Representing Nuprl Proof Objects in ACL2: toward a proof checker for Nuprl

James L. Caldwell[*] and John Cowles[*]

Department of Computer Science, University of Wyoming, Laramie Wyoming
{jlc,cowles}@cs.uwyo.edu

Abstract. Stipulations on the correctness of proofs produced in a formal system include that the axioms and proof rules are the intended ones and that the proof has been properly constructed (*i.e.* it is a correct instantiation of the axioms and proof rules.) In software implementations of formal systems, correctness additionally depends both on the correctness of the program implementing the system and on the hardware it is executed on. Once we implement a system in software and execute it on a computer, we have moved from the abstract world of mathematics into the physical world; absolute correctness can never be achieved here. We can only strive to increase our confidence that the system is producing correct results. In the process of creating proofs, foundational systems like Nuprl construct formal proof objects. These proof objects can be independently checked to verify they are correct instantiations of the axioms and proof rules thereby increasing confidence that the putative proof object faithfully represents a proof in the formal system. Note that this kind of proof checking does not address issues related to the models of the proof system, it simply provides more evidence that a proof has been correctly constructed. The Nuprl implementation consists of more than 100K lines of Lisp and tactic code implemented in ML. Although parts of the system consist of legacy codes going as far back as the late 1970's (Edinburgh LCF), and even though the Nuprl system has been extensively used since 1986 in formalizing a significant body of mathematics, the chances that the implementation is correct are slim. Verifying the system itself is infeasible, instead we propose to increase confidence in Nuprl proofs by independently checking them in ACL2. In this paper we describe: (i.) the ACL2 formalization of Nuprl terms, proof rules, and proofs, (ii.) first steps in the implementation of a proof checker, and (iii.) discuss issues related to the future of the project.

1 Introduction

Theorem proving is the archetypal example of a computationally hard (or even undecidable) problem for which a witness (*i.e.* a proof) is easy to check. Finding a proof requires expert human guidance, while checking a formal proof can often be done in linear time. There is a vast literature on checking formal proofs,

[*] This research is supported by ONR grant N00014-01-1-0765

we mention only a few [3, 13, 8, 15, 12]. We note that in [13, pg.215], Pollack suggests that our goal, checking Nuprl proofs, may not be possible. We hope to refute this claim and, more importantly, to provide independent certification of Nuprl proofs thereby increasing confidence in the proof objects constructed in the system. The work of McCune and Shumsky on Ivy [12] is especially applicable; Ivy integrates ACL2 with Otter provers to both preprocess inputs to Otter and to check outputs. But our goals are different since our intention is to create an independent criteria of correctness for Nuprl proofs.

Nuprl proof checking is certainly not a linear time operation. Of the 167 proof rules included in the standard Nuprl distribution, 152 of them can be simply and uniformly checked by a combination of pattern matching and verifying the well-formedness of the instances of the rule premises. Indeed, the Nuprl implementation includes a refiner that does exactly this to generate instances of the rules in the proof process. The remaining 15 proof rules have more complex side conditions which are established in Nuprl by calls to underlying Lisp code. The most extreme example of a rule having computationally hard side-conditions is `arith`. It invokes a decision procedure for an extended theory of linear arithmetic thus, checking an instance of `arith` is as hard as deciding this theory.

We believe ACL2[11, 10] provides the ideal environment for implementing a proof checker for Nuprl. It contains its own decision procedure for linear arithmetic [6], provides support for pattern matching and will allow us to prove properties of the checker itself.

For the purposes of proof checking, aside from a few side-conditions, a semantic understanding Nuprl's type-theory is not strictly necessary (this is the essence of proof checking) but we hope Nuprl's type theory will be of independent interest to the ACL2 community and we urge the reader to examine Allen's recent documentation of Nuprl's type theory [2]. A comparison between Nuprl and an earlier incarnation of ACL2 (NQTHM[1])was published by Basin and Kaufmann[4]; they compared the two systems by verifying Ramsey's theorem in each.

This paper describes Nuprl's internal representations of sequents, proof rules, and proofs and then describes how they are translated into a form usable in ACL2. To date, our checker only verifies that a Nuprl proof object satisfies the most gross structural constraints. We describe how we plan to extend the existing checker so that users can gain the highest levels of confidence in the proofs certified by the checker.

2 Overall Strategy

This section is largely speculative in that it describes the strategy we plan to implement for checking Nuprl proofs, but much remains to be done. However, we argue below that even the little that has been done increases confidence in Nuprl proofs.

[1] Although NQTHM is not ACL2, the comparison should still be interesting to ACL2 users.

Our strategy for using ACL2 to check Nuprl proofs is a three step process.

i.) From within Nuprl, a representation of a primitive Nuprl proof object suitable for input to ACL2 is generated. This step is implemented by an ML program within Nuprl.
ii.) The representation of the putative Nuprl proof is read into ACL2 and it is used to guide the synthesize a term of the ACL2 logic, we will refer to this term as `is-proof`.
iii.) The `is-proof` term is submitted to the ACL2 prover.

If ACL2 accepts the `is-proof` term as a theorem we claim that we have increased confidence that the Nuprl proof is indeed a proof. We argue that any reasonable person who understands the commonly accepted criteria for what it means to be a formal proof must assent to increased confidence as well. On the other hand, if ACL2 fails in its attempt to prove the `is-proof` term, we understand there *may* be a problem with Nuprl, with the translation process, with our criteria for what counts as a proof, or with ACL2. A failed certification would motivate further investigations.

There are many justifications for our claim of increased confidence. Of course, the fact that Caldwell and Cowles are known to be careful people who are not known to be liars is one form of justification. Similarly, the fact that ACL2 has been implemented by well known, respected and trusted individuals and that it has been used for many similar projects is another form of justification. Beyond these facts; that the ACL2 code implementing the proof checker is available for inspection by interested parties makes it unlikely that the checker relies on some unacceptable trick that would invalidate it. Finally, we intended to prove properties (in ACL2) about the checker system. These will include properties about the program that constructs the `is-proof` term, *e.g.* that the entire Nuprl proof is accounted for in the `is-proof` term; that no information has been added or lost. We also intend to prove properties about the `is-proof` term itself. `is-proof` is intended to assert commonly accepted criteria for formal proofs; facts about the relationship between Nuprl's proof rule schemata and fully instantiated proofs. Any reasonably knowledgeable person who understands the standard criteria for concepts such as *being a rule schema*, *being a substitution instance*, or *being equivalent modulo renaming of bound variables* should accept that certification by the checker does indeed add credence to correctness of the certified proof.

We should emphasize that we are not making claims about the correctness of Nuprl itself. Indeed, our goal is to give an independent criteria for the correctness of the proof objects constructed by Nuprl. Our current implementation of the checker simply verifies that the gross structure of the proof tree is correct; each node has the correct number of children. This is a kind of weak certification, but it *is* sufficient to catch a certain class of errors. We should note that these errors are not entirely unlikely since Nuprl users do not typically examine proofs at the level of primitive rule instances but instead view them at the level of tactic inferences. A missing sub-goal in a primitive rule application might never be observed by the typical user. Of course it is far more likely that users would notice extra subgoals. This structural property of proofs is only one of the criteria reasonable people would expect to hold of a Nuprl proof. There are many others. Indeed, since we are not verifying Nuprl but are establishing an independent criteria for accepting a Nuprl proof, there may be no end to the properties reasonable people might propose we either include in our independent certification or verify about the `is-proof` term.

3 Formalizing Core Nuprl in ACL2

Sequent proof systems for classical and intuitionistic logic were first presented by Gentzen [7]. Somewhat surprisingly, restricting sequents by allowing at most one formula in the succedent is enough to move from classical to intuitionistic logic. As we describe below, Nuprl sequents and proof objects are modified versions of pure Gentzen systems. This account of Nuprl proof structure is of independent interest since it has not been described elsewhere. The Nuprl representation naturally supports views of proofs at the both the tactic level and at the level of primitive rule instances.

The Nuprl system is implemented in two levels: an underlying system core implemented in Lisp and an interface to the core implemented in ML. The underlying Lisp implementation consists of approximately 60K lines of code and includes the implementation of ML, the internal representation of proof objects, as well as code for the refiner, an evaluator, the user interface, and decision procedures for equality and arithmetic. The ML interface to the system includes about 40K lines of ML code which is primarily tactic code but also includes ML primitives to manipulate terms and proofs.

Nuprl proofs are compactly stored on disk in a Lisp-readable form consisting of: the name of the theorem, its status (complete, partial, or in error), the sequent term which is the goal of the proof, the tactic script used to generate the proof, the extract term of the proof, and a list of lemma references made in the proof. Unless we intended to duplicate the entire tactic system (which we do not) the compact stored form is not useful for our purposes. Instead, we intend to check primitive Nuprl proofs. During proof development, or when a previously saved (partial) proof has been loaded from disk and expanded, the internal representation is a fully expanded (primitive) sequent proof. It is this primitive proof form we check. Indeed, should the checker find errors in Nuprl proofs, the most likely source of these errors would be the tactic system.

The fact that we are checking Nuprl's internal representation (and not the one used to compactly store proofs) means there is a translation step between the internal representation and a form suitable for use by the ACL2 checker. We will refer to the translation as the print-form of the proof. The translator is a piece of Nuprl ML code that traverses a proof object and outputs the print-form. The translator could be implemented at either Lisp level or the ML level. For ease of implementation and to provide independence from underlying Lisp implementations of the system [2] the translator is implemented at the ML level.

[2] Nuprl is currently ported to Allegro-CL, AKCL, and CMU-CL

The translator recursively walks the proof tree emitting addressed nodes of the proof as it goes. It may be of interest that the initial implementation of translator, which attempted to generated the print-form in memory before outputting to disk, frequently exhausted memory on even medium sized proof objects. These problems lead to the print-form representation using hierarchically addressed proof nodes.

In the following sections we describe the Nuprl representation of sequents, proof rules and proofs and how they are translated into the print-form for the checker.

3.1 Nuprl Sequents

Consider the section of the print-form grammar describing a Nuprl sequent.

```
<sequent>           ::= ( 'sequent <hypothesis list> <conclusion> )
<hypothesis list>   ::= true list of hypotheses
<hypothesis>        ::= ( <declared variable> <term> <Boolean> )
<conclusion>        ::= <term>
<declared variable> ::= <quoted string>
```

Thus, a Nuprl sequent consists of a hypothesis list and a conclusion. The conclusion is simply a Nuprl term. The hypothesis list is a list of triples each of which consists of: a variable name (used to refer to the hypothesis), a Nuprl term (which is the content of the hypothesis), and a Boolean value. The last element of a triple indicates whether a hypothesis is hidden or not.

Nuprl sequents also carry information about their inhabitants. We can extend the grammar for sequents to include this information (although our current implementation ignores extracts.)

```
<sequent> ::= ( 'sequent <hypothesis list> <conclusion> )
            | ( 'sequent <hypothesis list> <conclusion> <extract> )
<extract> ::= <term>
```

The extract is a term of the computation system, *i.e.* it is a program. Consider the Nuprl sequent $\Gamma \vdash C$ `ext` t, here t is the extract term. If we have a proof of $\Gamma \vdash C$ `ext` t, then we know $\Gamma \vdash t \in C$, *i.e.* we know that the term t has type C in the context specified by the hypothesis list Γ. Nuprl proofs are developed by top down refinement. Only after a proof is completed is it possible to instantiate the extract term, it is synthesized from the extract terms below it in the proof tree. As you will see below, Nuprl proof rules specify how extract terms are constructed from the extracts of the subgoals. The moral of this story is that Nuprl proofs implicitly contain programs which can be made explicit by extracting them from completed proofs. Sequents carry a placeholder for the extract term which is instantiated after a proof has been completed.

This brings us back to the issue of hidden hypotheses. Hidden hypotheses are a technical aspect of Nuprl's type theory, they provide the mechanism for implementing the so-called *non-propositional types* [1], which include the quotient type, comprehension subtypes, and the intersection type. Essentially, a non-propositional type is one whose inhabitants do not carry enough information to prove their own well-formedness. It is fair to think of hidden hypotheses as true, but lacking computational evidence for their truth. In practice, the effect on the Nuprl implementation is that, decomposing a non-propositional type may result in hypotheses for which the computational content is unknown, *i.e.* they can not be allowed to find their way into the extract of a proof. Thus, if a hypothesis is not hidden, the variable referring to it may appear in the extract of the proof. If a hypothesis is hidden, the variable referring to it may not appear in the extract of the proof. The Nuprl proof rules indicate when a hypothesis is to be hidden (or unhidden), and the Nuprl refiner raises an exception if the instantiation of a proof rule would incorrectly include a reference to a hidden hypothesis in the extract.

This property, whether a hidden hypothesis ever finds its way into an extract, is one we will independently check. Of course, by doing so we are not checking the correctness of the proof rules themselves nor are we checking whether they correctly specify when a variable is to be hidden or unhidden, but simply that a hidden variable never finds its way into an extract.

3.2 Nuprl Proof Rules

Abstractly, a proof rule is a quadruple consisting of: a name, a goal pattern[3], a list of rule parameters, and a list of subgoal patterns. The goal pattern specifies a family of sequents by using meta-variables for its various components. The subgoals of the rule are specified using meta-variables occurring in the goal pattern, rule parameters, and a substitution operator. Multiple occurrences of a term meta-variable may denote distinct, but alpha-equivalent, terms (*i.e.* terms are equivalent modulo substitution of bound variables.)

Consider the following rule having no subgoals.

```
* RULE hypothesis
 H, x:A, J ⊢ A ext x
    BY hypothesis #$i
    No Subgoals
```

The name of the rule is `hypothesis`. The goal pattern is `H, x:A, J ⊢ A`. Here, `H` and `J` are meta-variables denoting hypothesis lists, `x` is a variable meta-variable and `A` is a term meta-variable. The only parameter is `#$i`, a non-negative integer parameter, which indicates the position of the hypothesis `x:A` in the hypothesis list `H, x:A, J`. There are no subgoals so this rule serves as an axiom.

The hypothesis rule proves any sequent which is a substitution instance of the goal pattern. Note that the alpha-equality condition on term meta-variables

[3] The goal pattern for a Nuprl rule includes a specification of how the extract term is constructed from the extracts of it's subgoals. Currently, we do not consider these terms but will in the future.

means a sequent of the form `H, x:A, J ⊢ A'` matches this pattern if `A` and `A'` are alpha-equivalent.

As another example, consider the following rule for equality of dependent-functions.

```
* RULE functionExtensionality
 H ⊢ f = g ∈ x:A ⟶ B ext t
    BY functionExtensionality level{i} (y:C ⟶ D) (z:E ⟶ F) u
    H, u:A ⊢ f(u) = g(u) ∈ B[u/x] ext t
    H ⊢ A ∈ 𝕌{i}
    H ⊢ f ∈ y:C ⟶ D
    H ⊢ g ∈ z:E ⟶ F
```

The goal pattern is specified by the sequent `H ⊢ f = g ∈ x:A ⟶ B`. Here `H` is a hypothesis list meta-variable, `f, g, A`, and `B` are term meta-variables and `x` is variable meta-variable. The parameters to the rule include a universe level parameter `level{i}`, two term patterns `(y:C ⟶ D)` and `(z:E ⟶ F)` and a variable meta-variable `u`. There are four subgoals specified in terms of the meta-variables which appear in the goal pattern and the rule parameters. Any sequent which is a substitution instance of the goal pattern may be proved by discharging the four subgoals generated by building the specified sequents from the terms matching the corresponding meta-variables. Note that the first subgoal is specified using the substitution operator `B[u/x]` which denotes the substitution of the variable `u` for the variable `x` in the term matching the meta-variable `B`. The curious reader may wonder why the well-formedness subgoals do not have extract terms associated with them; they do, they are simply not displayed since they are always the constant term `Ax`.

Not all Nuprl proof rules are substitution instances of a rule pattern and rule parameters. We call these proof rules *non-uniform*. Non-Uniform rules are implemented in Nuprl by calling routines at the Lisp level. These are accessed via rules containing a term of the form `CallLisp(<Lisp function>)`. In the standard library there are 13 non-uniform rules. Each requires special attention in the proof checker. Many are trivial, but included among the non-uniform rules are decision procedures for arithmetic and for type equality terms.

When a `CallLisp (<F>)` term is invoked, the Lisp function `F` is called with the current proof node as an argument (which also gives `F` access to the arguments passed through the rule parameters). If `F` succeeds, the application of the rule succeeds. The function `F` may return information, passing it back through meta-variables in the rule specification.

Here is the `arith` rule found in the standard library which decides linear arithmetic.

```
* RULE arith
 H ⊢ C ext t
    BY arith 𝕌{i}
       Let (SubGoals, t) = CallLisp(ARITH)
    SubGoals
```

The goal pattern for this rule matches any sequent. The name of the rule is `arith` and it takes a universe term as a parameter. The subgoals generated by the rule are not specified here, they are generated by an external call to the Lisp function `ARITH` which, if it succeeds, returns a pair consisting of a list of subgoals and an extract term `t`. The `ARITH` decision procedure detects arithmetically true conclusions and arithmetic contradictions in the hypothesis list.

To check this rule we will use theorem proving capability in ACL2. The checker will only ever be asked to verify successful instances of the rule.

We have included the specifications of the other 12 non-uniform rules in Appendix B.

3.3 Nuprl Proofs

Nuprl proofs are interactively created by top-down refinement. Users interact with the system using a proof editor. The editor allows users to navigate through a proof structure, and to refine leaf nodes by applying tactics[4]. A Nuprl tactic [9, pp.35] is a function of the following type.

```
tactic = (proof -> ((proof list) × ((proof list) -> proof)))
```

A tactic is applied to a leaf node of a proof[5] and returns a pair consisting of list of remaining subgoals and a validation. The validation is a function that maps proofs of the subgoals to a proof of the refined node.

The internal representation of Nuprl's proof objects are designed to support multiple views of the proof. Ordinarily, proof objects are viewed at the level of tactic inferences. This is how the Nuprl editor displays proofs. Since proofs are constructed by making inference at the tactic level, this is the natural view. A goal of formalizing mathematics is not simply to check proofs, but to provide formally justified explanations. Typically, tactics encapsulate some cognitively meaningful chunk of reasoning and thus, as far as the explanatory function of proofs goes, this is the most useful view of a proof.

The formal justification for a proof lies in the structure of the primitive proof tree constructed by the tactics. Nuprl proof objects include both views.

We present an account of the Nuprl proof structure in phases. As a first approximation, we model proofs as inductively defined structure having refined and unrefined nodes. We call this structure a *pure-quasi-proof*: it is "pure" because it does not include tactic refinements, it is a pure Gentzen style sequent proof; it is a "quasi-proof" because it only defines the gross structure of a proof, it does not include the constraint that it is actually a tree of correct substitution instances of rules.

```
<pure-quasi-proof> ::=
    ( unrefined <sequent> )
  | ( rule-refined <sequent> <rule instance> [ <pure-quasi-proof>* ])
```

[4] In rare cases users will directly apply proof rules, but this is uncommon.

[5] The type referred to as `proof` is an abuse in that it includes both complete and partial trees *e.g.* a single unrefined node also has the Nuprl type proof.

```
<rule instance>     ::= ( 'rule-instance <rule name> <argument list> )
<rule name>         ::= <quoted string>
<argument list>     ::= list of terms and parameters
```

Note that an unrefined node is simply an unproved leaf. A refined node consists of a sequent, the rule being applied including any relevant parameters, and a (possibly empty) list of quasi-proofs, these are the proof obligations generated by the application of the rule.

These quasi-proofs are primitive proof trees. We add another constructor to better approximate the true Nuprl proof representation.

```
<quasi-proof>   ::=
    ( unrefined <sequent> )
  | ( rule-refined <sequent> <rule instance> [ <quasi-proof>* ])
  | ( tactic-refined <sequent> <tactic> <tactic-proof> [ <quasi-proof>* ])
<tactic-proof>  ::= <quasi-proof>
```

In this model, we distinguish between nodes refined by a tactic and nodes refined by a primitive rule. A `tactic-refined` node includes the sequent the tactic has been applied to, the tactic that was applied, a tactic proof and a list of quasi-proofs. The tactic-proof is the quasi-proof generated by applying the tactic. To understand how the tactic-proof relates to the entire proof we consider the cases when the tactic completes a proof and the case when it does not. If the tactic completes the proof, there will be no unrefined leafs of the associated tactic-proof and the accompanying quasi-proof list will be empty. If the tactic does not complete the proof, there will be unrefined leafs in the tactic-proof. The accompanying quasi-proof list will have as many quasi-proofs as there are unrefined leafs in the tactic-proof. The sequents associated with the unrefined leafs will match the sequents associated with the list of quasi-proofs.

Here is the quasi-proof representation of a proof.

```
(tactic-refined
  ⌜⊢ ∀P,Q:𝕌{i}.  P ∧ Q ⇒ Q ∧ P⌝
  ⌜D 0 THENA MemCD⌝
  tactic-proof-1
  [(tactic-refined
    ⌜P:𝕌{i} ⊢ ∀Q:𝕌{i}. P ∧ Q ⇒ Q ∧ P⌝
    ⌜Auto⌝
    tactic-proof-2
    [])
  ])
```

The application of the tactic ⌜`D 0 THENA MemCD`⌝ generates one subgoal. This subgoal is discharged by the application of the ⌜`Auto`⌝ tactic [6]

Because the list of quasi-proofs associated with the application of ⌜`Auto`⌝ is empty, we know *tactic-proof-2* is a complete quasi-proof, it has no unrefined

[6] The auto tactic could have completed the proof, we have added a step here to illustrate how the tactic proof is separated out from the primitive proof.

nodes. The *tactic-proof-1* is a quasi-proof having one unrefined node, and the sequent associated with that node is ⌜P:𝕌{i} ⊢ ∀Q:𝕌{i}. P ∧ Q ⇒ Q ∧ P⌝. We display the quasi-proof *tactic-proof-1* in Figure 1.

```
(rule-refined
  ⌜⊢ ∀P,Q:𝕌{i}.  P ∧ Q ⇒ Q ∧ P⌝
  ⌜direct_computation [1:∀P,Q:𝕌{i}.  P ∧ Q ⇒ Q ∧ P]⌝
  [(rule-refined
    ⌜⊢ P:𝕌{i} → (∀Q:𝕌{i}. P ∧ Q ⇒ Q ∧ P)⌝
    lambdaFormation level{i'} P
    [(unrefined
      ⌜P: 𝕌{i} ⊢ ∀Q:𝕌{i}. P ∧ Q ⇒ Q ∧ P⌝)
      (rule-refined
       ⌜⊢ 𝕌{i} = 𝕌{i}∈ 𝕌{i'}⌝
       ⌜reverse_direct_computation [1:𝕌{i} ∈ 𝕌{i'}]⌝
       [(rule-refined
         ⌜⊢ 𝕌{i} ∈ 𝕌{i'}⌝
         ⌜direct_computation [1:𝕌{i} ∈ 𝕌{i'}]⌝
          [(rule-refined
            ⌜⊢ 𝕌{i} = 𝕌{i}⌝
            universeEquality ()
            []
           )])])])])
```

Fig. 1. *tactic-proof-1*

3.4 The proof node addressing scheme

In the quasi-proof data-type, tactic-proofs and subgoals are also quasi-proofs. In the print form, we eliminate these inductive components and rely on a hierarchical addressing scheme to be able to access them. In this scheme, a proof node address is a list of natural numbers. The addressing scheme is defined as follows.

i.) The root node of a quasi-proof has the empty list `'()` as its address.

ii.) If the quasi-proof at address *addr* is of the form
`(rule-refined` s r `[`$q_1 \, q_2 \cdots q_n$`])`
then the address of the proof node for q_i is `(rcons` ***addr i***`)`, *i.e.* is the list obtained by concatenating i to the right of the list *addr*.

iii.) If the quasi-proof at address *addr* is of the form
`(tactic-refined` s r q `[`$q_1 \, q_2 \cdots q_n$ `])`
then the address of the proof node for q is `(rcons` ***addr 0***`)` and the address of the proof node for q_i is `(rcons` ***addr i***`)`.

For the purposes of checking the formal proofs, the checker reconstructs the equivalent of a pure-quasi-proof from a quasi-proof, *i.e.* it ignores all the `tactic-refined` nodes in the proof and must be able to match unrefined leafs in the tactic-proof with the roots of the appropriate subgoal in the list of quasi-proofs. Thus any `tactic-refined` nodes are *not* included in the final list of proof nodes output to a file.

Figure 2 shows a partial quasi-proof annotated with addresses which appear after the node constructor name.

```
(tactic-refined  ()
  ⌜⊢ ∀P,Q:𝕌{i}.  P ∧ Q ⇒ Q ∧ P⌝
  ⌜D 0 THENA MemCD⌝
  (rule-refined (0)
    ⌜⊢ ∀P,Q:𝕌{i}.  P ∧ Q ⇒ Q ∧ P⌝
    ⌜direct_computation [1:∀P,Q:𝕌{i}.  P ∧ Q ⇒ Q ∧ P]⌝
    [(rule-refined (0 1)
      ⌜⊢ P:𝕌{i} → (∀Q:𝕌{i}. P ∧ Q ⇒ Q ∧ P)⌝
      lambdaFormation level{i} P
      [(unrefined (0 1 1)
        ⌜P: 𝕌{i} ⊢ ∀Q:𝕌{i}. P ∧ Q ⇒ Q ∧ P⌝)
        (rule-refined (0 1 2)
         ⌜⊢ 𝕌{i} = 𝕌{i}∈ 𝕌{i'}⌝
         ⌜reverse_direct_computation [1:𝕌{i} ∈ 𝕌{i'}]⌝
         [(rule-refined (0 1 2 1)
           ⌜⊢ 𝕌{i} ∈ 𝕌{i'}⌝
           ⌜direct_computation [1:𝕌{i} ∈ 𝕌{i'}]⌝
            [(rule-refined (0 1 2 1 1)
              ⌜⊢ 𝕌{i} = 𝕌{i}⌝
              universeEquality ()
              []
            )])])])])
  [(tactic-refined (1)
    ⌜P:𝕌{i} ⊢ ∀Q:𝕌{i}. P ∧ Q ⇒ Q ∧ P⌝
    ⌜Auto⌝
    tactic-proof-2
    [])
  ])
```

Fig. 2. *quasi-proof annotated with addresses*

3.5 Representation of Nuprl Proofs in ACL2

The following is the first part of a grammar describing how Nuprl proofs are represented in ACL2. A more complete version of the grammar can be found in appendix A.

```
<proof list>      ::= ( 'proof . <proof node list> )
```

```
<proof node list> ::= true list of proof nodes
<proof node>      ::= ( <address> <sequent> )
                    | ( <address> <sequent> <rule instance> )
<address>         ::= ( 'address . <number list> )
<sequent>         ::= ( 'sequent <hypothesis list> <conclusion> )
<rule instance>   ::= ...
<hypothesis list> ::= ...
```

The grammar is implemented in ACL2 using records similar to the ones described in the section The ACL2 Record Facilities, located in the ACL2 implementation file `basis.Lisp`.

For each entity (record) used in this representation of the grammar, a recognizer for the entity, accessors for decomposing the entity, and a keyword command for displaying the entity are provided.

As a simple example, suppose the nonterminal `<node>` is described by the productions

```
<node> ::= ( a <symbol> <natural number> )
         | ( a <symbol> )
```

where a `<symbol>` is any Lisp symbol and a `<natural number>` is any nonnegative integer. The recognizer is called `|node|`, the accessors are invoked using `access` with `|node|` plus a key word, and the display command is invoked with `:d-node` (`d` for "display"). Evaluation of each expression on the left of the `->` returns the value on the right:

```
(|node| '(a b 2)) -> t
(|node| '(a 1 2)) -> nil

(|node| '(a b)) -> t
(|node| '(b b)) -> nil

(access |node| '(a b 2) :|symbol|) -> b
(access |node| '(a b 2) :|natural number|) -> 2

(access |node| '(a b) :|symbol|) -> b
(access |node| '(a b) :|natural number|) -> nil
```

Each command line below is followed by the output produced by the command:

```
NUPRL !>:d-|node| (a b 2)
node:
 symbol: B
 natural number: 2
T
-----
NUPRL !>:d-|node| (a b)
node:
 symbol: B
 natural number: NONE
```

```
T
-----
NUPRL !>:d-|node| (a 1 2)

ACL2 Error in ACL2::TOP-LEVEL:  The guard for the function symbol
|D-node-FN|, which is (|node| X), is violated by the arguments in
the call (|D-node-FN| '(A 1 2)).
-----
```

To help automate producing the recognizers, accessors, and keyword commands as the grammar went through many modifications, the grammar is translated into Common Lisp readable form in two steps.

1. Minimal hand edit.
   - Enclose entire list of productions with matching `[` & `]`.
   - Enclose each production with matching `[` & `]`.
   - Replace `|`(vertical bar) with `!`(bang).
   - Replace `true list of things` with `list <thing>`.
   - Replace "any Lisp type" with "Lisp typep."

```
[ ;; begin productions
 [<proof list>       ::= ( 'proof . <proof node list> )]
 [<proof node list>  ::= list <proof node>]
 [<proof node>       ::= ( <address> <sequent> )
                       ! ( <address> <sequent> <rule instance> )]
 [<address>          ::= ( 'address . <number list> )]
 [<sequent>          ::= ( 'sequent <hypothesis list> <conclusion> )]
 [<rule instance>    ::= ...]
 [<hypothesis list>  ::= ...]
       ... ]
```

2. Change AKCL Common Lisp `*readtable*`
   - Make `[` & `]` behave as `(` & `)`.
   - Make `!`(bang), `:`, and `=` behave as white space.
   - Make `<` & `>` behave as multiple escapes.

The result of using the changed `*readtable*` in AKCL produces a Common Lisp readable version of the grammar.

```
((|proof list| ('PROOF . |proof node list|))
 (|proof node list| LIST |proof node|)
 (|proof node| (|address| |sequent|)
               (|address| |sequent| |rule instance|))
 (|address| ('ADDRESS . |number list|))
 (|sequent| ('SEQUENT |hypothesis list| |conclusion|))
 (|rule instance| ...)
 (|hypothesis list| ...)
       ... )
```

4 Checking Nuprl Proofs

Our current implementation is able to check the gross structural properties of Nuprl proofs. We have implemented the translator which outputs the print form of a proof suitable for processing by ACL2. Utilities for navigating proof nodes by addresses have been implemented in ACL2 and they have been used to implement a simple checker.

The properties we can check include that the proper number of subgoals have been generated for each proof rule. For uniform proof rules, the number of subgoals is fixed. There are three non-uniform proof rules where evaluating the `CallLisp(`*F*`)` term returns a list of subgoals, the number of subgoals may vary. For these rules the checker simply assumes the number of subgoals generated is correct.

The correctness of the tactic system depends on the fact that the sequent in each unrefined leaf of a tactic-proof must match the sequent in one of the subgoals of the node. Thus, if the tactic refinement is of the following form:

$$\texttt{(tactic-refined}\ s\ \ r\ \ q\ \ \texttt{[}q_1\, q_2\, \cdots\, q_n\texttt{])}$$

there must be exactly n unrefined nodes in q and, if the sequents in those nodes are $s_1, s_2, \cdots s_n$, then there is a permutation $\rho : \{1..n\} \rightarrow \{1..n\}$ such that the sequent proved by q_i is $s_{\rho(i)}$. This matching of number and form of sequents is currently being checked.

5 Conclusions and Future Work

Our goal is to produce a proof checker that, applied to a Nuprl proof, adds significantly to the proof's credibility. Our current proof checker only checks certain structural features of a Nuprl proof. To achieve our goal much remains to be done.

The largest challenge for us as developers of the proof checker will be to certify the rules that invoke decision procedures. Also, to check the direct computation rules, which may fold or unfold definitions, requires access to those definitions within the proof. We are not currently carrying the information in the proof. Similarly, applications of the `lemma` rule which instantiates a theorem as a hypothesis will require we have, at least, the statement of the lemma. The issues of exactly what information and how much and how to package it in the print form is unresolved.

Independent scrutiny of Nuprl proof objects raises a number of challenges for the Nuprl projects. The largest of these is management of the `because` rule. This rule generates no subgoals completing a proof, it asserts the truth of any sequent. This primitive rule is frequently instantiated by tactics which use previously proved theorems. This reuse is based on a proof caching scheme. Statistics of rules uses show that instances of the `because` rule may account for 15% or more of the rules in a proof. When the `Auto` tactic discharges a well-formedness goal by appealing to the proof cache, it simply completes the proof by invoking the because rule. Most of these instances are can be eliminated by turning off the proof-caching mechanism, a user settable option. Certification of proofs run with caching turned off will provide evidence that the mechanism itself does not interfere with validity. However, other uses of the because rule are more fundamental. The SupInf tactic, which implements Bledsoe's Sup Inf decision procedure[5, 14] for linear arithmetic, invokes because if the decision procedure says *yes*. We do not believe the rule itself is objectionable, but if Nuprl proofs are to be believed, the uses of because must be accounted for. This is currently an active project in the Nuprl group at Cornell.

Acknowledgments

The authors would like to thank Stuart Allen at Cornell for his efforts in support of this project. We also thank Ralph Wachter at ONR for his support.

References

1. S. Allen, R. Constable, D. Howe, and W. Aitken. The semantics of reflected proof. *Proc. of Fifth Symp. on Logic in Comp. Sci., IEEE*, June 1990.
2. Stuart Allen. *Nuprl Basics*. Cornell University, 2001. www.cs.cornell.edu/Info/People/sfa/Nuprl/NuprlPrimitives/.
3. Jon Barwise. Mathematical proofs of computer systems correctness. *Notices of the American Mathematical Society*, 36:844–851, 1989.
4. David A. Basin and M. Kaufmann. The Boyer-Moore prover and Nuprl: An experimental comparison. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 89–119. Cambridge University Press, 1991.
5. W. Bledsoe. A new method for proving certain Presburger formulas. In *Proc. of the 4 th Joint Conf. on Artificial Intelligence*, pages 15–21, 1975.
6. R. S. Boyer and J Strother Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. Technical Report ICSCA-CMP-44, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1985. citeseer.nj.nec.com/boyer85integrating.html.
7. Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The collected papers of Gerhard Gentzen*, pages 68–131. North-Holland, 1969.
8. Michael J. C. Gordon. Representation and validation of mechanically generated proofs (final report). citeseer.nj.nec.com/154955.html.
9. Paul B. Jackson. *Enhancing the Nuprl proof development system and applying it to computational abstract algebra*. PhD thesis, Cornell University, 1995.
10. M. Kaufmann, P. Manolios, and J. Moore. *Computer-Aided reasoning: ACL2 Case Studies*, volume 4 of *Advances in Formal Methods*. Kluwer, 2000.
11. M. Kaufmann, P. Manolios, and J. Moore. *Computer-Aided reasoning: An approach*, volume 3 of *Advances in Formal Methods*. Kluwer, 2000.
12. William McCune and Olga Shumsky. Ivy: A preprocessor and proof checker for first-order logic". [10], chapter 16, pages 265–281.
13. Robert Pollack. How to believe a machine-checked proof. In G. Sambin and J. Smith, editors, *Twenty-Five Years of Constructive Type Theory*. Oxford University Press, 1998.
14. Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the ACM*, 24(4):529–543, 1977.

15. W. Wong. Recording and Checking HOL Proofs. In E.T. Schubert, P.J. Windley, and J. Alves-Foss, editors, *8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 971 of *LNCS*, pages 353–368, Aspen Grove, Utah, USA, 1995. Springer-Verlag.

A Nuprl in ACL2

The following grammar describes how Nuprl proofs are represented in ACL2.

```
<proof list>       ::= ( 'proof . <proof node list> )
<proof node list>  ::= true list of proof nodes
<proof node>       ::= ( <address> <sequent> )
                     | ( <address> <sequent> <rule instance> )
<address>          ::= ( 'address . <number list> )
<sequent>          ::= ( 'sequent <hypothesis list> <conclusion> )
<rule instance>    ::= ( 'rule-instance <rule name> <argument list> )
<hypothesis list>  ::= true list of hypotheses
<hypothesis>       ::= ( <declared variable> <term> <Boolean> )
<conclusion>       ::= <term>
<rule name>        ::= <quoted string>
<argument list>    ::= true list of terms
<declared variable>::= <quoted string>
<Boolean>          ::= T
                     | NIL
<term>             ::= ( ( <opid> . <parameter list> ) <bterm list> )
<bterm list>       ::= true list of bterms
<bterm>            ::= ( <bound-var list> <term> )
<bound-var list>   ::= true list of bound variables
<bound-var>        ::= <quoted string>
<opid>             ::= <token>
<parameter list>   ::= true list of parameters
<parameter>        ::= <simple parameter>
                     | <meta parameter>
<simple parameter> ::= ( <natural number> . |natural-parameter|   )
                     | ( <quoted string>  . |string-parameter|    )
                     | ( <quoted string>  . |token-parameter|     )
                     | ( <quoted string>  . |var-parameter|       )
                     | ( <Boolean>        . |bool-parameter|      )
                     | ( <level exp>      . |level-exp-parameter| )
<meta parameter>   ::= ( <meta variable> . |natural-meta-parameter| )
                     | ( <meta variable> . |string-meta-parameter|  )
                     | ( <meta variable> . |token-meta-parameter|   )
                     | ( <meta variable> . |var-meta-parameter|     )
                     | ( <meta variable> . |bool-meta-parameter|    )
<level exp>        ::= true list of token-nbr pairs
<token nbr>        ::= ( <token> . <natural number> )
<meta variable>    ::= any lisp string starting with the character $
<token>            ::= any lisp symbol
<number list>      ::= true list of natural numbers
```

```
<natural number>   ::= any nonnegative integer
<quoted string>    ::= any lisp string
```

Here are a few of the many varieties of Nuprl terms specifically mentioned in the proof rules.

```
<equal term>   ::=
    any term,
     ( ( <opid> . <parameter list> ) <bterm list> )
    of the form
   ( (|equal|) ( ( <bound-var list> <universe term> ) <bterm> <bterm> ) )

<or term>      ::=
    any term,
     ( ( <opid> . <parameter list> ) <bterm list> )
    of the form
     ( (|or|) ( <bterm> <bterm> ) )

<product term> ::=
    any term,
     ( ( <opid> . <parameter list> ) <bterm list> )
    of the form
     ( (|product|) ( <bterm> <bterm> ) )

<union term>   ::=
    any term,
     ( ( <opid> . <parameter list> ) <bterm list> )
    of the form
     ( (|union|) ( <bterm> <bterm> ) )

<universe term>::=
    any term,
     ( ( <opid> . <parameter list> ) <bterm list> )
    of the form
     ( (|universe| ( <level exp> . |level-exp-parameter| )) NIL)
```

B Non-uniform Nuprl proof rules

```
* RULE universeFormation
 H ⊢ 𝕌{i} ext 𝕌{j}
   BY universeFormation level{j}
      Let () = CallLisp(UNIVERSE-FORMATION)
   No Subgoals

* RULE universeEquality
 H ⊢ 𝕌{j} = 𝕌{j} ∈ 𝕌{i}
   BY universeEquality ()
      Let () = CallLisp(LE-UNIVERSE-EQUALITY)
   No Subgoals
```

```
* RULE cumulativity
 H ⊢ t = t ∈ 𝕌{i}
    BY cumulativity level{j}
       Let () = CallLisp(LE_CUMULATIVITY)
    H ⊢ t = t ∈ 𝕌{j}

* RULE direct_computation
 H ⊢ T ext t
    BY direct_computation S
       Let C = CallLisp(DIRECT-COMPUTATION)
    H ⊢ C ext t

* RULE direct_computation_hypothesis
H, x:A, J ⊢ T ext t
    BY direct_computation_hypothesis #$i S
       Let B = CallLisp(DIRECT-COMPUTATION-HYPOTHESIS)
    H, x:B, J ⊢ T ext t

* RULE reverse_direct_computation
 H ⊢ T ext t
    BY reverse_direct_computation S
       Let C = CallLisp(REVERSE-DIRECT-COMPUTATION)
    H ⊢ C ext t

* RULE reverse_direct_computation_hypothesis
H, x:A, J ⊢ T ext t
    BY reverse_direct_computation_hypothesis #$i S
       Let B = CallLisp(REVERSE-DIRECT-COMPUTATION-HYPOTHESIS)
    H, x:B, J ⊢ T ext t

* RULE lemma
 H ⊢ C ext t
    BY lemma "$theorem"
       Let (t, C) = CallLisp(LE-LEMMA)
    No Subgoals

* RULE instantiate
 H ⊢ T ext t[$psl]
    BY instantiate J C parameter-substitution-list{$psl:psl}
       Let () = CallLisp(INSTANTIATE)
    J ⊢ C ext t

* RULE extract
 H ⊢ t = t ∈ C
    BY extract "$theorem"
       Let (t, C) = CallLisp(LE-LEMMA)
    No Subgoals

* RULE equality
 H ⊢ s = t ∈ T
    BY equality ()
       Let () = CallLisp(LE_EQUALITY)
    No Subgoals
```

```
* RULE arith
 H ⊢ C ext t
    BY arith 𝕌{i}
       Let (SubGoals, t) = CallLisp(ARITH)
    SubGoals

* RULE recEquality
 H ⊢ rec(z1.t1) = rec(z2.t2) ∈ 𝕌{i}
    BY recEquality y
       Let () = CallLisp(REC-EQUALITY)
    H, y:𝕌{i}⊢ t1[y/z1] = t2[y/z2] ∈ 𝕌{i}

*RULE sqequalRule
  H ⊢ a ∼ b ext t
   BY sqequal ()
     Let (SubGoals, t) = CallLisp(SQEQ)
   SubGoals

*RULE sqequalIntensionalEquality
  H ⊢ (a ∼ b) = (c ∼ d) ∈ 𝕌{i}
   BY sqequalIntensionalEquality ()
     Let SubGoals = CallLisp(SQEQ-EQUALITY)
   SubGoals
```