Nuprl as a Generic Theorem Prover

Roderick Moten

Abstract

Logical Frameworks are one way to provide generic theorem provers. This paper describes another method using loose semantics. In the paper, we explain loose semantics, describes its use in building a programming calculus in the style of Back's refinement calculus, and relates the idea to Logical Framework or General Logic. Viewing Nuprl as a generic theorem prover using loose semantics can be used to describe the inference engine of Nuprl 4. This is the first attempt to explain the system design of Nuprl and relate it to the code.

1 Introduction

A generic theorem prover may be considered as a work bench for constructing mechanized reasoners for logics. Suppose that a generic theorem prover had logic independent devices that supported proof development such as an editor for manipulating proofs, a repository for storing objects, a facility for pretty printing terms, and a language for writing scripts to automate reasoning. A mechanized reasoner for a logic could be produced by encoding it into a generic theorem prover and inheriting the devices of the prover.

Generally, generic theorem provers are implementations of logical frameworks, such as Isabelle [19] and ELF [5]. In these provers, a type theory is given as the prover's logic. The type theory is hard coded into the inference mechanism of the prover. The object logics are encoded into the the type theory. Proofs in the object logic are constructed within the type theory.

Another approach to implementing a generic theorem prover is with loose semantics. That is, the inference mechanism of the prover is independent of any logic. It is parameterized by the axioms and inference rules of the object logic. The prover has a representation language for encoding object logics into the prover. Proofs are carried out directly in the object logic using the devices of the generic theorem prover to assist proof construction.

Designing routines for performing multiple steps of inference using heuristics, such as tactics, for an object logic are easier to design in a generic theorem prover with loose semantics than with one implementing a logical framework. In a generic theorem prover with loose semantics, these routines make choices based on knowledge of the object logic and the inference strategy. However, in a logical framework theorem prover, these routines also require knowledge of the logical framework.

The Nuprl theorem proving system can be exploited as a logical framework or a generic theorem prover with loose semantics. Constable and Howe depicted Nuprl as a logical framework using constructive type theory [9, 4]. They used Nuprl 3, the premier version of Nuprl [8]. The three primary components of Nuprl 3 were a repository for storing objects, an editor for manipulating proofs, and an inference mechanism. They are referred to as the library, the proof editor, and the refiner, respectively. The typing rules and the evaluation rules of constructive type theory where hard coded into the refiner.

Nuprl 4, the latest version of Nuprl [16], is implemented using loose semantics. The architecture of Nuprl 4 resembles the architecture of Nuprl 3, but inference rules of all logics including constructive type theory must be passed as parameters to the Nuprl 4 refiner. The Standard ML implementation of the Nuprl 4 refiner also allows the user to supply evaluation rules for non-canonical terms of an object logic. This feature is not available in the Nuprl 4 refiner implemented in Lisp.

In this paper, we demonstrate that Nuprl 4 uses loose semantics by showing that the refiner is a generic goal directed reasoner. In Section 2, we give an overview of the refiner, and in Section 3 we give its mathematical description. In Section 4, we use the refiner to encode the refinement calculus to demonstrate its capability as a generic goal directed reasoner.

1.1 Math Preliminaries

We will be using set theory to describe the refiner. The set of sequences of elements of a set A is denoted by A^*. The elements of A^* are represented as $(a_1, \ldots, a_n)$ for $a_i \in A$. The empty sequence is represented as $()$. The set of non-empty sequences of A, A^+, is the set $A^* - \{()\}$. Given sets $A_1, A_2, \ldots A_n$, $A_1 \times A_2 \times \cdots \times A_n$ is a set with elements $\langle a_1, a_2, \ldots, a_n \rangle$ where $a_i \in A_i$. If $A = A_1 = A_2 = \cdots = A_n$ then we write $A_1 \times A_2 \times \cdots \times A_n$ as A^n.

2 Overview of the Refiner

The refiner consists of the term module, the rule interpreter, and the proof manager. The term module defines the term language and computation over terms. The term language is the core of Nuprl's representation language. Most data, such as inference rules, ML code and rewrite rules, are represented as terms. The rule interpreter is responsible for performing primitive inference based on inference rules supplied by the user. Proof construction using tactics is handled by the proof manager. In addition, the proof manager can synthesize from a proof an object representing a solution to a problem based on the semantics of a particular object logic.

The refiner performs inference using goal directed reasoning. Goal directed reasoning is the process of finding a solution to a problem by decomposing the problem into more tractable subproblems. After the subproblems have been solved their solutions are used to compose a solution for the original problem [20]. LCF employed goal directed reasoning to develop the tactic method of problem solving (See [11] page 57). In LCF, a problem is called a goal. A tactic is a function that decomposes a goal into subproblems or subgoals and produces a function called a validation. A validation constructs a solution from the solutions of the subgoals. Finding the proof of a conjecture and finding an inhabitant of a type are the most common problems solved using the LCF approach [10, 14]. For these problems the solutions are proofs and terms of type theory, respectively.

The refiner performs inference similar to LCF's methodology to problem solving. Conjectures of an object logic are represented in the refiner as goals. Goals are decomposed using inference rules of the object logic. An inference rule in the refiner can be perceived as a schematic specification of an LCF tactic. In other words, an inference rule specifies for a class of goals a means of decomposing a goal into subgoals and constructing a solution for the goal from the solutions of the subgoals using pattern matching and substitution. Applying an inference rule also produces a partial solution. The process known as *extraction* composes partial solutions into a single solution for a goal. The refiner and LCF differ in the method in which tactics are used to perform multiple inferences. From a tactic, the refiner generates an object representing a derivation of several inference rule applications. However, as mentioned above, in LCF a tactic produces subgoals and a validation.

3 Mathematical Description of the Refiner

The mathematical description of the refiner stems from the formal account of the Nuprl term and proof structure provided by the reflected Nuprl type theory [2, 3, 7]. The description of the refiner serves as documentation for the Lisp implementation of the refiner and as a design specification of the refiner in Standard ML.

3.1 Nuprl Terms

We represent Nuprl terms as a subset of a set of second order terms $\mathcal{T}$. The terms $\mathcal{T}$ are a modified and simplified version of the second order terms presented using the lambda calculus by Huet and Lang [15]. The abstraction terms of Huet and Lang's language help explain bound terms in Nuprl. Our language, unlike Huet and Lang's, is untyped.

To begin, we assume that we have a countable infinite set of identifiers $\mathcal{I}$

and modifiers[1] $\mathcal{M}$. Elements of $\mathcal{I}$ are written in teletype font, for example `joy`. There are several classes of modifiers, for example variable symbols and natural numbers. Each class of modifiers has associated with it a unique identifier. We write a modifier x in the class with tag $\mathtt{t}$ as $x : \mathtt{t}$. The tags for variable symbols and natural numbers are `v` and `n`, respectively. An *operator* is an identifier paired with a possibly empty sequence of modifiers. Operators are written as $f\{p_1, \ldots, p_n\}$ where $f \in I$ and each $p_i \in \mathcal{M}$, for example $\mathtt{love}\{x : \mathtt{v}, 3 : \mathtt{n}\}$. We let Op be the set of operators.

We define the set of terms $\mathcal{T}$ inductively as follows.

1. If $a \in Op$, then $a \in \mathcal{T}$.
2. If $x \in \mathcal{I}$ and $t \in \mathcal{T}$, then $x.t \in \mathcal{T}$ called a **bound term**.
3. If $t_1, \ldots, t_n \in \mathcal{T}$ and $a \in Op$ then $a(t_1; \ldots; t_n) \in \mathcal{T}$.

We abbreviate nested bound terms of the form $t = x_1. \cdots .x_n.t$ as $x_1, x_2, \ldots, x_n.t$.

The first order variables of $\mathcal{T}$ are the terms of the form $\mathtt{variable}\{x\!:\!\mathtt{v}\}$. We abbreviate variables to their variable symbol, that is $\mathtt{variable}\{x\!:\!\mathtt{v}\}$ is written as x. We fix $\mathcal{V}$ to be the set of first order variables of $\mathcal{T}$. Terms of the form $\mathtt{variable}\{a\!:\!\mathtt{v}\}(t_1;\ \ldots;\ t_n)$ where each t_i is not a bound term are *second order variables instances.*

The free variables of a term t is the set of variables $FV(t)$. More specifically, if $t \in \mathcal{V}$ then $FV(t) = \{t\}$. If $t = a(t_1; \ldots; t_n)$, then $FV(t) = FV(t_1) \cup \cdots \cup FV(t_n)$ If $t = x.t'$, then $FV(t) = FV(t') - \{x\}$. As usual, t and t' are alpha equal, $t =_\alpha t'$, if they are equal up to bound variable renaming.

Given $t, s \in \mathcal{T}$ and $x \in \mathcal{V}$, we denote $t[s/x]$ as capture avoiding first order substitution of free occurrences of x with s in t. A *substitution pair* is a pair $\langle x, t\rangle$, where $x \in \mathcal{V}$ and $t \in \mathcal{T}$. We call x the target and t the replacement. A *substitution* is a finite set σ of substitution pairs where for every $t \in \mathcal{T}$, $\sigma t \in \mathcal{T}$ obtained inductively as follows.

1. If $t \in Op$ and $\exists\langle t, t'\rangle \in \sigma$, then $\sigma t = t'$; otherwise $\sigma t = t$.
2. If $t = a(t_1; \ldots; t_n)$, then
$$\sigma t \quad = \quad t'[\sigma t_1/x_1, \ldots, \sigma t_n/x_n],$$
if t is a second order variable instance and $\langle a, x_1 \ldots x_n.t'\rangle \in \sigma$; otherwise
$$\sigma t \quad = \quad a(\sigma t_1; \ldots; \sigma t_n).$$
3. If $t = x_1, \ldots, x_n.t'$ then $\sigma t = x_1, \ldots, x_n.\sigma' t'$, where σ' is σ with all pairs $\langle x_i, s\rangle$ for some $i \in \{1, \ldots, n\}$ removed if s is not a bound term.

[1] Traditionally, modifiers are called parameters. Referring to parameters as modifiers comes from Jason Hickey.

Note that σt is second order substitution as long as bound variables in t and the free variables of replacements of σ are distinct so that capture is prohibited. Huet and Lang use types in their definition of substitution to ensure substitution is second order and not higher order. We rely directly on the structure of the terms.

The terms of Nuprl, $\mathcal{T}_\nu$, are the terms of $\mathcal{T}$ that only contain bound terms as proper subterms. Given a substitution σ and term $t \in \mathcal{T}_\nu$, we desire σt to be in $\mathcal{T}_\nu$. The following theorem presents the conditions that ensures $\sigma t \in \mathcal{T}_\nu$.

Theorem 3.1 Let σ be a substitution and $t \in \mathcal{T}_\nu$. Then $\sigma t \in \mathcal{T}_\nu$ if the following criteria hold.

1. If $t \in V$ and $\langle t, t' \rangle \in \sigma$ then $t' \in \mathcal{T}_\nu$.
2. For any subterm of t that is a second order variable instance, say $a(t_1; \ldots; t_n)$, if $\langle a, x_1, \ldots, x_n.t' \rangle$ is in σ then $t' \in \mathcal{T}_\nu$.

We generalize term computation defined in Nuprl 3 (See Appendix C in [8]). Term computation in Nuprl 3 was defined with respect to the non–canonical terms of Nuprl type theory. Term computation was performed by using rewrite rules generated from theorems (`term_of` terms) and the reduction rules for the non–canonical terms. In this presentation, term computation is defined with respect to a set of rewrite rules and reduction rules.

Let Δ be a set of rewrite rules and Θ be a set of reduction rules. To distinguish between rewrite rules and reduction rules, we write rewrite rules as $t \Rightarrow t'$ and reduction rules as $t \rightarrow t'$. A *computation rule* is a triple

$$\langle s, (a_1, \ldots, a_n), (r_1 \rightarrow c_1, \ldots, r_k \rightarrow c_k) \rangle \in \mathcal{T}_\nu \times \mathcal{V}^* \times \Theta^+ \tag{1}$$

with the constraint $\{a_1 \ldots, a_n\} \subseteq FV(s)$. We use (1) intuitively to compute on $t \in \mathcal{T}_\nu$ as follows. Suppose $\sigma s = t$. Obtain t' from t by computing on some of the subterms of t, namely the subterms that are paired with $a_1, \ldots, a_n$ in σ. If t' matches a redex from one of the reduction rules, say $t' = \sigma' r_i$, then the result of the computing with the computation rule is $\sigma' c_i$. More formally, given $t, t' \in \mathcal{T}_\nu$, t computes to t' in at most m steps, $t \Downarrow_m t'$, for $m \geq 0$, if one of the following holds.

1. $m = 0$ and $t =_\alpha t'$.
2. There is a rewrite rule $s \Rightarrow s' \in \Delta$ and a substitution σ such that $t = \sigma s$, $t' = \sigma s'$, and $m \geq 1$.
3. There is a computation rule $\langle s, (a_1, \ldots, a_n), (r_1 \rightarrow c_1, \ldots, r_k \rightarrow c_k) \rangle$ and substitution σ such that $t = \sigma s$, $a_1 \Downarrow_{m_1} t_1, \ldots, a_n \Downarrow_{m_n} t_n$, where $m > \sum_{i=1}^{n} m_i$. In addition, for some j, $1 \leq j \leq k$, there exists a substitution σ' such that $\sigma'' r_j = \sigma(s[t_1/a_1, \ldots t_n/a_n])$ and $t' = \sigma' c_j$, where σ'' is obtained from σ by removing all substitution pairs that contain a_i as a target.

3.2 Rule Interpreter

In this section we describe the rule interpreter, the component of the refiner which carries out primitive inference with respect to goal directed reasoning.

A *goal* is a sequence of declarations $(x_1 : H_1, \ldots, x_n : H_n)$ paired with a Nuprl term C where for each i, $x_i \in \mathcal{V}$, $H_i \in \mathcal{T}_\nu$, $FV(H_i) \subseteq \{x_1, \ldots, x_{i-1}\}$ for $i \in \{1, \ldots, n\}$ and $FV(C) \subseteq \{x_1, \ldots, x_n\}$. We write the goal as

$$x_1 : H_1, \ldots, x_n : H_n \vdash C.$$

Contrary to the definition of a goal in [4], the Nuprl 4 refiner assigns no semantics to a goal. A goal is a purely syntactic object. A goal is used to represent problems for a particular problem domain. Usually, the sequence of declarations represents a context in which to solve the conclusion.

Goals are decomposed into subgoals using decomposition rules. The decomposition rules are encodings of the inference rules of an object logic. The refiner provides a specification language for users to specify decomposition rules. The specification language was designed by Rich Eaton while generalizing the syntax of the rules of the Nuprl type theory. The language is capable of expressing decomposition rules that are schemata as well as those that are decision procedures. In this paper, we only describe the specification and application of rules which can be specified as schemata.

The goals in decomposition rules are represented as schemes. That is, scheme variables can occur in place of a sequence of declarations and variables. For example, in the &-introduction decomposition rule below, H, A, B, a and b are scheme variables.

```
H ⊢ A & B ext pair(a;b)
  by AndIntro
  H ⊢ A ext a
  H ⊢ B ext b
```

We refer to goals in decomposition rules as *goal schemes.* The top goal scheme is the main goal scheme, and the others are subgoal schemes. For the &-introduction decomposition rule, the main goal scheme is `H ⊢ A & B`, and `H ⊢ A` and `H ⊢ B` are the subgoal schemes.

A *scheme substitution* is a set of substitution pairs whose targets are scheme variables and whose replacements are either terms, variables, or sequences of declaration. All scheme variables occur free in a term. Scheme substitution combines substitution as defined in Section 3.1 with textual replacement. For instance, instantiating the scheme $H \vdash t[s/x]$ with scheme substitution τ is $\tau H \vdash (\tau t)[\tau s/\tau x]$ instead of $\sigma H \vdash \tau(t[s/x])$. For clarity, we use $\tau, \tau', \tau_1, \tau_1', \ldots$ to range over scheme substitutions.

The rule interpreter decomposes a goal using decomposition rules as follows. Consider again the decomposition rule for &-introduction, and the goal

$$X : \texttt{Prop}, Y : \texttt{Prop} \vdash X \Rightarrow Y \& X \Rightarrow Y. \tag{2}$$

The rule interpreter matches (2) against the main goal scheme of the decomposition rule, producing a scheme substitution with pairs $\langle \texttt{H}, (X : Prop, Y : Prop) \rangle$, $\langle \texttt{A}, X \rangle$, and $\langle \texttt{B}, Y \rangle$. The rule interpreter uses this substitution to instantiate the subgoal schemes of &-introduction to generate the goals

$$X : Prop(i), Y : Prop(i) \vdash X \Rightarrow Y \tag{3}$$

and

$$X : Prop(i), Y : Prop(i) \vdash Y \Rightarrow X. \tag{4}$$

In addition to specifying goal decomposition, decomposition rules also specify construction of solutions of goals. Suppose that the solution for (3) is s_1 and the solution for (4) is s_2. Then according to the &–introduction decomposition rule, the solution of (2) is $\texttt{pair(}s_1\texttt{;}s_2\texttt{)}$. In a decomposition rule, the term following `ext` is called the extract. A solution is generated for a goal from the extract adjacent to the main goal scheme that the goal matches. The inclusion of the extracts of the subgoal schemes in the extract of the main goal scheme stipulates that the solution of a goal matching the main goal scheme is constructed from the solutions of its subgoals.

The general form of an decomposition rule is given below.

$$\begin{array}{l}
H \;\vdash\; C \quad \texttt{ext}\ t \\
\quad\;\; \texttt{by}\ r\ a_1\ \ldots\ a_m \\
\quad H_1 \;\vdash\; C_1\ \texttt{ext}\ t_1 \\
\qquad\quad \vdots \\
\quad H_n \;\vdash\; C_n\ \texttt{ext}\ t_n \\
\texttt{with}\ f_1\ \cdots\ f_k
\end{array}$$

We abbreviate it as

$$g\ \texttt{ext}\ t\ \texttt{by}\ r\ a_1\ \ldots a_m\ g_1\ \texttt{ext}\ t_1\ \cdots\ g_n\ \texttt{ext}\ t_n\ \texttt{with}\ f_1\ \cdots\ f_k\texttt{,}$$

where g is $H \vdash C$, and g_i is $H_i \vdash C_i$ for each g_i. The name of the decomposition rule is r, and $a_1 \ldots a_m$ are its argument schemes. *Argument schemes* are designed to match arguments to decomposition rules which can be terms, natural numbers for referring to a specific declaration, or other entities. Side conditions are checked by evaluating the partial functions $f_1, \ldots, f_n$ on substitutions. Moreover, these functions can invoke procedures that generate terms which can be instantiated in the subgoal schemes and the main goal scheme's extract. In order to generate solutions correctly, we require that decomposition rules be well formed. More specifically an decomposition rule

$$g \texttt{ ext } t \texttt{ by } r\ a_1\ \ldots a_m\ g_1 \texttt{ ext } t_1\ \cdots\ g_n \texttt{ ext } t_n \texttt{ with } f_1,\ \cdots\ f_k$$

is well–formed if each t_i is a distinct variable or is a closed term. For the rest of the paper, we assume that all decomposition rules are well–formed.

Given a set of decomposition rules Γ, we define a *justification* based on Γ as $r\ a'_1 \ldots a'_m$ if there exists an decomposition rule

$$q \texttt{ ext } t \texttt{ by } r\ a_1\ \ldots a_m\ q_1 \texttt{ ext } t_1\ \cdots\ q_n \texttt{ ext } t_n \texttt{ with } f_1\ \cdots\ f_k$$

in Γ and a substitution scheme τ such that $\tau a_i = a'_i$ for each a_i. We write the justification as $r\tau$. Given a goal g, $r\tau$ *decomposes* g into subgoals $g_1, \ldots, g_n$ and partial solution $\varepsilon(e)$, written as $g \stackrel{r\tau}{\rightharpoonup} \langle (g_1, \ldots, g_n), \varepsilon(e) \rangle$, where the following hold.

1. $g = \tau q$.
2. $\tau' = f_k(f_{k-1}(\cdots(f_1 \tau)\cdots))$.
3. For each g_i, $g_i = \tau' q_i$.
4. If t_i is a variable, then there is a variable $x \in FV(e)$ such that $x = t_i$.
5. The partial solution $\varepsilon(e)$ is a function from ${\mathcal{T}_\nu}^n$ to $\mathcal{T}_\nu$, such that $\varepsilon(e)(s_1, \ldots, s_n)$ is the term obtained by substituting t_i with s_i if $t_i \in FV(e)$.

If $n = 0$, then we say $r\tau$ *achieves* g. For a given set of decomposition rules, Γ, we denote all the justifications based on Γ as $\hat{\Gamma}$.

A *solution* of a goal g with respect to a set of decomposition rules Γ is constructed inductively as follows. If for some $r \in \hat{\Gamma}$, $g \stackrel{r}{\rightharpoonup} \langle (), \varepsilon(e) \rangle$, then $\varepsilon(e)()$ is the solution of g with respect to Γ. If for some $r \in \hat{\Gamma}$, $g \stackrel{r}{\rightharpoonup} \langle (g_1, \ldots, g_n), \varepsilon(e) \rangle$ and $s_1, \ldots s_n$ are the solutions of $g_1, \ldots, g_n$ with respect to Γ, respectively, then $\varepsilon(e)(s_1, \ldots s_n)$ is the solution of g with respect to Γ. A goal g is *solvable* with respect to Γ if g has a solution with respect to Γ.

3.3 Proof Manager

The proof manager is the component of the refiner which carries out tactic applications and combines partial solutions to generate solutions. In describing the proof manager, we implicitly refer to a fixed set of decomposition rules.

A *goal decomposition tree*[2] is a tree whose nodes are either goals or triples containing a goal, justification, and a partial solution. The interior nodes of a goal decomposition are always triples, while a leaf can be a triple or a goal. A leaf is *open* if it is a goal. In addition, for any interior node $\langle g, r, e \rangle$, $g \stackrel{r}{\rightharpoonup} \langle (g_1, \ldots, g_n), e \rangle$, and the i–th child is either g_i or $\langle g_i, r_i, e_i \rangle$ for some justification r_i and partial solution e_i.

[2] Traditionally, known as the primitive refinement tree.

Given a goal decomposition tree G with n open leaves, $n \geq 0$, and q a node in G, we define $ext_G(q)$ as a function from sequences of terms to terms as follows. If q is a the i-th open leaf encountered during left-to-right depth first traversal of G, then

$$ext_G(q) \quad = \quad \lambda(s_1, \ldots, s_n).s_i.$$

If $q = \langle g, r, e \rangle$ with children $q_1, \ldots, q_m$, $m \geq 0$, then

$$ext_G(q) \quad = \quad \lambda s.e(ext_G(q_1)(s), \ldots, ext_G(q_n)(s)).$$

If q is the root of G, we write $ext_G(q)$ as $ext(G)$. Note if q is the leaf $\langle g, r, e \rangle$, then $ext_G(q)() = e()$, the solution of g. Therefore, we can establish by induction that if none of the leaves of G are open that $ext(G)()$ is the solution to the goal at the root of G. We state this as the following theorem.

Theorem 3.2 A goal g is solvable if and only if there exists a goal decomposition tree G of g without any open leaves.

Corollary 3.3 If G is a goal decomposition tree of g without any open leaves, then $ext(G)$ is a solution of g.

3.4 Tactic Refinement

A *proof* is a finite tree whose nodes are either a goal or a triple containing a goal, a partial solution, and a tactic. A *tactic* is a partial function from goals to goal decomposition trees.[3] A node labeled with a goal, partial solution, and a tactic is *complete*; otherwise it is *incomplete*. A proof is *finished* if all its nodes are complete; otherwise it is *unfinished*. A proof is *raw* if it contains one incomplete node. The *premises* of a proof are its incomplete leaves. The *frontier* of a proof is the premises of the proof. The *main goal* of a proof is the goal of the proof's root node. We let $\mathcal{P}$ be the set of proofs.

The purpose of a tactic is to perform multiple steps of inference in a single step. Therefore, we desire only to retain part of the goal decomposition tree returned by a tactic, namely the root and the open leaves. We define the *tactic refinement* of proof P to a proof Q by the tactic t, $P \overset{t}{\mapsto} Q$, as follows. Let g be the main goal of P, and let $t(g) = G$. Let $g_1, \ldots, g_n$, for $n \geq 0$, be the open leaves of G where g_i is the i-th open leaf encountered during left–to–right depth first traversal of G. Let $e = ext(G)$. If the goal at the root of G is g, then $P \overset{t}{\mapsto} Q$ where the root of Q is labeled $\langle g, t, e \rangle$ and its children in left to right order are $g_1, \ldots, g_n$.

Given $P, Q \in \mathcal{P}$ and tactic t, t refines P to Q, $P \overset{t}{\Rightarrow} Q$, if there exists a premise P' of P, $P' \overset{t}{\mapsto} Q'$, and Q is P with P' replaced with Q'. The *tactic*

[3] Tactics in Nuprl are actually partial functions from proofs to proofs which are directly or indirectly composed of tactics from goals to goal decomposition trees.

refined proofs, $\hat{\mathcal{P}_T}$, are the proofs P such that P is raw, or there exists a tactic and a proof $Q \in \hat{\mathcal{P}_T}$ such that $Q \stackrel{t}{\Rightarrow} P$.

We extend $ext()$ to extract partial solutions from tactic proofs. The extension requires no significant changes to the original definition of $ext()$. Similar to goal decomposition trees, we can determine whether the main goal of a proof is solvable based on the structure of the proof.

Theorem 3.4 The main goal g of $P \in \hat{\mathcal{P}_T}$ is solvable if and only if P is finished.

Corollary 3.5 If $P \in \hat{\mathcal{P}_T}$ is finished then the solution of the main goal of P is $ext(P)$.

4 The Refiner as a Refinement Calculus Inference Engine

The refinement calculus is a methodology for deriving programs in Dijkstra's guarded command programming language from their specifications [6]. In the refinement calculus, specifications and programs are combined into a single language. The refinement calculus relates programs using the binary relation "refines to", written as $\sqsubseteq$, over programs. Intuitively, if for two programs S and S' such that $S \sqsubseteq S'$, then for any specification that S' implements S also implements. If S is a specification and S' is a pure guarded command program, then S' is an implementation of S. In [17], the refinement calculus is presented with several laws for refining specifications into concrete programs. Using the refiner for mechanizing reasoning for the refinement calculus stems from the presentation of the refinement calculus to exploit David Gries' methodology for program derivation in a goal directed fashion [18, 12].

The object language of the refinement calculus consists of first order logic with natural numbers and programs. This language can be divided into three syntactic categories, expressions, formulas, and programs. Expressions are normally called terms in most presentations of first order logic. We present expressions as the following abstract syntax where e and e' range over expressions.

$$e ::= x|n|e + e'|e \times e'|e - e'|e \div e'$$

Above, x ranges over variable symbols, and n ranges over natural numbers.

The meta–variables ϕ and ψ range over formulas of the refinement calculus in the following abstract syntax for formulas.

$$\phi ::= e = e'|e < e'|\neg\phi|\phi\wedge\psi|\phi\vee\psi|\phi\Rightarrow\psi|\forall x.\phi|\exists x.\phi$$

We intend to only encode a subset of guarded command programs. The abstract syntax for this subset including specifications is given below where P and Q range over programs.

$$P ::= x := e|w : [\phi, \psi]|P; Q|\textbf{if } \phi \textbf{ then } P \textbf{ else } Q \textbf{ fi}|\textbf{do } \phi \longrightarrow P \textbf{ od}$$

$$
\begin{array}{rcl}
\nu(x := e) & = & \mathtt{assign}(x; \nu(e)) \\
\nu(w : [\phi, \psi]) & = & \mathtt{spec}(\mathtt{frame}(x_1; \ldots; x_n); \nu(\phi); \nu(\psi)) \\
 & & \text{where } w = \{x_1, \ldots, x_n\} \\
\nu(P; Q) & = & \mathtt{seq}(\nu(P); \nu(Q)) \\
\nu(\mathbf{if}\ \phi\ \mathbf{then}\ P\ \mathbf{else}\ Q\ \mathbf{fi}) & = & \mathtt{if}(\nu(\phi); \nu(P); \nu(Q)) \\
\nu(\mathbf{do}\ \phi \longrightarrow P\ \mathbf{od}) & = & \mathtt{do}(\nu(\phi); \nu(P))
\end{array}
$$

Figure 1: Encoding programs as Nuprl terms.

The meta–variable w ranges over sets of variable symbols. Specifications are represented as programs of the form $w : [\phi, \psi]$. The set of variable symbols w is called the *frame*. All the free variables of ϕ and ψ are contained in w. Specifications are not executable. A program is executable as long as none of its subprograms are specifications or contains a formula with a quantifier.

To encode the object language of the refinement calculus, we use the mapping ν, which maps members of the syntactic categories of the object language to Nuprl terms. We omit showing the encoding of expressions and formulas due to space. Figure 1 depicts the encoding of programs into Nuprl terms.

Using the refinement calculus we want to solve the problem of deriving a program that implements a specification. In other words, deriving an executable program P for a specification $w : [\phi, \psi]$, where $w = \{x_1, \ldots, x_n\}$ for $n \geq 0$, such that $w : [\phi, \psi] \sqsubseteq P$. We encode specifications as conclusions of goals and executable programs as terms in the refiner. For example, the above specification is encoded as the goal

$$x_1 : \mathtt{nil}, \ldots x_n : \mathtt{nil} \vdash \mathtt{spec}(\mathtt{frame}(x_1; \ldots; x_n); \nu(\phi); \nu(\psi))$$

and is displayed as

$$x_1, \ldots, x_n \vdash x_1, \ldots, x_n : [\nu(\phi), \nu(\psi)].$$

Since each x_i occurs free in the conclusion, a corresponding declaration identifier x_i must be present in the declaration list, in order to be consist with the definition of goal. The term `nil` is insignificant and used only as a placed holder. If the object language had types, then `nil` would be replaced with the type of x_i. In attempting to derive an executable program that refines a specification, certain formulas have to be proven true. Therefore, the conclusions to some goals will be formulas. The solution to these goals will be first order logic proofs.

For brevity, we only give the laws of the refinement calculus that introduce program statements and omit those for frame variables and formulas. The laws

Assignment :

$$\frac{w, x : [\phi, \psi]}{x := e} \quad (x = e)\wedge\phi\Rightarrow\psi[e/x]$$

SequentialComposition :

$$\frac{w : [\phi, \psi]}{w : [\phi, \theta]; w : [\theta, \psi]} \quad \phi\Rightarrow\theta$$

If :

$$\frac{w : [\phi, \psi]}{\textbf{if } \theta \textbf{ then } w : [\phi\wedge\theta, \psi] \textbf{ else } w : [\phi\wedge\neg\theta, \psi] \textbf{ fi}}$$

Do :

$$\frac{w : [\phi\wedge\psi, \phi'\wedge\psi]}{\textbf{do } \theta \longrightarrow w : [\theta\wedge\psi, \psi] \textbf{ od}} \quad \neg\theta\wedge\psi\Rightarrow\phi'$$

Figure 2: Refinement Calculus Laws

are written as $\frac{S}{P}$, and are interpreted as $S \sqsubseteq P$. The laws are given in Figure 2. The frame w, x in the Assignment law abbreviates $w \cup \{x\}$.

Figure 3 contains the encoding of the refinement calculus laws from Figure 2 as Nuprl decomposition rules. Given a law, we encode it as a Nuprl decomposition rule as follows. The encoding of the specification above the bar becomes the main goal scheme of the decomposition rule. The conclusion of the subgoals are the side conditions of the laws and the specifications in the program statements. If the conclusion of the goal scheme is obtained from a specification, then the extract represents an executable program which refines the specification. Therefore, it appears in the extract of the main goal scheme. Extracts that are a single letter are variables. The extracts of goal schemes whose conclusions are encodings of formulas are the term `nil`.

Figure 4 contains a tactic proof of the goal representing the specification

$$x, y, z : [\text{true}, x \leq z\wedge y \leq z\wedge(x = z\vee y = z)].$$

The tactic IfThenElseTac applies the inference rule IfThenElse to a goal and performs boolean simplification on the subgoals. The Tactic AssignmentTac applies inference rule Assignment to a goal and deduces the subgoals using knowledge about first order logic and set theory. From the tactic proof in Figure 4 we can extract the executable program `if x≥y then z:=x else z:=y fi`.

To determine whether the encoding is *faithful* we do the following. Let $\mathcal{G}$, $\mathcal{S}$, and $\mathcal{R}$ be the specifications, executable programs, and the laws of the refinement calculus, respectively. To determine faithfulness, we show that for any specification $s \in \mathcal{G}$, there exists an executable program $p \in \mathcal{S}$ such that

```
C ⊢ w:[pre,post] ext x := e
        Assignment x e
    C ⊢ pre ⇒ post[e/x] ext nil
    C ⊢ x ∈ w ext nil

C ⊢ w:[pre,post] ext p;q
        SequentialComposition mid
    C ⊢ w:[pre,mid] ext p
    C ⊢ w:[mid,post] ext q
    C ⊢ pre ⇒ mid ext nil

C ⊢ w:[pre,post] ext if b → p ▯ ¬b → q
        IfThenElse b
    C ⊢ w:[b ∧ pre,post] ext p
    C ⊢ w:[¬b ∧ pre,post] ext q

C ⊢ w:[pre ∧ inv,inv ∧ post] ext do G → p od
        Iteration G
      C ⊢ w:[inv ∧ G, inv] ext p
      C ⊢ ¬G ∧ inv ⇒ post ext nil
```

Figure 3: Encoding of Refinement Calculus laws in Nuprl

```
x,y,z ⊢ x,y,z:[true, x≤z ∧ y≤z ∧ (x=z ∨ y=z)]
        IfThenElseTac x≥y
    x,y,z ⊢ x,y,z:[x≥y, x≤z ∧ y≤z ∧ (x=z ∨ y=z)]
            AssignTac x z
    x,y,z ⊢ x,y,z:[x<y, x≤z ∧ y≤z ∧ (x=z ∨ y=z)]
            AssignTac y z
```

Figure 4: Tactic Proof of $x, y, z : [\mathrm{true}, x \leq z \wedge y \leq z \wedge (x = z \vee y = z)]$

$s \sqsubseteq p$ if and only if there is a goal decomposition tree G of $\nu(s)$ with respect to the encoding of $\mathcal{R}$ so that $\nu(p) = ext(G)$. In general, if we represent a logic as $\langle \mathcal{G}, \mathcal{S}, \mathcal{R}, \alpha \rangle$ where $\alpha \subseteq \mathcal{G} \times \mathcal{S}$, R is a finite set of partial functions from $\mathcal{G} \times \mathcal{S}$ to $(\mathcal{G} \times \mathcal{S})^*$, and for $r \in \mathcal{R}$, $p \in \alpha$, and $r(p) = (p_1, \ldots, p_n)$ for $n \geq 0$ that each $p_i \in \alpha$. Let $\tilde{}$ represent the encoding of the logic in Nuprl. More specifically, let $\tilde{\mathcal{G}}$, $\tilde{\mathcal{S}}$, $\tilde{\mathcal{R}}$ be the encodings of $\mathcal{G}$, $\mathcal{S}$, and $\mathcal{R}$ in Nuprl, respectively. Then $\tilde{}$ is a faithful encoding if for all $g \in \mathcal{G}$ there exists an $s \in \mathcal{S}$ such that $\langle g, s \rangle \in \alpha$ if and only if there exists a decomposition tree G of $\tilde{g}$ with respect to $\tilde{R}$ such that $\tilde{s} = ext(G)$.

5 Discussion

We conclude that the Nuprl refiner performs inference using loose semantics. As a result Nuprl can be used as a workbench for building mechanized theorem provers. By using Nuprl as a workbench, a tool can be constructed for developing imperative programs in a goal–directed fashion by using the encoding of the refinement calculus in Section 4 and writing tactics to automate reasoning. We believe this approach is less costly than developing a tool from scratch. One could use a generic theorem prover implementing a logical framework as a workbench to develop such a tool. However, in addition to automating reasoning for the refinement calculus, the tactics must accommodate for reasoning within the logical framework. The degree of difficult this may cause depends on the encoding of the refinement calculus into the general logic and the expressiveness of the general logic. Another approach is *abstract refinement*; the concept defined by Timothy Griffin for constructing proof development systems using loose semantics [13]. Tactics in abstract refinement and LCF have the same definition. In abstract refinement, tactics are constructed to perform inference in the object logic independently of any encoding of the object logic's inference rules. Tactics are ensured of mimicking inference in the object logic by accepting only the outcomes of tactic applications obtainable by a deduction in the object logic. Determining faithfulness is simple as long as verifying valid deductions is decidable.

6 Future Work

In the future, we intend to redesign the refiner to execute tactics in parallel on a multi–processor. As result, having the potential to implement *concurrent tactics*. Our first step toward implementing concurrent tactics was the development of a Standard ML runtime library that permits several Standard ML processes to communicate via shared memory. In addition, we intend to further develop the refinement calculus within Nuprl. Our goal is to provide high level tactics similar to the tactics for automating the tedious aspects of hardware verification in HOL and Nuprl [1]. Thereby, obtaining a tool for deriving imperative programs from their specifications.

7 Acknowledgments

I would like to thank Stuart Allen, Rich Eaton, and Bob Constable for discussing Nuprl's system design with me. I specially appreciate Stuart Allen's assistance in fine tuning the mathematical description of the refiner. I also would like to thank Chet Murthy for his advice on re-implementing the Nuprl 4 refiner.

References

[1] M. Aagaard, M. Leeser, and P. Windley. Toward a super duper hardware tactic. In *Higher Order Theorem Proving and its Applications.* Lecture Notes in Computer Science, 780, Springer-Verlag, 1993.

[2] W. E. Aitken. *Reflecting on Nuprl.* PhD thesis, Cornell University, soon.

[3] W. E. Aitken, R. L. Constable, et al. Reflecting on Nuprl lessons 1–4. Lecture from graduate type theory course at Cornell University, Sept. 1992.

[4] S. F. Allen. *A Non-type-theoretic Semantics for a Type Theoretic Language.* PhD thesis, Cornell University, 1987.

[5] A. Avron, F. Honsell, and I. Mason. Using typed lambda calculus to implement formal systems on a machine. Technical Report LFCS Report Series ECS-LFCS-87-31, "Laboratory for the Foundations of Computer Sciene, Edinburgh University", 1987.

[6] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[7] R. Constable, S. Allen, and D. Howe. Reflecting the open–ended computation system of constructive type theory. In F. Bauer, editor, *Logic, Algebra and Computation*, pages 267–288. Springer-Verlag, 1991.

[8] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Englewood Cliffs, NJ, 1986.

[9] R. L. Constable and D. J. Howe. Nuprl as a general logic. In *Logic and Computer Science*, pages 77–90. Academic Press, 1990.

[10] M. Gordon. Hol: A proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.

[11] M. Gordon, A. Milner, and C. WadsWorth. *Edinbugh LCF.* Lecture Notes in Computer Science 78. Springer-Verlag, London, 1979.

[12] D. Gries. *Science Of Programming.* Springer-Verlag, New York, 1981.

[13] T. G. Griffin. *Notational Definition and Top-Down Refinement for Interactive Proof Development Systems.* PhD thesis, Cornell University, 1989.

[14] G. Huet et al. The Coq Proof Assistant User's Guide: Version 5.10. Unpublished, 1995.

[15] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[16] P. Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra.* PhD thesis, Cornell University, 1995.

[17] C. Morgan and T. Vickers. Types and invaariants in the refinement calculus. In C. Morgan and T. Vickers, editors, *On the Refinement Calculus*, pages 127–154. Springer-Verlag, 1992.

[18] R. G. Nickson and L. J. Groves. Metavariables and conditional refinements in the refinement calculus. In *Sixth Refinement Workshop*, pages 167–187. Springer Verlag, 1994.

[19] L. C. Paulson. *Isabelle: A Generic Theorem Prover.* Lecture Notes in Computer Science 828. Springer-Verlag, Berlin, 1994.

[20] E. Rich and K. Knight. *Artificial Intelligence.* McGraw Hill, New York, 1991.