of a summand. For each summand, the values `proof.`v of the free variables are used to calculate (i) the non-R^* premises, (ii) the arguments of the R^* premises, and (iii) the arguments of the (always R^*) conclusion, of the inference rule that the summand corresponds to.[3] For each summand,

[1] We omit some details from the code in this paper. Full details are in the supporting materials.

[2] Our initial proof trees contained data structures for R^* assertions which explicitly captured the rule conclusion. Storage of the free variables made this unnecessary.

[3] The arguments of the conclusion and R^* premises are always just `proof.`v variables in this example. But if the BASE rule had, say, premise $R(f(g(x)),h(y,x))$ and conclusion $R^*(x+y,g(y))$, then the code for `:base` would be something like

```
:base (and (r (f (g proof.x)) (h proof.y proof.x))
           (equal a (+ proof.x proof.y))
           (equal b (g proof.y)))
```

validity requires that (i) the calculated non-R^* premises hold, (ii) the R^* premise proof trees prove that R^* holds on the calculated arguments, and (iii) the formal parameters `a` and `b` are equal to the calculated arguments of the conclusion. This ties together every part of the inference rule instance, including the premise proof trees.

While the mention of `(r* a b)` above was informal, it becomes formal via the definition

```
(defun-sk r* (a b)
  (exists (proof) (and (r*-proofp proof) (r*-proof-validp proof a b))))
```

That is, `r*` holds exactly on the values for which there is a valid proof tree.

With the above definition, the inference rules are theorems:

```
(defthm r*-base (implies (r x y) (r* x y)))
(defthm r*-refl (r* x x))
(defthm r*-trans (implies (and (r* x y) (r* y z)) (r* x z)))
```

These are proved via the `r*-suff` theorem generated by the `defun-sk`, instantiated with a suitable proof tree, which in the case of `r*-trans` also involves the witness proof trees for the premises, given by the `r*-witness` function that `defun-sk` introduces for the existential:

```
; hints to prove r*-trans:
:in-theory (enable r*-proof-validp)
:expand ((r* x y) (r* y z)) ; expose premise proof trees
:use (:instance r*-suff ; proof of (r* x z) from (r* x y) and (r* y z):
                (proof (r*-proof-trans x y z (r*-witness x y) (r*-witness y z)))
                (a x)  ; 1st argument of r* in conclusion (r* x z)
                (b z)) ; 2nd argument of r* in conclusion (r* x z)
```

Additionally, any alternative binary relation satisfying the inference rules, introduced as

```
(encapsulate
  (((r*-alt * *) => *)) ; alternative relation
  (local (defun r*-alt (a b) (r* a b))) ; r* as witness
  (defthm r*-alt-base (implies (r x y) (r*-alt x y)))
  (defthm r*-alt-refl (r*-alt x x))
  (defthm r*-alt-trans (implies (and (r*-alt x y) (r*-alt y z)) (r*-alt x z))))
```

is proved (by induction on `r*-proof-validp`, using the `r*-alt-...` theorems above) to satisfy

```
(defthm r*-alt-when-r*-proof-validp
  (implies (r*-proof-validp proof a b) (r*-alt a b)))
```

and therefore to satisfy

```
(defthm r*-alt-when-r* (implies (r* a b) (r*-alt a b)))
```

thus establishing that `r*` is indeed the smallest relation satisfying the inference rules.

4 Automation in ACL2

The approach in Section 3 is mechanized in our `definductive` tool [21, `definductive`], whose call

```
(definductive refl-trans-closure
  :preds ((r* a b))
  :irules ((base ((r x y)) (r* x y))
           (refl () (r* x x))
           (trans ((r* x y) (r* y z)) (r* x z))))
```

generates (essentially) the events in Section 3.

4.1 General Form

The general form is

```
(definductive name ; name of the collection of events, e.g. for XDOC
  :preds  (pred1 pred2 ...) ; predicates being defined
  :irules (rule1 rule2 ...) ; inference rules defining the predicates
  ...) ; other optional inputs, e.g. for XDOC generation
```

Each `:preds` entry has the form `(p x1 x2 ...)`, where `p` is a symbol for the name of the predicate and `x1`, `x2`, etc. are symbols for its formal parameters. Each `:irules` entry has the form `(name (prem1 prem2 ...) concl)`, where `name` is a symbol for the name of the rule, `prem1`, `prem2`, etc. are terms for the premises, and `concl` is a term for the conclusion.

More precisely, each `concl` has the form `(p t1 t2 ...)`, where `p` is a predicate in `:preds` and `t1`, `t2`, etc. are valid terms before the call of `definductive`. When `definductive` is called, `(p t1 t2 ...)` itself is not a valid term because `p` must be a new name, but it becomes a valid term after the call. Each `prem` either has the same form as just described (possibly with a different predicate `q` in `:preds`) or is a valid term before the call of `definductive`; for example, in Section 1, the two premises of TRANS have the first form, while the premise of BASE has the second form. Since the predicates being defined only appear at the top level of premises and conclusions, they fulfill a syntactic condition of 'positive occurrences' [1], which suffices for the monotonicity condition in Section 2, as elaborated in Section 4.8.

The proofs generated by `definductive` are designed to always work. They do not rely on the current ACL2 theory, but instead they carry hints with quoted theories, calculated for the specific events. But there is no meta proof that they always work; a proof failure is regarded as a bug in `definductive`.

4.2 Proof Trees

In a call of `definductive`, a predicate `p` *directly depends on* a predicate `q` (both in `:preds`, and not necessarily distinct) exactly when there is a rule in `:irules` with `p` in the conclusion and `q` in some premise; and `p` *depends on* `q` exactly when they satisfy the transitive closure of the direct dependency relation. These dependencies partition the predicates in `:preds` into disjoint subsets, called *cliques*, where two distinct predicates are in the same clique exactly when each predicate depends on the other. Each clique consists of either a single predicate (which may depend on itself or not)[4] or two or more mutually dependent predicates. For example, `refl-trans-closure` above has a singleton clique consisting of `r*`, which depends on itself.

The cliques of a `definductive` call are organized in a strict partial order: a clique `C` *precedes* a clique `D` exactly when some predicate in `D` depends on some predicate in `C` but no predicate in `C` depends on any predicate in `D` (otherwise `C` and `D` would be the same clique).[5] For example, `refl-trans-closure` above has just one clique, while[6]

[4] The customary case is that it depends on itself, i.e. there is at least one inference rule with the predicate both in the conclusion and in some premise. But `definductive` does not prevent defining a predicate that does not appear in any premises, e.g. `(definductive just-0 :preds ((p x)) :irules ((zero () (p 0))))`.

[5] In graph-theoretic terms, cliques are the strongly connected components of the direct dependency graph, and clique `C` precedes `D` exactly when `C` is reachable from `D` in the resulting component graph.

[6] The following example is purely illustrative. Natural numbers are already defined in ACL2, and trees of natural numbers and lists thereof are better defined via mutually recursive recognizers rather than inference rules.

```
(definductive nat-trees
  :preds ((nat n) (tree tr) (trees trs))
  :irules (;; natural numbers:
           (zero () (nat 0))
           (succ ((nat n)) (nat (1+ n)))
           ;; trees of natural numbers:
           (leaf ((nat n)) (tree n))
           (branch ((nat n) (trees trs)) (tree (cons n trs)))
           ;; lists of trees of natural numbers:
           (empty () (trees nil))
           (cons ((tree tr) (trees trs)) (trees (cons tr trs)))))
```

has a clique consisting of `nat` that precedes a clique consisting of `tree` and `trees`. That is, `tree` and `trees` depend on each other, and both depend on `nat`, while `nat` depends only on itself.

The *p*`-proof` types are generated by `definductive` via an event per clique, in an order consistent with the cliques' precedence. If a clique is a singleton, the event is a single `fty::deftagsum`, recursive exactly when the predicate depends on itself, like `r*-proof` in Section 3. If a clique has two or more mutually dependent predicates, the event is a `fty::deftypes` containing two or more mutually recursive `fty::deftagsum`s, one per predicate; for the clique of `tree` and `trees` above, the event is

```
(fty::deftypes tree-proof-clique ; named after one of the types
  (fty::deftagsum tree-proof ; proof trees for predicate 'tree'
    (:leaf (n (premise1-proof nat-proof)))
    (:branch (n trs (premise1-proof nat-proof) (premise2-proof trees-proof))))
  (fty::deftagsum trees-proof ; proof trees for predicate 'trees'
    (:empty ()) ; no free variables and no premise proof trees
    (:cons (tr trs (premise1-proof tree-proof) (premise2-proof trees-proof)))))
```

FTY types cannot be empty, because their fixers are total functions that return values in the types. For a singly recursive sum type like `r*-proof`, the requirement amounts to the existence of some non-recursive summand, like `:base` and `:refl`. For mutually recursive sum types like `tree-proof` and `trees-proof`, a non-recursive summand for each type, like `:leaf` and `:empty`, is sufficient, but not necessary, as exemplified by[7]

```
(definductive even-and-odd
  :preds ((even n) (odd n))
  :irules ((zero () (even 0))
           (from-even ((even n)) (odd (1+ n)))
           (from-odd ((odd n)) (even (1+ n)))))
```

which has a clique consisting of `even` and `odd`, with proof tree types

```
(fty::deftypes even-proof-clique
  (fty::deftagsum even-proof ; level 0 (explained below)
    (:zero ())
    (:from-odd (n (premise1-proof odd-proof)))
    :measure (two-nats-measure (acl2-count proof) 0))
  (fty::deftagsum odd-proof ; level 1 (explained below)
    (:from-even (n (premise1-proof even-proof)))
    :base-case-override :from-even ; dependency from level 1 to level 0
    :measure (two-nats-measure (acl2-count proof) 1)))
```

[7] Another purely illustrative example, since ACL2 already has predicates for even and odd numbers.

While even-proof has a non-recursive summand, odd-proof only has a recursive one, but that summand only recurses to even-proof. This suffices to populate odd-proof, via :from-even, because even-proof is populated via :zero. The :base-case-override, along with the lexicographic measures, helps FTY recognize this (see the FTY documentation for details).

In general, a clique must be stratified into disjoint subsets, called *levels*: a predicate p is at level ℓ exactly when ℓ is the smallest natural number such that there is some rule with p in the conclusion each of whose premises contains no predicates in :preds, or only predicates that are either in preceding cliques or in the same clique but at levels below ℓ. In the example above, even is at level 0 and odd is at level 1. The stratification into levels is checked by definductive, which generates the appropriate :base-case-overrides and lexicographic :measures when there is more than one level. In the nat-trees example above, tree and trees are both at level 0, so no :base-case-overrides or lexicographic :measures are needed. The predicate in a singleton clique can only be at level 0.

The required non-emptiness of the proof tree types does not imply the non-emptiness of the predicates being defined, as it is possible that the p-proof-validp function deems all proof trees invalid. This is the case when every inference rule has a false or contradictory premise. On the other hand, emptiness of the proof tree types would imply emptiness of the predicates. See Section 4.8.

4.3 Validity of Proof Trees

The p-proof-validp functions are generated by definductive via an event per clique of predicates, mirroring the proof tree types. They have the same form as r*-proof-validp in Section 3, but p-proof-validp may call q-proof-validp in a summand for a rule with p in the conclusion and q in a premise; the recursion is the same as that of the proof tree types.

4.4 Predicate Definitions

The predicates p in :preds are generated by definductive as in Section 3, via defun-sk, one event per predicate, but with an additional requirement that the proof tree is minimal. For example, for r* it is

```
(defun-sk r* (a b)
  (exists (proof) (and (r*-proofp proof)
                       (r*-proof-validp proof a b)
                       (r*-proof-minimalp proof a b)))
  :skolem-name r*-proof) ; witness function of the existential
```

where r*-proof-minimalp is a function also generated by definductive

```
(defun-sk r*-proof-minimalp (proof a b)
  (forall (proof2)
          (implies (and (r*-proofp proof2) (r*-proof-validp proof2 a b))
                   (<= (r*-proof-count proof) (r*-proof-count proof2)))))
```

saying that every proof tree proof2 valid for the same a and b is not smaller than proof (where the 'size' is determined by the FTY-generated r*-proof-count function).

The minimality requirement on the proof tree is new as compared to the corresponding definition in Section 3. It was omitted from there because it does not change the meaning of the predicate. The existence of any valid proof tree implies the existence of a minimal proof tree, which definductive proves for each particular predicate. The reason for this additional constraint is just to affect *which* proof tree is witnessed, as explained in further detail in Section 4.5.

The r*-suff theorem generated automatically by the defun-sk is harder to use because of the extra r*-proof-minimalp hypothesis (i.e. not any proof tree will do, but only a minimal one); thus, definductive generates a theorem r*-when-proof-validp that is like r*-suff but without the r*-proof-minimalp hypothesis, to be used in proofs instead of r*-suff (see Section 5.2).

4.5 Inference Rule Induction

The above minimality requirement on proof trees exists so that definductive can generate an induction scheme applicable directly to the predicates being defined instead of their proof trees. The scheme has the form of a (clique of) recursive function(s) whose exact results are irrelevant but whose recursive structure provides the scheme (a standard technique in ACL2).

For r*, the function is

```
(defun r*-induct (a b)
  (declare (xargs :measure (r*-proof-count (r*-proof a b))))
  (and (r* a b)
       (let ((proof (r*-proof a b)))
         (r*-proof-case proof :base nil
                              :refl nil
                              :trans (list (r*-induct proof.x proof.y)
                                           (r*-induct proof.y proof.z))))))
```

which terminates because, in each recursive call, the proof tree of the premise is (structurally) smaller than the one of the conclusion, and r*-proof picks a minimal proof tree for that premise, which is therefore also smaller than the one for the conclusion. Without the minimality requirement, r*-proof could pick a larger proof tree for a premise than the one that is part of the proof tree for the conclusion. This induction scheme lets one assume, for each inference rule, that the property being proved holds on all the premises, when proving that it holds on the conclusion. We call this *(inference) rule induction*.

For non-singleton predicate cliques, as in nat-trees and even-and-odd above, the *p*-induct functions are mutually recursive, and accompanied by a flag macro [21, make-flag], which instantiates the induction scheme (as with other inductions from mutually recursive functions in ACL2).

For singleton predicate cliques, an ACL2 induction rule [21, induction] is also generated; see Section 5.4 for an example of its use. ACL2 induction rules only apply to singly recursive functions.

4.6 Inference Rule Theorems

For each rule in :irules, definductive generates a theorem whose formula is the rule, in implication form, like the r*-base, r*-refl, and r*-trans theorems in Section 3. Each theorem is proved as explained in Section 3, one event per theorem, without induction. Each theorem is named after the predicate and the rule name; definductive allows rules to have the same name, so long as they have different conclusion predicates; this yields unique theorem names, unique proof type summands, etc.

The inference rule theorems are always valid rewrite rules, because their conclusions are calls of the predicates being defined. Since inference rules are used for specification, they should optimize clarity and readability, which is not always optimal for rewriting, e.g. in the ordering of premises for instantiating free variables [21, free-variables]. Better rewrite rules, e.g. with reordered premises, can be proved as theorems from the inference rule theorems; see Section 5.1.

4.7 Predicate Minimality

The *p* `-alt` constrained functions are generated by `definductive` in a single `encapsulate` event, following the pattern of Section 3. Given that each *p* `-alt` is locally defined as *p*, the proofs of the constraining theorems are straightforward, from the rule theorems for *p*.

As in Section 3, to generate the minimality theorems saying that *p* implies *p* `-alt`, `definductive` first generates theorems saying that each *p* `-proof-validp` implies *p* `-alt`. The latter are proved by induction on the *p* `-proof-validp` functions, one flag event [21, `make-flag`] per clique with two or more predicates, or a single theorem event per singleton clique; `:use` hints with suitable instances of the rule theorems that constrain the *p* `-alt` predicates prove the cases of the induction.[8]

Note the distinction between the minimality of the predicates just discussed (i.e. the 'smallest' requirement of the inductive definition) and the minimality of the proof trees discussed in Section 4.4 and Section 4.5 (i.e. a device to generate the induction scheme).

4.8 Monotonicity and Positivity

Each call of `definductive` generates theorems saying that the predicates it defines are the smallest ones satisfying the inference rules. As mentioned earlier in this section, the ability to do that relies on the syntactic restrictions on the inference rules, which ensure the monotonicity discussed in Section 2. Below is a sketch of the proof of monotonicity, but the correctness of the events generated by `definductive` does not rely on any meta proof, because they include the specific theorems for the predicates.

Let Φ be a function, defined in the style of Section 2, derived from some set of inference rules that satisfy the `definductive` restrictions. Let $\rho_1 \subseteq \rho_2$ be two relations. Take an arbitrary $\tau \in \Phi(\rho_1)$; then τ is a tuple corresponding to the conclusion of an instance of one of the inference rules where all the premises hold. We show that the same applies to $\Phi(\rho_2)$: the premises that do not reference the input relation still hold; the other premises are membership checks over ρ_2, which hold because the corresponding tuples are members of ρ_1, and $\rho_1 \subseteq \rho_2$; thus $\tau \in \Phi(\rho_2)$. Since τ was arbitrary, $\Phi(\rho_1) \subseteq \Phi(\rho_2)$. This generalizes to mutually inductive relations.

These restrictions are stricter than general positivity [1], and beyond even the strict positivity used by languages such as Rocq [12] and Lean [2] for inductive type definitions. Conjunctions like `(and (`*p* `...) (`*q* `...))`, where *p* and *q* are predicates being defined, or disjunctions like `(or (`*p* `...) (`*q* `...))` are strictly positive, but currently rejected by `definductive`. This is not a serious restriction, as such conjunctions can be rewritten into multiple premises, and such disjunctions into multiple rules; future versions of `definductive` may allow them. A more serious restriction concerns universal quantifications over infinite domains,[9] which preserve strict positivity but are not expressible in `definductive`: they would require `defun-sk`s that cannot be defined simultaneously with the predicates, and the corresponding proof trees would be infinitely branching. Quantification over finite domains (e.g. lists) can be accomplished in `definductive` via companion predicates that iterate over the finite domains and that

[8] For efficiency, `definductive` generates computed hints that limit the `:use` hints to the cases of the induction where they are actually needed. The cases are recognized from occurrences of terms of the form `(equal (`*p* `-proof-kind proof) :rule)`, where `:rule` is a proof tree type summand tag. In an early version of `definductive` that generated all the `:use` hints for all the cases, the proofs were quite slow with a few tens of inference rules, due to many (propositional) case splits.

[9] An example is how Rocq and Lean define the accessibility predicate over a relation R:

$$\frac{\forall y.\, R(y,x) \Longrightarrow acc_R(y)}{acc_R(x)}\ \text{ACC}$$

are defined together with the main predicates, by mutually dependent inference rules (see Section 6 for an example).[10] Negations like `(not (p ...))` and implications like `(implies (p ...) ...)` violate general positivity and are not expressible in `definductive`.

Other constraints described above require that the proof tree types be non-empty. This is only necessary for the admissibility of the FTY types—not for monotonicity. Consider the following definition, currently rejected by `definductive`:

```
(definductive no-proof-trees
  :preds ((p x))
  :irules ((recurse ((p x)) (p x))))
```

Such a definition would be strange, but not incoherent. The least fixpoint exists; it is the empty predicate. In fact, all definitions of this kind, i.e. calls of `definductive` with a predicate that cannot be assigned a level, would produce an empty predicate if allowed.

4.9 User's View

The previous subsections describe the workings of `definductive`, but a user of the tool does not need to understand all of those details. The user needs to understand:

- The definition of the *p*`-proof` proof trees and their organization into cliques, both of which mirror the inference rules. The tool handles their admission behind the scenes, by generating `:measure`s and `:base-case-override`s as needed.
- The definition of the *p*`-proof-validp` predicates, which also mirror the inference rules, and which provide the proof tree induction scheme (see Section 5.3).
- The definition of the *p* predicates, in terms of proof tree existence. The *p*`-proof-minimalp` predicates are an internal device: they may show up in proof subgoals but can normally be ignored; the user-facing theorem is *p*`-when-proof-validp` rather than *p*`-suff` (see Section 5.2).
- The definition of the *p*`-induct` functions, which provide the inference rule induction scheme (see Section 5.4).
- The inference rule theorems (see Section 5.1).
- The *p*`-alt` constrained predicates and their accompanying minimality theorems, e.g. to show inclusion of the *p* predicates in others that satisfy the inference rules, by functional instantiation (see Section 5.5).

5 Reasoning in ACL2

The events described in Section 4 provide ways to reason about the predicates defined by `definductive`, which are discussed in this section. As more experience is gained with `definductive`, better and more automated reasoning techniques might be developed.

5.1 Inference Rule Theorems

The inference rule theorems generated by `definductive` (see Section 4.6) are, logically speaking, introduction rules for the defined predicates, i.e. they prove calls of the predicates.

[10] It is in fact common, in ACL2, for functions to have mutually recursive companions that iterate over finite structures, e.g. `pseudo-termp` and `pseudo-term-listp`.

Among other applications, they are useful for proving *derived* inference rules, which are implications, often written like the defining inference rules, that correspond to compositions of the latter. For instance, given the rules in Section 1, the derived rule

$$\frac{R^*(x,y) \qquad R(y,z)}{R^*(x,z)}\ \text{TRANS-BASE-RIGHT}$$

corresponds to

$$\frac{R^*(x,y) \qquad \dfrac{R(y,z)}{R^*(y,z)}\ \text{BASE}}{R^*(x,z)}\ \text{TRANS}$$

Given a call of `definductive`, derived rules can be proved as theorems, suitably composing theorems for other rules. For example, given the `refl-trans-closure` call in Section 4,

```
(defthm trans-base-right (implies (and (r* x y) (r y z)) (r* x z)))
```

proves by rewriting with `r*-base` and `r*-trans`,[11] realizing the composition depicted above. As another example, given the `even-and-odd` call in Section 4.2, the derived rule

```
(defthm even-add-2 (implies (even n) (even (+ 2 n))))
```

proves from `odd-from-even` and `even-from-odd`.

This also works for varying (i.e. not fixed) compositions of inference rules, e.g.

```
(defthm even-add-double (implies (and (natp m) (even n)) (even (+ (* 2 m) n))))
```

proves by induction on `m`, corresponding to the composition of `m` applications of `even-add-2`.

5.2 Proof Tree Construction

The *p*`-when-proof-validp` theorems discussed in Section 4.4 are also, logically speaking, introduction rules for the defined predicates. They have the form

```
(defthm p-when-proof-validp
  (implies (and (p-proof-validp proof x1 x2 ...)
                (p-proofp proof))
           (p x1 x2 ...)))
```

To prove a call of *p* on certain values, this theorem needs a valid proof tree for *p* on those values, which can be explicitly constructed in ACL2, proved to be valid, and used to establish the hypothesis above.

[11] For the derived rule

$$\frac{R(x,y) \qquad R^*(y,z)}{R^*(x,z)}\ \text{TRANS-BASE-LEFT}$$

the theorem

```
(defthm trans-base-left (implies (and (r x y) (r* y z)) (r* x z)))
```

needs a `:use` hint with `r*-base`, because of the way ACL2 finds instantiations for free variables [21, `free-variables`]; this is not specific to `definductive`. An alternative to the `:use` hint is to prove, and rewrite with, a derived rule

$$\frac{R^*(y,z) \qquad R^*(x,y)}{R^*(x,z)}\ \text{TRANS-SWAPPED}$$

Similar observations apply to other theorems shown in this section.

For instance, given the `even-and-odd` call in Section 4, to show that ACL2's `evenp` and `oddp` imply `even` and `odd` on natural numbers, a proof tree for each natural number `n` can be constructed via

```
(mutual-recursion
 (defun proof-for-even (n) ; proof tree that n is even
   (declare (xargs :guard (and (natp n) (evenp n))))
   (b* (((when (zp n)) (even-proof-zero)) ; base case
        (odd-proof (proof-for-odd (1- n)))) ; proof tree that n-1 is odd
     (make-even-proof-from-odd :n (1- n) :premise1-proof odd-proof)))
 (defun proof-for-odd (n) ; proof tree that n is odd
   (declare (xargs :guard (and (natp n) (oddp n))))
   (b* (((when (zp n)) ...) ; impossible -- result is irrelevant
        (even-proof (proof-for-even (1- n)))) ; proof tree that n-1 is even
     (make-odd-proof-from-even :n (1- n) :premise1-proof even-proof))))
```

The proof trees are shown valid for the respective natural numbers via

```
(defthm-proof-for-even/odd-flag ; flag event for the mutual recursion above
  (defthm even-proof-validp-of-proof-for-even
    (implies (and (natp n) (evenp n)) (even-proof-validp (proof-for-even n) n))
    :flag proof-for-even)
  (defthm odd-proof-validp-of-proof-for-odd
    (implies (and (natp n) (oddp n)) (odd-proof-validp (proof-for-odd n) n))
    :flag proof-for-odd))
```

from which `even-when-proof-validp` and `odd-when-proof-validp` prove the desired theorems

```
(defthm even-when-evenp (implies (and (natp n) (evenp n)) (even n))
  :hints
  (("Goal" :use (:instance even-when-proof-validp (proof (proof-for-even n))))))
(defthm odd-when-oddp (implies (and (natp n) (oddp n)) (odd n))
  :hints
  (("Goal" :use (:instance odd-when-proof-validp (proof (proof-for-odd n))))))
```

Derived inference rules can also be proved by constructing proof trees for their conclusions from proof trees for their premises, as an alternative to the approach in Section 5.1. It is the same proof, either shallowly or deeply embedded in ACL2. While the approach in Section 5.1 is lighter, it may need `:use` hints or additional rewrite rules (cf. Footnote 11); with the proof tree construction approach, it is generally enough to enable the proof tree validity predicates.

A proof tree for *p* may be constructed from a proof tree for *q* (possibly the same predicate) to prove certain theorems of the form `(implies ...`*q*`... ...`*p*`...)`. For instance, given the `even-and-odd` call in Section 4, a simple example is

```
(defthm even-of-pred-when-odd (implies (odd n) (even (1- n)))
  ... :use (:instance even-when-proof-validp
                      (proof (odd-proof-from-even->premise1-proof (odd-proof n)))
                      (n (1- n)))...)
```

where the proof tree for `(even (1- n))` is constructed from the proof tree for `(odd n)`.

5.3 Proof Tree Induction

An induction scheme from the inductive definition of the *p* predicate(s) is that of the *p*`-proof-validp` function(s). Certain theorems with the predicate(s) as hypotheses can be proved via that induction scheme, which captures the idea of "all the ways in which the predicate(s) can hold".

For instance, given the `even-and-odd` call in Section 4, the fact that `even` and `odd` only hold on natural numbers is proved by induction via

```
(defthm-even-proof-validp-clique-flag
  (defthm natp-when-even-proof-validp
    (implies (even-proof-validp proof n) (natp n))
    :flag even-proof-validp)
  (defthm natp-when-odd-proof-validp
    (implies (odd-proof-validp proof n) (natp n))
    :flag odd-proof-validp))
```

from which it follows, opening the definitions of `even` and `odd`, that

```
(defthm natp-when-even/odd (implies (or (even n) (odd n)) (natp n)))
```

The same approach proves that ACL2's `evenp` and `oddp` hold when `even` and `odd` do.

5.4 Inference Rule Induction

Another induction scheme from `definductive` is that of the *p*`-induct` functions discussed in Section 4.5. It can be used in situations similar to the ones mentioned in Section 5.3, and it also captures the idea of "all the ways in which the predicate(s) can hold".

For example, given the `refl-trans-closure` call in Section 4, and a function `f` that preserves `r`

```
(defthm r-of-f (implies (r x y) (r (f x) (f y))))
```

the fact that `f` also preserves `r*` is proved by inference rule induction via

```
(defthm r*-of-f (implies (r* a b) (r* (f a) (f b)))
  :hints (("Goal" :induct (r* a b) ; inference rule induction
                  :in-theory (enable r*-base r*-refl r*-trans ...)
                  :expand ((r* a b) (r*-proof-validp (r*-proof a b) a b)))))
```

As shown by the hints, the proof still needs to deal with proof trees and their validity, but inference rule induction obviates the need for proving a lemma on proof trees by induction on them. The `:induct (r* a b)` hint works because `definductive` also generates an induction rule (see Section 4.5).

5.5 Functional Instantiation of the Minimality Theorems

The *p*`-alt-when-`*p* theorems can be functionally instantiated to prove the inclusion of the *p* predicates in some other predicates. The proof obligations of the functional instantiation are that those other predicates satisfy all the inference rules.

For example, the `r*-of-f` theorem in Section 5.4 can be alternatively proved via

```
(defthm r*-of-f (implies (r* a b) (r* (f a) (f b)))
  :hints (("Goal"
           :use (:functional-instance r*-alt-when-r*
                                      (r*-alt (lambda (a b) (r* (f a) (f b)))))
           :in-theory (enable r*-base r*-refl r*-trans))))
```

6 Example Application

We are using `definductive` in an ACL2 formalization of Remora, a rank-polymorphic higher-order functional programming language that supports any-dimensional arrays (scalars, vectors, matrices,

tensors, etc.) and automatically lifts operations from lower to higher dimensions (which is what 'rank-polymorphic' means) [19, 18]. The formalization is available in the ACL2 community books [22, `[books]/kestrel/remora`] and documented in the manual [21, `remora`]. This formalization is work in progress and a topic for another paper; here we focus on its use of `definductive`.

After codifying the Remora static and dynamic semantics as an executable checker and interpreter in ACL2, we started using `definductive` to formulate a more general higher-level specification of both semantics, mirroring the inference rules in the aforementioned Remora publications. At the time of this writing, we have defined a large part of the static semantics, which we are actively working on completing, and then we will tackle the dynamic semantics. Using the `definductive` reasoning facilities, we have proved a number of properties of the current inference rules for the static semantics; we are working on proving more and deeper properties, including the ones in the Remora publications, as well as the correctness of the executable checker and interpreter.

Remora features array types of the form `(A` *`T`* `(dims` *`d1 d2 ...`*`))`,[12] where *`T`* is an atom (i.e. non-array) type (e.g. a type of primitive data, or a function type), and *`d1`*, *`d2`*, etc. are non-negative integers representing the dimensions of the array. For example, `(A Int (dims 3 4))` is for 3-by-4 matrices of integers, `(A Bool (dims 5))` is for vectors of 5 booleans, `(A (-> Float Float) (dims 2 3 4))` is for 2-by-3-by-4 tensors of unary float operations, and `(A Int (dims))` is for scalar integers (distinct from vectors of 1 integer); null dimensions, e.g. `(A Int (dims 3 0 4))`, are allowed. The dimensions *`d1`*, *`d2`*, etc. may be not only constants, but also expressions with variables and a few arithmetic operations, e.g. `(A Int (dims $d (* 2 $d)))`[13] is a generic type of twice-as-wide-as-tall matrices of integers. The `(dims ...)` form is not the only way to denote the list of dimensions (called a 'shape' in Remora) of an array type: expressions involving `dims` and variables and concatenation may be used, e.g. `(A Int (++ (dims (+ 1 $d)) @s))`[14] is a generic type of arrays with at least one non-null dimension, where `@s` may consist of zero or more additional dimensions.

The Remora static semantics involves checking whether certain pairs of types are equivalent. Two array types are equivalent iff they have equivalent element types and equivalent shapes, e.g.

```
(A Int (++ (dims (* 2 $d) 4) @s))
```

is equivalent to

```
(A Int (++ (dims (+ $d $d)) (++ (dims (- (+ $e 4) $e)) @s)))
```

because the element types are syntactically equal (the strongest form of equivalence) and the shapes, although syntactically different, evaluate to the same ground shape (i.e. sequence of natural numbers) for all possible assignments of natural numbers, or sequences thereof, to the variables `$d`, `$e`, and `@s`.

We have used `definductive` to define the equivalence of shapes inductively via inference rules that formalize: for dimensions, an equational theory over multivariate polynomials, with variadic additions and multiplications, and with a Lisp-like notion of variadic subtraction; for shapes, an equational theory over sequences (i.e. a free monoid) of dimensions, with variadic concatenation operations.

For example, the inference rule for the commutativity of addition reads

```
(add-comm ((dimp dim1) (dimp dim2))                    ; name and premises
          (dim-eq (dim+ dim1 dim2) (dim+ dim2 dim1))) ; conclusion
```

and the inference rule for the associativity of concatenation reads

```
(append-assoc ((shapep shape1) (shapep shape2) (shapep shape3))
              (shape-eq (shp++ (shp++ shape1 shape2) shape3)
                        (shp++ shape1 (shp++ shape2 shape3))))
```

[12] The concrete syntax of Remora is based on S-expressions.

[13] In Remora, dimension variables start with `$`.

[14] In Remora, shape variables start with `@`.

Since these and similar rules are formulated on binary versions of the variadic operations, we also have inference rules to normalize away non-binary versions, e.g.[15]

```
(add0 () (dim-eq (dim+) (dim-const 0))) ; sum of none
(add1 ((dimp dim)) (dim-eq (dim+ dim) dim)) ; sum of one
(add3m ((dimp dim1) (dimp dim2) (dim-listp dims) (consp dims))
       (dim-eq (dim-add (list* dim1 dim2 dims)) ; sum of three or more
               (dim-add (cons (dim+ dim1 dim2) dims))))
```

All these rules, with others, are part of the inductive definition of dimension and shape equivalence.

We proved known properties of the defining algebraic laws as derived inference rules, e.g.

```
(defthm dim-eq-neg-add
  (implies (and (dimp dim1) (dimp dim2))
           (dim-eq (dim- (dim+ dim1 dim2))
                   (dim+ (dim- dim1) (dim- dim2)))))
```

using the proof construction technique in Section 5.2.

We also proved that every dimension is equivalent to a normalized one. For addition, we defined a predicate `dim-binaddp` saying whether a dimension contains only binary additions, and a predicate saying whether a dimension is equivalent to one with only binary additions

```
(defun-sk dim-eq-to-binadd-p (dim)
  (exists (dim1) (and (dim-eq dim dim1) (dim-binaddp dim1))))
```

We defined a normalization function `binarize-add-in-dim` that also returns a proof tree of the equivalence; we proved that the dimension result always satisfies `dim-binaddp` and that the proof tree result always proves the equivalence of the argument and result dimensions. This proof-generating normalization function provides a witness for the existential quantifier in `dim-eq-to-binadd-p`, showing that this predicate holds for every dimension. Besides addition, we proved analogous properties for all the other variadic operations, for both dimensions and shapes. These properties validate our normalization rules.

We have also used `definductive` to define the static validity of the constructs of Remora, such as typing judgments for expressions of the form `(expr-ok ... expr type)`, where the ellipsis omits details of the context (types of free variables, and other information), `expr` is an expression, and `type` is a type. The judgment says that `expr` is statically valid with type `type`.

For instance, (a simplified form of)[16] the inference rule for function application reads

```
(app ((expr-ok ... fun (type-fun (type-array type-in shape-in)
                                 (type-array type-out shape-out)))
      (expr-ok ... arg (type-array type-in shape-in)))
     (expr-ok ... (expr-app fun arg) (type-array type-out shape-out)))
```

The function application expression is `(expr-app fun arg)`, where `fun` is the function expression and `arg` is the argument expression. The first premise says that `fun` is statically valid with a function type; the second premise says that `arg` is statically valid with the input type of the function; the conclusion says that `(expr-app fun arg)` is statically valid with the output type of the function.

All these predicates are inductively defined together with versions that iterate over lists, mentioned in Section 4.8 as the approach to handle quantification over finite domains. For instance, `expr-ok` is accompanied by `exprs-ok`, where `(exprs-ok ... exprs types)` says that each expression in the list `exprs` is statically valid with the corresponding type in the list `types`. The inference rule for Remora

[15] The macro `dim+` takes zero or more arguments. It expands to a call of the constructor `dim-add` on the list of its arguments.

[16] The full form would need more background on Remora than appropriate to the purpose of this paper.

non-empty frame expressions (which assemble higher-dimensional values from equally shaped lower-dimensional values) has a premise that iterates over a list of sub-expressions

```
(frame-nonempty (... (exprs-ok ... exprs ...))
                (expr-ok ... (expr-frame ... exprs) ...))
```

The companion predicate is characterized by structural inference rules for lists

```
(nil (...) (exprs-ok ... nil nil)) ; empty list
(cons (... (expr-ok ... expr type) (exprs-ok ... exprs types))
      (exprs-ok ... (cons expr exprs) (cons type types))) ; non-empty list
```

which imply that the lists of expressions and types have the same length.

7 Related Work

The reification of proof trees to define predicates inductively via inference rules has several precedents in ACL2 [16, 20, 5, 3] and NQTHM [17], but only for specific predicates and rules, and not always with explicit predicate definitions with existentials. To our knowledge, `definductive` is the first ACL2 tool that automates the approach for general predicates and rules. The aforementioned precedents independently provide validation for the usefulness of the approach and motivation for its general automation.

Proof trees overcome first-order logic's inability to directly express the 'smallest' requirement of inductive definitions. The recently developed set theory in ACL2 [21, `zfc`] can represent predicates as sets and quantify over them, but then predicates are logical individuals instead of logical functions, and it remains to investigate how seamlessly the two levels can interoperate, with most ACL2 libraries not phrased in terms of set theory. Higher-order logic can express the 'smallest' requirement directly, but higher-order theorem provers generally provide constructs to automate the definition from user inputs similar to `definductive`, i.e. predicate signatures and inference rules.

Isabelle [8] supports inductive definitions via inference rules [13]. Each inference rule must have a conclusion of the form $t \in S$, where S is a set (i.e. predicate) being defined and t is a term: this is the same as in our `definductive` tool. However, premises can have the more general form $t \in \mathcal{M}(S)$, where $\mathcal{M}$ is a monotone operator. From the inference rules, Isabelle automatically generates the appropriate definitions and theorems. It also supports coinductive definitions, via greatest, instead of least, fixpoints.

HOL [7] supports inductive definitions via inference rules that are similar to Isabelle's in terms of restrictions on the premises. It also supports coinductive definitions.

PVS [14] supports inductive and coinductive predicate definitions, via inference rules expressed as a single definition per predicate, interpreted as a least or greatest fixpoint, whose body is the disjunction of the conjunctions of premises corresponding to each rule, each conjunction possibly surrounded with an explicit existential quantification for the variables in the premises but not in the conclusion. The restrictions on these rules are similar to Isabelle's; monotonicity proof obligations are generated as needed.

Rocq [15] is built on the Calculus of Inductive Constructions [4, 12]: its basic mechanism for building compound types, with type constructors as introduction rules, also works for inductive predicates, which are a kind of type in Rocq's logic. Occurrences of the type must be strictly positive in the premises, i.e. they may not occur anywhere in the domain of a function type (where implication and universal quantification are kinds of function types). Strict positivity is less restrictive than `definductive`'s restrictions, but more restrictive than Isabelle's monotonicity condition. For case analysis and induction schemes, Rocq introduces helper functions using the primitive `match` and `fix` operators. Rocq also supports coinductive datatypes, using `cofix` instead of the `fix` primitive.

Lean [11] is built on a dependent type theory closely related to Rocq's, and supports inductive types (including predicates) with the same strict positivity restriction. Lean generates a primitive recursor directly from each type [2]. Lean also supports a separate mechanism for defining (co)inductive predicates using the `inductive_fixpoint` and `coinductive_fixpoint` termination clauses [10, 9]; the former is a least fixpoint, and the latter a greatest fixpoint. Unlike the general mechanism for inductive types, this does not require strict positivity; instead, it requires monotonicity, like Isabelle, HOL, and PVS.

8 Future Work

The predicates generated by `definductive` are not guard-verified, because the inference rules may contain arbitrary user-supplied terms, which may make the validity predicates, and thus the defined predicates, not even guard-verifiable. But it is generally possible to make them guard-verifiable by including appropriate premises, in the appropriate order, in the inference rules; then the guards can be verified, often automatically, after the `definductive` call. A possible extension of `definductive` is an option to generate trailing guard-verification events, or attempt guard verification as part of the function definitions, perhaps with an additional option to supply hints. A more elaborate approach is to have `definductive` calculate the guard obligations prior to submitting the events, presented to the user as applicability conditions to prove [21, `event-macro-applicability-conditions`], from which the guard verification proofs are automatically generated.

Since the inference rules define the predicates, they should be generally optimized for clarity, which is sometimes at odds with their use as rewrite rules in ACL2; this is even more the case when including premises for guard verification as discussed above. While better rewrite rules can be manually proved as derived inference rules, `definductive` could optionally generate such theorems in certain cases, including alternatives that bind different free variables in premises (cf. Section 4.6).

Besides being a theorem, a derived inference rule is implicitly accompanied by an ability to construct a proof tree that links its conclusion to the proof trees of its premises. Such construction could be reified into proof tree construction code that `definductive` or a companion macro could generate, based on suitable user inputs.

Although `definductive` generates non-executable predicates, the predicates may be computable, and in a way that can be systematically determined from the rules, as in bidirectional typing [6]. An interesting extension of `definductive` is to optionally generate executable definitions of the predicates based on analysis of the inference rules, also proving their equivalence to the non-executable definitions in terms of existence of proof trees.

9 Conclusion

The `definductive` tool brings to ACL2 a capability that has long been present in higher-order theorem provers. It supports the declarative definition of (possibly non-computable) predicates that naturally arise when formalizing logics and programming languages; it provides ways to reason about the predicates, including relating them to executable implementations. The tool is being successfully used in the formalization of Remora, discussed in Section 6.

The realization of this capability in ACL2 is less straightforward and more restrictive than in higher-order theorem provers. The main additional restriction is the one on universal quantification over infinite domains, discussed in Section 4.8, but so far we have not found it to be a practical limitation.

AI Use

Both authors routinely use AI (e.g. Claude Code) in their work, along with ACL2, Emacs, and other tools. But they do not just trust the AI for anything critical, carefully reviewing and revising everything it produces. They ultimately rely on their own judgment and expertise, and remain solely responsible for the content of this paper.

References

[1] Peter Aczel (1977): *An Introduction to Inductive Definitions*. In Jon Barwise, editor: *Handbook of Mathematical Logic*, *Studies in Logic and the Foundations of Mathematics* 90, Elsevier, pp. 739–782, doi:10.1016/S0049-237X(08)71120-0.

[2] Mario Carneiro (2019): *The Type Theory of Lean*. Master's thesis, Carnegie Mellon University. Available at `https://github.com/digama0/lean-type-theory`.

[3] Alessandro Coglio, Eric McCarthy & Eric Smith (2023): *Formal Verification of Zero-Knowledge Circuits*. In: *18th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2023)*, *Electronic Proceedings in Theoretical Computer Science (EPTCS)* 393, pp. 94–112, doi:10.4204/EPTCS.393.9.

[4] Thierry Coquand & Christine Paulin (1988): *Inductively Defined Types*. In Per Martin-Löf & Grigori Mints, editors: *International Conference on Computer Logic (COLOG-88)*, *Lecture Notes in Computer Science* 417, Springer, pp. 50–66, doi:10.1007/3-540-52335-9_47.

[5] Jared Davis (2009): *A Self-Verifying Theorem Prover*. Ph.D. thesis, The University of Texas at Austin.

[6] Jana Dunfield & Neel Krishnaswami (2021): *Bidirectional Typing*. *ACM Computing Surveys* 54(5), pp. 98:1–98:38, doi:10.1145/3450952.

[7] *The HOL Theorem Prover*. Available at `https://hol-theorem-prover.org`.

[8] *The Isabelle Theorem Prover*. Available at `https://isabelle.in.tum.de`.

[9] *The Lean Language Reference, Version 4.33.0: The `coinductive_fixpoint` Clause*. Available at `https://lean-lang.org/doc/reference/4.33.0/Definitions/Recursive-Definitions/#coinductive-fixpoint-clause`.

[10] *The Lean Language Reference, Version 4.33.0: The `inductive_fixpoint` Clause*. Available at `https://lean-lang.org/doc/reference/4.33.0/Definitions/Recursive-Definitions/#inductive-fixpoint-clause`.

[11] *The Lean Prover*. Available at `https://lean-lang.org`.

[12] Christine Paulin-Mohring (1993): *Inductive Definitions in the System Coq — Rules and Properties*. In Marc Bezem & Jan Friso Groote, editors: *International Conference on Typed Lambda Calculi and Applications (TLCA '93)*, *Lecture Notes in Computer Science* 664, Springer, pp. 328–345, doi:10.1007/BFb0037116.

[13] Lawrence C. Paulson (1994): *A Fixedpoint Approach to Implementing (Co)inductive Definitions*. In Alan Bundy, editor: *Automated Deduction — CADE-12*, *Lecture Notes in Computer Science* 814, Springer, pp. 148–161, doi:10.1007/3-540-58156-1_11.

[14] *The PVS Theorem Prover*. Available at `https://sri-fm.github.io/pvs`.

[15] *The Rocq Prover*. Available at `https://rocq-prover.org`.

[16] José-Luis Ruiz-Reina, José-Antonio Alonso, María-José Hidalgo & Francisco-Jesús Martín-Mateos (2000): *Formalizing Rewriting in the ACL2 Theorem Prover*. In John A. Campbell & Eugenio Roanes-Lozano, editors: *International Conference on Artificial Intelligence and Symbolic Computation (AISC 2000)*, *Lecture Notes in Artificial Intelligence* 1930, Springer, pp. 92–106, doi:10.1007/3-540-44990-6_7.

[17] Natarajan Shankar (1988): *A Mechanical Proof of the Church–Rosser Theorem*. *Journal of the ACM* 35(3), pp. 475–522, doi:10.1145/44483.44484.

[18] Justin Slepak (2019): *A Typed Programming Language – The Semantics of Rank Polymorphism*. Ph.D. thesis, Northeastern University. Available at `https://www.khoury.northeastern.edu/~jrslepak/Dissertation.pdf`.

[19] Justin Slepak, Olin Shivers & Panagiotis Manolios (2019): *The Semantics of Rank Polymorphism*. *CoRR* abs/1907.00509, doi:10.48550/arXiv.1907.00509. arXiv:1907.00509.

[20] Sol Swords & William R. Cook (2006): *Soundness of the Simply Typed Lambda Calculus in ACL2*. In: *6th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2-2006)*, Association for Computing Machinery, pp. 35–39, doi:10.1145/1217975.1217982.

[21] The ACL2 Community: *The ACL2 Theorem Prover and Community Books: Documentation*. Available at `http://acl2.org/manual`.

[22] The ACL2 Community: *The ACL2 Theorem Prover and Community Books: Source Code*. Available at `http://github.com/acl2/acl2`.