Netty: A Prover's Assistant

Eric Hehner and Lev Naiman
Department of Computer Science
University of Toronto
40 St. George Street, Toronto, Canada
Email: naiman@cs.toronto.edu

***Abstract*—Netty is a prover's assistant. It supports a calculational style of proof, and allows the creation of proofs that are correct by construction. It provides an intuitive interface that allows direct manipulation of expressions. The key idea is that instead of tactics for proof the user only needs to select the next step of the proof from a list of suggestions. These suggestions are usually the result of unification. Netty provides mechanisms to facilitate proving, such as making use of local assumptions and the use of monotonicity in subproofs. Program refinement from specification can be done in Netty similarly to any other kind of proof.**

***Index Terms*—proof; calculation; tool**

I. Introduction

Proof assistants have been created with various purposes. One such purpose is complete automation; this is useful when we need a proof and we care about the result rather than the process. Such provers have been successful, although the theories they can reason about are limited. Some add freedom of theory by being only partially automatic, requiring some user guidance to produce a proof, and are known as interactive theorem provers. Examples of such provers include HOL, PVS, and Coq [3][10][4]. Some tools have specific modes for program verification. Other tools are specific to program verification or refinement, and hence are restrictive to the theories allowed [8]. Tools like Isabelle [9] require the user to use tactics for proving, which requires the user to learn another meta language. The most similar existing tool to Netty is KeY [2], since it allows users to pick a rule to apply from a list of applicable rules. In contrast, Netty allows users to pick the result of applying rules. KeY does not support the use of local assumptions or monotonicity, and is restricted to the theories it allows.

Despite their usefulness, these tools present a difficulty when used to teach logic. There are often complicated tactics for proof, and perhaps some meta-language for performing certain operations, or a scripting language for creating user-defined tactics. This means that a user must learn a concept that is almost as complicated as programming before being able to prove any expression, regardless of how simple it is.

Netty [7] is a prover's assistant named for Antonetta van Gasteren (1952-2002), a pioneer of calculational proof. It is based on work by Robert Will [11]. Its main purpose is pedagogical; it aims to foster understanding about how a proof is constructed and why each step is allowed. It supports a calculational type of proof, that is similar to the successfully used Structured Derivations [1]. It allows the direct manipulation of expressions and subexpressions to advance a proof. Advancing a proof is usually done by picking an expression from a list of suggestions that Netty provides to be the next line. The importance of this is that it allows a user to concentrate on the proof itself rather than learning how to use the tool.

The paper is organized as follows: Section II discusses the use and structure of calculational proof. Section III shows how the main parts of Netty are integrated. Section IV shows the structure, display and navigation of proofs in Netty. Section V-E is about advancing a proof; it discusses how Netty generates suggestions to proceed with proofs and how the user interacts with the tool to advance the proof. This section also discusses the special mechanisms in proof, and how suggestions are filtered. Section VI shows how a program refinement is performed (identical to any other proof).

II. Calculational Proof

Calculational proof is a fixed and structured format for presenting proofs. It makes proofs and calculations equivalent in that each step has an explicit justification, usually a law. A human prover may have a reason for constructing a proof, and they may have a proof strategy in mind, but these are not our concerns; our concern is to provide a tool that makes proof construction easy and fully formal. The advantage of a fully formal proof is that its correctness is machine checkable. It has been adopted by several researchers in formal methods and used effectively for teaching mathematics at a high-school level [1]. A calculational proof is a bunch of expressions with connective operators between them, whose transitive relation allows us to conclude something about the first and last expression of the proof.

For example, a calculational proof that there is no smallest integer might look like this:

$$\begin{array}{ll}
\quad\neg\exists\langle n : int \rightarrow \forall\langle m : int \rightarrow n \leq m\rangle\rangle & \text{Specialization} \\
\Leftarrow \neg\exists\langle n : int \rightarrow \langle m : int \rightarrow n \leq m\rangle\,(n-1)\rangle & \text{Function Apply} \\
= \neg\exists\langle n : int \rightarrow n \leq n-1\rangle & \text{Identity Law} \\
= \neg\exists\langle n : int \rightarrow n-0 \leq n-1\rangle & \text{Cancellation} \\
= \neg\exists\langle n : int \rightarrow 0 \leq -1\rangle & \text{Ordering} \\
= \neg\exists\langle n : int \rightarrow \bot\rangle & \text{Quantifier Identity} \\
= \top &
\end{array}$$

The top line of this proof can be read "there does not exists n of type integer, such that for all m of type integer,

n is less than or equal to m", and the bottom line can be read as "true". The angle brackets serve as the scope of a function and as the scope of a quantified variable (discussed in Section V-C). Notational peculiarities are not the point here; any reader interested in the notational details is referred to [6]. We say that a proof proves an expression '$expn$' if we show that $expn = \top$ or $\top \Rightarrow expn$, and we say that it proves $\neg expn$ if $expn = \bot$ or $expn \Rightarrow \bot$. Hence in this example we say we proved $\neg\exists\langle n : int \rightarrow \forall\langle m : int \rightarrow n \leq m\rangle\rangle$. Each line in the proof has a hint to its right telling how the next line is created. For example, the line $\neg\exists\langle n : int \rightarrow n \leq n - 1\rangle$ has the hint 'Identity Law' saying that n is replaced by $n - 0$ in the next line.

III. Overview

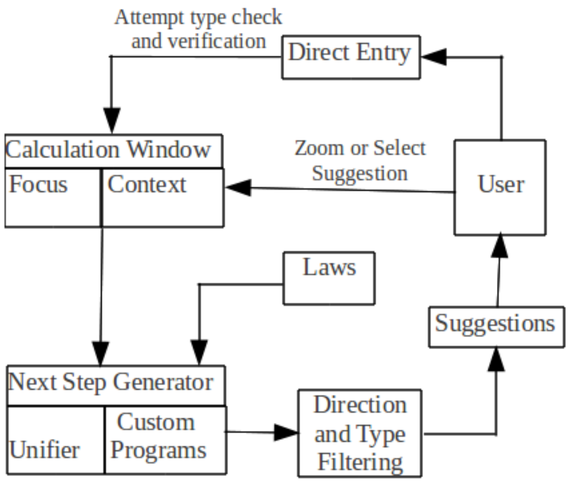


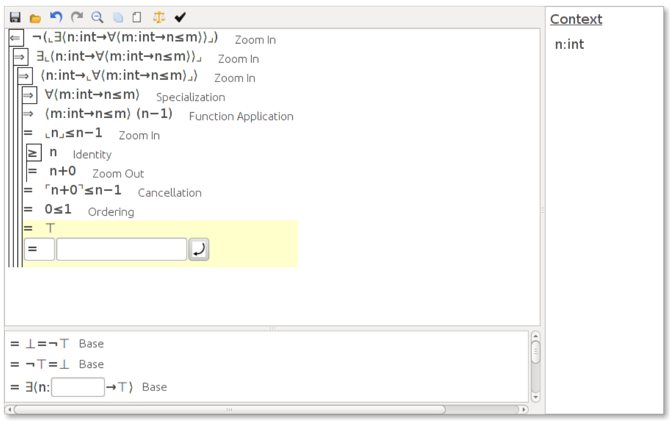
Fig. 1. Backbone

This flowchart illustrates the basic components of Netty. Once a user starts a proof, next steps are generated and filtered as described in Section V. The user can then accept a suggestion, directly enter the next expression, or navigate the proof as described in Section IV.

IV. Proof Display and Navigation

Figure 2 is a screenshot of Netty's calculation window. It is divided into three parts: the proof pane (top left), the suggestions pane (bottom left), and the context pane (right).

The proof pane contains the proof that has been built so far, in the format described in the examples to follow. The suggestion pane contains valid possible next steps in the proof. The context pane displays the laws that are local to the current context. In addition, there is one line selected by the user from which to continue the proof, and it will be referred to as the 'focus'. The box contains the direction in which the proof is allowed to proceed, and hence limits the suggestions that Netty provides for advancing the proof. The proof in Figure 3 serves to explicitly show every step of a Netty proof. Without the boxed directions and vertical lines, we would have a proof



$\boxed{\Leftarrow}$ ¬(⌞∃⟨n:int→∀⟨m:int→n≤m⟩⟩⌟) Zoom In
$\boxed{\Rightarrow}$ ∃⌞⟨n:int→∀⟨m:int→n≤m⟩⟩⌟ Zoom In
$\boxed{\Rightarrow}$ ⟨n:int→⌞∀⟨m:int→n≤m⟩⌟⟩ Zoom In
$\boxed{\Rightarrow}$ ∀⟨m:int→n≤m⟩ Specialization
⇒ ⟨m:int→n≤m⟩ (n−1) Function Application
= ⌞n⌟≤n−1 Zoom In
$\boxed{\geq}$ n Identity
= n+0 Zoom Out
= ⌜n+0⌝≤n−1 Cancellation
= 0≤1 Ordering
= ⊤
=

= ⊥=¬⊤ Base
= ¬⊤=⊥ Base
= ∃⟨n: →⊤⟩ Base

Context
n:int

Fig. 2. Calculation Window

that very closely resembles a Structured Derivation that has a main proof and several nested subproofs. The purpose of the vertical lines is structural; they serve to mark the extent of a proof (or subproof). The low corner brackets mark the subexpression that is used in the following subproof, and the high corner brackets mark the result of the subproof.

0.	$\boxed{\Leftarrow}\neg\llcorner\exists\langle n : int \rightarrow \forall\langle m : int \rightarrow n \leq m\rangle\rangle\lrcorner$	Zoom-in
1.	$\boxed{\Rightarrow}\exists\llcorner\langle n : int \rightarrow \forall\langle m : int \rightarrow n \leq m\rangle\rangle\lrcorner$	Zoom-in
2.	$\boxed{\Rightarrow}\langle n : int \rightarrow \llcorner\forall\langle m : int \rightarrow n \leq m\rangle\lrcorner\rangle$	Zoom-in
3.	$\boxed{\Rightarrow}\forall\langle m : int \rightarrow n \leq m\rangle$	Specialization
4.	$\Rightarrow \langle m : int \rightarrow n \leq m\rangle\,(n - 1)$	Function Apply
5.	$= \llcorner n\lrcorner \leq n - 1$	Zoom-in
6.	$\boxed{\geq}n$	Identity Law
7.	$= n + 0$	Zoom-out
8.	$= \ulcorner n + 0\urcorner \leq n - 1$	Cancellation
9.	$= 0 \leq -1$	Ordering
10.	$= \bot$	Zoom-out
11.	$\Rightarrow \langle n : int \rightarrow \ulcorner\bot\urcorner\rangle$	Zoom-out
12.	$\Rightarrow \exists\ulcorner\langle n : int \rightarrow \bot\rangle\urcorner$	Zoom-out
13.	$\Leftarrow \neg\ulcorner\exists\langle n : int \rightarrow \bot\rangle\urcorner$	Quantifier Identity
14.	$= \top$	

Fig. 3. Detailed Proof

A. Directions and Connectives

The idea of the direction is that in order to conclude some relationship between the first and last line of a proof the connectives for each line must have a transitive relation. For example, the direction $\Leftarrow$ in line 0 allows lines 13 and 14 to use either the $=$ or $\Leftarrow$ connective; the direction $\Rightarrow$ in line 3 allows lines 4, 5, 8, 9, and 10 to use either the $=$ or $\Rightarrow$ connective; the direction $\geq$ in line 6 allows line 7 to use any of the connectives $=, \geq$ or $>$. A direction of $=$ allows only the $=$ connective. Notice that the direction symbol must have the

$$\begin{array}{|ll}
\boxed{\Leftarrow} \neg\exists\langle n : int \to \llcorner\forall\langle m : int \to n \le m\rangle\lrcorner\rangle & \text{Specialization} \\
\Leftarrow \neg\exists\langle n : int \to \ulcorner\llcorner\langle m : int \to n \le m\rangle\,(n-1)\lrcorner\urcorner\rangle & \text{Function} \\
 & \text{Application} \\
= \neg\exists\langle n : int \to \ulcorner\llcorner n\lrcorner \le n-1\urcorner\rangle & \text{Identity Law} \\
= \neg\exists\langle n : int \to \ulcorner\llcorner n-0\urcorner \le n-1\lrcorner\rangle & \text{Cancellation} \\
= \neg\exists\langle n : int \to \ulcorner\llcorner 0 \le -1\lrcorner\urcorner\rangle & \text{Ordering} \\
= \neg\exists\langle n : int \to \ulcorner\bot\urcorner\rangle & \text{Quantifier Identity} \\
= \top &
\end{array}$$

Fig. 4. Compressed Proof

same type as the expression in the proof. In such a way Netty can combine several theories, such as booleans, numbers, and sets.

The direction remains constant throughout a single level of proof, but can change as we zoom into or out of a subproof. In line 0 the direction is $\Leftarrow$ but we zoom past the $\neg$ sign changing the direction in line 1 to $\Rightarrow$.

The line in a parent proof directly after a subproof must also have a connective. Without the subproof it is just a regular line of proof, except that the justification for it is what the subproof proves. It is as if the subproof was a lemma about a subexpression of an expression in the main proof, which is used just like any law, preserving any monotonic properties. This is shown in line 7 of the example in Figure 4. The connective for the subsequent line is determined as follows:

1. If the subproof proves equality, the new leading connective is $=$
2. Otherwise the new leading connective is the direction of the line we zoomed in from

The subsequent line is the same as the line before the subproof, except that the subexpression that was considered in the subproof is replaced with the last line of the subproof.

Such support for monotonicity is not strictly necessary in order to produce proofs. However, it saves many unnecessary steps, and is especially useful for program refinement where a specification is strengthened to a program.

B. Zoom

The suggestions that are created for each line are dependent on the whole expression, mostly because laws involve unification. It would be terribly inefficient and cluttered if we were to produce suggestions that included manipulations on each subexpression. This is why unlike Fitch-style natural deductions [5] we allow subexpression selection. By selecting a subexpression of the focus we say we zoom-in, and hence create a subproof. We can also return one level higher to the parent proof by pushing a key, and we call this zooming out. Only the direct subexpressions of the focus are available for zoom. For example, while doing the proof in Figure 3, on line 5 we have the expression $n \le n - 1$. The parts that can be zoomed into to create a subproof are n on the left and $n - 1$ on the right.

Zooming is done one level at a time, and getting to a deeper subexpression is simply several zoom actions. The idea is to make selection gestures simple, and not to require a high degree of accuracy in clicking. It might appear useful to be able to select a deeper subexpression directly in order to proceed faster in the proof. However, this is actually not faster. Selecting the right expression takes longer, and navigating to other subexpressions also takes longer. Finally, the presentation layer removes any unnecessary lines, as shown in figure 4.

In Netty expressions are stored in a tree structure with four main classes: literals, variables, scopes, and (function) applications. Literals represent values like 1, $\top$, or 3.14. Variables are not bound to any specific value, but they can be instantiated during unification. Both variables and literals never have any child nodes. Scopes are used to formally introduce variables, give variables types, and to serve as functions. A scope is of the form $\langle var : domain \to body\rangle$. Applications have an operator (of arbitrary fixity and number of keywords) as the root, and operands as children. The reason that the internal expression structure does not use curried functions for all operators is to allow easy manipulation and use of associativity in the presentation layer. The zoom mechanism then works simply by making a sub-proof initially contain the sub-expression that was selected by the user.

V. Advancing a Proof

A. Unification

The most used algorithm to generate next steps for a proof is unification. The standard unification algorithm is used, with the exception that only the law will have variables that can be unified with sub-expressions of the focus. We differentiate variables and constants by universally quantifying variables. For example, having the law $\forall\langle x : int \to x \ge 4 \Rightarrow x > pi\rangle$ the unifier would attempt to unify $x \ge 4 \Rightarrow x > pi$ with the focus, treating only x as a variable but pi as a constant. This both provides a form for laws that is fully formal, and allows Netty to distinguish variables from constants. In addition, if an expression matches a law completely, then one of the suggestions given is $\top$. Similarly, if the focus is $\top$ then all laws match. In addition, variables that are introduced in local scopes differ from each other even if they have the same identifier. In the one-point law $\exists\langle v : D \to v = a \wedge fv\rangle = \langle v : D \to fv\rangle\ a$ the v on the left side is a different variable than the v on the right side, and can hence be unified with different expressions.

Several law files can be used in Netty; these are plain-text files which are read and parsed by Netty. They can be created, deleted and modified by using any text editor. The laws themselves are simply expressions.

Every line in a proof has some justification at its right hand side (or between lines if more space is needed). This justification is usually the application of some law, and hence it is the name of the law that was applied. In addition, sometimes there are steps that were not justified by a law, or where the type checker could not determine if the focus was of the right type for the law. In these cases, we have a warning symbol, indicating that a certain step is unchecked.

We have the Law Query display mechanism to show a user how the next expression came about (that is, to demonstrate unification). It can also be used to illustrate how the unification algorithm works; for example, suppose somewhere in a proof, we have the two lines:

$$\begin{array}{ll} \quad x \wedge y \Rightarrow (y \vee z \Rightarrow (z \Rightarrow x)) & \text{portation} \\ = x \wedge y \wedge (y \vee z) \Rightarrow (z \Rightarrow x) & \end{array}$$

A click and hold will replace those lines in place by:

$$\begin{array}{ll} \quad a \quad \Rightarrow (\quad b \quad \Rightarrow \; c\,) & \text{portation} \\ = \quad a \quad \wedge \quad b \quad \Rightarrow c & \end{array}$$

The fully formal law is:

$$\forall\langle a, b, c : bool \rightarrow (a \wedge b \Rightarrow c) = (a \Rightarrow (b \Rightarrow c))\rangle$$

The Law Query shows the correspondence (unification) between the variables of the law body and subexpression in the proof.

Occasionally one side of a law matches the focus, and the other side of the law has unconstrained variables. Suppose we have 0 as the focus. Then the right side of the law $x - x = 0$ would match it, and the resulting suggestion would be $= x - x$, leaving x unconstrained [7]. We can have an arbitrary expression placed instead of x, and we cannot generate all possible suggestions for it. Instead, for each unconstrained variable Netty has a dialog box in which a user can type. What is typed in one box for a given variable is inserted into all boxes for that variable, so that the user only has to type it in once.

B. Context

Context is a bunch of local laws in a proof. The idea of context is to be able to make use of local assumptions and to allow the user to only worry about the current expression. At the top-level proof we have no local laws except for the ones the user loaded from law files (discussed in Section V-E). Zooming into subexpressions adds context, and zooming out removes them. This implies that subproofs inherit context from their parent proofs. Context expressions are used exactly like laws, and hence suggestions are generated from them in the exact same manner. The mechanism of context removes the need for any explicit declaration of assumptions, since they can simply be added as an antecedent to the top-level expression. Here are some examples of context rules [6]:

- From $a \wedge b$, if we zoom in on a , we gain context b .
- From $a \vee b$, if we zoom in on a , we gain context $\neg b$.
- From $a \Rightarrow b$, if we zoom in on a , we gain context $\neg b$.
- From $a \Rightarrow b$, if we zoom in on b , we gain context a .
- From $\textbf{if } a \textbf{ then } b \textbf{ else } c \textbf{ fi}$, if we zoom in on a , we gain context $b \neq c$.
- From $\textbf{if } a \textbf{ then } b \textbf{ else } c \textbf{ fi}$, if we zoom in on b , we gain context a .
- From $\textbf{if } a \textbf{ then } b \textbf{ else } c \textbf{ fi}$, if we zoom in on c , we gain context $\neg a$.
- From $\langle var : domain \rightarrow body\rangle$, if we zoom in on $body$, we gain context $var : domain$

To understand the context rules, consider the first one. If we assume b when we zoom into a and b turns out to be true, then we made the right choice. However, if b turns out to be false, there is no harm done in any change to a that assumed b, since the value of the entire expression remains the same (false). Similarly, in the body of a function we can assume var:$domain$, since a function must be applied to an element of its domain.

Internally there is a stack of lists that keeps track of context; we add a list of expressions to the context pane on a zoom-in if a context rule is satisfied, and we pop a list on a zoom-out. A zoom-in can add more than one expression to the current context, since t he context expressions are then broken down; each conjunct is gained as a separate law, and if the expression gained is a negation, we push it down the expression tree and perform a deep negation.

C. Scope

Expressions can contain functions, which declare variables. A function has the form $\langle var : domain \rightarrow body\rangle$. A user can zoom into the body similarly to a zoom on any other expression. Any mention of var within the function scope refers to the locally declared var, which is different than any other variable outside the scope with the same name. When we zoom into the scope we might already have in the current context expressions that include var. However, since it is really not the same variable, such expressions cannot be used inside the scope. Netty displays such unusable context expressions at the bottom of the context list in grey. This is to indicate that although we have not lost the context, it cannot be used at present.

The type of variables is gained from their declaration within a scope. If for example we want to prove $\neg a \Rightarrow (\neg b \Rightarrow \neg a)$, we need to start with $\langle a, b : bool \rightarrow \neg a \Rightarrow (\neg b \Rightarrow \neg a)\rangle$, which gives the context of $a : bool$ and $b : bool$ when we zoom into the function body. This way we maintain full formality and give information to the type-checker.

D. Type Checking

The type checker is currently very basic. Literals are given types when the expression is parsed, and variables are given a type if they are declared through a function scope to have a certain type. In addition, the type checker can be invoked with a list of context expressions. In that case the type checker attempts to find the type for any variables whose type has not yet been determined. Functions can have multiple operands and resulting types, which are read in through a configuration file. For example, the type of $\wedge$ might be $bool \rightarrow bool \rightarrow bool$. If both operands are of type $bool$, then the type checker concludes that the type of the expression is $bool$. The reason that functions can have many types is both to allow overloading of functions and to allow greater freedom of theory; this method allows users to define a theory simply through its axioms.

E. Custom Suggestions

The most common way to proceed with a proof is to pick a suggestion by clicking on one from the suggestion pane. Most of the suggestions there are a result of unification with a law. In addition to laws, we use programs to generate suggestions. For example, it would be rather tedious to prove that $5+4=9$ using only the construction axiom of natural numbers. Instead we use a program that performs addition and outputs $=9$ as a suggestion to $5+4$. From the user's perspective there is little difference, since the suggestions and justifications appear in the exact same way.

Custom programs can make use of the available context, law and current expression to generate suggestions. For example, an assignment statement such as $x{:=}e$ means that x is assigned e, but every other variable in the state space remains the same. If x and y are state variables, $x{:=}e \;=\; x'{=}e \wedge y'{=}y$. However, in this theory the state space is not explicitly stated in the assignment, but would be available in the context since state variables need to be declared. The custom suggestion generator searches the context and laws for all state variable declarations and outputs a suggestion. In a sense, custom programs are the equivalent of tactics in other provers, since they provide a means other than unification to proceed with a proof.

There is one type of suggestion that is special: function application. If we have $\langle v : d \to b\rangle\ x$ as the focus, the suggestion given is the result of applying the function to its argument. Function application is not done by unification but by a program, and hence it appears in the same manner as a suggestion for addition. The difference is that in order to apply a function to an argument it must be in the domain of the function. For trivial checks, such as $1 : nat$ or $a : nat$ where we have the type of a in context, we have a simple type checker to perform that check. However, in Netty the concept of type is more general: the type of a variable is just some bunch that the variable is an element or sub-bunch of [6]. This means that with the added power of this theory comes the burden of non-trivial type checking. This is resolved in the Logical Gaps and Direct Entry section.

F. Logical Gaps and Direct Entry

Direct entry is provided through a dialog box below the current line which allows a user to type the next line of the proof. This allows a proof to proceed when a part of it is still unknown or there is no law for it yet. Proofs in Netty are not required to be completely formal at all stages, and steps without formal justification are marked with a warning sign to indicate a possible logical gap in the proof. A user can return to the unsafe line at any time to complete the proof.

Another method of introducing a logical gap is where the type-checker for function application cannot determine if the operands are of the right type. In that case we have a subproof between the previous line and the focus that requires the user to prove the operands are of the right type. For example, if the function application in figure 5 was done somewhere in a proof, the three dots indicate the need to complete a subproof.

$$\begin{array}{l}
\boxed{=}\ \langle n : nat \to n-1\rangle\ (i \times i)\\
\quad \boxed{\Leftarrow}\ i \times i : nat\\
\qquad \ldots\\
\quad \Leftarrow \top\\
= i \times i - 1
\end{array}$$

Fig. 5. Type Proof

This kind of subproof is different in that it does not result from zooming into a subexpression. It can be viewed as a lemma proved in context. A similar gap in the proof would happen if we were to apply a law where the type of the operands is unclear. For the standard types, such as *bool* and *nat* the type checker resolves almost all such problems. It is only for complex types like lists where such a burden of proof is necessary, as it might be non-trivial.

VI. Program Refinement

Program refinement is done in the exact same manner as any other proof using the program theory described in [6]. In the following example the task is to write a program that cubes a number n using only addition, subtraction, and test for zero. We will use two additional state variable x and y. The initial specification is $x' = n^3$.

$$\begin{array}{rl}
0. & \boxed{\Leftarrow}\ \mathbf{var}\ x, y, n : nat \cdot \llcorner x' = n^3 \lrcorner\\
1. & \quad \boxed{\Leftarrow}\ x' = n^3\\
2. & \quad \Leftarrow x' = n^3 \wedge y' = n^2 \wedge n' = n\\
3. & \quad \Leftarrow \mathbf{if}\ n = 0\ \mathbf{then}\ x := 0.\ y := 0\ \mathbf{else}\\
 & \qquad n := n-1.\ x' = n^3 \wedge y' = n^2 \wedge n' = n.\ n := n+1.\\
 & \qquad \llcorner x' = x + 3y + 3n - 2 \wedge y' = y + 2n - 1 \wedge n' = n \lrcorner\ \mathbf{fi}\\
4. & \qquad \boxed{\Leftarrow}\ x' = x + 3y + 3n - 2 \wedge y' = y + 2n - 1 \wedge n' = n\\
5. & \qquad = x := x + y + y + y - n - n - n - 2.\\
 & \qquad\quad y := y + n + n - 1
\end{array}$$

Fig. 6. Refinement

In Figure 6, we see an example of steps that are not fully formal. Netty allows hiding lines, which might be used when certain lines are deemed obvious in the proof. Lines 3 and 5 are the results of either direct entry, line hiding, or a combination of the two. If any line in the hidden or directly entered lines is unsafe, the resulting line will have a warning symbol as a justification. Allowing unsafe lines allows the user to take larger steps in refinement and then return to fill in the gaps. The resulting program without the proof, which we call *cube*, is:

```
procedure $cube$ is
  if $x = 0$ then $x := 0.\ y := 0$
  else $n := n - 1.\ cube.\ n := n + 1.$
      $x := x + y + y + y - n - n - n - 2.$
      $y := y + n + n - 1$
  fi
```

As in the first example, the only necessary actions are a combination of applying laws to change the focus and zooming in on a subexpression to refine. Once all subexpressions have been refined to a program the user will have refined the entire program. The benefit that the zoom mechanism provides is that it allows a user to safely ignore any other subexpression, while having full use of all the local laws in the current context. For example, in order to perform assignments Netty must know all of the state variables. This information is obtained simply by examining the context laws; if we have in context $x : nat$ and $x' : nat$, Netty will conclude that x is a state variable of type nat.

VII. Conclusion and Future Work

Netty has been designed to make proving in the calculational style easy, and we incorporate programming by refinement smoothly as a special case. A lot of attention has been paid to the user interface and ease of use. All methods of proceeding with a proof have been delegated to the generation of next step suggestions, while providing the user with a convenient method of utilizing local assumptions and monotonicity.

Netty has been implemented in Java, using standard GUI libraries such as swing. We have not yet done any empirical studies to test the usability and effectiveness of Netty. We plan to test the effectiveness of the tool in a fourth-year course in formal methods and in a circuit design course.

Netty is powerful enough to include program theory such as the one in [6]. Instead of having to translate the code into another language, it would be desirable to execute it directly. We currently intend to use Scheme to implement the execution of expressions that have been refined to a program. It would also be desirable to advance the capabilities of the type-checker. Improvements would include the accommodation of union-types, and checking some non-trivial expression equality. Currently Netty has a concrete grammar that restricts the available theories. It would be desirable to allow the user complete freedom of theory, including how operators are defined.

Currently, Netty presents suggestions simply in the order that laws are entered in a law file. Ideally, suggestions should be ordered with the most likely steps at the top of the list. In addition, suggestions can be extended to patterns of steps. There are several benefits to this such as allowing faster proving and a more intuitive progression through the proof while maintaining full formality. This could be absolutely invaluable to learning. The implementation of this sort of pattern detection will likely involve a machine learning algorithm. Currently the most suitable kinds of techniques appear to be reinforcement learning techniques. We would need to use data from our empirical studies as training data for the algorithm.

Acknowledgment

We would like to thank Robert Will for laying the groundwork for the tool and initial discussion. We would also like to thank David Kordalewski for discussing ideas and for assistance in implementing Netty. We would also like to thank the referees for helpful suggestions that have improved the paper.

References

[1] R. Back. *Structured derivations: a unified proof style for teaching mathematics*. Formal Aspects of Computing, vol. 22, no. 5, pp. 629-661, Sep 2010.

[2] B. Beckert, R. Hähnle, and Peter H. Schmitt *Verification of Object-Oriented Software. The KeY Approach*. Springer-Verlang, Berlin, 2007.

[3] A.D. Brucker, L. Brügger, M.P. Krieger, and B. Wolff. *HOL-TestGen 1.5.0 User Guide*. ETH Zurich, Technical Report 670, 2010.

[4] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. Version 8.2, July 2009. ¡http://coq.inria.fr/refman/¿ 31.08.2011.

[5] F.B. Fitch. *Symbolic Logic*. The Ronald Press Company, New York, 1952.

[6] E.C.R. Hehner. *a Practical Theory of Programming*. Springer, New York, 1993. ¡http://www.cs.utoronto.ca/∼hehner/aPToP¿ 31.08.2011.

[7] E.C.R. Hehner, R.J. Will, L. Naiman, and D. Kordalewski. *The Netty Project*. University of Toronto, 2011. ¡http://www.cs.utoronto.ca/∼hehner/Netty¿ 31.08.2011.

[8] A.Y.C. Lai. *A Tool for A Formal Refinement Method*. MSc Thesis, University of Toronto, January 2000.

[9] L.C. Paulson. *Isabelle: A generic theorem prover*. Springer-Verlag, Berlin, 1994.

[10] N. Shankar, S. Owre, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Prover Guide*. Version 2.4, November 2001.

[11] R.J. Will. *Constructing Calculations from Consecutive Choices: a Tool to Make Proofs More Transparent*. MSc Thesis, University of Toronto, January 2010.