Run your Research, Mind the Binders

James Cheney
University of Edinburgh
jcheney@inf.ed.ac.uk

Alberto Momigliano
Università degli Studi di Milano
momigliano@di.unimi.it

Matteo Pessina
Università degli Studi di Milano
matteo.pessina3@studenti.di.unimi.it

αCheck is a light-weight property-based testing tool built on top of αProlog. Being based on nominal logic programming, it is particularly suited to the validation of the meta-theory of formal systems. To substantiate this claim we compare its performances w.r.t. its main competitors in the logical framework niche, namely the QuickCheck/Nitpick combination offered by Isabelle/HOL and the random testing facility in PLT-Redex. We briefly sketch the architecture of αCheck and mention some future directions for the tool and for benchmarking mechanized meta-theory model-checking systems more generally.

1 Introduction

Notwithstanding recent progress, formal verification of the meta-theory of formal systems *via* proof assistants is still hard work — especially if either the specification or the intended theorem is wrong. While a failed proof attempt can give revealing insights about what went wrong in the case of deep results, it is almost always not the best way to debug a specification or a theorem, when they are "morally right", in the sense that minor modifications will make the proof go through. For this class of *shallow* bugs, a tool that automatically finds counterexamples can be surprisingly effective [5, 11] and can even complement formal proof attempts by warning when the property being proved is false [2]. The beauty of such *meta-theory model checking* is that, compared to other general forms of system validation, the properties that should hold are already given by means of the theorems that the calculus under study is supposed to satisfy. Of course, those need to be fine tuned for testing to be effective, but we are mostly free of the thorny issue of specification/invariant generation.

In fact, such tools are now gaining some traction in the field as we discuss in Section 4, see in particular the QuickCheck/Nitpick combination offered in Isabelle/HOL [2] and random testing in PLT-Redex [11], the latter being a particularly persuasive demonstration of the effectiveness of such an approach.[1] However, none of them has any direct support for binding syntax. To quote from [11]:

> "Redex offers little support for handling binding constructs in object languages. It provides a generic function for obtaining a fresh variable, but no help in defining capture-avoiding substitution or α-equivalence ... In one case ... managing binders constitutes a significant portion of the overall time spent ..."

This lack of support can make testing either ineffective (Isabelle/HOL) or requiring an additional amount of coding, which may need to be duplicated in every case study (PLT-Redex, in particular the fine tuning of random generators):

> "Generators derived from grammars ... require substantial massaging to achieve high test coverage. This deficiency is particularly pressing in the case of typed object languages, where the massaging code almost duplicates the specification of the type system". (ibid.)

[1] Exhaustive counterexample search over finite spaces in Bedwyr [1] is somewhat different, as we touch upon next.

We were among the first to propose automated counterexample search for language specifications [5], including a prototype based on the αProlog nominal logic programming language [6]. In contrast to QuickCheck/Nitpick and PLT Redex, our approach supports binding syntax directly and uses logic programming to perform exhaustive symbolic search for counterexamples. To date, there has been no empirical comparison of our approach with these alternatives, due in part to the absence of an easy-to-use implementation of our approach. This paper reports on work in progress on a re-implementation of our approach called αCheck, a property-based testing tool for mechanized meta-theory model-checking. We first give a short tour of αCheck and review related systems and our approach, and then report on case studies comparing αCheck with our main competitors. We conclude with a discussion of future steps for αCheck and for benchmarking mechanized meta-theory model-checking systems more generally.

2 A Brief Tour of the Checker

We use a simply-typed λ-calculus augmented with constructors for integers and lists, following the PLT-Redex benchmark `sltk.lists.rkt` from [9], which we will also use in Section 5.3. Note the presence of an (uncaught) *error* expression, to model run time errors such taking the head of an empty list. The language is formally declared as follows:

Types	A, B	::=	$int \mid ilist \mid A \to B$
Terms	M	::=	$x \mid \lambda x{:}A.\ M \mid M_1\ M_2 \mid c \mid error$
Constants	c	::=	$n \mid nil \mid cons \mid hd \mid tl$
Values	V	::=	$c \mid \lambda x{:}A.\ M \mid cons\ V_1\ V_2$

We start by declare in our tool the syntax of terms, constant and types, while values will be carved out *via* an appropriate predicate. A similar predicate `is_err` characterizes the threading of the *error* expression in the operational semantics, following standard practice. We assume some familiarity with nominal logic and just mention that `lam(x\var(x),intTy)` is concrete syntax for $\lambda x{:}int.\ x$.

```
ty: type.          id: name_type.  exp: type.              cst: type.
intTy: ty.         listTy: ty.     funTy: (ty,ty) -> ty.
var: id -> exp.    c: cst -> exp.  app: (exp,exp) -> exp. lam: (id\exp,ty) -> exp. error:exp
cons: cst.         hd: cst.        tl: cst.               nil: cst.
toInt: int -> cst.
```

We follow this up with the static and dynamic semantics, where we omit the judgments for `value` and `substitution`, which are analogous to the ones in [5]. Note that *error* has any type and constants are typed *via* a table `tcc`, omitted. The freshness predicate $t \# u$ indicates that name t does not appear free in term u; in particular, if u is also a name then freshness is name-inequality.

```
type ctx = [(id,ty)].

pred tc (ctx,exp,ty).
tc(_,error,T).
tc(_,c(C),T)                              :- tcc(C,T).
tc([(X,T)|G],var X,T).
tc([(Y,_)|G],var X,T)                     :- X # Y, tc(G,var(X),T).
tc(G,app(M,N),U)                          :- tc(G,M,funTy(T,U)), tc(G,N,T).
tc(G,lam(x\M),funTy(T,U))                 :- x # G, tc([(x, T) |G],M,U).

pred step(exp,exp).
step(app(c(hd),app(app(c(cons),X),_)),X).
```

```
step(app(c(tl),app(app(c(cons),_),XS)),XS).
step(app(lam(x\M),N),P)                    :- value N, substp(M,x,N,P).
step(app(M1,M2),app(M1',M2))               :- step(M1,M1').
step(app(M1,M2),app(M1,M2'))               :- value(M1),step(M2,M2').

pred is_err(exp).
is_err(error).
is_err(app(c(hd),c(nil))).
is_err(app(c(tl),c(nil))).
is_err(app(E1,E2))                         :- is_err(E1).
is_err(app(V1,E2))                         :- value(V1),is_err(E2).
```

The idea is to test some properties, trying to falsify them up to a certain bound. Following the PLT-Redex development, we will concentrate here only on checking that the following preservation and progress properties hold. We remark, as we have shown in [5], that it is often worthwhile to check also auxiliary lemmas, e.g. that substitution is functional or that weakening holds for the typing judgments, since this can help uncover bugs that may not show up if only the main statement is tested. So, we add in a separate file these *checking directives*, where the logical variables are universally quantified:

```
#check "preserv" 7 : tc([],E,T), step(E,E')  => tc([],E',T).
#check "progress" 7: tc([],E,T) => progress(E).
```

Here, '7' is the resource bound (i.e. maximum proof depth) for exploring the search-space in this case; `progress` is a predicate encoding the propery of "being either a value, an error, or able to make a step". The tool will not find any counterexample, because, well, those properties are actually true of the given setup. Now, following the Redex manual [9], let us insert a typo that swaps the range and domain types of the function in the application rule, which now reads:

```
tc(G,app(M,N),U) :- exists T. tc(G,M,funTy(T,U)), tc(G,N,U). % was funTy(U,T)
```

We ask ourselves: what properties become false? Which remain true? The checker returns immediately with this counterexample to progress:

```
E = app(c(hd),c(toInt(N)))
T = intTy
```

This is concrete syntax for $hd(n)$, an expression erroneously well-typed and obviously stuck (where n is any number). Preservation meets a similar fate:

```
M = app(lam(x\app(var(x),error),funTy(T,intTy)),c(toInt(N)))
M' = app(c(toInt(N)),error)
T = intTy
```

3 System Architecture

To provide context for the case studies discussed in the rest of the paper, we review the αCheck system architecture. Given a pure αProlog program specifying a formal system, we consider properties of the form $\forall \vec{X}.\ \mathit{N}\vec{a}.\ H_1 \wedge \cdots \wedge H_n \supset A$, where $\vec{X}$ and $\vec{a}$ include all of the free variables and names of the hypotheses H_i and test formula A. To find a counterexample means (logically) to solve the goal $\exists \vec{X}.\mathit{N}\vec{a}.H_1 \wedge \cdots \wedge H_n \wedge \neg A$; a *counterexample* is a substitution θ such that $\theta(H_1), \ldots, \theta(H_n)$ all hold but the conclusion $\theta(A)$ does not. (Recall that the Gabbay-Pitts N-quantifier is self-dual, so $\neg \mathit{N}a.\phi \iff \mathit{N}a.\neg\phi$ holds.) Moreover, αProlog's default depth-first search is typically ineffective for searching for counterexamples, so instead, we employ *iterative deepening* search up to a given resource bound.

Since we live in a logic programming world, the choice of what we mean by "not holding" is crucial, as we must choose an appropriate notion of *negation*. We explore two approaches, the first being the

standard *negation-as-failure* rule (NF) and the other based on the technique of *negation elimination* (NE) [12], a source-to-source transformation replacing negated subgoals with calls to equivalent positively defined predicates. A detailed explanation of the two approaches can be found in [5].

The main differences (for the purposes of this paper) are that NF requires *grounding* the free variables of the test before solving the negated goal, to avoid the usual non-logical behavior of negation-as-failure. We currently support NF by automatically generating grounding predicates. In contrast, NE does not require grounding, but does require generating predicates for term inequality and non-freshness of names in addition to the negated predicates. Further, existential subgoals are negated using *extensional universal quantifiers*, which perform case unfolding. The generated code typically has a high branching factor due to this and other redundancies, and is not aggressively optimized. We also consider a special case NE^- that uses standard universal quantification instead (which is faster but less complete). In practice, NF is often (but not always) faster than NE; however, we have a stronger correctness guarantee for NE, since αProlog's implementation of negation-as-failure is not currently backed up by a semantics or correctness proof.

4 Related Work

Our approach owes both to bounded model checking as well as to property-based testing on the QuickCheck tradition [7]. Here we mention only systems that directly have been used for meta-theory model checking.

The system where proofs and disproofs are better integrated is arguably Isabelle/HOL [2]: it offers a QuickCheck-like combination of random, exhaustive and symbolic testing, as well as *Nitpick* [3], a higher-order model finder in the *Alloy* lineage supporting (some) (co)inductive definitions. It works translating a significant fragment of Isabelle/HOL into first-order relational logic and then invoking Alloy's SAT-based model enumerator. Isabelle's QuickCheck, due to some remarkable work on automatic inference of (smart) generators, is very friendly to use but it is still restricted to the *executable* fragments of functional specifications. Nitpick in a sense tries to do too much, addressing most of higher-order logic. In our experience, most of the inductive definitions of related to meta-theory model-checking fall out of the handled fragment.[2]

The other major contender is *PLT-Redex* [9], an executable DSL for mechanizing semantic models built on top of *DrRacket*, with special support for evaluation semantics. Redex has been the first environment to adopt the idea of random testing 'a la QuickCheck for validation of the meta-theory of object language, with significant success [11]. As we have mentioned, the main drawbacks are again the lack of support for binders and low coverage of test generators stemming from grammars definitions. The user is therefore required to write her own generators, a task which tends to be demanding.

Finally, the Bedwyr system [1] supports exhaustive search for finite domains, applied for example to bisimulation checking for finitary pi-calculus. In principle, Bedwyr should support some form of meta-theory model-checking (provided suitable finite generators are implemented for a given object language), but we have not yet performed experiments comparing its behavior with αCheck or other systems in detail.

[2] Note that QuickCheck and Nitpick's do not interact well with *Nominal* Isabelle [15], as it requires strengthening its support for computation with names, permutations and abstract syntax modulo α-conversion.

5 Case Studies

All test have been performed under Ubuntu 14.4 on a Intel Core i7 CPU 870, 2.93GHz with 8GB RAM. These tests must be taken with a lot of salt: not only our tool is under active development, in particular yielding very different search-spaces for NF and NE, but the comparison with the other systems is only roughly indicative, having to factor out differences between logic and functional programming, as well the sheer scale, and therefore indirectness, of a system such as Isabelle/HOL.

5.1 Basic Properties of Lists

While not terribly exciting, these benchmarks, proposed and measured in [4] and taken from Isabelle *List.thy* theory are useful to set up a rough comparison with Isabelle's QuickCheck. We show the checks in our logic programming formulation, leaving to the reader the obvious meaning, noting only that we use numerals as datatype.

```
D1: distinct([X|XS])=> distinct(XS).
D2: distinct(XS),remove1(X,XS,YS)=> distinct(YS).
D3: distinct(XS),distinct(YS),zip(XS,YS,ZS)=> distinct(ZS).
S1: sorted(XS),remove_dupls(XS,YS)=> sorted(YS).
S2: sorted(XS),insert(X,XS,YS)=> sorted(YS).
S3: sorted(XS),length(XS,N),less_equal(I,J),less(J,N),nth(I,XS,X),nth(J,XS,Y)=>less_equal(X,Y).
```

		9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
D1	S	0	0	0	0.2	0.7	3.8	22	135	862								
	NF	0	0	0	0	0	0	0	0	0	0.07	0.12	0.2	0.32	0.52	0.83	1.36	2.22
	NE	0	0	0	0	0	0	0	0	0	0.06	0.11	0.18	0.3	0.49	1.8	1.3	2.1
D2	S	0	0	0.1	0.4	2.5	16	98	671									
	NF	0	0	0	0	0	0	0	0	0	0	0.07	0.19	0.32	0.51	0.83	1.36	2.23
	NE	0	0	0	0	0	0	0	0	0	0.6	0.11	0.18	0.3	0.49	0.8	1.32	2.17
D3	S	4.3	157															
	NF	0	0	0	0.08	0.14	0.35	0.76	1	3	6	12	24	45	82	155	286	580
	NE	0	0	0	0.08	0.13	0.32	0.68	1.3	3	6	11	22	42	79	150	280	586
S1	S	0	0	0	0	0	0	0	0	0.10	0.2	0.3	0.8	1.7	3.6	7.8	17	36
	NF	0	0	0	0	0	0	0	0	0	0	0.6	0.08	0.11	0.15	0.21	0.27	0.35
	NE	0	0	0	0	0	0	0	0	0	0	0.06	0.08	0.11	0.15	0.2	0.27	0.36
S2	S	0	0	0	0	0	0.1	0.1	0.2	0.5	1.1	2.5	5.5	12	28	61	135	292
	NF	0	0	0	0	0	0	0	0	0	0	0	0.05	0.07	0.1	0.13	0.18	0.23
	NE	0	0	0	0	0	0	0	0	0	0.06	0.08	0.11	0.15	0.19	0.25	0.33	0.44
S3	S	0	0	0	0	0.1	0.1	0.2	0.4	0.9	2.2	5.1	12	26	59	136	311	708
	NF	0	0	0.05	0.08	0.13	0.2	0.32	0.48	0.73	1	1.5	2.2	3.2	4.5	6.4	8.9	12
	NE	0	0	0	0.05	0.08	0.12	0.18	0.27	0.4	0.57	0.83	1.1	1.6	2.2	3.2	4.3	5.7

Table 1: QuickCheck's benchmark: S for smart generators, 0 for time < 50 ms, empty cells for timeout after 1h.

Table 2 5.1 shows the run time to validate those (true) properties *w.r.t.* a correct implementation up to a given size (25), that in our case we interpret as depth-bound. We extrapolated from Table 2 in [4] the *S* (for *smart generator*) rows. (We omit the results for *exhaustive* and *narrowing-based* testing; the point of their inclusion in [4] was to show how smart generation outperforms the latter two over checks with hard-to-satisfy premises.) The fact that we are so largely superior is probably due to smart generation trying to replicate in a functional setting what logic programming naturally offers. Note however that tests in Isabelle/QuickCheck are efficiently run by code generation at the ML level, while our bounded solver is

just a non-optimized logic programming interpreter – to name one, it does not have yet first-argument indexing.

Here NE somewhat outperforms NF, in part because the negated predicates (`distinct`, `sorted` etc.) are fairly simple, in part because it does not require extensional quantification. One could conjecture that mode information, which is a central ingredient in smart generation, may help to reduce some of NF's excessive term generation burden, as in *D3*. Anecdotal evidence, which needs to be investigated further, suggests instead that it may not yield a net gain in our setting. For example, mode information will dictate this version of *S2*, where `sortedm` is modified to be grounding:

```
S2m: sortedm(Xs),gen_nat(X),insert(X,Xs,Ys) => sortedm(Ys).
```

This happens to be in our tool under NF much slower over *S2* – 16 seconds over 0.7.

5.2 Security Type Systems

To compare Nitpick with our approach, we selected a case study mentioned in [3]: an encoding of the Volpano, Irvine and Smith [16] security type system, whereby the basic imperative language *IMP* is endowed with a type system that prevents information flow from private to public variables.[3] Blanchette and Nipkow reports that by inserting a mutation in the typing rule for command sequencing, they get a counterexample to the crucial *non-interference* property.

For our test, we actually selected the more general version of the type system formalized in [13], where the security levels are generalized from *high* and *low* to natural numbers. Given a fixed assignment *sec* of such security levels to variables, then lifted to arithmetic and booleans expressions, the typing judgment $l \vdash c$ reads as "command c does not contain any information flow to variables lower then l and only safe flows to variables $\geq l$. We show a selection of the rules, where the over-strike denotes the inserted mutation.

$$\frac{sec\ a \leq sec\ x \qquad l \leq sec\ x}{l \vdash x := a} \qquad \frac{l \vdash c_1 \qquad \cancel{l \vdash c_2}}{l \vdash c_1; c_2} \qquad \frac{max\ (sec\ b\ l) \vdash c_1 \qquad max\ (sec\ b\ l) \vdash c_2}{l \vdash IF\ b\ THEN\ c_1\ ELSE\ c_2}$$

The properties of interest relate states that agree on the value of each variable *below* a certain security level, denoted as $\sigma_1 \approx_{\leq l} \sigma_2$ (resp. $\sigma_1 \approx_{< l} \sigma_2$) iff $\forall x.\ sec\ x \leq l \rightarrow \sigma_1(x) = \sigma_2(x)$ (resp. $<$). Given a standard big-step evaluation semantics for IMP, relating an initial state σ and a command c to a final state τ:

Confinement If $\langle c, \sigma \rangle \downarrow \tau$ and $l \vdash c$ then $\sigma \approx_{< l} \tau$;

Non-interference If $\langle c, \sigma \rangle \downarrow \sigma'$, $\langle c, \tau \rangle \downarrow \tau'$, $\sigma \approx_{\leq l} \tau$ and $0 \vdash c$ then $\sigma' \approx_{\leq l} \tau'$;

Differently from the Isabelle/HOL mostly functional setting, our encoding is fully relational, where, for example, states and security assignments cannot be seen as partial functions but are reified in association lists. We also pay a price in not being able to rely on built-in types such as integers, but have to deploy our clearly inefficient versions. Finally, this case study does not exercise binders intensely, as we are only using nominal techniques in representing program variables as names and freshness to guarantee well-formedness of states and of the table encoding the variable security settings. Nevertheless, the experimental evidence is quite pleasing, as we sum up in Table 2: our tool finds very quickly this counterexample to confinement, where c is $(SKIP\ ; x := 1)$, $sec\ x = 0$, $l = 1$ and σ maps x to 0. This would not hold were the typing rule to check the second premise. A not too dissimilar counterexample falsifies non-interference: c is $(SKIP\ ; x := y)$, $sec\ x, y = 0, 1$, $l = 0$ and σ maps y to 0 and x undefined (i.e. to a logic

[3] We readily acknowledge that this is quite trivial, compared to much more in-depth experiments such as [10].

variable), while τ maps y to 1 and keeps x undefined. Compare it to Nitpick, which, using the following settings `nitpick_params[sat_solver=MiniSat_JNI,max_threads=1,check_potential]` finds *(x := -1 + y; y := -1 + y)* as a confinement bug fairly quickly. However, it warns that it could be *spurious*, and the *auto* tactic is unable to rule it genuine. Secondly, Nitpick fails to deal with non-interference due to the standard problem that "the conjecture lies outside Nitpick's supported fragment", meaning that the evaluation relation cannot be proven well-founded (and rightly so), requiring therefore its problematic unrolling.

Check	**Nitpick**	**NF**	**NE**	**NE$^-$**
Confinement	0.6	0.04	0.04	0.03
Non-interference	timeout	9.46	0.32	0.29

Table 2: The VIS case study: time out at 30 secs, max depth 5 (for confinement) and 8 (for non-interference)

5.3 Head-to-Head with PLT-Redex

bug#	class	description	cex
1	S	as in Sec 2	*hd 0*
2	M	*(cons v) v* value has been omitted	*(cons 0) nil*
3	S	swap of order of types in function pos of app	$(\lambda x{:}int.\ cons\ )\ cons$
4	S	type of cons is incorrect	**not found**
5	S	tail reduction returns the head	*tl ((cons 1) nil))*
6	M	hd reduction acts on partially applied cons	*(hd ((cons 1) nil)))*
7	M	no evaluation rhs of app	*1 + (2 + 3)*
8	U	lookup always returns int	$(\lambda x{:}ilist.\ cons\ x)\ nil$
9	S	vars may not match in lookup	$(\lambda x{:}int.\ \lambda y{:}ilist\ x)\ 1)$

Table 3: Stlc benchmark list

Here we report our results in checking the rest of the mutations mentioned in `http://docs.racket-lang.org/redex/benchmark.html` *w.r.t.* the calculus of Section 2 and listed in Table 3. This lists the mutation with a "difficulty" rating (Simple/Medium/Unusual) and the smallest expressions, found by Redex, that falsifies preservation *or* progress.

The two encodings are somewhat different: Redex has very good support for evaluation contexts, while we use congruence rules. Being untyped, the Redex econding treats *error* as a string, which is then procedurally handled in the statement of preservation and progress, whereas for us it is part of the language. Since [11] Redex allows the user to write judgments such as typing in a declarative style akin to Ott's style [14], provided they can be given a functional mode, but slightly more complex systems, such as typing rule for a polymorphically version of a similar calculus, require very indirect encoding, e.g. CPS-style. We do not model addition on integers, as we currently require our code to be pure in the logical sense, i.e. no appeal to built-in arithmetics, as opposed to Redex that maps integers to Racket's ones. This is why we cannot find bug 4, which requires integers to be used and not just built. *W.r.t.* lines of code, our encodings is roughly $1/4$ of Redex's, not counting Redex's built-in generators and substitution function. The adopted checking philosophy is also somewhat different: they choose to test preservation

and progress together, using a cascade of three built-in generators and collect all the counterexamples found within a timeout of n seconds. Redex response is very quick, in this benchmark under 1 second.

While our performances, as shown in Table 2 5.3 could be better, we basically catch the same bugs with very similar counterexamples. Bugs 8 and 9 can also be caught by checking the substitution lemma:
`x # (G,E),tc(G,E,T),tc([(x,T)|G],E',T'),valid_ctx(G) => tc(G,subst(E',x,E),T').`
showing the usefulness of testing intermediate results. In conclusion, we'd like to point out that the above mutations, as witnessed by Redex's counterexamples, do not exercise binders that much, especially compared to some of the case studies in [11]. This probably explains why Redex's built-in generators are here surprisingly effective, although they do not discriminate between free or bound variables, nor do they produce well-typed terms as we do by construction. This seems to suggest that the typing rules are not exercised in depth. Finally, the fact that random testing *without* shrinking finds roughly the same counterexamples as exhaustive search suggests that this benchmark is not too meaningful and that a harder case study is required.

bug#	check	NF	NE$^-$	cex
1	pres	0.3	1.46	$(\lambda x{:}ilist.\ x\ err)\ n$
	prog	0	3.89	as Redex
2	prog	0.29		as Redex
3	pres	0	0	$(\lambda x{:}ilist.\ n)\ n$
	prog	0	4.39	as Redex
5	pres	5.77	23.2	as Redex
6	prog	29.8		as Redex
7	prog	0.9		$hd\ ((\lambda x{:}int.\ error)\ n)$
8	pres	0.1	0.1	$(\lambda x{:}ilist.\ x)\ nil$
9	pres	0.1	0.1	$(\lambda x{:}int.\ y)\ n$

Table 4: αCheck results on Stlc benchmarks. Timeout 30 sec, smallest cex shown.

6 Conclusions and Future Work

This work-in-progress paper presents case studies comparing Isabelle's QuickCheck/Nitpick, PLT Redex, and αCheck. Our experiments support a preliminary conclusion that our approach has advantages compared to the competition in terms of the amount of work required to represent and check a system correctly (taking binding into account) and does not pay a terrible price in terms of speed, notwithstanding being largely unoptimized. On the other hand, αCheck has some clear limitations compared to these systems: for example, QuickCheck/Nitpick's embedding in Isabelle/HOL supports both proof and checking of the same specification in an integrated way, and as part of the Racket ecosystem Redex supports among others operational semantics animation, sophisticated documentation generation and efficient compilation of tests. We have also not yet performed a detailed comparison with Bedwyr, which appears to be the closest in terms of the underlying technology.

Despite its incompleteness, performing this comparison has been valuable to us, particular in identifying bugs and limitations in αProlog and its checking facilities. We conclude by listing some future directions, both for our own goal of developing αCheck into a serious tool for mechanized meta-theory exploration and for empirical research about such tools generally.

- **Combining performance and usability.** There appear to be opportunities to improve the performance of negation elimination. αCheck is still not easy to use correctly, and checks sometimes need to be written differently depending on whether NF or NE is used.
- **Integration.** We have resisted the temptation to complicate αCheck with document generation or theorem proving capabilities, contrasting with more monolithic tools such as Redex or Isabelle/HOL. Bidirectional transformations (bx) [8] could be useful for synchronizing specifications among multiple specialized tools.
- **Benchmarking.** Previous publications on property-based testing for mechanized meta-theory (including our own) have generally not been accompanied by detailed enough descriptions of experimental methodology to ensure reproducibility. Developing a suite of benchmark/challenge problems and standards for reporting results would, we hope, help clarify the strengths and weaknesses of different approaches and guide future research towards improvements.

References

[1] David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur & Alwen Tiu (2007): *The Bedwyr System for Model Checking over Syntactic Expressions*. In Frank Pfenning, editor: *CADE*, *LNCS* 4603, Springer, pp. 391–397. Available at `http://dx.doi.org/10.1007/978-3-540-73595-3_28`.

[2] Jasmin Christian Blanchette, Lukas Bulwahn & Tobias Nipkow (2011): *Automatic Proof and Disproof in Isabelle/HOL*. In Cesare Tinelli & Viorica Sofronie-Stokkermans, editors: *FroCoS*, *Lecture Notes in Computer Science* 6989, Springer, pp. 12–27. Available at `http://dx.doi.org/10.1007/978-3-642-24364-6_2`.

[3] Jasmin Christian Blanchette & Tobias Nipkow (2010): *Nitpick: A Counterexample generator for higher-order logic based on a relational model finder*. In M. Kaufmann & L. Paulson, editors: *Interactive Theorem Proving (ITP 2010)*, *LNCS* 6172, Springer, pp. 131–146.

[4] Lukas Bulwahn (2012): *Smart Testing of Functional Programs in Isabelle*. In Nikolaj Bjørner & Andrei Voronkov, editors: *LPAR*, *Lecture Notes in Computer Science* 7180, Springer, pp. 153–167. Available at `http://dx.doi.org/10.1007/978-3-642-28717-6_14`.

[5] James Cheney & Alberto Momigliano (2007): *Mechanized metatheory model-checking*. In Michael Leuschel & Andreas Podelski, editors: *PPDP*, ACM, pp. 75–86. Available at `http://doi.acm.org/10.1145/1273920.1273931`.

[6] James Cheney & Christian Urban (2008): *Nominal Logic Programming*. *ACM Transactions on Programming Languages and Systems* 30(5), p. 26.

[7] Koen Claessen & John Hughes (2000): *QuickCheck: a lightweight tool for random testing of Haskell programs*. In: *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, ACM, pp. 268–279.

[8] Krzysztof Czarnecki, J. N. Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr & James F. Terwilliger (2009): *Bidirectional Transformations: A Cross-Discipline Perspective*. In: *ICMT*, *LNCS* 5563, pp. 260–283.

[9] Robert Bruce Findler, Casey Klein & Burke Fetscher (2015): *Redex: Practical Semantics Engineering*.

[10] Catalin Hritcu & co authors (2013): *Testing Noninterference, Quickly*. In: *ICFP*, ACM, pp. 455–468.

[11] Casey Klein & co authors (2012): *Run your research: on the effectiveness of lightweight mechanization*. In: *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, ACM, pp. 285–296. Available at `http://doi.acm.org/10.1145/2103656.2103691`.

[12] Alberto Momigliano (2000): *Elimination of Negation in a Logical Framework*. In Peter Clote & Helmut Schwichtenberg, editors: *CSL*, *Lecture Notes in Computer Science* 1862, Springer, pp. 411–426. Available at `http://link.springer.de/link/service/series/0558/bibs/1862/18620411.htm`.

[13] Tobias Nipkow & Gerwin Klein (2014): *Concrete Semantics - With Isabelle/HOL*. Springer.

[14] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar & Rok Strnisa (2010): *Ott: Effective tool support for the working semanticist*. *J. Funct. Program.* 20(1), pp. 71–122, doi:http://dx.doi.org/10.1017/S0956796809990293.

[15] Christian Urban & Cezary Kaliszyk (2012): *General Bindings and Alpha-Equivalence in Nominal Isabelle*. *Logical Methods in Computer Science* 8(2), doi:10.2168/LMCS-8(2:14)2012.

[16] Dennis Volpano, Cynthia Irvine & Geoffrey Smith (1996): *A Sound Type System for Secure Flow Analysis*. *J. Comput. Secur.* 4(2-3), pp. 167–187.