# Advances in Property-Based Testing for $\alpha$Prolog

James Cheney[1(✉)], Alberto Momigliano[2], and Matteo Pessina[2]

[1] University of Edinburgh, Edinburgh, UK
jcheney@inf.ed.ac.uk
[2] Università degli Studi di Milano, Milan, Italy
momigliano@di.unimi.it, matteo.pessina3@studenti.unimi.it

**Abstract.** $\alpha$Check is a light-weight property-based testing tool built on top of $\alpha$Prolog, a logic programming language based on nominal logic. $\alpha$Prolog is particularly suited to the validation of the meta-theory of formal systems, for example correctness of compiler translations involving name-binding, alpha-equivalence and capture-avoiding substitution. In this paper we describe an alternative to the negation elimination algorithm underlying $\alpha$Check that substantially improves its effectiveness. To substantiate this claim we compare the checker performances w.r.t. two of 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.

## 1 Introduction

Formal compiler verification has come a long way from McCarthy and Painter's "Correctness of a Compiler for Arithmetic Expression" (1967), as witnessed by the success of *CompCert* and subsequent projects [21,35]. However outstanding these achievements are, they are not a magic wand for every-day compiler writers: not only CompCert was designed with verification in mind, whereby the implementation and the verification were a single process, but there are only a few dozen people in the world able and willing to carry out such an endeavour. By verification, CompCert means the preservation of certain simulation relations between source, intermediate and target code; however, the translations involved are relatively simple compared to those employed by modern optimizing compilers. Despite some initial work [1,7], handling more realistic optimizations seems even harder, e.g. the verification of the *call arity* analysis and transformation in the Glasgow Haskell Compiler (GHC):

> "The [Nominal] Isabelle development corresponding to this paper, including the definition of the syntax and the semantics, contains roughly 12,000 lines of code with 1,200 lemmas (many small, some large) in 75 theories, created over the course of 9 months" (page 11, [7]).

For the rest of us, hence, it is back to compiler testing, which is basically synonymous with passing a hand-written fixed validation suite. This is not completely satisfactory, as the coverage of those tests is difficult to assess and

B.K. Aichernig and C.A. Furia (Eds.): TAP 2016, LNCS 9762, pp. 37–56, 2016.
DOI: 10.1007/978-3-319-41135-4_3

because, being fixed, these suites will not uncover new bugs. In the last few years, *randomized differential testing* [24] has been suggested in combination with automatic generation of (expressive) test programs, most notably for C compilers with the *Csmith* tool [36] and to a lesser extent for GHC [30]. The oracle is *comparison checking*: Csmith feeds randomly generated programs to several compilers and flags the minority one(s), that is, those reporting different outputs from the majority of the other compilers under test, as incorrect. Similarly, the outcome of GHC on a random program with or without an optimization enabled is compared.

*Property-based testing*, as pioneered by QuickCheck [12], seems to leverage the automatic generation of test cases with the use of *logical specifications* (the properties), making validation possible not only in a differential way, but internally, *w.r.t.* (an abstraction of) the behavior of the source and intermediate code. In fact, compiler verification/validation is a prominent example of the more general field of verification of the *meta-theory* of formal systems. For many classes of (typically) shallow bugs, a tool that automatically finds counterexamples can be surprisingly effective and can complement formal proof attempts by warning when the property we wish to prove has easily-found counterexamples. 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 traction in the field of semantics engineering, see in particular the QuickCheck/Nitpick combination offered in Isabelle/HOL [4] and random testing in PLT-Redex [18]. However, a particular dimension to validating for example optimizations in a compiler such as GHC, whose intermediate language is a variant of the polymorphically typed $\lambda$-calculus, is a correct, simple and effective handling of *binding signatures* and associated notions such as $\alpha$-equivalence and capture avoiding substitutions. A small but not insignificant part of the success of the CompCert project is due to not having to deal with any notion of binder[1]. The ability to encode possibly non-algorithmic relations (such as typing) in a declarative way would also be a plus.

The nominal logic programming language $\alpha$Prolog [11] offers all those facilities. Additionally, it was among the first to propose a form of property based testing for language specifications with the *$\alpha$Check* tool [9]. 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. Systems lacking this kind of support may end up with ineffective testing capabilities or requiring an additional amount of coding, which needs to be duplicated in every case study:

[1] X. Leroy, personal communication. In fact, the encoding in [22] does not respect $\alpha$-equivalence, nor does it implement substitutions in a capture avoiding way.

"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 $\alpha$-equivalence [...] In one case [...] managing binders constitutes a significant portion of the overall time spent [...] 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" (page 5, [18]).

$\alpha$Check extends $\alpha$Prolog with tools for searching for counterexamples, that is, substitutions that makes the antecedent of a specification true and the conclusion false. In logic programming terms this means fixing a notion of *negation*. To begin with, $\alpha$Check adopted the infamous *negation-as-failure* (NF) operation, "which put pains thousandfold upon the" logic programmers. As many good things in life, its conceptual simplicity and efficiency is marred by significant problems:

- the lack of an agreed intended semantics against which to carry a soundness proof: this concern is significant because the semantics of negation as failure has not yet been investigated for nominal logic programming;
- even assuming such a semantics, we know that *NF* is unsound for non-ground goals; hence all free variables must be instantiated before solving the negated conclusion. This is obviously exponentially expensive in an exhaustive search setting and may prevent optimizations by goal reordering.

To remedy this $\alpha$Check also offered *negation elimination* (NE) [3,26], a source-to-source transformation that replaces negated subgoals to calls to equivalent positively defined predicates. *NE* by-passes the previous issues arising for *NF* since, in the absence of local (existential) variables, it yields an ordinary ($\alpha$)Prolog program, whose intended model is included in the complement of the model of the source program. In particular, it avoids the expensive term generation step needed for *NF*, it has been proved correct, and it may open up other opportunities for optimization. Unfortunately, in the experiments reported in our initial implementation of $\alpha$Check [9], *NE* turned out to be slower than *NF*.

Perhaps to the reader's chagrin, this paper does not tackle the validation of compiler optimizations (yet). Rather, it lays the foundations by:

1. describing an alternative implementation of negation elimination, dubbed *NEs*—"s" for simplified: this improves significantly over the performance of *NE* as described in [9] by producing negative programs that are equivalent, but much more succinct, so much as to make the method competitive *w.r.t. NF*;
2. and by evaluating our checker in comparison with some of its competitors in the logical framework niche, namely QuickCheck/Nitpick [4] and PLT-Redex [18]. To the best of our knowledge, this is the first time any of these three tools have been compared experimentally.

In the next section we give a tutorial presentation of the tool and move then to the formal description of the logical engine (Sect. 3). In Sect. 4, we detail the

*NEs* algorithm and its implementation, whereas Sect. 5 carries out the promised comparison on two case studies, a prototypical $\lambda$-calculus with lists and a basic type system for secure information flow. The sources for $\alpha$Prolog and $\alpha$Check can be found at https://github.com/aprolog-lang/aprolog. Supplementary material, including the full listing of the case studies presented here and an online appendix containing additional experiments and some formal notions used in Sect. 3, but omitted here for the sake of space, are available at [10]. We assume some familiarity with logic programming.

## 2 A Brief Tour of $\alpha$Check

We specify the formal systems and the properties we wish to check as Horn logic programs in $\alpha$Prolog [11], a logic programming language based on *nominal logic*, a first-order theory axiomatizing names and name-binding introduced by Pitts [32].

In $\alpha$Prolog, there are several built-in types, functions, and relations with special behavior. There are distinguished *name types* that are populated with infinitely many *name constants*. In program text, a lower-case identifier is considered to be a name constant by default if it has not already been declared as something else. Names can be used in *abstractions*, written `a\M` in programs, considered equal up to $\alpha$-renaming of the bound name. Thus, where one writes $\lambda x.M$, $\forall x.M$, etc. in a paper exposition, in $\alpha$Prolog one writes `lam(x\M)`, `forall(x\M)`, etc. In addition, the *freshness* relation `a # t` holds between a name `a` and a term `t` that does not contain a free occurrence of `a`. Thus, $x \notin FV(t)$ is written in $\alpha$Prolog as `x # t`; in particular, if $t$ is also a name then freshness is name-inequality. For convenience, $\alpha$Prolog provides a function-definition syntax, but this is just translated to an equivalent (but more verbose) relational implementation via *flattening*.

Horn logic programs over these operations suffice to define a wide variety of object languages, type systems, and operational semantics in a convenient way. To give a feel of the interaction with the checker, here we encode a simply-typed $\lambda$-calculus augmented with constructors for integers and lists, following the PLT-Redex benchmark `sltk.lists.rkt` from http://docs.racket-lang.org/redex/benchmark.html, which we will examine more deeply in Sect. 5.1. The language is formally declared as follows:

$$\begin{array}{lll}
\text{Types} & A, B & ::= \mathit{int} \mid \mathit{ilist} \mid A \to B \\
\text{Terms} & M & ::= x \mid \lambda x{:}A.\ M \mid M_1\ M_2 \mid c \mid \mathit{err} \\
\text{Constants} & c & ::= n \mid \mathit{nil} \mid \mathit{cons} \mid \mathit{hd} \mid \mathit{tl} \\
\text{Values} & V & ::= c \mid \lambda x{:}A.\ M \mid \mathit{cons}\ V \mid \mathit{cons}\ V_1\ V_2
\end{array}$$

We start (see the top of Fig. 1) by declaring the syntax of terms, constants and types, while we carve out values *via* an appropriate predicate. A similar predicate `is_err` characterizes the threading in the operational semantics of the *err* expression, used to model run time errors such as taking the head of an empty list.

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

type ctx = [(id,ty)].

pred tc (ctx,exp,ty).
tc(_,err,T).
tc(_,c(C),T)                                :- tcf(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,T),funTy(T,U))                 :- x # G, tc([(x,T) |G],M,U).

pred step(exp,exp).
step(app(c(hd),app(app(c(cons),V),VS)),V) :- value(V), value(VS).
step(app(c(tl),app(app(c(cons),V),VS)),VS):- value(V), value(VS).
step(app(lam(x\M,T),V), subst(M,x,V))      :- value(V).
step(app(M1,M2),app(M1',M2))                :- step(M1,M1').
step(app(V1,M2),app(M1,M2'))                :- value(V1), step(M2,M2').

pred is_err(exp).
is_err(err).
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).
```

**Fig. 1.** Encoding of the example calculus in αProlog

We follow this up (see the remainder of Fig. 1) with the static semantics (predicate `tc`) and dynamic semantics (one-step reduction predicate `step`), where we omit the judgments for the `value` predicate and `subst` function, which are analogous to the ones in [9]. Note that *err* has any type and constants are typed *via* a table `tcf`, also omitted.

Horn clauses can also be used as specifications of desired program properties of such an encoding, including basic lemmas concerning substitution as well as main theorems such as preservation, progress, and type soundness. This is realized *via* checking *directives*

```
#check "spec" n : H1, ..., Hn => A.
```

where `spec` is a label naming the property, `n` is a parameter that bounds the search space, and `H1` through `Hn` and `A` are atomic formulas describing the preconditions and conclusion of the property. As with program clauses, the specification formula is implicitly universally quantified. Following the PLT-Redex development, we concentrate here only on checking that preservation and progress hold.

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

Here, `progress` is a predicate encoding the property 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 (hopefully) true of the given setup. Now, 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) :- tc(G,M,funTy(T,U)), tc(G,N,U). % was funTy(U,T)
```

Does any property become false? The checker returns immediately with this counterexample to progress:

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

This is abstract syntax for *hd n*, an expression erroneously well-typed and obviously stuck. Preservation meets a similar fate: $(\lambda x{:}T \to \mathit{int}.\ x\ \mathit{err})\ n$ steps to an ill-typed term.

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

## 3 The Core Language

In this section we give the essential notions concerning the core syntax, to which the surface syntax used in the previous section desugars, and semantics of $\alpha$Prolog programs.

An $\alpha$Prolog *signature* is composed by sets $\Sigma_D$ and $\Sigma_N$ of, respectively, base types $\delta$, which includes a type $o$ of *propositions*, and name types $\nu$; a collection $\Sigma_P$ of *predicate symbols* $p : \tau \to o$ and one $\Sigma_F$ of *function symbol* declarations $f : \tau \to \delta$. Types $\tau$ are formed as specified by the following grammar:

$$\tau ::= \delta \mid \tau \times \tau' \mid \mathbf{1} \mid \nu \mid \langle\nu\rangle\tau$$

where $\delta \in \Sigma_D$ and $\nu \in \Sigma_N$ and $\mathbf{1}$ is the unit type. Given a signature, the language of *terms* is defined over sets $V = \{X, Y, Z, \ldots\}$ of logical variables and sets $A = \{\mathsf{a}, \mathsf{b}, \ldots\}$ of names:

$$\begin{aligned} t, u &::= \mathsf{a} \mid \pi \cdot X \mid \langle\rangle \mid \langle t, u\rangle \mid \langle\mathsf{a}\rangle t \mid f(t) \\ \pi &::= \mathsf{id} \mid (\mathsf{a}\ \mathsf{b}) \circ \pi \end{aligned}$$

where $\pi$ are permutations, which we omit in case $\mathsf{id} \cdot X$, $\langle\rangle$ is unit, $\langle t, u\rangle$ is a pair and $\langle\mathsf{a}\rangle t$ is the abstract syntax for name-abstraction. The result of applying the permutation $\pi$ (considered as a function) to $\mathsf{a}$ is written $\pi(\mathsf{a})$. Typing for these terms is standard, with the main novelty being that name-abstractions $\langle \mathsf{a} \rangle t$ have abstraction types $\langle \nu \rangle \tau$ provided $\mathsf{a} : \nu$ and $t : \tau$.

The *freshness* ($s \mathrel{\#_\tau} u$) and *equality* ($t \approx_\tau u$) constraints, where $s$ is a term of some name type $\nu$, are the new features provided by nominal logic. The former relation is defined on ground terms by the following inference rules, where $f : \tau \to \delta \in \Sigma_F$:

$$\frac{\mathsf{a} \neq \mathsf{b}}{\mathsf{a} \mathrel{\#_\nu} \mathsf{b}} \qquad \frac{}{\mathsf{a} \mathrel{\#_{\mathbf{1}}} \langle\rangle} \qquad \frac{\mathsf{a} \mathrel{\#_\tau} t}{\mathsf{a} \mathrel{\#_\delta} f(t)} \qquad \frac{\mathsf{a} \mathrel{\#_{\tau_1}} t_1 \quad \mathsf{a} \mathrel{\#_{\tau_2}} t_2}{\mathsf{a} \mathrel{\#_{\tau_1 \times \tau_2}} \langle t_1, t_2 \rangle} \qquad \frac{\mathsf{a} \mathrel{\#_{\nu'}} \mathsf{b} \quad \mathsf{a} \mathrel{\#_\tau} t}{\mathsf{a} \mathrel{\#_{\langle \nu' \rangle \tau}} \langle \mathsf{b} \rangle t} \qquad \frac{}{\mathsf{a} \mathrel{\#_{\langle \nu' \rangle \tau}} \langle \mathsf{a} \rangle t}$$

In the same way we define the equality relation, which identifies terms modulo $\alpha$-equivalence, where $(\mathsf{a}\ \mathsf{b}) \cdot u$ denotes *swapping* two names in a term:

$$\frac{}{\mathsf{a} \approx_\nu \mathsf{a}} \qquad \frac{}{\langle\rangle \approx_{\mathbf{1}} \langle\rangle} \qquad \frac{t_1 \approx_{\tau_1} u_1 \quad t_2 \approx_{\tau_2} u_2}{\langle t_1, t_2 \rangle \approx_{\tau_1 \times \tau_2} \langle u_1, u_2 \rangle} \qquad \frac{t \approx_\tau u}{f(t) \approx_\delta f(u)}$$

$$\frac{\mathsf{a} \approx_\nu \mathsf{b} \quad t \approx_\tau u}{\langle \mathsf{a} \rangle t \approx_{\langle \nu \rangle \tau} \langle \mathsf{b} \rangle u} \qquad \frac{\mathsf{a} \mathrel{\#_\nu} \mathsf{b} \quad \mathsf{a} \mathrel{\#_\nu} u \quad t \approx_\tau (\mathsf{a}\ \mathsf{b}) \cdot u}{\langle \mathsf{a} \rangle t \approx_{\langle \nu \rangle \tau} \langle \mathsf{b} \rangle u}$$

Given a signature, *goals* $G$ and *program clauses* $D$ have the following form:

$$\begin{aligned} A &::= t \approx u \mid t \mathrel{\#} u \\ G &::= \bot \mid \top \mid A \mid p(t) \mid G \wedge G' \mid G \vee G' \mid \exists X{:}\tau.\ G \mid \text{И}\mathsf{a}{:}\nu.\ G \mid \forall^* X{:}\tau.\ G \\ D &::= \top \mid p(t) \mid D \wedge D' \mid G \supset D \mid \forall X : \tau.\ D \mid \bot \mid D \vee D' \end{aligned}$$

The productions shown in black yield a fragment of nominal logic called И-goal clauses [11], for which resolution based on nominal unification is sound and complete. This is in contrast to the general case where the more complicated *equivariant unification* problem must be solved [8]. We rely on the fact that $D$ formulas in a program $\Delta$ can always be normalized to sets of clauses of the form $\forall \boldsymbol{X}{:}\boldsymbol{\tau}.\ G \supset p(t)$, denoted $\mathrm{def}(p, \Delta)$. The *fresh-name* quantifier И, firstly introduced in [32], quantifies over names not occurring in a formula (or in the values of its variables). The extensions shown in red here in the language BNF (and in its proof-theoretic semantics in Fig. 2) instead are constructs brought in from the negation elimination procedure (Sect. 4.1) and which will not appear in any source programs. In particular, an unusual feature is the *extensional* universal quantifier $\forall^*$ [15]. Differently from the *intensional* universal quantifier $\forall$, for which $\forall X{:}\tau.\ G$ holds if and only if $G[x/X]$ holds, where $x$ is an eigenvariable representing any terms of type $\tau$, $\forall^* X{:}\tau.\ G$ succeeds if and only if $G[t/X]$ does for *every* ground term of type $\tau$.

*Constraints* are $G$-formulas of the following form:

$$C ::= \top \mid t \approx u \mid t \mathrel{\#} u \mid C \wedge C' \mid \exists X{:}\tau.\ C \mid \text{И}\mathsf{a}{:}\nu.\ C$$

We write $\mathcal{K}$ for a set of constraints and $\Gamma$ for a context keeping track of the types of variables and names. Constraint-solving is modeled by the judgment $\Gamma; \mathcal{K} \models C$, which holds if for all maps $\theta$ from variables in $\Gamma$ to ground terms if $\theta \models \mathcal{K}$ then $\theta \models C$. The latter notion of satisfiability is standard, modulo handling of names: for example $\theta \models \mathsf{И}\mathsf{a}{:}\nu.\ C$ iff for some $\mathsf{b}$ fresh for $\theta$ and $C$, $\theta \models C[\mathsf{b}/\mathsf{a}]$.

$$\frac{\Gamma;\mathcal{K} \models A}{\Gamma;\Delta;\mathcal{K} \Rightarrow A}\ con \qquad \frac{\Gamma;\Delta;\mathcal{K} \Rightarrow G_1 \quad \Gamma;\Delta;\mathcal{K} \Rightarrow G_2}{\Gamma;\Delta;\mathcal{K} \Rightarrow G_1 \wedge G_2}\ \wedge R$$

$$\frac{\Gamma;\Delta;\mathcal{K} \Rightarrow G_i}{\Gamma;\Delta;\mathcal{K} \Rightarrow G_1 \vee G_2}\ \vee R_i \qquad \frac{\Gamma;\mathcal{K} \models \exists X{:}\tau.\ C \quad \Gamma, X{:}\tau;\Delta;\mathcal{K}, C \Rightarrow G}{\Gamma;\Delta;\mathcal{K} \Rightarrow \exists X{:}\tau.\ G}\ \exists R$$

$$\frac{\Gamma;\mathcal{K} \models \mathsf{И}\mathsf{a}{:}\nu.\ C \quad \Gamma\#\mathsf{a}{:}\nu;\Delta;\mathcal{K}, C \Rightarrow G}{\Gamma;\Delta;\mathcal{K} \Rightarrow \mathsf{И}\mathsf{a}{:}\nu.\ G}\ \mathsf{И}R$$

$$\frac{}{\Gamma;\Delta;\mathcal{K} \Rightarrow \top}\ \top R \qquad \frac{\Gamma;\Delta;\mathcal{K} \xrightarrow{D} Q \quad D \in \Delta}{\Gamma;\Delta;\mathcal{K} \Rightarrow Q}\ sel$$

$$\frac{\bigwedge\{\Gamma, X{:}\tau;\Delta;\mathcal{K}, C \Rightarrow G \mid \Gamma;\mathcal{K} \models \exists X{:}\tau.\ C\}}{\Gamma;\Delta;\mathcal{K} \Rightarrow \forall^* X{:}\tau.\ G}\ \forall^*\omega$$

$$\frac{\Gamma;\mathcal{K} \models t \approx u}{\Gamma;\Delta;\mathcal{K} \xrightarrow{p(t)} p(u)}\ hyp \qquad \frac{\Gamma;\Delta;\mathcal{K} \xrightarrow{D_i} Q}{\Gamma;\Delta;\mathcal{K} \xrightarrow{D_1 \wedge D_2} Q}\ \wedge L_i$$

$$\frac{\Gamma;\Delta;\mathcal{K} \xrightarrow{D} Q \quad \Gamma;\Delta;\mathcal{K} \Rightarrow G}{\Gamma;\Delta;\mathcal{K} \xrightarrow{G \supset D} Q}\ \supset L$$

$$\frac{\Gamma;\mathcal{K} \models \exists X{:}\tau.\ C \quad \Gamma, X{:}\tau;\Delta;\mathcal{K}, C \xrightarrow{D} Q}{\Gamma;\Delta;\mathcal{K} \xrightarrow{\forall X{:}\tau.\ D} Q}\ \forall L$$

$$\frac{}{\Gamma;\Delta;\mathcal{K} \xrightarrow{\perp} Q}\ \perp L \qquad \frac{\Gamma;\Delta;\mathcal{K} \xrightarrow{D_1} Q \quad \Gamma;\Delta;\mathcal{K} \xrightarrow{D_2} Q}{\Gamma;\Delta;\mathcal{K} \xrightarrow{D_1 \vee D_2} Q}\ \vee L$$

**Fig. 2.** Proof search semantics of $\alpha$Prolog programs

We can describe an idealized interpreter for $\alpha$Prolog with the "amalgamated" proof-theoretic semantics introduced in [11] and inspired by similar techniques stemming from CLP [20] — see Fig. 2, sporting two kind of judgments, goal-directed proof search $\Gamma;\Delta;\mathcal{K} \Rightarrow G$ and focused proof search $\Gamma;\Delta;\mathcal{K} \xrightarrow{D} Q$. This semantics allows us to concentrate on the high-level proof search issues, without requiring to introduce or manage low-level operational details concerning constraint solving. We refer the reader to [11] for more explanation and ways to make those judgments operational. Note that the rule $\forall^*\omega$ says that goals of the form $\forall^* X{:}\tau.G$ can be proved if $\Gamma, X{:}\tau;\Delta;\mathcal{K}, C \Rightarrow G$ is provable for every constraint $C$ such that $\Gamma;\mathcal{K} \models \exists X{:}\tau.\ C$ holds. Since this is hardly practical, the number of candidate constraints $C$ being infinite, we approximate it by modifying the interpreter so as to perform a form of case analysis: at every stage, as dictated by the type of the quantified variable, we can either instantiate $X$ by performing a one-layer type-driven case distinction and further recur to expose the next layer by introducing new $\forall^*$ quantifiers, or we can break the recursion by instantiation with an eigenvariable.

## 4 Specification Checking

Informally, `#check` specifications correspond to specification formulas of the form

$$\text{И}\mathsf{a}.\forall \boldsymbol{X}.\ G \supset A \tag{1}$$

where $G$ is a goal and $A$ an atomic formula (including equality and freshness constraints). Since the И-quantifier is self-dual, the negation of (1) is of the form $\text{И}\mathsf{a}.\exists \boldsymbol{X}.G \wedge \neg A$. A *(finite) counterexample* is a closed substitution $\theta$ providing values for $\boldsymbol{X}$ such that $\theta(G)$ is derivable, but the conclusion $\theta(A)$ is not. 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.*

In αCheck the reference implementation reads negation as *finite failure* (*not*):

$$\text{И}\mathsf{a}.\exists \boldsymbol{X}{:}\boldsymbol{\tau}.\ G \wedge gen[\![\boldsymbol{\tau}]\!](\boldsymbol{X}) \wedge not(A) \tag{2}$$

where $gen[\![\boldsymbol{\tau}]\!]$ are type-indexed predicates that *exhaustively* enumerate the inhabitants of $\boldsymbol{\tau}$. For example, $gen[\![\texttt{ty}]\!]$ yields the predicate:

```
gen_ty(intTy).          gen_ty(listTy).
gen_ty(funTy(T1,T2)) :- gen_ty(T1), gen_ty(T2).
```

A check such as (2) can simply be executed as a goal in the αProlog interpreter, using the number of resolution steps permitted to solve each subgoal as a bound on the search space. This method, combined with a complete search strategy such as iterative deepening, will find a counterexample, if one exists. This realization of specification checking is simple and effective, while not escaping the traditional problems associated with such an operational notion of negation.

### 4.1 Negation Elimination

Negation Elimination [3,26] is a source-to-source transformation that replaces negated subgoals with calls to a combination of equivalent positively defined predicates. In the absence of local (existential) variables, *NE* yields an ordinary (α)Prolog program, whose intended model is included in the complement of the model of the source program. In other terms, a predicate and its complement are mutually *exclusive*. *Exhaustivity*, that is whether a program and its complement coincide with the Herbrand base of the program's signature may or may not hold, depending on the decidability of the predicate in question; nevertheless, this property, though desirable, is neither frequent nor necessary in a model checking context. When local variables are present, the derived positivized program features the *extensional* universal quantifier presented in the previous section.

The generation of complementary predicates can be split into two phases: *term complementation* and *clause complementation.*

*Term Complementation.* A cause of atomic goal failure is when its arguments do not unify with any of the program clause heads in its definition. The idea is then to generate the complement of the term structure in each clause head by constructing a set of terms that differ in at least one position. However, and similarly to the higher-order logic case, the complement of a nominal term containing free or bound names cannot be represented by a *finite* set of nominal terms. For our application nonetheless, we can pre-process clauses so that the standard complementation algorithm for (linear) first order terms applies [19]. This forces terms in source clause heads to be linear and free of names (including swapping and abstractions), by replacing them with logical variables, and, in case they occurred in abstractions, by constraining them in the clause body by a *concretion* to a fresh variable. A concretion, written $t@a$, is the elimination form for abstractions and can be implemented by translating a goal $G$ with an occurrence of $[t@a]$ (notation $G[t@a]$) to $\exists X.t \approx \langle a\rangle X \wedge G[X]$. For example, the clause for typing lambdas is normalized as:

```
tc(G,lam(M,T),funTy(T,U)):- new x. tc([(x,T) |G],M@x,U).
```

Hence, we can use a type-directed version of first-order term complementation, $not[\![\tau]\!] : \tau \to \tau\ set$ and prove its correctness in term of *exclusivity* following [3,27]: the intersection of the set of ground instances of a term and its complement is empty. *Exhaustivity* also holds, but will not be needed. The definition of $not[\![\tau]\!]$ is in the appendix [10], but we offer the following example:

$$\begin{aligned}&not[\![\texttt{exp}]\!](\texttt{app}(\texttt{c}(\texttt{hd}),\_)) =\\&\{\texttt{lam}(\_,\_),\texttt{err},\texttt{c}(\_),\texttt{var}(\_),\texttt{app}(\texttt{c}(\texttt{tl}),\_),\texttt{app}(\texttt{c}(\texttt{nil}),\_),\texttt{app}(\texttt{c}(\texttt{toInt}(\_)),\_),\\&\texttt{app}(\texttt{var}(\_),\_),\texttt{app}(\texttt{err},\_),\texttt{app}(\texttt{lam}(\_,\_),\_),\texttt{app}(\texttt{app}(\_,\_),\_)\}\end{aligned}$$

*Clause Complementation.* The idea of the clause complementation algorithm is to compute the complement of each head of a predicate definition using term complementation, while clause bodies are negated pushing negation inwards until atoms are reached and replaced by their complement and the negation of constraints is computed. The contributions (in fact a disjunction) of each of the original clauses are finally merged. The whole procedure can be seen as a negation normal form procedure, which is consistent with the operational semantics of the language.

The first ingredient is complementing the equality and freshness constraints, yielding ($\alpha$-)inequality $neq[\![\tau]\!]$ and non-freshness $nfr[\![\nu, \delta]\!]$: we implement these

$$
\begin{array}{rclrcl}
not^G(\top) &=& \bot & not^D(\top) &=& \bot \\
not^G(\bot) &=& \top & not^D(\bot) &=& \top \\
not^G(p(t)) &=& p^\neg(t) & not^D(G \supset p(t)) &=& \bigwedge\{\forall(p^\neg(u)) \mid u \in not[\![\tau]\!](t)\} \wedge \\
 & & & & & (not^G(G) \supset p^\neg(t)) \\
not^G(t \approx_\tau u) &=& neq[\![\tau]\!](t,u) & & & \\
not^G(a \#_\tau u) &=& nfr[\![\nu,\tau]\!](a,u) & & & \\
not^G(G \wedge G') &=& not^G(G) \vee not^G(G') & not^D(D \wedge D') &=& not^D(D) \vee not^D(D') \\
not^G(G \vee G') &=& not^G(G) \wedge not^G(G') & not^D(D \vee D') &=& not^D(D) \wedge not^D(D') \\
not^G(\forall^* X{:}\tau.\ G) &=& \exists X{:}\tau.\ not^G(G) & not^D(\forall X{:}\tau.\ D) &=& \forall X{:}\tau.\ not^D(D) \\
not^G(\exists X{:}\tau.\ G) &=& \forall^* X{:}\tau.\ not^G(G) & & & \\
not^G(\reflectbox{N}\mathsf{a}{:}\nu.\ G) &=& \reflectbox{N}\mathsf{a}{:}\nu.\ not^G(G) & not^D(\Delta) &=& not^D(\mathrm{def}(p,\Delta))
\end{array}
$$

**Fig. 3.** Negation of a goal and of clause

using type-directed code generation within the αProlog interpreter and refer again to the appendix [10] for their generic definition.

Figure 3 shows goal and clause complementation: most cases of the former, *via* the $not^G$ function, are intuitive, being classical tautologies. Note that the self-duality of the И-quantifier allows goal negation to be applied recursively. Complementing existential goals is where we introduce *extensional* quantification and invoke its proof-theory.

Clause complementation is where things get interesting and differ from the previous algorithm [9]. The complement of a clause $G \supset p(t)$ must contain a "factual" part, built *via* term complementation, motivating failure due to clash with (some term in) the head. We obtain the rest by negating the body with $not^G(G)$. We take clause complementation *definition-wise*, that is the negation of a program is the conjunction of the negation of all its predicate definitions. An example may help: negating the typing clauses for constants and application (tc from Fig. 2) produces the following disjunction:

```
(not_tc(_,err,_) /\ not_tc(_,var(_),_) /\ not_tc(_,app(_,_),_) /\
 not_tc(_,lam(_,_),_) /\ not_tc(_,c(C),T):- neq(tcf(C), T))
 \/
(not_tc(_,err,_) /\ not_tc(_,var(_),_) /\ not_tc(_,c(_),_) /\
 not_tc(_,lam(_,_),_) /\
 not_tc(G,app(M,N),U):- forall* T. not_tc(G,M,funTy(T,U)) /\
 not_tc(G,app(M,N),U):- forall* T. not_tc(G,N,T))
```

Notwithstanding the top-level disjunction, we are *not* committing to any form of disjunctive logic programming: the key observation is that '$\vee$' can be restricted to a program constructor *inside* a predicate definition; therefore it can be eliminated by simulating unification in the definition:

$$(G_1 \supset Q_1) \vee (G_2 \supset Q_2) \equiv \theta(G_1 \wedge G_2 \supset Q_1)$$

where $\theta = \mathrm{mgu}(Q_1, Q_2)$. Because $\vee$ is commutative and associative we can perform this merging operation in any order. However, as with many bottom-up operations, merging tends to produce a lot of redundancies in terms of clauses that are instances of each other. We have implemented *backward* and *forward* subsumption [23], by using an extension of the $\alpha$Prolog interpreter itself to check entailment between newly generated clauses and the current database (and vice-versa). Despite the fact that this subsumption check is *partial*, because the current unification algorithm does not handle equivariant unification with mixed prefixes [25] and extensional quantification [8], it makes all the difference: the `not_is_err` predicate definition decreases from an unacceptable 128 clauses to a much more reasonable 18. The final definition of `not_tc` follows, where we (as in Prolog) use the semicolon as concrete syntax for disjunction in the body:

```
not_tc(_,c(C),T)              :- neq_ty(tcf(C),T).
not_tc([],var(_),_).
not_tc([(X,T)|G],var(X'),T') :- (neq_ty(T,T'); fresh_id(X,X')),
                                   not_tc(G,var(X'),T').
not_tc(G,app(M,N),U)          :- forall* T:ty. not_tc(G,M,funTy(T,U));
                                                 not_tc(G,N,T).
not_tc(G,app(M,N),listTy)     :- forall* T:ty. not_tc(G,M,funTy(T,listTy));
                                                 not_tc(G,N,T).
not_tc(G,app(M,N),intTy)      :- forall* T:ty. not_tc(G,M,funTy(T,intTy));
                                                 not_tc(G,N,T).
not_tc(_,lam(_),listTy).
not_tc(_,lam(_),intTy).
not_tc(G,lam(M,T),funTy(T,U)):- new x:id. not_tc([(x,T)|G],M@x,U).
```

Regardless of the presence of two subsumed clauses in the `app` case that our approach failed to detect, it is a big improvement in comparison to the 38 clauses generated by the previous algorithm [9]. And in exhaustive search, every clause counts.

Having synthesized the negation of the `tc` predicate, $\alpha$Check will use it internally while searching, for instance in the preservation check, for

$$\exists E.\exists T.\ \texttt{tc}([], E, T), \texttt{step}(E, E'), \texttt{not\_tc}([], E', T)$$

Soundness of clause complementation is crucial for the purpose of model checking; we again express it in terms of exclusivity. The proof follows the lines of [26].

**Theorem 1 (Exclusivity).** *Let $\mathcal{K}$ be consistent. It is not the case that:*

- $\Gamma; \Delta; \mathcal{K} \Rightarrow G$ *and* $\Gamma; not^D(\Delta); \mathcal{K} \Rightarrow not^G(G)$;
- $\Gamma; \Delta; \mathcal{K} \xrightarrow{D} Q$ *and* $\Gamma; not^D(\Delta); \mathcal{K} \xrightarrow{not^D(D)} not^G(Q)$.

## 5 Case Studies

We have chosen as case studies here the *Stlc* benchmark suite, introduced in Sect. 2, and an encoding of the Volpano et al. security type system [34], as suggested in [5]. For the sake of space, we report *at the same time* our comparison between the various forms of negation, in particular *NEs* vs. *NE*, *and* the other systems of reference, accordingly, PLT-Redex and Nitpick.

*PLT-Redex* [13] is an executable DSL for mechanizing semantic models built on top of *DrRacket*. Redex has been the first environment to adopt the idea of random testing a la QuickCheck for validating the meta-theory of object languages, with significant success [18]. As we have mentioned, the main drawbacks are the lack of support for binders and low coverage of test generators stemming from grammar definitions. The user is therefore required to write her own generators, a task which tends to be demanding.

The system where proofs and disproofs are best integrated is arguably Isabelle/HOL [4]. In the appendix [10] we report some comparison with its version of QuickCheck, but here we concentrate on *Nitpick* [5], a higher-order model finder in the *Alloy* lineage supporting (co)inductive definitions. Nitpick works translating a significant fragment of HOL into first-order relational logic and then invoking Alloy's SAT-based model enumerator. The tool has been used effectively in several case studies, most notably weak memory models for C++ [6]. It would be natural to couple Isabelle/HOL's QuickCheck and/or Nitpick's capabilities with *Nominal* Isabelle [33], but this would require strengthening the latter's support for computation with names, permutations and abstract syntax modulo $\alpha$-conversion. So, at the time of writing, $\alpha$Check is unique as a model checker for binding signatures and specifications.

All test have been performed under Ubuntu 15.4 on a Intel Core i7 CPU 870, 2.93 GHz with 8 GB RAM. We time-out the computation when it exceeds 200 seconds. We report 0 when the time is <0.01. These tests must be taken with a lot of salt: not only is our tool under active development but the comparison with the other systems is only roughly indicative, having to factor differences between logic and functional programming (PLT-Redex), as well as the sheer scale and scope of counter-examples search in a system such as Isabelle/HOL.

## 5.1 Head-to-Head with PLT-Redex

We first measure the amount of *time to exhaust the search space* (TESS) using the three versions of negations supported in $\alpha$Check, over a bug-free version of the *Stlc* benchmark for $n = 1, 2, \ldots$ up to the point where we time-out. This gives some indication of how much of the search space the three techniques explore, keeping in mind that what is traversed is very different in shape; hence the more reliable comparison is between *NE* and *NEs*. As the results depicted in Fig. 4 suggests,

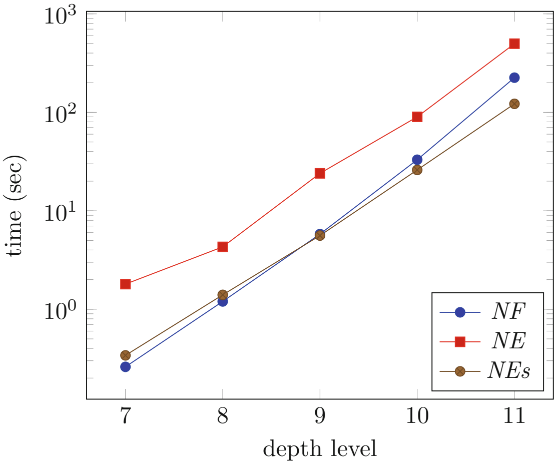

**Fig. 4.** Loglinear-plot of TESS on prog theorem

**Table 1.** TFCE on the *stlc* benchmark, Redex-style encoding

| bug | check | NF | NE | NEs | cex | Description/Class |
|---|---|---|---|---|---|---|
| 1 | pres | 0.3 (7) | 1 (7) | 0.37 (7) | $(\lambda x.xerr)n$ | range of function in app rule matched to the arg. (S) |
| | prog | 0 (5) | 3.31 (9) | 0.27 (5) | $hd\ n$ | |
| 2 | prog | 0.27 (8) | t.o. (11) | 85.3 (12) | $(cons\ n)\ nil$ | value $(cons\ v)\ v$ omitted (M) |
| 3 | pres | 0.04 (6) | 0.04 (6) | 0.3 (6) | $(\lambda x.n)m$ | order of types swapped in function pos of app (S) |
| | prog | 0 (5) | 3.71 (9) | 0.27 (8) | $hd\ n$ | |
| 4 | prog | t.o | t.o | t.o | ? | The type of cons is incorrect (S) |
| 5 | pres | t.o. (9) | t.o. (10) | 41.5 (10) | $tl\ ((cons\ n)\ err)$ | tail red. returns the head (S) |
| 6 | prog | 29.8 (11) | t.o. (11) | t.o. (12) | $hd\ ((cons\ n)\ nil)$ | hd red. on part. appl. cons (M) |
| 7 | prog | 1.04 (9) | 18.5 (10) | 1.1 (9) | $hd((\lambda x.err)n)$ | no eval for argument of app (M) |
| 8 | pres | 0.02 (5) | 0.03 (5) | 0.1 (5) | $(\lambda x.x)nil$ | lookup always returns int (U) |
| 9 | pres | 0 (5) | 0.02 (5) | 0.1 (5) | $(\lambda x.y)n$ | vars do not match in lookup (S) |

*NEs* shows a clear improvement over *NE*, while *NF* holds its ground, however hindered by the explosive exhaustive generation of terms.

However, our mission is finding counterexamples and so we compare the *time to find counterexamples* (TFCE) using *NF*, *NE*, *NEs* on the said benchmarks. We list in Table 1 the 9 mutations from the cited site. Every row describes the mutation inserted with an informal classification inherited from ibidem—(S)imple, (M)edium or (U)nusual, better read as artificial. We also list the counterexamples found by $\alpha$Check under *NF* (NE(s) being analogous but less instantiated) and the depths at which those are found or a time-out occurred.

The results in Table 1 show a remarkable improvement of *NEs* over *NE*, in terms of counter-examples that were timed-out (bug 2 and 5), as well as major speedups of more than an order of magnitude (bugs 3 (ii) and 7). Further, *NEs* never under-performs *NE*, probably because it locates counterexample at a lower depth. In rare occasions (bug 5 again) *NEs* even outperforms *NF* and in several cases it is comparable (bug 1, 3, 7, 8 and 9). Of course there are occasions (2 and 6), where *NF* is still dominant, as *NEs* counter-examples live at steeper depths (12 and 16, respectively) that cannot yet be achieved within the time-out.

We do not report TFCE of PLT-Redex, because, being based on randomized testing, what we really should measure is time spent *on average* to find a bug. The two encodings are quite different: Redex has very good support for evaluation contexts, while we use congruence rules. Being untyped, the Redex encoding treats *err* 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 [18], Redex allows the user to write certain judgments in a declarative style, provided they can be given a functional mode, but more complex systems, such as typing for a polymorphic version of a similar calculus, require very indirect encoding, e.g. CPS-style. We simulate addition on integers with numerals (omitted from the code snippets presented in Sect. 2 for the sake of space), as we currently require our code to be pure in the logical sense, as opposed to Redex that maps integers to Racket's ones. *W.r.t.* lines of code, the size of our encoding is roughly

1/4 of the Redex version, 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.

The performance of the negation elimination variants in this benchmark is not too impressive. However, if we adopt a different style of encoding (let's call it PCF, akin to what we used in [9]), where constructors such as `hd` are *not* treated as constants, but are first class, e.g.:

```
tc(G,hd(E),intTy)       :- tc(G,E,listTy).
step(hd(cons(H,Tl)), H) :- value(H),value(Tl).
```

then all counter-examples are found very quickly, as reported in Table 2. In bug 4, *NEs* struggles to get at depth 13: on the other hand PLT-Redex fails to find that very bug. Bug 6 as well as several counterexamples disappear as not well-typed. This improved efficiency may be due to the reduced amount of nesting of terms, which means lower depth of exhaustive exploration. This is not a concern for random generation and (compiled) functional execution as in PLT-Redex.

**Table 2.** TFCE on the *Stlc* benchmark, PCF-style encoding. *NEs* cex shown

| bug# | check | NF | NE | NEs | cex |
|---|---|---|---|---|---|
| 1 | pres | 0.05 (5) | 2.79 (5) | 0.04 (5) | $(\lambda x.hdx)N$ |
| 2 | prog | 0 (4) | 7.76 (9) | 0.8 (7) | *(cons N) nil* |
| 3 | pres | 0 (4) | 0.05 (4) | 0 (4) | $(\lambda x.nil)nil$ |
| 4 | prog | 0.15 (7) | t.o. (10) | 199.1 (12) | *N + (cons N nil)* |
| 5 | pres | 0(4) | 0.04 (4) | 0(4) | *tl (cons N) nil* |
| 7 | prog | 5.82 (9) | 151.2 (11) | 19.54. (10) | $(\lambda x.nil)(N+M)$ |
| 8 | pres | 0.01 (4) | 0.04 (4) | 0.1 (4) | $(\lambda x.x)nil$ |
| 9 | pres | 0 (4) | 0.04 (4) | 0.1 (4) | $(\lambda x.y)\ N$ |

## 5.2 Nitpicking Security Type Systems

To compare Nitpick with our approach, we use the security type system due to Volpano, Irvine and Smith [34], whereby the basic imperative language *IMP* is endowed with a type system that prevents information flow from private to public variables[2]. For our test, we actually selected the more general version of the type system formalized in [28], 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 Boolean expressions, the typing judgment $l \vdash c$ reads as "command $c$ does not contain any information flow to

[2] For an interesting case study regarding instead *dynamic* information flow and carried out in Haskell, see [17]. A large part of the paper is dedicated to the fine tuning of custom generators and shrinkers.

variables $< l$ and only safe flows to variables $\geq l$." Following [28], we call this system *syntax-directed.*

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

**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 $\sigma' \approx_{\leq l} \tau'$;

We extend this exercise by considering also a *declarative* version (*std*) $l \vdash_d c$ of the syntax directed system, where anti-monotonicity is taken as a primitive rule instead of an admissible one as in the previous system; finally we encode also a syntax-directed *termination-sensitive* (*stT*) version $l \vdash_{\Downarrow} c$, where non-terminating programs do not leak information and its declarative cousin (*stTd*) $l \vdash_{\Downarrow d} c$. We then insert some mutations in all those systems, as detailed in Table 3 and investigate whether the following equivalences among those systems still hold:

**st↔std** $l \vdash c$ iff $l \vdash_d c$ and **stT↔stTd** $l \vdash_{\Downarrow} c$ iff $l \vdash_{\Downarrow d} c$.

Again the experimental evidence is quite pleasing as far as *NE* vs. *NEs* goes, where the latter is largely superior (5 (ii), 1 (i), 7 (ii)). In one case *NEs* improves on *NF* (1 (ii)) and in general competes with it save for 4 (ii) and 5 (i) and (ii). To have an idea of the counterexamples found by $\alpha$Check, the command (`SKIP`; `x`:=`1`), *sec* `x` = `0`, `l` = `1` and state $\sigma$ mapping $x$ to 0 falsifies confinement 1 (i); in fact, this would not hold were the typing rule to check the second premise. A not too dissimilar counterexample falsifies non-interference 1 (ii): $c$ is (`SKIP`; `x`:=`y`), *sec* `x`, `y` = `0`, `1`, `l` = `0` and $\sigma$ maps $y$ to 0 and $x$ undefined

**Table 3.** $\alpha$Check vs. Nitpick on the Volpano benchmark suite. (sp) indicates that Nitpick produced a spurious counterexample.

| bug | check | Nitpick | NF | NE | NEs | Description |
|---|---|---|---|---|---|---|
| 1 | conf | (sp) | 0.03 (5) | 4.4 (8) | 2.1 (7) | second premise of seq rule omitted |
| | non-inter | t.o. | 9.13 (8) | 6.71 (8) | 6.1 (8) | ditto |
| 2 | non-inter | (sp) | 3.3 (8) | 2.1 (8) | 1.9 (8) | var swap in $\leq$ premise of assn rule |
| 3 | st→std | 0.95 | t.o | t.o | t.o | inversion of $\leq$ in antimono rule ditto |
| | std→st | 0.75 | 0.8 (7) | 0.3 (7) | 0.3 (7) | |
| 4 | st→std | | | | | $\leq$ assumption omitted in IF: **true** |
| | std→st | 1.3 | 0.9 (7) | t.o. | t.o | ditto |
| 5 | st→std | 5.1(sp) | 24.5 (11) | t.o | t.o | as 2 but on decl. version of the rule |
| | std→st | 1.1 | 0.2 (7) | t.o. | 24.6 (11) | ditto |
| 6 | stT→stTd | 5.1(sp) | t.o | t.o | t.o | as 2 but on term. version of the rule |
| | stTd→stT | 1.0 | 0.01 (5) | 0.32 (7) | 0.05 (6) | ditto |
| 7 | stT→stTd | | | | | as 4 but on term-decl. rule: **true** |
| | stTd→stT | 1.6 | 1.7 (8) | 12.5 (9) | 1.2(8) | ditto |

(i.e. to a logic variable), while $\tau$ maps $y$ to 1 and keeps $x$ undefined. We note in passing that here extensional quantification is indispensable, since ordinary generic quantification is unable to instantiate security levels so as to find the relevant bugs.

The comparison with Nitpick[3] is more mixed. On one hand Nitpick fails to find 1 (ii) within the timeout and in other four cases it reports *spurious* counterexamples, which on manual analysis turn out to be good. On the other it nails down, quite quickly, two other cases where $\alpha$Check fails to converge at all (3 (i), 6 (i)). This despite the facts that relations such as evaluations, $\vdash_d$ and $\vdash_{\Downarrow d}$, are reported not well founded requiring therefore a problematic unrolling.

The crux of the matter is that differently from Isabelle/HOL's mostly functional setting (except for inductive definition of evaluation and typing), our encoding is fully relational: states and security assignments cannot be seen as partial functions but are reified in association lists. Moreover, we pay a significant price in not being able to rely on built-in types such as integers, but have to deploy our clearly inefficient versions. This means that to falsify simple computations such as $n \leq m$, we need to provide a derivation for that failure. Finally, this case study does not do justice to the realm where $\alpha$Prolog excels, namely it does not exercise binders intensely: 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. Yet, we could not select more binding intensive examples due to the current difficulties with running Nitpick under *Nominal* Isabelle.

## 6 Conclusions and Future Work

We have presented a new implementation of the *NE* algorithm underlying our model checker $\alpha$Check and experimental evidence showing satisfying improvements *w.r.t.* the previous incarnation, so as to make it competitive with the *NF* reference implementation. The comparison with PLT-Redex and Nitpick, systems of considerable additional maturity, is also, in our opinion, favourable: $\alpha$Check is able to find similar counterexamples in comparable amounts of time; it is able to find some counterexamples that Redex or Nitpick respectively do not; and in no case does it report spurious counterexamples. Having said that, our comparison is at most just suggestive and certainly partial, as many other proof assistants have incorporated some notion of PBT, e.g. [29,31]. A notable absence here is a comparison with what at first sight is a close relative, the Bedwyr system [2], a logic programming engine that allows a form of model checking directly on syntactic expressions possibly containing binding. Since Bedwyr uses depth-first search, checking properties for infinite domains should be approximated by writing logic programs encoding generators for a finite portion of that model. Our initial experiments in encoding the *Stlc* benchmark in Bedwyr have failed to find any counterexample, but this could be imputed simply to

[3] Settings: `[sat_solver=MiniSat_JNI,max_threads=1,timeout=200]`.

our lack of experience with the system. Recent work about "augmented focusing systems" [16] could overcome this problem.

All the mutations we have inserted so far have injected faults in the specifications, not in the checks. This make sense for our intended use; however, it would be interesting to see how our tool would fare *w.r.t.* mutation testing of *theorems*.

*Exhaustive* term generation has served us well so far, but it is natural to ask whether *random* generation could have a role in $\alpha$Check, either by simply randomizing term generation under *NF* or more generally the logic programming interpreter itself, in the vein of [14]. More practically, providing generators and reflection mechanism for built-in datatypes and associated operators is a priority.

Finally, we would like to implement improvements in nominal equational unification algorithms, which would make subsumption complete, *via equivariant* unification [8], and more ambitiously introduce *narrowing*, so that functions could be computed rather then simulated relationally. In the long run, this could open the door to use $\alpha$Check as a light-weight model checker for (a fragment) of Nominal Isabelle.

## References

1. Aspinall, D., Beringer, L., Momigliano, A.: Optimisation validation. Electron. Notes Theor. Comput. Sci. **176**(3), 37–59 (2007)
2. Baelde, D., Gacek, A., Miller, D., Nadathur, G., Tiu, A.F.: The Bedwyr system for model checking over syntactic expressions. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 391–397. Springer, Heidelberg (2007)
3. Barbuti, R., Mancarella, P., Pedreschi, D., Turini, F.: A transformational approach to negation in logic programming. J. Log. Program. **8**, 201–228 (1990)
4. Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic proof and disproof in Isabelle/HOL. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 12–27. Springer, Heidelberg (2011)
5. Blanchette, J.C., Nipkow, T.: Nitpick: a counterexample generator for higher-order logic based on a relational model finder. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer, Heidelberg (2010)
6. Blanchette, J.C., Weber, T., Batty, M., Owens, S., Sarkar, S.: Nitpicking C++ concurrency. In: Schneider-Kamp, P., Hanus, M. (eds.) Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, pp. 113–124. ACM (2011)
7. Breitner, J.: Formally proving a compiler transformation safe. In: Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell, Haskell 2015, pp. 35–46. ACM, New York (2015)
8. Cheney, J.: Equivariant unification. J. Autom. Reasoning **45**(3), 267–300 (2010)
9. Cheney, J., Momigliano, A.: Mechanized metatheory model-checking. In: Leuschel, M., Podelski, A. (eds.) PPDP, pp. 75–86. ACM (2007)
10. Cheney, J., Momigliano, A., Pessina, M.: Appendix to Advances in property-based testing for $\alpha$Prolog (2016). http://momigliano.di.unimi.it/alphaCheck.html
11. Cheney, J., Urban, C.: Nominal logic programming. ACM Trans. Program. Lang. Syst. **30**(5), 26 (2008)

12. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), pp. 268–279. ACM (2000)
13. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. The MIT Press, Massachusetts (2009)
14. Fetscher, B., Claessen, K., Pałka, M., Hughes, J., Findler, R.B.: Making random judgments: automatically generating well-typed terms from the definition of a type-system. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 383–405. Springer, Heidelberg (2015)
15. Harland, J.: Success and failure for hereditary Harrop formulae. J. Log. Program. **17**(1), 1–29 (1993)
16. Heath, Q., Miller, D.: A framework for proof certificates in finite state exploration. In: Kaliszyk, C., Paskevich, A. (eds.) Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, 2–3 Aug 2015, vol. 186. EPTCS, pp. 11–26 (2015)
17. Hritcu, C., Hughes, J., Pierce, B.C., Spector-Zabusky, A., Vytiniotis, D., Azevedo de Amorim, A., Lampropoulos, L.: Testing noninterference, quickly. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, pp. 455–468. ACM, New York (2013)
18. Klein, C., Clements, J., Dimoulas, C., Eastlund, C., Felleisen, M., Flatt, M., McCarthy, J.A., Rafkind, J., Tobin-Hochstadt, S., Findler, R.B.: 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 2012, pp. 285–296. ACM, New York (2012)
19. Lassez, J.-L., Marriott, K.: Explicit representation of terms defined by counter examples. J. Autom. Reasoning **3**(3), 301–318 (1987)
20. Leach, J., Nieva, S., Rodríguez-Artalejo, M.: Constraint logic programming with hereditary Harrop formulas. TPLP **1**(4), 409–445 (2001)
21. Leroy, X.: Formal verification of a realistic compiler. CACM **52**(7), 107–115 (2009)
22. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Inf. Comput. **207**(2), 284–304 (2009)
23. Loveland, W.D., Nadathur, G.: Proof procedures for logic programming. Technical report, Durham, NC, USA (1994)
24. McKeeman, W.M.: Differential testing for software. Digit. Tech. J. **10**(1), 100–107 (1998)
25. Miller, D.: Unification under a mixed prefix. J. Symb. Comput. **14**(4), 321–358 (1992)
26. Momigliano, A.: Elimination of negation in a logical framework. In: Clote, P.G., Schwichtenberg, H. (eds.) CSL 2000. LNCS, vol. 1862, p. 411. Springer, Heidelberg (2000)
27. Momigliano, A., Pfenning, F.: Higher-order pattern complement and the strict lambda-calculus. ACM Trans. Comput. Log. **4**(4), 493–529 (2003)
28. Nipkow, T., Klein, G.: Concrete Semantics-with Isabelle/HOL. Springer, Heidelberg (2014)
29. Owre, S.: Random testing in PVS. In: Workshop on Automated Formal Methods (AFM) (2006)
30. Palka, M.H., Claessen, K., Russo, A., Hughes, J.: Testing an optimising compiler by generating random lambda terms. In: AST 2011, pp. 91–97. ACM (2011)
31. Paraskevopoulou, Z., Hritcu, C., Dénès, M., Lampropoulos, L., Pierce, B.C.: Foundational property-based testing. In: Urban, C., Zhang, X. (eds.) Interactive Theorem Proving. LNCS, vol. 9236, pp. 325–343. Springer, Heidelberg (2015)

32. Pitts, A.M.: Nominal logic, a first order theory of names and binding. Inf. Comput. **183**, 165–193 (2003)
33. Urban, C., Kaliszyk, C.: General bindings and alpha-equivalence in nominal Isabelle. Log. Methods Comput. Sci. **8**(2), 1–35 (2012)
34. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2–3), 167–187 (1996)
35. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: CompCertTSO: a verified compiler for relaxed-memory concurrency. J. ACM **60**(3), 22:1–22:50 (2013)
36. Yang, X., Chen, Y., Eide, E., Regehr, J.: Finding and understanding bugs in c compilers. In: PLDI 2011, pp. 283–294. ACM, New York (2011)