1 Introduction

1.1 Background

Formal verification is an exciting field of computer science and mathematics that is dedicated to writing rigorous mathematical proofs in the form of computer programs. These programs are then used to produce better software which is proven to accomplish the goals it was written for. The most basic form of software verification is unit testing, where engineers write out example input and output values for their functions, and confirm that their function behaves correctly on these inputs. While testing this way can increase confidence that a function is correct, there is no direct evidence that the function still works on untested inputs. With formal verification, this doubt is removed - the program is rigorously proven to be correct in all cases. Two important classes of verification tools are model-checkers and proof assistants.

A model checker is a tool in which a user can describe a system, describe properties, and check whether the system satisfies those desired properties. Some examples are Alloy/-Electrum, where users can use specify their system in terms of constraints on its behavior [JSS00]. Another prominent example is the Cryptographic Protocol Shapes Analyzer (CPSA), which is specifically optimized for finding vulnerabilities in cryptographic protocols [Lis+16]. Model checkers like Alloy and CPSA provide a language for users to specify their systems with. They prove claims by reducing statements from their input language into SAT, and then asking a SAT or SMT solver to search for counter-examples [JSS00]. If no counter-examples are found, then a claim is true! One advantage of this approach is that if a system doesn’t operate the way the designer expects it to, then the counter-example will show the designer exactly what the issue is, allowing them to update their system accordingly. However, there is a flaw in that there is no explanation towards why the claims are true, and claims can only be proven up to the bounds that the underlying SAT solver enforces on its input.

Proof assistants are programming languages where users can write mathematical statements and mathematical proofs. Since the proofs are written in a program, a computer can machine-check this proof, indicating that the proof is undeniably valid. If a proof is written by hand, it is possible that the writer and readers both make a mistake such that the proof
is not actually valid, but the statement it "proved" is considered proven. This issue does not occur if proofs are machine-checked. The idea that proofs correspond to programs is called the Curry-Howard isomorphism [SU98], and it is used as a backbone for popular proof assistants such as Lean [Car19].

Proof assistants are advantageous compared to model checkers in that machine-checked proofs are not limited by the bounds of SAT or SMT solvers, and that proofs at least go some way towards explaining why a claim is true. However, not all proof assistants have the ability to generate counter-examples for propositions, which means they lack the ability to give feedback to users in the same way that model checkers can. This work attempts to resolve this issue by investigating how a proof assistant based in dependent type theory (in particular, Lean) can be improved to generate counter-examples.

1.2 Counter-Examples in Proof Assistants

Isabelle is a proof assistant in which statements are written in higher-order logic (HOL) [NWP02]. Nitpick is a counter-example generator for Isabelle that is designed by Blanchette et al, though it can be used to generate counter-examples for any HOL-based proof assistant, not just Isabelle [BN10]. Isabelle is popularly used, and Nitpick has been used by users such as Guttman et al [GSW11]. Nunchaku is an even stronger successor to Nitpick that has also been incorporated into Isabelle already [CB16, Bla17]. Isabelle is actively used in formal verification. For example, its CryptHOL library has been used to prove theorems from multi-party computation [BLS20, BAG20]. As an active tool in Isabelle, Nunchaku is at the forefront of formal verification.

While Nunchaku already works for Isabelle, it cannot handle statements from Lean and Coq yet, because these are based on dependent type theory [Car19, BC13], which allows users to define types that cannot be defined in HOL. A dependent type is a type which changes based on the value of some variable. One example is the type of vectors of size $n$. This is a dependent type because a vector with 3 elements will be a vector of size 3, but it will not be a vector of size 5. In this way, whether an expression has the type vector of size $n$ is dependent on the actual value of $n$. HOL is called a simple type theory, because dependent types cannot be defined in HOL.

While Nunchaku can and does generate counter-examples for statements from HOL, it does not yet have the ability to generate counter-examples for statements from dependent type theory, which means it must be extended before it can be used for Lean or Coq. Given a dependently-typed expression, it is possible to construct an equivalent simply-typed expression, which means that it is possible to translate expressions from Lean or Coq into a form that Nunchaku can already generate counter-examples for.

2 Goal

Since Nunchaku already has a strong understanding of HOL, one way to extend it to work over dependent types would be to describe a way to translate dependently-typed expressions into HOL. Then, Nunchaku’s input language can be extended to accept dependent types, and the translation can be applied as a first step for generating counter-examples.
Cruanes and Blanchette have described a way to start this translation [CB16], but their work only focuses on some examples, and does not contain a fully specify translation. In this work, I define and input language and output language for the translation, and then provide pseudocode for the actual translation. Finally, I briefly discuss steps that must occur while implementing this translation, and mention a couple of applications.

3 Specification

3.1 Input Language (CIC)

In order to define the translation, it is necessary to formalize the input and output languages. The input dependent type theory will be the Calculus of Inductive Constructions, which is used as a backbone for both Lean and Coq [Car19, BC13]. The following is a grammar for CIC, as used to define the type system of Lean [Car19]:

\[
\begin{align*}
\Gamma &::= \cdot \mid \Gamma, x : e \\
\ell &::= u \mid 0 \mid S\ell \mid \text{max}(\ell, \ell) \mid \text{imax}(\ell, \ell) \\
K &::= \cdot \mid (k : e) + K \\
e &::= x \mid U\ell \mid e e \mid \lambda x : e. e \mid \forall x : e. e \\
&\quad | \text{let} x : e := e \text{ in } e \\
&\quad | \mu x : e. K \mid k_{\mu x : e.K} \mid \text{rec}_{\mu x : e.K} \\
&\quad | c_\bar{u}.
\end{align*}
\]

The \(\Gamma\) production rules define a context mapping variables to their types. The \(\ell\) production rules define universe levels. The \(K\) production rules define a list of constructors for an inductive type, and the \(e\) production rules define the expressions in CIC. Since the goal is to translate expressions from CIC into simple type theory, only the expression rules are relevant to this work, though the reader can explore [Car19] for further reading about the other rules.

Among the rules for expressions, \(x\) denotes a variable, \(U\ell\) denotes a type universe, \(e e\) denotes function application, \(\lambda x : e. e\) denotes function definition, and \(\forall x : e. e\) denotes universal quantification, where the type of \(x\) is an expression because this is a dependent type theory. Similarly, the type assigned to a variable in a \text{let}-binding is an expression. The next few rules handle inductive type definitions, where \(\mu x : e. K\) defines a new inductive type with the given list of constructors and \(k_{\mu x : e.K}\) refers to a specific constructor. The recursor \(\text{rec}_{\mu x : e.K}\) for an inductive type is used by Lean to use induction to prove statements about that type. In this work, the recursor can be treated as a constant, but more information about it is available in [Car19]. Finally, \(c_\bar{u}\) is used to denote a constant that lives in universe level \(\bar{u}\).

The rest of this subsection is dedicated to examples where this grammar is used to define inductive types. The natural numbers can be defined as an inductive type:

\[
\mathbb{N} := \mu N : \text{Type}. \ (\text{zero} : N) + (\text{succ} : N \rightarrow N),
\]

where \(\text{Type}\) is the type that all types have (so that \(\mathbb{N} : \text{Type}\)), and \(N \rightarrow N\) is shorthand for \((\forall n : N. N)\). Here, \(\text{zero}\) returns 0 and \(\text{succ}\) is the successor relation, where \((\text{succ } n)\) returns \(n + 1\). All natural numbers can be constructed this way:
and so on. Polymorphic types can be constructed similarly, as functions of types. For example, consider linked lists. The goal is to produce \( \text{List} : \text{Type} \to \text{Type} \) such that for any type \( A \), \( \text{List} \ A \) is the type of linked lists whose elements have type \( A \). There are two constructors for linked lists, \( \text{empty} \) and \( \text{link} \), yielding the definition

\[
\text{List} := \mu L : (\forall A : \text{Type}. \ A) + (\text{link} : A \to L A \to L A).
\]

Then, for example, \( \text{List} \ \mathbb{N} \) is the type of linked lists containing natural numbers, with constructors \( \text{empty} : \text{List} \ \mathbb{N} \) and \( \text{link} : \mathbb{N} \to \text{List} \ \mathbb{N} \to \text{List} \ \mathbb{N} \). Dependent types can be constructed inductively as well. Consider the type of vectors with size \( n \), where \( n \) is a natural number. There are two constructors: \( \text{nil} \), which produces an empty vector (with type \( \text{Vec} \ 0 \)), and \( \text{cons} \), which adds a given element to a vector of size \( n \) (type \( \text{Vec} \ n \)), outputting a vector with size \( n + 1 \) (type \( \text{Vec} \ (\text{succ} \ n) \)). Since vectors are also polymorphic, the data definition for vectors becomes

\[
\text{Vec} := \mu V : (\forall A : \text{Type}. \ \forall n : \mathbb{N}. \ A) + (\text{nil} : V A 0) + (\text{cons} : A \to V A n \to V A (\text{succ} n)).
\]

Then, for example, \( \text{Vec} \ \mathbb{N} \ 3 \) is the type of vectors with size 3 that contain natural numbers.

### 3.2 Output Language (HOL)

The goal is to use Nunchaku to develop counter-examples to propositions written in CIC. Nunchaku is a descendent of Nitpick, a counter-example generator for higher-order logic (HOL) \cite{CB16}. Nitpick and Nunchaku translate from higher-order logic into first-order logic (FORL) \cite{BN10}. Accordingly, the output language of the translation into dependent types should be similar to HOL. That way, the output from the translation can be piped into Nunchaku in order to find a counter-example. Let’s define the \textit{NunchakuInput} language for this purpose, by the following grammar:

\[
\sigma ::= \alpha | \tau | \sigma \to \sigma \\
\ K ::= \_ | k\sigma ; \ K \\
\ t ::= x\sigma | c\sigma | t \ t | \lambda x\sigma. \ t | \text{let} \ x\sigma := t \ \text{in} \ t | \text{data} \ \alpha^{(\alpha,...,\alpha)} : \{K\} \ \text{in} \ t.
\]

This is similar to the HOL grammar presented in \cite{BN10}, but extended to include let bindings and data definitions, which are present in Nunchaku. Since HOL is a simple type theory, types and terms cannot be used interchangably, unlike in CIC. The \( \sigma \) production rules are used to denote types, while the \( t \) production rules are used to denote terms. The \( K \) production rules are used to denote lists of type constructors.
Among types, $\alpha$ refers to type variables, $\tau$ refers to type constants (such as the boolean type), and $\sigma \rightarrow \sigma$ is used for function types. Among terms, $x^\sigma$ represents variables and $c^\sigma$ represents constants, both of which are denoted with their types. Applications are denoted by $(t \; \; t)$, and functions are denoted by $(\lambda x^\sigma. \; \; t)$. While let bindings are used to define new terms, data definitions are used to define new inductive types. Unlike CIC, they must be distinct here, as $\text{NunchakuInput}$ is simply-typed. Booleans $\text{T}$, $\text{F}$, and important logical operators like $\forall$, $\exists$, $\neg$, $\land$, and $\lor$ are defined as constants in HOL (and $\text{NunchakuInput}$), further reading about this is available in [JM93].

Using the examples from the previous subsection, the natural numbers can be defined by

$$\text{data } \text{N} : \{\text{zero}^{\text{N}}; \; \; \text{succ}^{\text{N}\rightarrow\text{N}}\} \; \; \text{in} \; \; \ldots,$$

while linked lists can be defined by

$$\text{data list}^{(A)} : \{\text{empty}^{\text{list} \; \; A}; \; \; \text{link}^{A\rightarrow\text{list} \; \; \rightarrow \; \; \text{list} \; \; A} \} \; \; \text{in} \; \; \ldots,$$

with the superscript is used to bind $A$ for use in the constructors, allowing linked lists to be polymorphic. Note that the superscript $A$ is binding a parameter that only describes the type of the elements in the list. $A$ does not describe the values of these elements, so this is not a dependent type. The type of $\text{vectors with type } n$ cannot directly be defined in $\text{NunchakuInput}$. This type will be used in section 4 as an example of the results of the translation.

4 Translation

A program in CIC really is sequence of definitions. Data types are defined, then functions over them and propositions about them are defined, and proofs are expressions as well (by the Curry-Howard isomorphism), so they are also defined the same way other expressions are. Definitions in CIC are written as let-bindings, so a program in CIC is really one big expression - it is a series of let-bindings. So the goal is to translate expressions from CIC into expressions in $\text{NunchakuInput}$. Recursion will be used to ensure that all sub-expressions are properly encoded, and pattern-matching will be used to handle all cases of expressions. The $\text{encode}$ function will require two arguments:

1. $\text{expr}$, the expression being encoded

2. $\text{depTypeMap}$, a record which maps type variables to a proposition that represents the type dependency. For example, a statement about $\text{vectors of size } n$ can be expressed as a statement about vectors, with an implication that the vector has exactly $n$ elements in it. So, the type of $\text{vectors of size } n$ would have a dependency proposition that the for all $n$, the size of a given vector is equal to $n$.

These arguments yield the type signature

$$\text{encode} : \text{CIC} \rightarrow (\text{CIC} \times \text{CIC}) \rightarrow \text{NunchakuInput},$$

because the expressions are written in CIC, as are the type variables and the propositions about dependent types.
Algorithm 1: Encoding CIC programs into NunchakuInput

function ENCODE(expr, depTypeMap):

match expr with
| f a ↦ let f' := ENCODE(f, depTypeMap) in
  let a' := ENCODE(f, depTypeMap) in
  f' a'
| λx : t. b ↦ let b' := ENCODE(b, depTypeMap) in
  if isDepType(t) then let depprop := depTypeMap(t) in
  λx'. (b' asserting depprop(x))
  else λx'. b'
| ∀x : t. b ↦ let b' := ENCODE(b, depTypeMap) in
  if isDepType(t) then let depprop := depTypeMap(t) in
  ∀x'. (depprop(x) ⇒ b')
  else ∀x'. b'
| let x := (µx' : t'. K) in b ↦
  let K' := removeDeps(K) in
  let b' := ENCODE(b, depTypeMap + (x → getDepProp(K)))) in
  data x' : {K'} in b'
| let x := v in b ↦ let v' := ENCODE(v, depTypeMap) in
  let b' := ENCODE(b, depTypeMap) in
  let x' := v' in b'
| _ ↦ expr ▷ variables, constants have same value after encoding (base case)

In most cases, the encoding can return the same expression as it received as input. For example, a variable \( x \) contains no dependencies, and neither does a constant, so they can just map to copies of themselves. There are a few special cases, where expressions consist of multiple sub-expressions, and all sub-expressions need to be encoded too. This occurs in the function application case. The function itself is encoded, the argument is encoded, and then the results are applied.

Let bindings are especially tricky, because in NunchakuInput, there is a difference between a let-binding and a data definition, but in CIC there is not. These two cases are separated and handled differently. In a let expression, there is a variable \( x \), a type \( t \), a value \( v \), and a body \( b \). If \( v \) is not a type, then \( v \) and \( b \) are encoded and a NunchakuInput let binding using them is produced. If \( v \) is a simple type, then the let binding is encoded into a data definition. If \( v \) is a dependent type, however, more steps are necessary.

The removeDeps helper function takes as input a list of constructors for a dependent type and produces a list of constructors for an independent data type. This type is equivalent to the union of all dependent types defined by the original constructors. The new simple type defined by this is what will be used in the NunchakuInput expressions that result from the encoding process. However, the type dependencies that the original types had must be preserved. These dependencies can be thought of as propositions. The getDepProp function takes the constructors for a dependent type as input, and encodes their type dependencies as a proposition. So finally, in a let binding with variable \( x \), type \( t \), value \( v \), and body
The dependencies are removed from the constructors of \(v\) and added as a proposition to \(depTypeMap\). This new version of \(depTypeMap\) is used to encode \(b\) so that the dependencies still exist there, and the result of encoding the let binding is used to construct a new simply-typed data definition with the encoded body.

Consider again the example of vectors of size \(n\). As mentioned in section 3, this can be defined in CIC by

\[
\text{let } Vec : \text{Type} := \mu V : (\forall A : \text{Type}, \forall n : \mathbb{N}. \text{Type}).
\begin{align*}
\quad (\text{nil} : V A \text{zero}) + \\
\quad (\text{cons} : A \rightarrow V A n \rightarrow V A (\text{succ} n))
\end{align*}
\text{in} \ldots.
\]

The only vectors of size 0 are defined by \(\text{nil}\), and the only vectors of size \(n + 1\) are defined by \(\text{cons}\). The dependency is that a vector must have size exactly \(n\) to be a vector of size \(n\). This dependency is removed by allowing vectors of any size to be produced by either constructor, so this type definition is encoded to become

\[
\text{data } Vec(A) : \{\text{nil}^{\text{Vec}} A ; \text{cons}^{A \rightarrow \text{Vec}} A \rightarrow \text{Vec} A\} \text{ in} \ldots
\]

and all vectors of any size can be defined using these constructors. To make sure the dependency on size of a given vector is still used in future statements, the key-value pair

\[(Vec, \forall n^{\mathbb{N}}. n = \text{size } v),\]

with \(v : Vec\) is added to \(depTypeMap\).

Function definitions are another special case. The body of the function must be encoded, but there is added complexity because the type of the argument may be a dependent type. In NunchakuInput this type will exist without the dependency, so the dependency must be instilled into the body of the new function. Since types in Lean are well-formed, all dependently-typed expressions actually subscribe to the condition they depend on (i.e. every vector with size 5 will actually have exactly 5 elements in it). Thus, Nunchaku only needs to find counter-examples where the dependency does hold. The ASSERTING keyword does exactly this. In Nunchaku, the statement

\[a \text{ ASSERTING } p(a)\]

tells the solver to only search for counter-examples in cases where \(p(a)\) is true [CB16]. So, given a function with a dependently-typed argument, \(depTypeMap\) is used to find the proposition that describes the dependency, and then the encoded version of the function has a body where the dependency proposition is asserted. For example, the function that inserts the number 8 onto a vector of size 5,

\[\lambda v : \text{Vec } \mathbb{N} 5. \text{cons } 8 v\]

is translated into the function

\[\lambda v^{\text{Vec } \mathbb{N}}. ((\text{cons } 8 v) \text{ ASSERTING } (5 = \text{size } v)),\]
where the assertion makes sure that this function is only applied to vectors that do have size 5.

The for-all quantification case is similar to the function case, but it just produces a proposition, meaning it doesn’t need an assertion to make sure it is only considered in certain cases. It can be considered whether the dependency holds or not, as long as the truth value when the dependency doesn’t hold is irrelevant. Implication is used to accomplish this. The body of the for-all statement is encoded, the proposition describing type dependency is found in `depTypeMap`, and the encoded version of the for-all statement has the encoded version of the body, guarded by an implication that the dependency proposition actually holds. For example, the statement

\[
\forall n : \mathbb{N}. \ n > 0 \implies (\forall v : \text{Vec} \mathbb{N} n. \ v \neq \text{nil})
\]

is translated into

\[
\forall n^\mathbb{N}. \ n > 0 \implies (\forall v^\text{Vec}^\mathbb{N}. \ n = \text{size} v \implies v \neq \text{nil}),
\]

with an implication making sure the dependency on the size of \(v\) is still there.

5 Discussion

5.1 Implementation

To benefit from this translation, it is necessary to implement it. There are two steps to that process: extending Nunchaku to include this, and connecting it to proof assistants such as Lean and Coq.

Nunchaku is structured as a pipeline that performs a series of transformations that encode programs written in its input language into formulas that can be passed to the SMT solver it is using [CB16]. Under the hood, there is an interface that can be used to define new transformations, which asks for an encode function and a decode function [CBP23]. The encode function would be exactly what is specified in sections 3 and 4. The decode helps Nunchaku display the output of its solver in a similar format to what the user wrote originally. Specifying a decoding function for these expressions is a potential goal for future work.

Once Nunchaku contains this translation, it can be added to Lean and Coq. Both Lean and Coq rely on proof tactics where, given a goal, the user can apply tactics to transform the goal until it is proven [MU21; HKP97]. Lean provides a metaprogramming framework which can be used to define new tactics [Ebn+17], so one can define a new tactic that takes in a goal and asks Nunchaku to generate counter-examples for it, allowing the user to reference Nunchaku directly while writing a proof.

5.2 Application

Once the translation is accessible from Coq or Lean, it can assist users who are designing verified specifications by writing programs in Lean or Coq that specify a system and then
prove it has the desired properties. Guttman et al note that Nitpick (Nunchaku’s predecessor) was useful for them because it allowed them to check whether properties were desirable before they committed to proving them [GSW11].

When a Lean or Coq user is still designing a system, they don’t know if their desired properties are truly desirable until they’ve checked models of the system. With a counter-example generator, each time a property is violated, the user can investigate the model that is shown as a counter-example and ask if it’s actually problematic or if the property itself wasn’t fully desirable. This allows the user to confirm that their specification is truly what they intended, in addition to formally proving that all desired properties hold.

6 Future Work

The most enticing future work for this project is to go from the specification here to an actual implementation. It would be excellent to add this to Nunchaku’s translation pipeline. This would allow developers to use this tool, and would make way for studies regarding what benefits it provides. One step along that way would be to specify and implement a decoding function, that takes input from HOL and pipes it back into CIC. Nunchaku could use that to better display the counter-examples that are generated, because they would appear in the same language that the user originally wrote them in.

Another interesting direction to take future work is to verify that truth value is preserved across the translation. Counter-examples are generated based on the encoded expressions. If their truth values do not match those of the original expressions, then finding a counter-example does not provide the user any information; it could be an issue with the translation rather than the actual proposition. For this reason, it is desirable to have a proof that an encoded expression is true if and only if the original is true. For maximal rigor, this proof can even be written in a proof assistant such as Lean.

7 Acknowledgements

I wish to extend unending gratitude towards Robert Lewis, my advisor throughout this work. His help and advice are always valuable, and he has been an incredible mentor throughout this process, especially as this project and our expectations appeared to change shape. I also would like to thank my friends and family for their constant support and care.

References


