Extending Nunchaku to Dependent Type Theory

Simon Cruanes
Inria Nancy – Grand Est, France
simon.cruanes@inria.fr

Jasmin Christian Blanchette
Inria Nancy – Grand Est, France
Max-Planck-Institut für Informatik, Saarbrücken, Germany
jasmin.blanchette@inria.fr

Nunchaku is a new higher-order counterexample generator based on a sequence of transformations from polymorphic higher-order logic to first-order logic. Unlike its predecessor Nitpick for Isabelle, it is designed as a stand-alone tool, with frontends for various proof assistants. In this short paper, we present some ideas to extend Nunchaku with partial support for dependent types and type classes, to make frontends for Coq and other systems based on dependent type theory more useful.

1 Introduction

In recent years, we have seen the emergence of "hammers"—integrations of automatic theorem provers in proof assistants, such as Sledgehammer and HOLyHammer [7]. As useful as they might be, these tools are mostly helpless in the face of an invalid conjecture. Novices and experts alike can enter invalid formulas and find themselves wasting hours (or days) on an impossible proof; once they identify and correct the error, the proof is often easy. To discover flaws early, some proof assistants include counterexample generators to debug putative theorems or specific subgoals in an interactive proof. When formalizing algebraic results in Isabelle/HOL, Guttmann et al. [21] remarked that

> Counterexample generators such as Nitpick complement the ATP [automatic theorem proving] systems and allow a proof and refutation game which is useful for developing and debugging formal specifications.

Nunchaku is a new fully automatic counterexample generator for higher-order logic (simple type theory) designed to be integrated into several proof assistants. It supports polymorphism, (co)algebraic datatypes, (co)recursive functions, and (co)inductive predicates. The tool is undergoing considerable development, and we expect that it will soon be sufficiently useful to mostly replace Nitpick [8] for Isabelle/HOL. The source code is freely available online.[1]

A Nunchaku frontend in a proof assistant provides a **nunchaku** command that can be invoked on conjectures to debug them. It collects the relevant definitions and axioms, translates them to higher-order logic along with the negated conjecture, invokes Nunchaku, and translates any model found to higher-order logic. We have developed a frontend for Isabelle/HOL [32]. We are also working on a frontend for the set-theoretic TLA$^+$ Proof System [18] and plan to develop frontends for other proof assistants.

This short paper discusses some of the issues that must be addressed to make frontends for Coq [4] and other systems based on dependent type theory (e.g., Agda, Lean, and Matita) applicable beyond their simple type theory fragment. We plan to elaborate and implement the approach in a Coq frontend, as part of the Inria technological development action "Contre-exemples utilisables par Isabelle et Coq."

[1] https://github.com/nunchaku-inria/nunchaku

Submitted to:
HaTT 2016

2 Overview of Nunchaku

Nunchaku is the spiritual successor to Nitpick but is designed as a stand-alone OCaml program, with its own input language. Whereas Nitpick generates a succession of finite problems for increasing cardinalities, Nunchaku translates its input to one first-order logic program that targets the finite model finding fragment of CVC4 [2], a state-of-the-art SMT (satisfiability modulo theories) solver. Using CVC4 as a backend allows Nunchaku to reason efficiently about arithmetic constraints and (co)algebraic datatypes [36] and to detect unsatisfiability in addition to satisfiability. Support for other backends, including Kodkod [43] (used by Nitpick) and Paradox [16], is in the works. We also plan to integrate backends based on code execution and narrowing, as provided by Quickcheck for Isabelle/HOL [10], to further increase the likelihood of finding counterexamples.

Nunchaku's input syntax is inspired by that of proof assistants based on higher-order logic (e.g., Isabelle/HOL) and by typed functional programming languages (e.g., OCaml). The following problem gives a flavor of the syntax:

$$
\begin{array}{l}
\textbf{data}\ \mathsf{nat} := \mathsf{Zero} \mid \mathsf{Suc}\ \mathsf{nat}. \\[4pt]
\textbf{pred}\ \mathsf{even} : \mathsf{nat} \to \mathsf{prop} := \\
\quad \mathsf{even}\ \mathsf{Zero}; \\
\quad \forall n.\ \mathsf{odd}\ n \Rightarrow \mathsf{even}\ (\mathsf{Suc}\ n) \\
\textbf{and}\ \mathsf{odd} : \mathsf{nat} \to \mathsf{prop} := \\
\quad \forall n.\ \mathsf{even}\ n \Rightarrow \mathsf{odd}\ (\mathsf{Suc}\ n). \\[4pt]
\textbf{val}\ m : \mathsf{nat}. \\[4pt]
\textbf{goal}\ \mathsf{even}\ m \wedge \neg\,(m = \mathsf{Zero}).
\end{array}
$$

The problem defines a datatype (nat) and two mutually recursive inductive predicates (even and odd), it declares a constant *m*, and it specifies a goal to satisfy ("*m* is even and nonzero"). For counterexample generation, the negated conjecture must be specified as the Nunchaku goal. For the example above, Nunchaku outputs the model

$$
\begin{array}{lll}
\textbf{val}\ \mathsf{even} & := & \lambda(n : nat).\ \text{IF}\ n = \mathsf{Zero} \vee n = \mathsf{Suc}\ (\mathsf{Suc}\ \mathsf{Zero})\ \text{THEN}\ \mathsf{true}\ \text{ELSE}\ ?_{\_\_}\ n. \\
\textbf{val}\ \mathsf{odd} & := & \lambda(n : nat).\ \text{IF}\ n = \mathsf{Suc}\ \mathsf{Zero}\ \text{THEN}\ \mathsf{true}\ \text{ELSE}\ ?_{\_\_}\ n. \\
\textbf{val}\ m & := & \mathsf{Suc}\ (\mathsf{Suc}\ \mathsf{Zero}).
\end{array}
$$

The output is a finite fragment of an infinite model. The notation '$?_{\_\_}$' is a placeholder for an unknown value or function. To most users, the interesting part is the interpretation of *m*; but it may help to inspect the partial model of even and odd to check if they have the expected semantics.

Given an input problem, Nunchaku parses it before applying a sequence of translations, each reducing the distance to the target fragment. In our example, the predicates even and odd are translated to recursive functions, then the recursive functions are encoded to allow finite model finding, by limiting their domains to an unspecified finite fragment. If Nunchaku finds a model of the goal, it translates it back to the input language, reversing each phase.

The translation pipeline includes the following phases (adapted from a previous paper [37]):

Type inference infers types and checks definitions;

Type skolemization replaces $\exists\alpha.\ \varphi[\alpha]$ with $\varphi[\tau]$, where τ is a fresh type;

Monomorphization specializes polymorphic definitions on their type arguments and removes unused definitions;

Elimination of equations translates multiple-equation definitions of recursive functions into a single nested pattern matching;

Specialization creates instances of functions with static arguments (i.e., an argument that is passed unchanged to all recursive calls);

Polarization specializes predicates into a version used in positive positions and a version used in negative positions;

Unrolling adds a decreasing argument to possibly ill-founded predicates;

Skolemization introduces Skolem symbols for term variables;

Elimination of (co)inductive predicates recasts a multiple-clause (co)inductive predicate definition into a recursive equation;

λ-Lifting eliminates λ-abstractions by introducing named functions;

Elimination of higher-order constructs substitutes SMT-style arrays for higher-order functions;

Elimination of recursive functions encodes recursive functions to allow finite model finding;

Elimination of pattern matching rewrites pattern-matching expressions using datatype discriminators and selectors;

Elimination of assertions encodes ASSERTING operator using logical connectives;

CVC4 invocation runs CVC4 to obtain a model.

Although our examples use datatypes and well-founded (terminating) recursion, Nunchaku also supports codatatypes and productive corecursion. In addition to finite values, cyclic α-regular codatatype values can arise in models (e.g., the infinite stream $1, 0, 9, 0, 9, 0, 9, \ldots$) [36].

While most of Nunchaku's constructs are fairly conventional, one is idiosyncratic and plays a key role in the translations described here: The ASSERTING operator, written t ASSERTING φ, attaches a formula φ—the *guard*—to a term t. It allows the evaluation of t only if φ is satisfied. The construct is equivalent to IF φ THEN t ELSE UNREACHABLE in other specification languages (e.g., the Haskell Bounded Model Checker [14]). Internally, Nunchaku pulls the ASSERTING guards outside of terms into the surrounding logical context, carefully distinguishing positive and negative contexts.

Nunchaku can only find classical models with functional extensionality, which are a subset of the models of constructive type theory. This means the tool, together with the envisioned encoding, will be sound but incomplete: All counterexamples will be genuine, but no counterexamples will be produced for classical theorems that do not hold intuitionistically. We doubt that this will seriously impair the usefulness of Nunchaku in practice.

3 Encoding Recursive Functions

When using finite model finding to generate counterexamples, a central issue is to translate infinite positive universal quantifiers in a sound way. The situation is hopeless for arbitrary axioms or hypotheses, but infinite quantifiers arising in well-behaved definitions can be encoded soundly. We describe Nunchaku's encoding of recursive functions [37], because it is one of the most crucial phases of the translation pipeline and it illustrates the ASSERTING construct in a comparatively simple setting.

Consider the following factorial example:

rec $\mathsf{fact} : \mathsf{int} \to \mathsf{int} :=$
$\forall n.\ \mathsf{fact}\ n = (\text{IF } n \leq 0 \text{ THEN } 1 \text{ ELSE } n * \mathsf{fact}\ (n-1)).$

val $m : \mathsf{int}.$

goal $\mathsf{fact}\ m > 100.$

(We conveniently assume that Nunchaku has a standard notion of integer arithmetic, as provided by its backend CVC4.) The encoding restricts quantification on fact's domain to an unspecified, but potentially finite, type α_{fact} that is isomorphic to a subset of fact's argument type and introduces projections $\gamma_{\mathsf{fact}} : \alpha_{\mathsf{fact}} \to \mathsf{int}$ and ASSERTING guards throughout the problem, as follows:

val $\mathsf{fact} : \mathsf{int} \to \mathsf{int}.$

axiom $\forall (a : \alpha_{\mathsf{fact}}).\ \mathsf{fact}\ (\gamma_{\mathsf{fact}}\ a) = (\text{IF } \gamma_{\mathsf{fact}}\ a \leq 0 \text{ THEN } 1$
$\text{ELSE } \gamma_{\mathsf{fact}}\ a * (\mathsf{fact}\ (\gamma_{\mathsf{fact}}\ a - 1) \ \text{ ASSERTING } \exists (b : \alpha_{\mathsf{fact}}).\ \gamma_{\mathsf{fact}}\ b = \gamma_{\mathsf{fact}}\ a - 1)).$

val $m : \mathsf{int}.$

goal $(\mathsf{fact}\ m \ \text{ ASSERTING } \exists (b : \alpha_{\mathsf{fact}}).\ \gamma_{\mathsf{fact}}\ b = m) > 100.$

The guards are propagated outward until they reach a polarized context, at which point they can be asserted using standard connectives:

val $\mathsf{fact} : \mathsf{int} \to \mathsf{int}.$

axiom $\forall (a : \alpha_{\mathsf{fact}}).\ \mathsf{fact}\ (\gamma_{\mathsf{fact}}\ a) = (\text{IF } \gamma_{\mathsf{fact}}\ a \leq 0 \text{ THEN } 1 \text{ ELSE } \gamma_{\mathsf{fact}}\ a * \mathsf{fact}\ (\gamma_{\mathsf{fact}}\ a - 1)$
$\wedge \neg\, \gamma_{\mathsf{fact}}\ a \leq 0 \wedge \exists (b : \alpha_{\mathsf{fact}}).\ \gamma_{\mathsf{fact}}\ b = \gamma_{\mathsf{fact}}\ a - 1).$

val $m : \mathsf{int}.$

goal $\mathsf{fact}\ m > 100 \wedge \exists (b : \alpha_{\mathsf{fact}}).\ \gamma_{\mathsf{fact}}\ b = m.$

The guards ensure that the result of recursive function calls is inspected (i.e., influences the truth value of the problem) only if the arguments are in the subset α_{fact}, for which the function is axiomatized.

4 Encoding Dependent Datatypes

We propose an extension to Nunchaku's type system with a simple flavor of dependent types. We assume a finite hierarchy of sorts. A Coq frontend would need to truncate the problem's hierarchy of universes. Our encoding is similar to the one proposed by Jacobs and Melham [24]. We, too, erase dependent parameters from types and use additional predicates to enforce constraints that would be lost otherwise—with the addition of dependent (co)datatypes. In (co)datatypes, we allow term parameters (such as the length of a list, of type nat) to occur as uniform parameters or as indices (i.e., each constructor can have a different value for this parameter), but type parameters should occur uniformly. We only forbid polymorphic recursion (type indices), because it is not compatible with the monomorphization step Nunchaku currently relies on.

In general, we consider dependent (co)datatype definitions of the form

$$
\begin{aligned}
&(\textbf{co})\textbf{data}\ \tau\ \overline{x}\ \overline{\alpha} := \\
&\quad c_1 : \overline{\sigma^1} \to \tau\ \overline{t^1}\ \overline{\alpha} \\
&\quad \vdots \\
&\mid\ c_k : \overline{\sigma^k} \to \tau\ \overline{t^k}\ \overline{\alpha}.
\end{aligned}
$$

where $\overline{x} := (x_i)_{i=1}^{m}$ is the tuple of term variables on which τ depends, $\overline{\alpha} := (\alpha_i)_{i=1}^{n}$ is the tuple of type variables, the types $(\sigma_i^k)_{i=1}^{\mathrm{arity}(c_k)}$ are the types of the arguments of the kth constructor, and the terms

$\overline{t^k} := (t^k_i)^m_{i=1}$ are the term arguments of the kth constructor's return type. More elaborate definitions, such as those interleaving type and term parameters in more intricate ways, are beyond the scope of this approach. We are aiming for a practical balance between expressiveness and ease of implementation.

Let $\tau'\ \overline{\alpha}$ be the encoding of τ where all term arguments have been removed. We introduce a predicate inv_τ, defined inductively (if τ is a datatype) or coinductively (if τ is a codatatype), that enforces the correspondence between $\overline{x}$ and $\tau'\ \overline{\alpha}$:

$$(\mathbf{co})\mathbf{pred}\ \mathsf{inv}_\tau : \Pi\overline{\alpha}.\ \overline{\alpha} \to \tau'\ \overline{\alpha} \to \mathsf{prop} :=$$
$$\bigwedge_{i=1}^{k} \left[\begin{array}{l} \forall \overline{x}\ (y_1 : a^i_1)\ \ldots\ (y_k : a^k_{\mathrm{arity}(c_k)}). \\ \quad \left(\bigwedge_{j=1, y^k_j : \tau}^{\mathrm{arity}(c_k)} \mathsf{inv}_\tau\ \overline{\alpha}\ y^k_j \right) \Rightarrow \mathsf{inv}_\tau\ \overline{\alpha}\ \ (c_k\ \overline{\alpha}\ \overline{y}) \end{array} \right].$$

The predicate inv_τ has one clause per constructor c_k of τ, which ensures that if the invariant holds for every argument $(y_j)^{\mathrm{arity}(c_k)}_{j=1}$ of c_k that has type τ (a recursive instance of τ), it also holds for $c_k\ \overline{\alpha}\ \overline{y}$.

When encoding terms, we process binders on dependently-typed variables recursively as follows: $\forall v : \tau\ \bar{t}\ \bar{u}.\ \varphi$ becomes $\forall v : \tau'\ \bar{u}.\ \mathsf{inv}_\tau\ \bar{t}\ v \Rightarrow \varphi$, and a function $\lambda(x : \tau\ \bar{t}\ \bar{u}).\ v$ is translated to $\lambda(x : \tau'\ \bar{u}).\ (v\ \text{ASSERTING}\ \mathsf{inv}_\tau\ \bar{t}\ x)$.

Functions whose type depends on terms remain parameterized by these terms after the translation, but their definition specifies a precondition that links the term parameters to the encoded dependent type. The use of ASSERTING to encode the precondition ensures that the function is evaluated only if the condition is met, irrespective of the context (positive, negative, or unpolarized) of the function. Finally, some specific constructs such as equality (in Coq, equality is a dependent datatype) are translated directly into Nunchaku counterparts.

As an example, consider the type of vectors of length n. Here, n is an index, and α is a uniform type parameter:

$$\begin{array}{l} \mathbf{data}\ \mathsf{vec} : \mathsf{nat} \to \mathsf{type} \to \mathsf{type} := \\ \quad \mathsf{nil}\ \alpha : \mathsf{vec}\ 0\ \alpha \\ \mid \forall (n : \mathsf{nat})\ (x : \alpha)\ (l : \mathsf{vec}\ n\ \alpha).\ \mathsf{cons}\ \alpha\ x\ l : \mathsf{vec}\ \ (\mathsf{S}\ n)\ \ \alpha. \end{array}$$

The encoded type vec' corresponds to the datatype of finite lists, and the invariant is

$$\begin{array}{l} \mathbf{pred}\ \mathsf{inv}_{\mathsf{vec}} : \mathsf{nat} \to \mathsf{vec}'\ \alpha \to \mathsf{prop} := \\ \quad \mathsf{inv}_{\mathsf{vec}}\ 0\ (\mathsf{nil}\ \alpha) \\ \mid \forall (n : \mathsf{nat})\ (x : \alpha)\ (l : \mathsf{vec}'\ \alpha).\ \mathsf{inv}_{\mathsf{vec}}\ n\ l \Rightarrow \mathsf{inv}_{\mathsf{vec}}\ \ (\mathsf{S}\ n)\ \ (\mathsf{cons}\ \alpha\ x\ l). \end{array}$$

A formula $\forall (v : \mathsf{vec}\ n\ \tau).\ \varphi$ is translated to $\forall (v : \mathsf{vec}'\ \tau).\ \mathsf{inv}_{\mathsf{vec}}\ n\ v \Rightarrow \varphi$. A function $\lambda (v : \mathsf{vec}\ n\ \tau).\ t$ is translated to $\lambda (v : \mathsf{vec}'\ \tau).\ (t\ \text{ASSERTING}\ \mathsf{inv}_{\mathsf{vec}}\ n\ v)$.

Thus, the function returning the length of a vector, $\lambda n\ (l : \mathsf{vec}\ n\ \alpha).\ n$, becomes

$$\lambda n\ (l : \mathsf{vec}'\ \alpha).\ (n\ \text{ASSERTING}\ \mathsf{inv}_{\mathsf{vec}}\ n)$$

The append function $\lambda m\, n\ (l_1 : \mathsf{vec}\ m\ \alpha)\ (l_2 : \mathsf{vec}\ n\ \alpha).\ t$ (omitting the body) becomes

$$\lambda m\, n\ (l_1 : \mathsf{vec}'\ \alpha)\ (l_2 : \mathsf{vec}'\ \alpha).\ (t\ \text{ASSERTING}\ \mathsf{inv}_{\mathsf{vec}}\ m\ l_1 \wedge \mathsf{inv}_{\mathsf{vec}}\ n\ l_2)$$

And the mult function that multiplies two matrices, $\lambda m\, n\, k\ (A : \mathsf{matrix}\ m\ n)\ (B : \mathsf{matrix}\ n\ k).\ t$, returning a value of type $\mathsf{matrix}\ m\ k$, becomes

$$\lambda m\ n\ k\ (A : \mathsf{matrix}')\ (B : \mathsf{matrix}').\ (t\ \text{ASSERTING}\ \mathsf{inv}_{\mathsf{matrix}}\ m\ n\ A \wedge \mathsf{inv}_{\mathsf{matrix}}\ n\ k\ B)$$

5 Encoding Dependent Records and Type Classes

Type classes are a powerful tool for abstraction in Coq, Isabelle/HOL, and other proof assistants [41,45]. However, in dependently typed proofs assistants such as Coq, they are usually encoded as dependent records combining types, values, and proofs. We assume that type classes have been explicitly resolved by the frontend's type inference and focus on their representation as a record of values and propositions. Consider the following example from basic algebra:

$$\begin{array}{l} \textbf{class}\ \mathsf{monoid}\ a\ \textbf{where} \\ \quad \mathsf{e} : a \\ \quad \mathsf{op} : a \to a \to a \\ \quad \mathsf{left\_neutral} : \forall x.\ \mathsf{op}\ \mathsf{e}\ x = x \\ \quad \mathsf{assoc} : \forall x\,y\,z.\ \mathsf{op}\ (\mathsf{op}\ x\ y)\ z = \mathsf{op}\ x\ (\mathsf{op}\ y\ z). \end{array}$$

This definition of monoids can be encoded in a straightforward way as a dependent record—that is, a datatype with a single four-argument constructor. The encoding from Section 4 could then be applied. Here, we propose a more specific encoding that avoids introducing an inductive predicate $\mathsf{inv}_{\mathsf{monoid}}$. This transformation does not use dependent types, and its result still contains the required invariants of each type class, thereby requiring models to satisfy them.

Following our proposed scheme, a type class is translated into a nondependent datatype with one constructor whose arguments are the data fields (e.g., e and op for monoid). The proofs of the axioms can be erased, since they serve no purpose for model finding, and the additional properties $\mathsf{left\_neutral}$ and assoc are directly inserted at appropriate places in the problem.

The definition of monoid is translated to

$$\mathsf{inst}_{\mathsf{monoid}} : \Pi a.\ a \to (a \to a \to a) \to \mathsf{monoid}\ a.$$

$$\begin{array}{l} \textbf{pred}\ \mathsf{left\_neutral}_{\mathsf{monoid}} : \Pi a.\ \mathsf{monoid}\ a \to \mathsf{prop} := \\ \quad \forall e\ op.\ (\forall x.\ op\ e\ x = x) \Rightarrow \mathsf{left\_neutral}_{\mathsf{monoid}}\ a\ (\mathsf{inst}_{\mathsf{monoid}}\ a\ e\ op). \end{array}$$

$$\begin{array}{l} \textbf{pred}\ \mathsf{assoc}_{\mathsf{monoid}} : \Pi a.\ \mathsf{monoid}\ a \to \mathsf{prop} := \\ \quad \forall e\ op.\ (\forall x\,y\,z.\ op\ (op\ x\ y)\ z = op\ x\ (op\ y\ z)) \Rightarrow \mathsf{assoc}_{\mathsf{monoid}}\ a\ (\mathsf{inst}_{\mathsf{monoid}}\ a\ e\ op). \end{array}$$

A function definition

$$\begin{array}{l} \textbf{rec}\ \mathsf{f} : \Pi a.\ \mathsf{monoid}\ a \Rightarrow a \to \tau := \\ \quad \forall (x : a).\ \mathsf{f}\ x = t. \end{array}$$

is translated to

$$\begin{array}{l} \textbf{rec}\ \mathsf{f} : \Pi a.\ \mathsf{monoid}\ a \to a \to \tau := \\ \quad \forall a\,x.\ \mathsf{f}\ a\ x = (t\ \textsc{asserting}\ \mathsf{left\_neutral}_{\mathsf{monoid}}\ a \wedge \mathsf{assoc}_{\mathsf{monoid}}\ a). \end{array}$$

In a proof assistant, users must explicitly register types as instances of type classes. For example, registering nat as a monoid instance might involve some syntax such as

$$\begin{array}{l} \textbf{instance}\ \mathsf{monoid}\ \mathsf{nat}\ \textbf{where} \\ \quad \mathsf{e} = 0 \\ \quad \mathsf{op} = (+) \\ \quad \mathsf{left\_neutral} = \langle \text{proof of } \mathsf{left\_neutral} \rangle \\ \quad \mathsf{assoc} = \langle \text{proof of } \mathsf{assoc} \rangle. \end{array}$$

These would not have to be specified to Nunchaku; in a semantic setting, any type that satisfies the type class axioms would be considered a member of the type class. (For essentially the same reason, only definitions and axioms need to be specified in Nunchaku problems, and not derived lemmas.) Nonetheless, it might be more efficient to provide the instantiations to Nunchaku, so that it can eliminate true conditions such as $\mathsf{left\_neutral}_{\mathsf{monoid}}\ \mathsf{nat} \wedge \mathsf{assoc}_{\mathsf{monoid}}\ \mathsf{nat}$ that can arise as a result of its monomorphization phase.

6 Related Work

There are many competing approaches to refuting logical formulas. The main ones are *finite model finding* and *code execution*. Alternatives include infinite model generation [11], counterexample-producing decision procedures [13], model checking [17], and saturations [1].

Finite model finding consists of enumerating all potential finite models, starting with a cardinality of one for the domains. Some model finders explore the search space directly; FINDER [40], SEM [46], Alloy's precursor [22], and Mace versions 3 and 4 [30] are of this type. Other tools reduce the problem to propositional satisfiability and invoke a SAT solver; these include early versions of Mace (or MACE) [31], Paradox [16], Kodkod [43] and its frontend Alloy [23], and FM-Darwin [3]. Finally, some theorem provers implement finite model finding on top of their proof calculus; this is the case for KIV [35], iProver [25], and CVC4 [38]. To make finite model finding more useful, techniques have been developed to search for partial fragments of infinite models [6, 19, 26, 37, 42].

The idea with code execution is to generate test inputs and evaluate the goal, seen as a functional program. For Haskell, QuickCheck [15] generates random inputs, SmallCheck [39] systematically enumerates inputs starting with small ones, and Lazy SmallCheck [39] relies on narrowing to avoid evaluating irrelevant subterms. A promising combination of bounded model checking and narrowing is implemented in HBMC, the Haskell Bounded Model Checker [14].

In proof assistants, Refute [44] and Nitpick [8] for Isabelle/HOL are based on finite model finding. QuickCheck-like systems have been developed for Agda [20], Isabelle/HOL [10], PVS [33], FoCaLiZe [12], and now Coq with QuickChick [34]. Agsy for Agda [27] employs narrowing. Isabelle's Quickcheck combines random testing, bounded exhaustive testing, and narrowing in one tool [10]. Finally, ACL2 [29] combines random testing and theorem proving.

Our experience with Isabelle is that Nitpick and Quickcheck have complementary strengths and weaknesses [5, Section 3.6] and that it would be a mistake to rely on a single strategy. For example, debugging the axiomatic specification of the C++ memory model [9] was a heavy combinatorial task where Nitpick's SAT solving excelled, whereas for the formalization of a Java-like language [28] it made more sense to develop an executable specification and invoke Quickcheck. Nunchaku currently stands firmly in the finite model finding world, but we plan to develop an alternative translation pipeline to generate Haskell code and invoke QuickCheck, SmallCheck, Lazy SmallCheck, and HBMC.

7 Conclusion

Nunchaku supports polymorphic higher-order logic by a series of transformations that yield a first-order problem suitable for finite model finding. This paper introduced further transformations that extend the translation pipeline to support dependent types and type classes as found in Coq and similar systems. More work is necessary to fully specify these transformations, prove them correct, and implement them. We plan an integration in Coq but will happily collaborate with the developers of other systems to build further frontends; in particular, we are already in contact with the developers of Lean, a promising new proof assistant based on type theory.

We generally contend that too much work has gone into engineering the individual proof assistants, and too little into developing compositional methods and tools with a broad applicability across systems. Nunchaku is our attempt at changing this state of affairs for counterexample generation.

Acknowledgment We are grateful to the anonymous reviewers for making many useful comments and suggestions and for pointing to related work. We also thank Mark Summerfield, who suggested many textual improvements. Cruanes is supported by the Inria technological development action "Contre-exemples utilisables par Isabelle et Coq" (CUIC). Nunchaku would not exist today had it not been for the foresight and support of Stephan Merz, Andrew Reynolds, and Cesare Tinelli.

References

[1] Leo Bachmair & Harald Ganzinger (2001): *Resolution theorem proving*. In Alan Robinson & Andrei Voronkov, editors: *Handbook of Automated Reasoning*, I, Elsevier, pp. 19–99.

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds & Cesare Tinelli (2011): *CVC4*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *CAV 2011*, *LNCS* 6806, Springer, pp. 171–177, doi:10.1007/978-3-642-22110-1_14.

[3] Peter Baumgartner, Alexander Fuchs, Hans de Nivelle & Cesare Tinelli (2009): *Computing finite models by reduction to function-free clause logic*. *J. Applied Logic* 7(1), pp. 58–74, doi:10.1016/j.jal.2007.07.005.

[4] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer.

[5] Jasmin Christian Blanchette (2012): *Automatic Proofs and Refutations for Higher-Order Logic*. Ph.D. thesis, Technische Universität München.

[6] Jasmin Christian Blanchette (2013): *Relational analysis of (co)inductive predicates, (co)inductive datatypes, and (co)recursive functions*. *Softw. Qual. J.* 21(1), pp. 101–126, doi:10.1007/s11219-011-9148-5.

[7] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C. Paulson & Josef Urban (2016): *Hammering towards QED*. *J. Formal. Reasoning* 9(1), pp. 101–148, doi:10.6092/issn.1972-5787/4593.

[8] Jasmin Christian Blanchette & Tobias Nipkow (2010): *Nitpick: A counterexample generator for higher-order logic based on a relational model finder*. In Matt Kaufmann & Lawrence C. Paulson, editors: *ITP 2010*, *LNCS* 6172, Springer, pp. 131–146, doi:10.1007/978-3-642-14052-5_11.

[9] Jasmin Christian Blanchette, Tjark Weber, Mark Batty, Scott Owens & Susmit Sarkar (2011): *Nitpicking C++ concurrency*. In: *PPDP 2011*, ACM, pp. 113–124, doi:10.1145/2003476.2003493.

[10] Lukas Bulwahn (2012): *The new Quickcheck for Isabelle: Random, exhaustive and symbolic testing under one roof*. In Chris Hawblitzel & Dale Miller, editors: *CPP 2012*, *LNCS* 7679, Springer, pp. 92–108, doi:10.1007/978-3-642-35308-6_10.

[11] Ricardo Caferra, Alexander Leitsch & Nicolas Peltier (2004): *Automated Model Building*. *Applied Logic* 31, Springer.

[12] Matthieu Carlier, Catherine Dubois & Arnaud Gotlieb (2012): *A first step in the design of a formally verified constraint-based testing tool: FocalTest*. In Achim D. Brucker & Jacques Julliand, editors: *TAP 2012*, *LNCS* 7305, Springer, pp. 35–50, doi:10.1007/978-3-642-30473-6_5.

[13] Amine Chaieb & Tobias Nipkow (2008): *Proof synthesis and reflection for linear arithmetic*. *J. Autom. Reasoning* 41(1), pp. 33–59, doi:10.1007/s10817-008-9101-x.

[14] Koen Claessen (2016): Private communication.

[15] Koen Claessen & John Hughes (2000): *QuickCheck: A lightweight tool for random testing of Haskell programs*. In: *ICFP '00*, ACM, pp. 268–279, doi:10.1145/357766.351266.

[16] Koen Claessen & Niklas Sörensson (2003): *New techniques that improve MACE-style model finding*. In: *MODEL*.

[17] Edmund M. Clarke, Jr., Orna Grumberg & Doron A. Peled (1999): *Model Checking*. MIT Press.

[18] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts & Hernán Vanzetto (2012): *TLA$^+$ proofs*. In Dimitra Giannakopoulou & Dominique Méry, editors: *FM 2012*, *LNCS* 7436, Springer, pp. 147–154, doi:10.1007/978-3-642-32759-9_14.

[19] Andriy Dunets, Gerhard Schellhorn & Wolfgang Reif (2010): *Automated flaw detection in algebraic specifications*. *J. Autom. Reasoning* 45(4), pp. 359–395, doi:10.1007/s10817-010-9166-1.

[20] Peter Dybjer, Qiao Haiyan & Makoto Takeyama (2003): *Combining testing and proving in dependent type theory*. In David A. Basin & Burkhart Wolff, editors: *TPHOLs 2003*, *LNCS* 2758, Springer, pp. 188–203, doi:10.1007/10930755_12.

[21] Walter Guttmann, Georg Struth & Tjark Weber (2011): *Automating algebraic methods in Isabelle*. In Shengchao Qin & Zongyan Qiu, editors: *ICFEM 2011*, *LNCS* 6991, Springer, pp. 617–632, doi:10.1007/978-3-642-24559-6_41.

[22] Daniel Jackson (1996): *Nitpick: A checkable specification language*. In: *FMSP '96*, pp. 60–69.

[23] Daniel Jackson (2006): *Software Abstractions: Logic, Language, and Analysis*. MIT Press.

[24] Bart Jacobs & Tom Melham (1993): *Translating dependent type theory into higher order logic*. In M. Bezem & J.F. Groote, editors: *TLCA 1993*, *LNCS* 664, Springer, pp. 209–229, doi:10.1007/BFb0037108.

[25] Konstantin Korovin (2013): *Non-cyclic sorts for first-order satisfiability*. In Pascal Fontaine, Christophe Ringeissen & Renate A. Schmidt, editors: *FroCoS 2013*, *LNCS* 8152, Springer, pp. 214–228, doi:10.1007/978-3-642-40885-4_15.

[26] Viktor Kuncak & Daniel Jackson (2005): *Relational analysis of algebraic datatypes*. In Michel Wermelinger & Harald Gall, editors: *ESEC/FSE 2005*, ACM, pp. 207–216, doi:10.1145/1081706.1081740.

[27] Fredrik Lindblad (2007): *Property directed generation of first-order test data*. In Marco Morazán, editor: *TFP 2007*, Intellect, pp. 105–123.

[28] Andreas Lochbihler & Lukas Bulwahn (2011): *Animating the formalised semantics of a Java-like language*. In Marko van Eekelen, Herman Geuvers, Julien Schmalz & Freek Wiedijk, editors: *ITP 2011*, *LNCS* 6898, Springer, pp. 216–232, doi:10.1007/978-3-642-22863-6_17.

[29] Panagiotis Manolios (2013): *Counterexample generation meets interactive theorem proving: Current results and future opportunities*. In Sandrine Blazy, Christine Paulin-Mohring & David Pichardie, editors: *ITP 2013*, *LNCS* 7998, Springer, p. 18, doi:10.1007/978-3-642-39634-2_4.

[30] William McCune: *Prover9 and Mace4*. `http://www.cs.unm.edu/~mccune/prover9/`.

[31] William McCune (1994): *A Davis–Putnam program and its application to finite first-order model search: quasigroup existence problems*. Technical Report, Argonne National Laboratory.

[32] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. *LNCS* 2283, Springer.

[33] Sam Owre (2006): *Random testing in PVS*. In: *AFM '06*.

[34] Zoe Paraskevopoulou, Catalin Hritcu, Maxime Dénès, Leonidas Lampropoulos & Benjamin C. Pierce (2015): *Foundational property-based testing*. In Christian Urban & Xingyuan Zhang, editors: *ITP 2015*, *LNCS* 9236, Springer, pp. 325–343, doi:10.1007/978-3-319-22102-1_22.

[35] Wolfgang Reif, Gerhard Schellhorn & Andreas Thums (2001): *Flaw detection in formal specifications*. In Rajeev Goré, Alexander Leitsch & Tobias Nipkow, editors: *IJCAR 2001*, *LNCS* 2083, Springer, pp. 642–657, doi:10.1007/3-540-45744-5_52.

[36] Andrew Reynolds & Jasmin Christian Blanchette (2015): *A decision procedure for (co)datatypes in SMT solvers*. In Amy Felty & Aart Middeldorp, editors: *CADE-25*, *LNCS* 9195, Springer, pp. 197–213, doi:10.1007/978-3-319-21401-6_13.

[37] Andrew Reynolds, Jasmin Christian Blanchette, Simon Cruanes & Cesare Tinelli (2016): *Model finding for recursive functions in SMT*. In N. Olivetti & A. Tiwari, editors: *IJCAR 2016*, *LNCS* 9706, Springer.

[38] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstić, Morgan Deters & Clark Barrett (2013): *Quantifier instantiation techniques for finite model finding in SMT*. In Maria Paola Bonacina, editor: *CADE-24*, *LNCS* 7898, Springer, pp. 377–391, doi:10.1007/978-3-642-38574-2_26.

[39] Colin Runciman, Matthew Naylor & Fredrik Lindblad (2008): *SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values*. In Andy Gill, editor: *Haskell 2008*, ACM, pp. 37–48, doi:10.1145/1411286.1411292.

[40] John K. Slaney (1994): *FINDER: Finite domain enumerator—System description*. In Alan Bundy, editor: *CADE-12*, *LNCS* 814, Springer, pp. 798–801, doi:10.1007/3-540-58156-1_63.

[41] Matthieu Sozeau & Nicolas Oury (2008): *First-class type classes*. In Otmane Ait Mohamed, César Muñoz & Sofiène Tahar, editors: *TPHOLs 2008*, *LNCS* 5170, Springer, pp. 278–293, doi:10.1007/978-3-540-71067-7_23.

[42] Philippe Suter, Ali Sinan Köksal & Viktor Kuncak (2011): *Satisfiability modulo recursive programs*. In Eran Yahav, editor: *SAS 2011*, *LNCS* 6887, Springer, pp. 298–315, doi:10.1007/978-3-642-23702-7_23.

[43] Emina Torlak & Daniel Jackson (2007): *Kodkod: A relational model finder*. In Orna Grumberg & Michael Huth, editors: *TACAS 2007*, *LNCS* 4424, Springer, pp. 632–647, doi:10.1007/978-3-540-71209-1_49.

[44] Tjark Weber (2008): *SAT-Based Finite Model Generation for Higher-Order Logic*. Ph.D. thesis, Technische Universität München.

[45] Markus Wenzel (1997): *Type classes and overloading in higher-order logic*. In Elsa L. Gunter & Amy Felty, editors: *TPHOLs 1997*, *LNCS* 1275, Springer, pp. 307–322, doi:10.1007/BFb0028402.

[46] Jian Zhang & Hantao Zhang (1995): *SEM: A system for enumerating models*. In Chris S. Mellish, editor: *IJCAI-95*, 1, Morgan Kaufmann, pp. 298–303.