Proving Calculational Proofs Correct

Andrew T. Walter Ankit Kumar Panagiotis Manolios

Khoury College
Northeastern University
Massachusetts, USA

walter.a@northeastern.edu kumar.anki@northeastern.edu p.manolios@northeastern.edu

Teaching proofs is a crucial component of any undergraduate-level program that covers formal reasoning. We have developed a calculational reasoning format and refined it over several years of teaching a freshman-level course, "Logic and Computation", to thousands of undergraduate students. In our companion paper [28], we presented our calculational proof format, gave an overview of the calculational proof checker (CPC) tool that we developed to help users write and validate proofs, described some of the technical and implementation details of CPC and provided several publicly available proofs written using our format. In this paper, we dive deeper into the implementation details of CPC, highlighting how proof validation works, which helps us argue that our proof checking process is sound.

1 Introduction

Calculational Proof Checker (CPC) is a tool designed to help teach undergraduate computer science students how to write proofs. In a previous work [28] we presented the calculational proof format used by CPC and gave an overview of its design and implementation. Here we provide additional details about CPC's implementation and provide an argument for CPC's soundness.

CPC was designed for Manolios' freshman-level CS2800 "Logic and Computation" course [18], which uses the ACL2 Sedan (ACL2s) theorem prover [4, 12] to introduce logic and formal reasoning. ACL2s extends ACL2 with several additional features, including the `defdata` data definition framework [8], the `cgen` counterexample generation framework [5–7, 9], a termination analysis system using calling context graphs [23] and ordinals [20–22] and property-based modeling and analysis. As nearly anyone who has taught a formal reasoning class can attest to, teaching students how to identify what is a proof and what is not is challenging, and teaching students how to write proofs is even more so. The choice of proof format is highly impactful from a pedagogical standpoint, and therefore we put substantial effort into developing ours based on many years of experience teaching CS2800. The proof format we use in CS2800 is heavily inspired by the calculational proof style popularized by Dijkstra [10, 11]. Dijkstra's proof format is appropriate here because (1) its linear proofs are easier to check in a local manner (2) explicit context forces students to identify which parts of the context are used to discharge each step and (3) it is designed for human consumption rather than for a proof assistant, making these skills highly transferable.

CPC checks proofs in three *phases*. Phases 0 and 1 are intended to find problems with the proof in a way that is aimed at generating actionable and high-quality feedback for the user, and were discussed in detail in the companion paper [28]. Phase 2 involves translating the proof into one or more ACL2s theorems with proof-builder [1, `proof-builder`] instructions and checking these theorems inside of ACL2s. Therefore, the soundness of CPC reduces to the soundness of the proof-builder and ACL2s.

Our contributions include: (1) a method for translating calculational proofs into ACL2s theorems checkable by an unmodified ACL2s instance, (2) a proof of soundness of CPC and (3) several exten-

Submitted to:
ACL2 2023

sions and libraries for ACL2s we developed for CPC. The source code for CPC is available in a public repository [27].

2 Proof Format

We illustrate our proof format with an example proof of a conjecture, shown in Figure 1. Notice that the proof document starts with ACL2s definitions of relevant functions required for the proof. We use ACL2s' `definec` to define functions with input and output types. We will discuss `definec` in more detail later, but for now one should read the definition `(definec aapp (a :tl b :tl) :tl ...)` as the definition of a function `aapp` that takes as argument two true lists `a` and `b` and returns a true list. We also use ACL2s' `property` form to state an ACL2 theorem. The first argument to that form describes type constraints on free variables used in the form; in this case, one can read `(property assoc-append (x :tl y :tl z :tl) <body>)` as `(defthm assoc-append (implies (and (tlp x) (tlp y) (tlp z)) <body>)`.

In our proof format, proofs and ACL2s expressions can be arbitrarily interleaved, allowing for example a user to define an ACL2s function, write a CPC proof about that function, and then use that proof to justify the admission of another ACL2s function. Users can also define helper lemmas in ACL2s using standard ACL2s proof techniques and subsequently apply those lemmas in a CPC proof, as is done in this example with the ACL2s lemma `assoc-append`.

The proof starts with a named conjecture "`revt-rrev-help`" followed by the expression to be proved. In this case, we want to prove the conjecture using induction, so we specify that we are doing a proof by induction and provide the function that gives rise to the induction scheme we want to use. We then provide a number of subproofs (Induction Case 0 through 2), one for each proof obligation that induction using the specified scheme gives rise to. In this case, each of these is an equational reasoning proof, though in general any number of them could be induction proofs instead. For each equational reasoning proof, we provide an expression to be proved. If the expression is of the form $A \rightarrow B \rightarrow C$ then we require that the user use the logical rule of exportation to eliminate the nesting of implications and state $(A \wedge B) \rightarrow C$ in the *exportation step*. This must be done recursively, *e.g.* if there is an expression of that form in the antecedent of an implication. If desired, the statement may also be transformed into an equivalent one using propositional logic rules during this step. A contract completion step (which will be discussed in detail later) comes next, if applicable. The user then writes out the proof context (the hypotheses of the contract completed statement, if it is an implication) as a labeled list of *context items*. Additional context items can be added in the *derived context*. In Induction Case 2, derived context items are used to derive the consequent of the induction hypothesis so that it can be used more easily in the proof. Each derived context item has a list of justifications. If one can derive `nil` (false) in a derived context item, there is no need to provide the rest of the sections for that proof. The proof *goal* (the consequent of the contract completed statement if it is an implication, or the contract completed statement itself otherwise) is then listed, followed by a sequence of *proof steps*. Each proof step consists of two statements separated by a relation and a set of justifications for that step.

A simplified and compacted version of our proof grammar is shown in Figure 2. For brevity we do not include the complete grammar of our proof format, but it is available in our repository [27]. Our companion paper [28] has more examples of proofs written for CPC, both from CS2800 assignments and Dijkstra's EWDs [10]. We recommend that interested readers review those example proofs.

```
(definec aapp (a :tl b :tl) :tl
  (if (endp a)
      b
    (cons (first a) (aapp (rest a) b))))

(definec rrev (x :tl) :tl
  (if (endp x)
      nil
    (aapp (rrev (rest x)) (list (first x)))))

(definec revt (x :tl acc :tl) :tl
  (if (endp x)
      acc
    (revt (rest x) (cons (first x) acc))))

(property assoc-append (x :tl y :tl z :tl)
    (equal (aapp x (aapp y z))
           (aapp (aapp x y) z)))

Lemma revt-rrev-help:
(implies (and (tlp x)
              (tlp acc))
         (equal (revt x acc)
                (aapp (rrev x) acc)))

Proof by: Induction on (revt x acc)

Induction Case 0:
;; Elided case where (not (and (tlp x) (tlp acc)))
QED

Induction Case 1:
(implies (endp x)
  (implies (and (tlp x) (tlp acc))
           (equal (revt x acc)
                  (aapp (rrev x) acc))))

Exportation:
(implies (and (tlp x) (tlp acc) (endp x))
         (equal (revt x acc)
                (aapp (rrev x) acc)))

Context:
C1. (tlp x)
C2. (tlp acc)
C3. (endp x)

Derived Context:
D1. (equal x nil) { C1, C3 }

Goal: (equal (revt x acc) (aapp (rrev x) acc))

Proof:
(revt x acc)
== { D1, Def revt }
acc
== { Def aapp }
(aapp nil acc)
== { Def rrev, D1 }
(aapp (rrev x) acc)

QED
```

```
Induction Case 2:
(implies (and (not (endp x))
  (implies
    (and (tlp (cdr x))
         (tlp (cons (car x) acc)))
         (equal (revt (cdr x) (cons (car x) acc))
                (aapp (rrev (cdr x))
                      (cons (car x) acc)))))
  (implies (and (tlp x) (tlp acc))
           (equal (revt x acc)
                  (aapp (rrev x) acc))))

Exportation:
(implies
  (and (tlp x)
       (tlp acc)
       (not (endp x))
       (implies
         (and (tlp (cdr x))
              (tlp (cons (car x) acc)))
              (equal (revt (cdr x)
                           (cons (car x) acc))
                     (aapp (rrev (cdr x))
                           (cons (car x) acc)))))
  (equal (revt x acc)
         (aapp (rrev x) acc)))

Context:
C1. (tlp x)
C2. (tlp acc)
C3. (not (endp x))
C4. (implies (and (tlp (cdr x))
                  (tlp (cons (car x) acc)))
      (equal (revt (cdr x) (cons (car x) acc))
             (aapp (rrev (cdr x))
                   (cons (car x) acc))))

Derived Context:
D1. (tlp (cdr x)) { C1, C3, Def tlp }
D2. (tlp (cons (car x) acc)) { C2, C3, Def tlp }
D3. (equal (revt (cdr x) (cons (car x) acc))
        (aapp (rrev (cdr x)) (cons (car x) acc)))
    { D1, D2, C4, MP }

Goal: (equal (revt x acc) (aapp (rrev x) acc))

Proof:
(revt x acc)
== { Def revt, C3 }
(revt (cdr x) (cons (car x) acc))
== { D3 }
(aapp (rrev (cdr x)) (cons (car x) acc))
== { Def aapp, car-cdr axioms }
(aapp (rrev (cdr x)) (aapp (list (car x)) acc))
== { Lemma assoc-append ((x (rrev (cdr x)))
                         (y (list (car x))) (z acc)) }
(aapp (aapp (rrev (cdr x)) (list (car x))) acc)
== { C3, Def rrev, car-cdr axioms }
(aapp (rrev x) acc)

QED

QED
```

Figure 1: An example proof written in our proof format. This proof file is available in our repo [27] at the path `example/ind-examples/pass/rrev.proof`.

⟨*ProofDocument*⟩ ::= (⟨*Proof*⟩ | ⟨*SExpression*⟩)+

⟨*Proof*⟩ ::= ⟨*Type*⟩ $\mathscr{V}$: $\mathscr{E}$ [Exportation: $\mathscr{E}$] [Contract Completion: $\mathscr{E}$] ⟨*Body*⟩ QED

⟨*Body*⟩ ::= ⟨*Simple*⟩ | ⟨*Inductive*⟩

⟨*Simple*⟩ ::= [Context: ⟨*Ctx*⟩] [Derived Context: ⟨*Dtx*⟩] Goal: $\mathscr{E}$ Proof: ⟨*Seq*⟩

⟨*Inductive*⟩ ::= Proof by: $\mathscr{E}$ [⟨*ContractCase*⟩] ⟨*BaseCase*⟩+ ⟨*InductionCase*⟩*

⟨*ContractCase*⟩ ::= Contract Case $\mathbb{N}$: $\mathscr{E}$ ⟨*Body*⟩ QED

⟨*BaseCase*⟩ ::= Base Case $\mathbb{N}$: $\mathscr{E}$ ⟨*Body*⟩ QED

⟨*InductionCase*⟩ ::= Induction Case $\mathbb{N}$: $\mathscr{E}$ ⟨*Body*⟩ QED

⟨*Type*⟩ ::= Conjecture | Property | Lemma | Theorem

⟨*Ctx*⟩ ::= (C$\mathbb{N}$: $\mathscr{E}$)*

⟨*Dtx*⟩ ::= (D$\mathbb{N}$: $\mathscr{E}$)*

⟨*Seq*⟩ ::= $\mathscr{B}$ ($\mathscr{R}$ {⟨*Hint*⟩ (, ⟨*Hint*⟩)*} $\mathscr{B}$)*

⟨*Hint*⟩ ::= ⟨*Type*⟩ $\mathscr{V}$ [$\mathscr{S}$] | C$\mathbb{N}$ | D$\mathbb{N}$ | Def $\mathscr{F}$ | $\mathscr{A}$ | algebra | obvious | PL | MP

Figure 2: EBNF grammar for our calculational proofs where, in the ACL2s universe, $\mathscr{V}$ is a fresh variable or natural number, $\mathscr{E}$ is an expression, $\mathbb{N}$ is a natural number, $\mathscr{B}$ is a Boolean expression, $\mathscr{R}$ is a binary relation on Boolean expressions, $\mathscr{S}$ is an association list used to represent a valid substitution, $\mathscr{F}$ is a valid function name, $\mathscr{A}$ is an axiom, and ⟨*SExpression*⟩ is an ACL2s event form. PL and MP stand for Propositional Logic and Modus Ponens hints respectively. Items in square brackets are optional.

3 System Architecture

We summarize the architecture of CPC here. We refer interested readers to our companion paper [28] for a detailed description. The architecture of CPC consists of three primary pieces — the user interface, the Xtext [31] language support, and the ACL2s backend. Xtext is a framework for building domain-specific languages, and automatically generates a lexer and parser from our proof format grammar. A user submits a proof document for checking through one of the CPC interfaces, which is parsed by Xtext and turned into an Xtext document. Next, an Xtext *validator* that we developed runs on the Xtext document, translating it into a form that is usable by the ACL2s backend and invoking that backend. The ACL2s backend then runs through the proof document and reports any issues back to the Xtext validator, which sends that information back to the user interface for reporting to the user. A major benefit of using Xtext in this way is the ability to associate errors detected by the backend with regions of the user's proof document. This means that, for example if a step is determined to be incorrect, we can produce error underlining for that step to help the user localize the error. Xtext also allows us to provide IDE features like syntax highlighting and code folding with minimal additional effort.

The ACL2s backend of CPC is implemented using our ACL2s Systems Programming methodology, which we described in an ACL2 Workshop paper last year [29]. That is, the backend is implemented mainly in "raw Lisp" and makes queries to ACL2s using the API described in our paper. This allows us to use programming constructs that are not legal in logic-mode ACL2s code, like the Common Lisp condition system.

As the proof format's grammar (see Figure 2) specifies, a proof document consists of a sequence of elements, where each element is either a proof or an event form (a call to any of the ACL2s event functions—in our examples, primarily `property`, `defdata` and `definec`). When CPC is run on a proof document, it processes each element of the proof document in sequence, evaluating the element in ACL2s using `ld` if the element is an event form, or performing proof checking and generation if the element is a proof. Operating in this way adds some complexity but also makes CPC more flexible. For example, a user can define an ACL2s function, write a proof about that function, and then use that proof to justify the admission of another ACL2s function. This would not be possible if we only supported documents where a set of ACL2s expressions (or a book) ran before all of the proofs. If an event form evaluation or a proof check fails, CPC will report an error to the user but will continue to operate on subsequent elements inside the proof document.

As previously stated, CPC performs proof checking in three phases. Phases 0 and 1 are primarily designed to find problems in a proof document that we can provide actionable and high-quality feedback for. In Phase 2, CPC will translate the proof file into appropriate ACL2s theorems complete with proof-builder commands, which are then run through ACL2s to confirm that the proofs in that file are correct. Our soundness argument is based entirely on Phase 2.

4 Proof Checking

Our prior work [28] describes Phases 0 and 1 in detail. Here we will summarize Phases 0 and 1 and describe some relevant aspects in more detail.

Phase 0 is a quick, syntactic check of the proof document performed by Xtext. This is provided as part of the parser that Xtext generates from our grammar. Phase 1 is performed in the ACL2s backend, and can itself be broken up into performing three checks. The first is to check that the initial setup of the proof—the contract completion, exportation, context and derived context, all of which we will discuss

in more detail later—is correct. The second is to check that all of the steps (for a non-inductive proof) or all of the subproofs (for an inductive proof) are correct. For a non-inductive proof, the third step is to confirm that the conjunction of the steps is sufficient to prove the statement under consideration. For an inductive proof, the third step is to confirm that the subproofs constitute the proof obligations of the induction proof that the user specified.

4.1 Guards and Contract Completion

Students in CS2800 are taught to reason about programs. Using ACL2s' `defdata` [1, `defdata`] is helpful as it is a natural way to introduce students to contract-driven development. Students write all of their functions using ACL2s' `definec`, which requires that the user specify the type of input arguments to the function as well as the type that the function outputs. `definec` [1, `definec`] is hooked into ACL2's guard system [1, `guard`] in such a way that a function defined using `definec` will have guards that assert that its arguments satisfy their specified types. Recall that the *guard obligations* for an expression is the sequence of conditions that must be true to satisfy the guards for every function call inside that expression. The *function contract* for a `definec` function is the statement that if all arguments to the function in some function call satisfy the specified input types (the input contract for the function is satisfied), that call evaluates to a value that satisfies the specified output type. When a `definec` form is evaluated, in addition to proving termination like `defun`, ACL2s will prove that the function's contract holds and will perform guard verification of the function's body. Guard verification is the process of proving that the guard obligations of an expression hold. Note that each `defdata` type has a "type predicate" associated with it—a function of one argument that evaluates to true if and only if that argument is a member of the corresponding type. The function contract theorems that `definec` submits are suffixed with `-CONTRACT` and `-CONTRACT-TP`.

We require that the statement a user is trying to prove in CPC has an empty sequence of guard obligations (equivalently, the guard obligations are all satisfied), or if this is not the case, we require that the user perform *contract completion* on the statement before proving it. Contract completion refers to the process of adding appropriate hypotheses to a statement to satisfy its guard obligations. We will also refer to the resulting statement after contract completion as the contract completion of the original statement. Performing contract completion on a statement of course changes the logical meaning of the statement. From a pedagogical standpoint, forcing users to perform contract completion helps us highlight the correspondence between the statements being proved and the code (the executable bodies of the functions in the statement). The way that `definec` works is relevant here—the logical definition of a function admitted using `definec` states that a call to the function with inputs that don't satisfy the function's input contract will evaluate to an arbitrary value satisfying the function's specified return type. A consequence of this is that a `definec` function cannot be expanded into its user-provided definition unless it is known that the function's input contract is satisfied. It is important to note that this is different from simply adding guards to a `defun`, as guards do not affect either the semantics of a function definition or the theorem prover [1, `guard-miscellany`]. Enforcing that statements are contract completed eliminates the possibility of errors or counterexamples due to guard violations ("type errors").

Note that the order in which hypotheses appear in a conjunction matters, as `and` in ACL2 is logically just syntactic sugar for `if` statements and the type information that a hypothesis provides might be necessary to satisfy the guards of a subsequent hypothesis. For example, if the original expression was `(implies (in e l) (consp l))` and the ACL2s definition of `in` requires that `(tlp l)`, the correct contract completion of the statement is `(implies (and (tlp l) (in e l)) (consp l))` and **not** `(implies (and (in e l) (tlp l)) (consp l))`.

It would be simpler to enforce that users provide contract completed statements at the get-go, but

having users perform contract completion inside of CPC has some advantages. In particular, having both the original statement and the contract completed statement inside of CPC allows us to check that the contract completion was done appropriately, *e.g.* that only the necessary hypotheses were added to satisfy the guard obligations. We do not guarantee CPC's soundness when the user provides a *non-trivial contract completion* — one that is not syntactically equivalent to the exported statement (if provided) or original statement (otherwise). To be clear, the only situtation in which a non-trivial contract completion must be provided is when the original statement has a non-empty sequence of guard obligations, that is, there is at least one function call in the statement with an input contract that is not provably always true. If a user provides a non-trivial contract completion, we currently produce a warning notifying the user of the potential for unsoundness and recommending they update the original conjecture so that it is contract completed.

An example of a statement with a trivial contract completion is `(implies (and (tlp x) (tlp y)) (equal (app x y) (app y x)))`. Since the ACL2s definition of `app` only requires that its arguments are true lists, the two antecedents ensure that the arguments to `app` are true lists and both `tlp` and `equal` have no guards, the guard obligations for this statement are trivially true and thus no antecedents need to be added during contract completion.

An example of a statement with a non-trivial contract completion is `(implies (in e l) (consp l))`, given the definition of `in` requires that its second argument is a true list. Since the guard obligations for this statement (just `(tlp l)`) are not trivially true, a non-trivial contract completion is required. In this case, `(tlp l)` must be added as an antecedent before the `(in e l)` hypothesis, so the only possible correct contract completion is `(implies (and (tlp l) (in e l)) (consp l))`.

4.2 Proof Building Blocks

The basic building block of an equational reasoning proof is a proof step — a statement that two expressions satisfy some relation, justified by one or more hints. In general, a step in an equational reasoning proof in our format will look like (with α and β being S-expressions and R being either a relation or an alias for a relation):

```
α
R { H_1, ..., H_n }
β
```

We say that this step is correct if and only if ACL2s can prove the statement `(R α β)` under an appropriate set of hypotheses and when constrained to an appropriate theory. As we will discuss shortly, the appropriate set of hypotheses and the appropriate theory both are influenced by the hints $H = \{H_1, ..., H_n\}$ that the user provided, but also by the context of the proof that the step is contained inside.

Hints

CPC supports several types of hints for justifying reasoning steps. These include `C`*i* and `D`*i* which refer to context and derived context items respectively, `def` *foo* which allows one to reference the definition of a function (allowing one to expand a function call into its body with an appropriate substitution) and `arithmetic` which allows many kinds of arithmetic manipulations. Some hints have aliases (for example, `arith` is an alias for `arithmetic`). Other hints only exist for readability—for example, we use `MP` (`Modus Ponens`) to indicate that a step or derived context item is justified by the conclusion of an implication after satisfying that implication's hypotheses, but it does not affect CPC's checking. Each hint for a proof step gives rise to zero or more of hypotheses (*Hyps*), *Rules* and *Lemma instantiations*, used in proving the proof step. *Rules* here refer to proved theorems in ACL2's database, which ACL2 can

automatically apply. We define functions hyps(h), rules(h) and instances(h) to be the set of hypotheses, rules and lemma instantiations (in a format amenable to ACL2) that a hint h gives rise to, respectively.

- `C`*i*: add the expression corresponding to the i^{th} context item as a hypothesis
- `D`*i*: add the expression corresponding to the i^{th} derived context item as a hypothesis
- `def` *foo*: enable the definition rule(s) for the function *foo*
- `cons axioms`: enable the following rules regarding `cons`: `(:rewrite car-cons)`, `(:rewrite cdr-cons)`, `car-cdr-elim`, `cons-equal`, `default-car`, `default-cdr`, `cons-car-cdr`
- `arithmetic`: enable the set of rules added by including the `arith-5` books
- `evaluation`: enable all rules of type `:executable-counterpart`
- `lemma` *foo*: add a lemma instance `:use` hint for *foo* with the given instantiation (if provided). This effectively instantiates the given lemma and adds the resulting expression as a hypothesis.

Theories

At different times during both Phases 1 and 2, it is useful to be able to ask ACL2s to prove a statement while limiting the types of reasoning that it can use. One of the ways we do this is by controlling the set of rules that ACL2s has access to. We define theories for certain sets of rules that are used inside CPC:

- `arith-5-theory` is the set of rules that are added by including the `"arithmetic-5/top"` book in a vanilla ACL2 instance.
- `min-theory` consists of ACL2's minimal theory (which includes only rules about basic built-in functions like `if` and `cons`) plus `(:executable-counterpart acl2::tau-system)`, `(:compound-recognizer booleanp-compound-recognizer)`, and `(:definition not)`. The former of these three rules enables ACL2 to perform some type-based reasoning, and the latter two are often useful for reasoning about propositional logic.
- `arith-theory` which consists of some basic facts about `+` and $*$.
- `type-prescription-theory` which consists of any rules of type `:type-prescription`
- `executable-theory` which consists of any rules of type `:executable-counterpart`.
- `contract-theory` which is `min-theory` plus `type-prescription-theory` and any rules with names ending in `"CONTRACT"` or `"CONTRACT-TP"`. The latter rules correspond to the function contracts for any functions admitted using `definec`.
- `min-executable-theory` which is the union of the rules in `min` and `executable`.

Type Hypotheses

Almost any proof involving a function defined with `definec` requires that the function's input contract is satisfied. In early versions of CPC, we found that this resulted in users needing to repeatedly include justifications in their steps corresponding to hypotheses that some free variables in the proof statement satisfy some type predicates. Given that users already must perform contract completion on their proof statement, this felt like an unnecessary burden. Therefore for any step or derived context item, CPC will automatically include hints that correspond to calls of type predicates. For example, in `Induction Case 2` in the proof example in Section 2, `C1. (tlp x)`, `C2. (tlp acc)`, `D1. (tlp (cdr x))` and `D2. (tlp (cons (car x) acc))` are all included "for free" as justifications of any proof step.

5 Soundness

Once the user has provided a proof that passes Phases 0 and 1, we would like to translate it into an ACL2s theorem. There are two benefits this brings: (1) we can reduce the soundness of CPC to that of ACL2s and (2) it enables one to perform a proof in CPC that might be challenging to do in ACL2s and then use the resulting theorem in ACL2s. The second benefit is not currently exposed in a convenient way to users of CPC, but we believe it would be easy to implement this feature.

Our soundness theorem is as follows: given a proof P without a non-trivial contract completion and whose proof statement is ϕ, if CPC validates P then ϕ is a valid statement in ACL2s, given the same ACL2s world prior to the validation of P. The witness for our soundness theorem is the ACL2s theorem that proves ϕ.

This theorem is easy to prove, as CPC will validate a proof only if it was able to prove that proof's statement in ACL2s, using the proof-builder instructions that CPC generates as described below. Note that we make no claims about completeness—CPC may reject a proof of a valid statement.

5.1 Proof Builder

Generating an ACL2s statement of a CPC theorem is straightforward, but we do not want to simply hand this statement off to ACL2s for an automatic proof—ACL2s may decide to attempt to take a different proof approach that requires a different set of lemmas, or may just fail to find a proof. Ultimately our goal is to determine whether or not the user's proof is correct, so we should be able to transform it and its justifications into a theorem that ACL2s can prove. For this reason, we use ACL2's proof-builder functionality, which allows us to command the theorem prover's behavior at a much lower level.

The proof-builder operates in a manner similar to an interactive proof assistant like Coq [3] or Isabelle [24]: there is a *proof state* consisting of a stack of goals, each of which contains a set of hypotheses and a statement to be proved, and one provides *instructions* that operate on the goal stack. These instructions range in granularity, with coarse instructions like `prove` (attempt to prove the current goal entirely automatically with ACL2's full power) to fine instructions like `dive` (focus on a particular subexpression in the current statement to be proved). ACL2's documentation provides information about many of the available proof-builder instructions [1, `proof-builder-commands`]. For CPC we developed several new proof-builder instructions, many of which are variants of existing instructions that succeed where the existing instructions would fail. For example, `:retain-or-skip` is exactly like the built-in `:retain` instructions, except that it will succeed even when all of the existing hypotheses are retained (producing no change in the proof-builder state). Many instructions have similar behavior that is desirable when a human is interacting directly with the proof-builder, but that is not when automatically generating instructions. These new proof-builder instructions are available in the ACL2 distribution, inside `books/acl2s/utilities.lisp`. All of the new instructions are listed below:

- `:claim-simple`: exactly like `:claim`, except that it does not automatically perform hypothesis promotion on the newly created goal.
- `:pro-or-skip`: exactly like `:pro`, except that it will succeed even when no promotion is possible.
- `:drop-or-skip`: exactly like `:drop`, except that it will succeed even when there are no top-level hypotheses and no arguments are provided.
- `:retain-or-skip`: exactly like `:retain`, except that it will succeed even when all of the existing hypotheses are retained.

- `:cg-or-skip`: exactly like `:cg`, except that it will succeed even when the specified goal to change to is the same as the current goal.
- `:instantiate`: instantiate a theorem as a hypothesis under the given substitution.
- `:split-in-theory`: exactly like `:split`, except that a theory can be provided to use instead of `minimal-theory`.
- `:by`: prove a goal using exactly an existing lemma under a given substitution.

5.2 Instruction Generation Algorithms

We will now describe how we generate proof-builder instructions for steps and derived context items, equational reasoning proofs, and inductive proofs. In the below algorithm listings, we will use a typewriter font face `like this` to denote S-expressions that we generate. Some additional comments on notation:

- $x +\!\!+ y$ denotes the sequence produced by appending the sequences x and y. If y is a set, then it is first transformed into a sequence by enumerating the elements of y in an arbitrary order.
- An ACL2 statement x is a *type predicate call* if and only if it is a function call with one argument and the function name is known by `defdata` to be a type predicate.
- Let $\text{hid}(x)$ be an identifier used by the proof-builder to refer to the hypothesis corresponding to the context or derived context item x.
- Let $\text{rules}(x)$ be the set of rules that a hint x gives rise to.
- Let $\text{instances}(x)$ be the set of lemma instantiations (in a format amenable to ACL2) that a hint x gives rise to.
- `IndObsAndNames(`*stmt, indterm*`)` calls ACL2's proof-builder to determine what goals are created when one tries to prove *stmt* by performing an induction on *indterm*. The output is a set of tuples $(obs, name)$ where *obs* is an ACL2 statement expressing one of the created goals and *name* is the name that ACL2 gave to that goal.

In the below algorithms, we elide the complexity of matching up the names that the proof-builder gives to the hypotheses with the names of context items that the user gave in the proof.

Hints to Instructions

The processes of generating proof-builder instructions for a step and for a derived context item are similar, so we will describe them together. Figure 3 shows the general form of a step and a derived context item. Using names from Figure 3, we will define the *equivalent expression* of a step to be `(`$R\ \alpha\ \beta$`)` and the equivalent expression of a derived context item to be γ. Algorithm 1 is the corresponding algorithm for this process. We pass the equivalent expression of the step or derived context item into the `stmt` input of the algorithm. Let EE be the equivalent expression of the step or derived context item in question.

The instructions that we generate should do the following: use `:claim-simple` to add a hypothesis that the equivalent expression of the step or derived context item holds, then prove that this hypothesis holds using the justifications that the user provided.

We first generate a `claim-simple` instruction with EE as the statement to cause the proof-builder to add EE as a hypothesis in the current goal. This also results in the creation of a new goal to prove EE

```
;; A step
α
R { H1, ..., Hn }
β
;; A derived context item
Dn. γ { H1, ..., Hn }
```

Figure 3: The general form of a proof step and a derived context item.

given the current set of hypotheses (before EE was added). We pass `:hints :none` to the `claim-simple` instruction so that ACL2 does not try to prove this new goal automatically. Then, we generate a `cg` instruction to change to the newly generated goal. Next, we calculate the guard obligations that this goal would have given the current context and generate a `claim` instruction with those obligations as its statement. This `claim` instruction will result in ACL2 trying to prove that the statement holds automatically. Next, based on $H_1,...,H_n$, we determine which context and derived context items should be available when proving EE. We then generate a `retain-or-skip` instruction with appropriate arguments to only retain the appropriate context and derived context items. We then determine based on $H_1,...,H_n$ what ACL2 rules should be available. We generate a `in-theory` instruction with appropriate arguments to enable and disable rules appropriately. Finally, we generate the instruction `(:finish :bash)`, which tells ACL2 to attempt to prove the current goal while limiting its abilities. If ACL2 is unable to prove the goal, it will raise an error and the proof attempt will result in a failure.

Equational reasoning proofs

Generating instructions for an equational reasoning proof is fairly straightforward; the algorithm is shown in Algorithm 2.

We start with `:pro-or-skip` to expand the proof statement's implication into antecedents and a consequent (if it is an implication). Then, we generate instructions to add a hypothesis for each derived context item and prove that it holds given the provided justifications. We do something very similar for each proof step. Then, we `:demote` to turn the goal and hypotheses into an ACL2 implication statement before repeatedly calling `(:split-in-theory min-executable-theory)` until the goal has been discharged. We use `:split-in-theory` (and therefore `:split`) here as it is a convenient way to invoke ACL2 with very limited reasoning ability (just simplification, preprocessing, and whatever rules are in the given theory).

Inductive proofs

The algorithm for generating proof-builder instructions for inductive proofs is provided in Algorithm 3.

Assume we have a proof by induction without a non-trivial contract completion. This can be thought of as several separate proofs, one for each induction proof obligation. For each of these subproofs, we will generate a separate ACL2 proof, complete with proof-builder instructions. Then, we generate an ACL2 proof for the top-level induction proof with instructions to perform a proof by induction using the induction scheme that the user specified, and generate instructions that discharge each proof obligation using the corresponding generated ACL2 proof. This approach requires that CPC determine the order of the subgoals that ACL2 will generate when asked to perform an induction proof with the given induction scheme so that we can map up the subproofs that the user performed with these subgoals, and thus in the instructions for each subgoal we can refer to the appropriate generated ACL2 proof. We generate

Algorithm 1: proof-builder instruction generation for a step or derived context item

Function `ProveUsingHints`(*stmt, hints, ctx*)

Input: *stmt* is the statement to prove, *hints* is the set of hints the user provided, and *ctx* is the set of context and derived context items it should be proved under.

$I \leftarrow []$;

`/* Add` *stmt* `as a hypothesis and as a new goal, do not attempt to prove it automatically, and switch to the new goal */`

$I \leftarrow I +\!\!+ [$`(:claim-simple` *stmt*`... :hints :none),:cg`$]$;

$hyps \leftarrow \{x \mid x \in hints \wedge x$ is a context or derived context hint $\}$;

$contracts \leftarrow \mathrm{GO}((\bigwedge_{h \in hyps} h) \rightarrow stmt)$;

if $contracts \neq$ `true` **then**

`/* Add` *contracts* `as a hypothesis and as a new goal & prove it automatically. */`

$I \leftarrow I +\!\!+ [$`(:claim` *contracts*`...)`$]$;

$typectx \leftarrow \{x \mid x \in ctx \wedge x$ is a type-predicate call $\}$;

$contractsidx \leftarrow$ a set containing the identifier of the *contracts* hypothesis if $contracts \neq$ `true` or $\emptyset$ otherwise;

`/* Only keep the hypotheses we should have given the hints the user provided */`

$I \leftarrow I +\!\!+ [$`(:retain-or-skip` $\{\mathrm{hid}(x) \mid x \in hyps \cup typectx \cup contractsidx\}$`)`$]$;

$hintrules \leftarrow \bigcup \{\mathrm{rules}(x) \mid x \in hints\}$;

`/* Ensure that only the rules that we should have access to given the user's hints are available */`

$I \leftarrow I +\!\!+ [$`(:in-theory (union-theories (theory 'contract-theory)` *hintrules*`))`$]$;

$lemmainstances \leftarrow \bigcup \{\mathrm{instances}(x) \mid x$ is a hint for $Dx_i\}$;

`/* Add any lemma instances that the user described in the hints */`

$I \leftarrow I +\!\!+ \{$`(:instantiate` x`)` $\mid x \in lemmainstances\}$;

`/* Ask ACL2 to automatically prove this goal without induction, and then reset to the original theory */`

$I \leftarrow I +\!\!+ [$`(:finish :bash),:in-theory`$]$;

return I

Algorithm 2: proof-builder instruction generation for a non-inductive conjecture

```
1  Function EquationalReasoningTranslate(C, D, R, P, H)
       Input: C and D are the sets of all non-derived context and derived context items for a proof
              respectively. R, P and H are the relations, step statements and hints for the proof's
              proof steps, indexed from the start of the proof. This function only operates on
              equational reasoning proofs.
2      I ← [];
       /* Perform exportation and expand implication into hyps/conclusion                    */
3      I ← I ++ [:pro-or-skip];
       /* Generate instructions for each derived context item                                */
4      foreach i ∈ [1..m] do
5          stmt ← the proof statement associated with Dx_i;
           /* Do not include any later derived context items in the context used to prove
              Dx_i                                                                           */
6          ctx ← C ∪ {Dx_j | j ∈ [1..i−1]};
7          I ← I ++ ProveUsingHints(stmt, hints, ctx);
       /* Generate instructions for each step                                                */
8      foreach i ∈ [1..n] do
9          stmt ← ( R_i P_i P_{i+1} );
10         ctx ← C ∪ D;
11         I ← I ++ ProveUsingHints(stmt, H_i, ctx);
       /* Turn the hypotheses and goal into an implication                                   */
12     I ← I ++ [:demote];
       /* Repeatedly call :split-in-theory until the proof is successful or we reach a
          fixpoint                                                                           */
13     I ← I ++
        [(:finish (:repeat-until-done (:split-in-theory min-executable-theory)))];
14     return I
```

these subgoals by using the ACL2 function `state-stack-from-instructions`, which allows one to get the state of the proof builder after running a sequence of instructions, and then reuse some existing CPC code to find a bijection between these subgoals and the induction proof cases that the user provided.

Once we have generated proof-builder instructions for an inductive proof, we generate `defthms` with proof-builder instructions for all of its subproofs. We then generate an `encapsulate` statement and insert all of the subproof `defthms` in the encapsulate as `local`. The inductive proof itself is not inserted into a `local` and is thus exported from the `encapsulate`. We do not want to export the subproof `defthms`, as they are only needed to show that the top-level inductive proof theorem holds.

Algorithm 3: proof-builder instruction generation for an inductive conjecture

Function `InductiveTranslate`(*M, stmt, indterm, PC*)

Input: This function only operates on inductive proofs. M is a function mapping the names of the proof cases of this inductive proof to corresponding ACL2 theorems. *stmt* is the proof statement for this inductive proof, and *indterm* is the induction term. PC is the set of proof cases given for this inductive proof, where name(PC_i) is the name of PC_i and stmt(PC_i) is the proof statement for PC_i.

`/* Generate the proof obligations an induction on indterm will give rise to */`

$obsnames \leftarrow$ `IndObsAndNames`(*stmt, indterm*);

`/* Perform exportation, expand implication into hyps/conclusion, perform induction */`

$I \leftarrow$ [`:pro-or-skip`, (`:induct` *indterm*)];

Attempt to find an injective mapping from *obsnames* to PC, where an element $(obs, name) \in obsnames$ is mapped to an element $PC_i \in PC$ iff the conjunction of the hypotheses of *obs* after exportation are propositionally equivalent to the conjunction of the hypotheses of stmt(PC_i) after exportation.;

if *no such mapping exists* **then**

    raise an error;

$inj \leftarrow$ the injective mapping;

foreach $(obs, name) \in obsnames$ **do**

    $injPC \leftarrow inj((obs, name))$;

    `/* Change to the induction obligation, use the existing proof to discharge it */`

    $I \leftarrow$ [(`:cg-or-skip` *name*), (`:finish :demote (:by` M(name$(injPC)$))))];

return I

6 Related Work

Our previous work [28] contains a longer discussion of works surrounding the use and mechanical verification of calculational proofs. Below we provide a summary of that discussion, as well as some related work in ACL2 in particular.

Calculational proofs were popularized by in the early 1990s by Dijkstra and Scholten [11], Gasteren [13] and Gries [14]. A series of works [2, 15, 16, 25] by Robinson, Stables, Back, Grundy and Wright resulted in the development of structured calculational proofs, an extension of the calculational proof style that

allows for the hierarchical decomposition of proofs. This format reduces to natural deduction, but maintains the benefits of calculational proofs while also allowing for improved readability and browsability of proofs.

Manolios argued for the formalization of calculational proofs and their mechanized checking in 2000 [19]. Mizar [26] is a system for checking calculational proofs first developed in the 1970s. Several systems inspired by Mizar have been developed since, including Isabelle/Isar [30] and Leino et. al's poC extension to Dafny [17]. These systems typically follow Mizar's format in not requiring the user to explicitly state the proof context. Mizar has only lightweight support for automated reasoning in proving that proof steps hold and only allows equality relations inside of proofs. Isar allows for arbitrary relations and provides access to Isabelle's powerful reasoning capabilities, like *simp* for Isabelle's simplifier, and *auto* for a combination of several tools [17]. poC only allows a predefined set of relations but is as declarative as Mizar is, while providing more powerful automated reasoning with its SMT solver backend.

7 Conclusion and Future work

We have presented an argument for the soundness of CPC, based on its translation of calculational proofs into ACL2s theorems with proof-builder instructions. We are interested in seeing how CPC can be used by "professional" users to design their proofs, and have some ideas about functionality that would be appropriate for these users (for example, automatic generation of context and induction proof obligations and a "bash" mode for eliding simple subproofs like contract cases in inductive proofs). We hope to continue to extend and improve CPC based on requests from students and the community, and plan on continuing to use it to help teach undergraduates how to write proofs.

Acknowledgments

We sincerely thank Ken Baclawski, Raisa Bhuiyan, Harsh Chamarthi, Peter Dillinger, Robert Gold, Jason Hemann, Andrew Johnson, Alex Knauth, Michael Lin, Riccardo Pucella, Sanat Shajan, Olin Shivers, David Sprague, Atharva Shukla, Ravi Sundaram, Stavros Tripakis, Thomas Wahl, Josh Wallin, Kanming Xu and Michael Zappa for their help with developing and teaching with CPC. We also thank all of the Teaching Assistants and students in CS2800, who are too numerous to list individually, who used and provided feedback on CPC.

References

[1] ACL2 Contributors: *ACL2 XDoc Documentation*. `https://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index.html`.

[2] Ralph Back, Jim Grundy & Joakim von Wright (1997): *Structured Calculational Proof*. Formal Aspects of Computing 9(5-6), pp. 469–483, doi:10.1007/BF01211456.

[3] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series, Springer, doi:10.1007/978-3-662-07964-5.

[4] Harsh Chamarthi, Peter C. Dillinger, Panagiotis Manolios & Daron Vroon (2011): *The "ACL2" Sedan Theorem Proving System*. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), doi:10.1007/978-3-642-19835-9_27.

[5] Harsh Raju Chamarthi (2016): *Interactive Non-theorem Disproving*. Ph.D. thesis, Northeastern University. Available at https://www.proquest.com/openview/e8a87be5efd3591f96cfaf16b4c79b73.

[6] Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann & Panagiotis Manolios (2011): *Integrating Testing and Interactive Theorem Proving*. In David S. Hardin & Julien Schmaltz, editors: *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications*, *EPTCS* 70, pp. 4–19, doi:10.4204/EPTCS.70.1.

[7] Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann & Panagiotis Manolios (2011): *Integrating Testing and Interactive Theorem Proving*. In: *International Workshop on the ACL2 Theorem Prover and its Applications*, EPTCS, doi:10.4204/EPTCS.70.1.

[8] Harsh Raju Chamarthi, Peter C. Dillinger & Panagiotis Manolios (2014): *Data Definitions in the ACL2 Sedan*. In: *Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and its Applications*, EPTCS, doi:10.4204/EPTCS.152.3.

[9] Harsh Raju Chamarthi & Panagiotis Manolios (2011): *Automated specification analysis using an interactive theorem prover*. In Per Bjesse & Anna Slobodová, editors: *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11*, FMCAD Inc., pp. 46–53. Available at http://dl.acm.org/citation.cfm?id=2157665.

[10] Edgar W. Dijkstra: *EWDs*. https://www.cs.utexas.edu/users/EWD/.

[11] Edsger W. Dijkstra & Carel S. Scholten (1990): *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science, Springer, doi:10.1007/978-1-4612-3228-5.

[12] Peter C. Dillinger, Panagiotis Manolios, Daron Vroon & J. Strother Moore (2007): *ACL2s: "The ACL2 Sedan"*. In: *Proceedings of the 7th Workshop on User Interfaces for Theorem Provers (UITP 2006)*, Electronic Notes in Theoretical Computer Science, doi:10.1016/j.entcs.2006.09.018.

[13] Antonetta J. M. van Gasteren (1990): *On the Shape of Mathematical Arguments*. *Lecture Notes in Computer Science* 445, Springer, doi:10.1007/BFb0020908.

[14] David Gries (1991): *Teaching Calculation and Discrimination: A More Effective Curriculum*. *Communications of the ACM* 34(3), pp. 44–55, doi:10.1145/102868.102870.

[15] Jim Grundy (1996): *A browsable format for proof presentation*. *Logic, Mathematics, and the Computer–Foundations: History, Philosophy and Applications* 14, pp. 171–178. Available at https://www.researchgate.net/publication/2359706_A_Browsable_Format_for_Proof_Presentation.

[16] Jim Grundy (1996): *Transformational Hierarchical Reasoning*. *The Computer Journal* 39(4), pp. 291–302, doi:10.1093/comjnl/39.4.291.

[17] K. Rustan M. Leino & Nadia Polikarpova (2013): *Verified Calculations*. In Ernie Cohen & Andrey Rybalchenko, editors: *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*, *Lecture Notes in Computer Science* 8164, Springer, pp. 170–190, doi:10.1007/978-3-642-54108-7_9.

[18] Panagiotis Manolios: *Logic and Computation Class*. https://www.ccs.neu.edu/home/pete/courses/Logic-and-Computation/2022-Fall/.

[19] Panagiotis Manolios & J Strother Moore (2001): *On the desirability of mechanizing calculational proofs*. *Information Processing Letters* 77(2-4), pp. 173–179, doi:10.1016/S0020-0190(00)00200-3.

[20] Panagiotis Manolios & Daron Vroon (2003): *Algorithms for Ordinal Arithmetic*. In Franz Baader, editor: *19th International Conference on Automated Deduction (CADE)*, *Lecture Notes in Computer Science* 2741, Springer, pp. 243–257, doi:10.1007/978-3-540-45085-6_19.

[21] Panagiotis Manolios & Daron Vroon (2004): *Integrating Reasoning About Ordinal Arithmetic into ACL2*. In Alan J. Hu & Andrew K. Martin, editors: *5th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, *Lecture Notes in Computer Science* 3312, Springer, pp. 82–97, doi:10.1007/978-3-540-30494-4_7.

[22] Panagiotis Manolios & Daron Vroon (2005): *Ordinal Arithmetic: Algorithms and Mechanization*. *Journal of Automated Reasoning* 34(4), pp. 387–423, doi:10.1007/s10817-005-9023-9.

[23] Panagiotis Manolios & Daron Vroon (2006): *Termination Analysis with Calling Context Graphs*. In Thomas Ball & Robert B. Jones, editors: *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, *Lecture Notes in Computer Science* 4144, Springer, pp. 401–414, doi:10.1007/11817963_36.

[24] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. *Lecture Notes in Computer Science* 2283, Springer, doi:10.1007/3-540-45949-9.

[25] Peter J. Robinson & John Staples (1993): *Formalizing a Hierarchical Structure of Practical Mathematical Reasoning*. *Journal of Logic and Computation* 3(1), pp. 47–61, doi:10.1093/logcom/3.1.47.

[26] Piotr Rudnicki (1992): *An overview of the Mizar project*. In: *Proceedings of the 1992 Workshop on Types for Proofs and Programs*.

[27] Andrew T. Walter, Ankit Kumar & Panagiotis Manolios: *Calculational Proof Checker repository*. `https://gitlab.com/acl2s/proof-checking/calculational-proof-checker`.

[28] Andrew T. Walter, Ankit Kumar & Panagiotis Manolios (2023): *Calculational Proofs in ACL2s*. arXiv:2307.12224.

[29] Andrew T. Walter & Panagiotis Manolios (2022): *ACL2s Systems Programming*. In: *Proceedings of the Seventeenth International Workshop on the ACL2 Theorem Prover and its Applications*, EPTCS, doi:10.4204/EPTCS.359.12.

[30] Makarius Wenzel (2007): *Isabelle/Isar–a generic framework for human-readable proof documents*. *Studies in Logic, Grammar and Rhetoric* 10(23). Available at `http://mizar.org/trybulec65`.

[31] Xtext Contributors: *Xtext*. Available at `https://www.eclipse.org/Xtext/`. Accessed on April 25th, 2022.