αCheck: a Mechanized Metatheory Model-checker

JAMES CHENEY∗
University of Edinburgh

ALBERTO MOMIGLIANO
DI, University of Milan

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

The problem of mechanically formalizing and proving metatheoretic properties of programming language calculi, type systems, operational semantics, and related formal systems has received considerable attention recently. However, the dual problem of searching for errors in such formalizations has attracted comparatively little attention. In this article, we present αCheck, a bounded model-checker for metatheoretic properties of formal systems specified using nominal logic. In contrast to the current state of the art for metatheory verification, our approach is fully automatic, does not require expertise in theorem proving on the part of the user, and produces counterexamples in the case that a flaw is detected. We present two implementations of this technique, one based on *negation-as-failure* and one based on *negation elimination*, along with experimental results showing that these techniques are fast enough to be used interactively to debug systems as they are developed.

KEYWORDS: nominal logic, model checking, counterexample search, negation elimination

1 Introduction

Much of modern programming languages research is founded on proving properties of interest by syntactic methods, such as cut elimination, strong normalization, or type soundness theorems (Pierce 2002). Convincing syntactic proofs are challenging to perform on paper for several reasons, including the presence of variable binding, substitution, and associated equational theories (such as α-equivalence in the λ-calculus and structural congruences in process calculi), the need to perform reasoning by simultaneous or well-founded induction on multiple terms or derivations, and the often large number of cases that must be considered. Paper proofs are believed to be unreliable due in part to the fact that they usually sketch only the essential part of the argument, while leaving out verification of the many subsidiary

∗ Supported by EPSRC grant GR/S63205/01 and a Royal Society University Research Fellowship while performing this research.

lemmas and side-conditions needed to ensure that all of the proof steps are correct and that all cases have been considered.

A great deal of attention, reinvigorated by the POPLMark Challenge (Aydemir et al. 2005), has been focused on the problem of *metatheory mechanization*, that is, formally verifying such properties using computational tools. Formal, machine-checkable proof is widely agreed to provide the highest possible standard of evidence for believing such a system is correct. However, all theorem proving/proof assistant systems that have been employed in metatheory verification (to name a few Twelf, Coq, Isabelle/HOL, HOL, Abella, Beluga) have steep learning curves; using them to verify the properties of a nontrivial system requires a significant effort even after the learning curve has been surmounted, because inductive theorem-proving is currently a brain-bound, not CPU-bound, process. Moreover, verification attempts provide little assistance in the case of an incorrect system, even though this is the common case during the development of such a system. Verification attempts can flounder due to either flaws in the system, mistakes on the user's part, or the need for new representations or proof techniques compatible with mechanized metatheory tools. Determining which of these is the case (and how best to proceed) is part of the arduous process of becoming a power user of a theorem-proving system.

These observations about formal verification are not new. They have long been used to motivate *model-checking* (Clarke et al. 2000). In model-checking, the user specifies the system and describes properties which it should satisfy; it is the computer's job to search for counterexamples or to determine that none exist. Although it was practical only for small finite-state systems when first proposed more than 30 years ago, improved techniques for searching the state space efficiently (such as *symbolic model checking*) have now made it feasible to verify industrial hardware designs. As a result, model checking has gained widespread acceptance in industry.

We argue that mechanically verified proof is neither the only nor always the most appropriate way of gaining confidence in the correctness of a formal system; moreover, it is almost never the most appropriate way to *debug* such a system, especially in early stages of development. This is certainly the case in the area of hardware verification, where model-checking has surpassed theorem-proving in industrial acceptance and applicability. For finite systems such as hardware designs, model checking is, in principle, able to either guarantee that the design is correct, or produce a concrete counterexample. Model-checking tools that are fully automatic can often leverage hardware advances more readily than interactive theorem provers that require human guidance. Model-checkers do not generally require as much expertise as theorem provers; once the model specification and formula languages have been learned, an engineer can formalize a design, specify desired properties, and let the system do the work. Researchers can (and have) focused on the orthogonal issue of representing and exploring the state space efficiently so that the answer is produced as quickly as possible. This separation of concerns has catalyzed great progress towards adoption of model-checking for real-world verification problems.

We advocate *mechanized metatheory model-checking* as a useful complement to established theorem-proving techniques for analyzing programming languages and related systems. Of course, such systems are usually infinite-state, so they cannot necessarily be verified through brute-force search techniques, but we can at least automate the search for counterexamples over bounded, but arbitrarily large, subsets of the search space. Such bounded model checking (failure to find a simple counterexample) provides a degree of confidence that a design is correct, albeit not as much confidence as full verification. Nevertheless, this approach shares other advantages of model-checking: it is CPU-bound, not brain-bound; it separates high-level specification concerns from low-level implementation issues; and it provides explicit counterexamples. Thus, bounded model checking is likely to be more helpful than verification typically is during the development of a system.

In this article we describe αCheck, a tool for checking desired properties of formal systems implemented in αProlog, a nominal logic programming language. Nominal logic programming combines the *nominal terms* and associated unification algorithm introduced by Urban et al. (2004) with *nominal logic* as explored by Pitts (2003), Gabbay and Cheney (2004) and Cheney (2016). In αProlog, many object languages can be specified using Horn clauses over abstract syntax trees with "concrete" names and binding modulo α-equivalence (Cheney and Urban 2008).

Roughly, the idea is to test properties/specifications of the form $H_1 \wedge \cdots \wedge H_n \supset A$ by searching *exhaustively* (up to a bound) for a substitution θ such that $\theta(H_1), \ldots, \theta(H_n)$ all hold but the conclusion $\theta(A)$ does 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*. We explore two approaches, starting with the standard *negation-as-failure* rule, known as *NAF* (Section 4). This choice inherits many of the positive characteristics of *NAF*, e.g. its implementation being simple and quite effective. However, it does not escape the traditional problems associated with an operational notion of negation, such as the need for full instantiation of all free variables before solving the negated conclusion and the presence of several competing semantics (three-valued completion, stable semantics etc. (Apt and Bol 1994)). The latter concern is significant because the semantics of negation as failure has not yet been investigated for nominal logic programming. As a radical solution to this impasse, we therefore adopt the technique of *negation elimination*, abridged as *NE* (Barbuti et al. 1990; Momigliano 2000), a source-to-source transformation replacing negated subgoals with calls to equivalent positively defined predicates (Section 5). In this way the resulting program is a *negation-free* αProlog program, possibly with a new form of universal quantification, which we call *extensional*. The net results brought by the disappearance of the issue of negation are the avoidance of the expensive term generation step needed to ground free variables, the recovery of a clean proof-theoretic semantics and the possibility of optimization of properties by goal reordering.

We maintain that our tool helps to find bugs in high-level specifications of programming languages and other calculi *automatically* and *effectively* (Section 2.2). The beauty of metatheory model checking is that, compared to other general forms of system validation, the properties that should hold are already given to the user/tester 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.

Our experience (Section 6) has been that while brute-force testing cannot yet find "deep" problems (such as the well-known unsoundness in old versions of ML involving polymorphism and references) by itself, it is extremely useful for eliminating "shallow" bugs such as typographical errors that are otherwise time-consuming and tedious to eliminate. This applies in particular to *regression* testing of specifications.

To sum up, the contributions of this paper are:

- the presentation of the idea of metatheory model-checking, as a complementary approach to the formal verification of properties of formal systems;
- the adaptation of negation elimination to a fragment of nominal logic programming, endowing αProlog with a sound and declarative notion of negation;
- the description of the α*Check* tool;
- an extensive set of experiments that show that the tool has encouraging performance and is immediately useful in the validation of the encoding of formal systems.

This paper is a major extension of our previous work (Cheney and Momigliano 2007), where we give full details about the correctness of the approach, we significantly enlarge the set of experiments and we give an extensive review of related work, which has notably expanded since the initial conference publication. In fact, the idea of using testing and counter-model generation alongside formal metatheory verification has, in the past few years, gone mainstream; this happened mainly by importing the idea of *property-based testing* pioneered by the QuickCheck system (Claessen and Hughes 2000) into environments for the specification of programming languages, e.g., *PLT-Redex* (Felleisen et al. 2009), or outright proof assistants such as Isabelle/HOL (Blanchette et al. 2011) and Coq (Paraskevopoulou et al. 2015). Our approach helped inspire some of these techniques, and remains complementary to most of them; we refer to Section 7 for a detailed comparison.

The structure of the remainder of the article is as follows. Following a brief introduction to αProlog, Section 2 presents αCheck at an informal, tutorial level. Section 3 introduces the syntax and semantics of a core language for αProlog, which we shall use in the rest of the article. Section 4 discusses a simple implementation of metatheory model-checking in αProlog based on negation-as-failure. Section 5 defines a negation elimination procedure for αProlog, including extensional universal quantification. Section 6 presents experimental results that show the feasibility and usefulness of metatheory model checking. Sections 7 and 8 discuss related and future work and conclude. Detailed proofs as well as the debugged code of the example in Section 2.2 can be found in the electronic appendix of this paper.

2 Tutorial example

2.1 αProlog background

We will specify the formal systems whose properties we wish to check, as well as the properties themselves, as Horn clause logic programs in αProlog (Cheney and

Urban 2008). αProlog is a logic programming language based on *nominal logic* and using *nominal terms* and their associated unification algorithm for resolution, just as Prolog is based on first-order logic and uses first-order terms and unification for resolution. Unlike ordinary Prolog, αProlog is typed; all constants, function symbols, and predicate symbols must be declared explicitly. We provide a brief review in this section and a more detailed discussion of a monomorphic core language for αProlog in Section 3; many more details, including examples illustrating how to map conventional notation for inference rules to αProlog and a detailed semantics, can be found in Cheney and Urban (2008). We provide further discussion of related work on nominal techniques in Section 7.

In αProlog, 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 name constant is generally a lower-case symbol that has not been declared as something else (such as a predicate or function symbol). Names can be used in *abstractions*, written `a\M` in programs. Abstractions are considered equal up to α-renaming of the bound name for the purposes of unification in αProlog. Thus, where one writes $\lambda x.M$, $\nu x.M$, etc. in a paper exposition, in αProlog one writes `lam(x\M)`, `nu(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, where one would write $x \notin FV(t)$ in a paper exposition, in αProlog one writes `x # t`.

Horn clause logic programs over these operations suffice to define a wide variety of core languages, type systems, and operational semantics in a convenient way. Moreover, Horn clauses can also be used as specifications of desired program properties, including basic lemmas concerning substitution as well as main theorems such as preservation, progress, and type soundness. We therefore consider the problem of checking *specifications*

```
#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. As a simple, running example, we consider the lambda-calculus with pairs, together with appropriate specifications of properties that one usually wishes to verify. The abstract syntax, substitution, static and dynamic semantics for this language are shown in Figure 1, and the αProlog encoding of the syntax of this language is shown in the first part of Figure 2.

Terms and substitution In contrast to other techniques such as higher-order abstract syntax, there is no built-in substitution operation in αProlog, so we must define it explicitly. Nevertheless, substitution can be defined declaratively, see Figure 2. For convenience, αProlog provides a function-definition syntax, but this is simply syntactic sugar for its relational implementation. Most cases are straightforward; the cases for variables and lambda-abstraction both use freshness subgoals to check that variables are distinct or do not appear fresh in other expressions.

$$\begin{array}{llll}
\text{Types} & A, B & ::= & \mathbf{1} \mid A * B \mid A \to B \\
\text{Terms} & M & ::= & x \mid \langle\rangle \mid \lambda x.\ M \mid M_1\ M_2 \mid \langle M_1, M_2\rangle \mid \mathit{fst}\ M \mid \mathit{snd}\ M \\
\text{Values} & V & ::= & \langle\rangle \mid \lambda x.\ M \mid \langle V_1\ \ V_2\rangle \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}$$

$$\begin{array}{rcll}
\langle\rangle\{M/x\} & = & \langle\rangle & \\
x\{M/x\} & = & M & \\
y\{M/x\} & = & y & (x \neq y) \\
(M_1 M_2)\{M/x\} & = & M_1\{M/x\} M_2\{M/x\} & \\
\langle M_1, M_2\rangle\{M/x\} & = & \langle M_1\{M/x\}, M_2\{M/x\}\rangle & \\
(\mathit{fst}\ M')\{M/x\} & = & \mathit{fst}\ (M'\{M/x\}) & \\
(\mathit{snd}\ M')\{M/x\} & = & \mathit{snd}\ (M'\{M/x\}) & \\
(\lambda y.M')\{M/x\} & = & \lambda y.M'\{M/x\} & (y \notin FV(x, M))
\end{array}$$

$$\frac{}{\vdash \langle\rangle : \mathbf{1}}\ \texttt{T-1} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \texttt{T-VAR} \qquad \frac{x \notin \Gamma \quad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B}\ \texttt{T-ABS}$$

$$\frac{\Gamma \vdash M_1 : A_1 \quad \Gamma \vdash M_2 : A_2}{\Gamma \vdash \langle M_1, M_2\rangle : A_1 * A_2}\ \texttt{T-PAIR} \qquad \frac{\Gamma \vdash M_1 : A \to B \quad \Gamma \vdash M_2 : A}{\Gamma \vdash M_1 M_2 : B}\ \texttt{T-APP}$$

$$\frac{\Gamma \vdash M : A_1 * A_2}{\Gamma \vdash \mathit{fst}\ M : A_1}\ \texttt{T-FST} \qquad \frac{\Gamma \vdash M : A_1 * A_2}{\Gamma \vdash \mathit{snd}\ M : A_2}\ \texttt{T-SND}$$

$$\frac{}{\lambda x : A.M\ V \leadsto M\{V/x\}}\ \texttt{E-ABS}$$

$$\frac{M_1 \leadsto M_1'}{M_1 M_2 \leadsto M_1' M_2}\ \texttt{E-APP1} \qquad \frac{M \leadsto M'}{V M \leadsto V M'}\ \texttt{E-APP2}$$

$$\frac{M_1 \leadsto M_1'}{\langle M_1, M_2\rangle \leadsto \langle M_1', M_2\rangle}\ \texttt{E-PAIR1} \qquad \frac{M \leadsto M'}{\langle V, M\rangle \leadsto V, M'}\ \texttt{E-PAIR2}$$

$$\frac{M \leadsto M'}{\mathit{fst}\ M \leadsto \mathit{fst}\ M'}\ \texttt{E-FST} \qquad \frac{M \leadsto M'}{\mathit{snd}\ M \leadsto \mathit{snd}\ M'}\ \texttt{E-SND}$$

$$\frac{}{\mathit{fst}\ \langle V_1, V_2\rangle \leadsto V_1}\ \texttt{E-FP} \qquad \frac{}{\mathit{snd}\ \langle V_1, V_2\rangle \leadsto V_2}\ \texttt{E-SP}$$

Fig. 1. Static and dynamic semantics of the λ-calculus with pairs

Despite these side-conditions, substitution is a total function on terms quotiented by α-equivalence; see Gabbay (2011) and Pitts (2013) for more details.

After the definition of the `sub` function, we have added some directives that state desired properties of substitution that we wish to check. First, the `sub_fun` property states that the result of substitution is uniquely defined. Since `sub` is internally translated to a relation in the current implementation, this is not immediate, so it should be checked. Second, `sub_id` checks that substituting a variable with itself has no effect. The `sub_fresh` property is the familiar lemma that substituting has no effect if the variable is not present in M; the last property `sub_sub` is a standard substitution commutation lemma.

Types and typechecking Next we turn to types and typechecking, shown in Figure 3. We introduce constructors for simple types, namely unit, pairing, and function types. The typechecking judgment is standard. In addition, we check some

```
id : name_type.
tm : type.
ty : type.

var  : id -> tm.
unit : tm.
app  : (tm,tm) -> tm.
lam  : id\tm -> tm.
pair : (tm,tm) -> tm.
fst  : tm -> tm.
snd  : tm -> tm.

func sub(tm,id,tm)    = tm.
sub(var(X),X,N)       = N.
sub(var(X),Y,N)       = var(Y) :- X # Y.
sub(app(M1,M2),Y,N)   = app(sub(M1,Y,N),sub(M2,Y,N)).
sub(lam(x\M),Y,N)     = lam(x\sub(M,Y,N)) :- x # (Y,N).
sub(unit,Y,N)         = unit.
sub(pair(M1,M2),Y,N)  = pair(sub(M1,Y,N),sub(M1,Y,N)).
sub(fst(M),Y,N)       = fst(sub(M,Y,M)).
sub(fst(M),Y,N)       = snd(sub(M,Y,N)).

#check "sub_fun"   5 :  sub(M,x,N) = M1, sub(M,x,N) = M2 => M1 = M2.
#check "sub_id"    5 :  sub(M,x,var(x)) = M.
#check "sub_fresh" 5 :  x # M => sub(M,x,N) = M.
#check "sub_sub"   5 :  x # N'
                     => sub(sub(M,x,N),y,N') = sub(sub(M,y,N'),x,sub(N,y,N')).
```

Fig. 2. αProlog specification of the λ-calculus: Terms and substitution

standard properties of typechecking, including weakening (`tc_weak`) and the substitution lemma (`tc_sub`). Note that since we are merely specifying, not proving, the substitution lemma, we do not have to state its general form. However, since contexts are encoded as lists of pairs of variables and types, to avoid false positives, we do have to explicitly define what it means for a context to be well-formed: contexts must *not* contain multiple bindings for the same variable. This is specified using the `wf_ctx` predicate.

Evaluation and soundness Now we arrive at the main point of this example, namely defining the operational semantics and checking that the type system is sound with respect to it, shown in Figure 4. We first define values, introduce one-step and multi-step call-by-value reduction relations, define the `progress` relation indicating that a term is not stuck, and specify type preservation (`tc_pres`), progress (`tc_prog`), and soundness (`tc_sound`) properties.

2.2 Specification checking

The alert reader may have noticed several errors in the programs in Figure 2 to Figure 4. In fact, *every* specification we have ascribed to it is violated. Some of

```
unitTy : ty.
==>    : ty -> ty -> ty.         infixr ==> 5.
**     : ty -> ty -> ty.         infixl ** 6.

type ctx = [(id,ty)].

pred wf_ctx(ctx).
wf_ctx([]).
wf_ctx([(X,T)|G]) :- X # G, wf_ctx(G).

pred tc(ctx,tm,ty).
tc([(V,T)|G],var(V), T).
tc(G,lam(x\E),T1 ==> T2) :- x # G, tc ([(x,T1)|G], E, T2).
tc(G,app(M,N),T)         :- tc(G,M,T ==> T0),
                            tc(G,N,T0).
tc(G,pair(M,N),T1 ** T2) :- tc(G,M,T1), tc(G,N,T2).
tc(G,fst(M),T1)          :- tc(G,M,T1 ** T2).
tc(G,snd(M),T1)          :- tc(G,M,T1 ** T2).
tc(G,unit,unitTy).

#check "tc_weak" 5 :  x # G, tc(G,E,T), wf_ctx(G) => tc([(x,T’)|G],E,T).
#check "tc_sub"  5 :  x # G, tc(G,E,T), tc([(x,T)|G],E’,T’), wf_ctx(G)
                   => tc(G,sub(E’,x,E),T’).
```

Fig. 3. αProlog specification of the λ-calculus: Types, contexts, and well-formedness

the bugs were introduced deliberately, others were discovered while debugging the specification using an early version of the tool. Before proceeding, the reader may wish to try to find all of these errors.

We now describe the results of a run of αCheck on the above program, using the negation-as-failure back end.[1] Complete source code for αProlog and running instructions for these examples can be found at `http://github.com/aprolog-lang/`.

First, consider the substitution specifications. αCheck produces the following (slightly sanitized) output for the first one:

```
Checking for counterexamples to
sub_fun: sub(M,x,N) = M1, sub(M,x,N) = M2 => M1 = M2
Checking depth 1 2
Counterexample found:
M =  fst(var(x))
M1 = fst(var(x))
M2 = snd(var(V))
N =  var(V)
```

The first error is due to the following bug:

```
sub(fst(M),Y,N) = snd(sub(M,Y,N))
```

should be

[1] Negation elimination finds somewhat different counter-examples, as we discuss in Section 6.

```
pred value(tm).
value(lam(_)).
value(unit).
value(pair(V,W)) :- value(V),value(W).

pred step(tm,tm).
step(app(lam(x\M),N),sub(N,x,M))  :- value(N).
step(app(M,N),app(M',N))          :- step(M,M').
step(app(V,N),app(V,N'))          :- value(V), step(N,N').
step(pair(M,N),pair(M',N))        :- step(M,M').
step(pair(V,N),pair(V,N'))        :- value(V), step(N,N').
step(fst(M),fst(M'))              :- step(M,M').
step(fst(pair(V1,V2)),V1)         :- value(V1), value(V2).
step(snd(M),snd(M'))              :- step(M,M').
step(snd(pair(V1,V2)),V2)         :- value(V1), value(V2).

pred progress(tm).
progress(V) :- value(V).
progress(M) :- step(M,_).

pred steps(exp,exp).
steps(M,M).
steps(M,P) :- step(M,N), steps(N,P).

#check "tc_pres" 5  :  tc([],M,T), step(M,M') => tc([],M',T).
#check "tc_prog" 5  :  tc([],E,T) => progress(E).
#check "tc_sound" 5 :  tc([],E,T), steps(E,E')  => tc([],E',T).
```

Fig. 4. αProlog specification of the λ-calculus: Reduction, type preservation, progress, and soundness

```
sub(snd(M),Y,N) = snd(sub(M,Y,N))
```

The second specification also reports an error:

```
Checking for counterexamples to
sub_id: sub(M,x,var(x)) = M
Checking depth 1
Counterexample found:
M = var(V1)
x # V1
```

which appears to be due to the typo in the clause

```
sub(var(X),Y,N) = var(Y) :- X # Y.
```

which should be

```
sub(var(X),Y,N) = var(X) :- X # Y.
```

After fixing these errors, no more counterexamples are found for `sub_fun`, but we have

```
Checking for counterexamples to
sub_id: sub(M,x,var(x)) = M
Checking depth 1 2 3
Counterexample found:
M = pair(var(x),unit)
```

Looking at the relevant clauses, we notice that

```
sub(pair(M1,M2),Y,N) = pair(sub(M1,Y,N),sub(M1,Y,N)).
```

should be

```
sub(pair(M1,M2),Y,N) = pair(sub(M1,Y,N),sub(M2,Y,N)).
```

After this fix, the only remaining counterexample involving substitution is

```
Checking for counterexamples to
sub_id: sub(M,x,var(x)) = M
Checking depth 1 2 3
Counterexample found:
M = fst(lam(y\var(y)))
```

The culprit is this clause

```
sub(fst(M),Y,N) = fst(sub(M,Y,M)).
```

which should be

```
sub(fst(M),Y,N) = fst(sub(M,Y,N)).
```

Once these bugs have been fixed, the `tc_sub` property checks out, but `tc_weak` and `tc_pres` are still violated:

```
Checking for counterexamples to
tc_weak: x # G, tc(G,E,T), wf_ctx(G) => tc([(x,T’)|G],E,T)
Checking depth 1 2 3
Counterexample found:
E =  var(V)
G =  [(V,unitTy)]
T =  unitTy
T’ = unitTy ** unitTy
--------
Checking for counterexamples to
tc_pres: tc([],M,T), step(M,M’) => tc([],M’,T)
Checking depth 1 2 3 4
Counterexample found:
M =  app(lam(x\var(x)),unit)
M’ = var(V)
T =  unitTy
```

For `tc_weak`, of course we add to the too-specific clause

```
tc([(V,T)|G],var(V), T).
```

the clause

```
tc([_| G],var(V),T) :- tc(G,var(V),T).
```

For `tc_pres`, `M` should never have type-checked at type `T`, and the culprit is the application rule:

```
tc(G,app(M,N),T)        :- tc(G,M,T ==> T0),
                           tc(G,N,T0).
```

Here, the types in the first subgoal are backwards, and should be

```
tc(G,app(M,N),T)        :- tc(G,M,T0 ==> T),
                           tc(G,N,T0).
```

Some bugs remain after these corrections, but they are all detected by αCheck. In particular, the clauses

```
tc(G,snd(M),T1)  :- tc(G,M,T1 ** T2).
step(app(lam(x\M),N),sub(N,x,M)) :- value(N).
```

should be changed to

```
tc(G,snd(M),T2)  :- tc(G,M,T1 ** T2).
step(app(lam(x\M),N),sub(M,x,N)) :- value(N).
```

After making these corrections, none of the specifications produce counterexamples up to the depth bounds shown.

3 Core language

The implementation of αProlog features a number of high-level conveniences including parameterized types such as lists, polymorphism, function definition notation, and non-logical features such as negation-as-failure and the "cut" proof-search pruning operator. For the purposes of metatheory model-checking we consider only input programs within a smaller, better-behaved fragment for which the semantics (and accompanying implementation techniques) are well-understood (Cheney and Urban 2008). In particular, to simplify the presentation we consider only monomorphic, non-parametric types; for convenience, our implementation handles lists as a special case.

A *signature* $\Sigma = (\Sigma_D, \Sigma_N, \Sigma_P, \Sigma_F)$ consists of sets Σ_D and Σ_N of base data types δ, including a distinguished type o of *propositions*, and name types ν, respectively, along with a collection Σ_P of *predicate symbols* $p : \tau \to o$ together with one Σ_F of *function symbol* declarations $f : \tau \to \delta$. Here, types τ are formed according to the following grammar:

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

where $\langle \nu \rangle \tau$ classifies name-abstractions, $\delta \in \Sigma_D$ and $\nu \in \Sigma_N$. We consider constants of type δ to be function symbols of arity $\mathbf{1} \to \delta$.

Given a signature Σ, the language of *terms* over sets V of (logical) variables $X, Y, Z, \ldots$ and A of names $\mathsf{a}, \mathsf{b}, \ldots$ is defined by the following grammar:

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

π denotes a permutation over names, and $\pi \cdot X$ its *suspended* action on a logic variable X. Suspended identity permutations are often omitted; that is, we write X for $\mathsf{id} \cdot X$. The abstract syntax $\langle \mathsf{a} \rangle t$ corresponds to the concrete syntax `a\t` for name-abstraction. We say that a term is *ground* if it has no variables (but possibly does contain names), otherwise it is *non-ground* or *open*. These terms are precisely those used in the *nominal unification* algorithm of Urban et al. (2004), and we will reuse a number of definitions from that paper and from Cheney and Urban (2008); the reader is encouraged to consult those papers for further explanation and examples.

We define the action of a permutation π on a name as follows:

$$\begin{array}{rcl} \mathsf{id}(\mathsf{a}) & = & \mathsf{a} \\ ((\mathsf{a}\,\mathsf{b}) \circ \pi)(\mathsf{c}) & = & \begin{cases} \mathsf{b} & \pi(\mathsf{c}) = \mathsf{a} \\ \mathsf{a} & \pi(\mathsf{c}) = \mathsf{b} \\ \mathsf{c} & \mathsf{c} \notin \{\mathsf{a}, \mathsf{b}\} \end{cases} \end{array}$$

Note that these permutations have *finite support*, that is, the set of names a such that $\pi(\mathsf{a}) \neq \mathsf{a}$ is finite, so $\pi(-)$ is the identity function on all but finitely many names. This fact plays an important role in the semantics of nominal logic and αProlog programs.

The swapping operation is extended to act on *ground* terms as follows:

$$\begin{array}{rclcrcl} \pi \cdot \langle\rangle & = & \langle\rangle & \qquad & \pi \cdot f(t) & = & f(\pi \cdot t) \\ \pi \cdot \langle t, u \rangle & = & \langle \pi \cdot t, \pi \cdot u \rangle & & \pi \cdot \mathsf{a} & = & \pi(\mathsf{a}) \\ \pi \cdot \langle \mathsf{a} \rangle t & = & \langle \pi \cdot \mathsf{a} \rangle \pi \cdot t & & & & \end{array}$$

Nominal logic includes two atomic formulas, *equality* ($t \approx_\tau u$) and *freshness* ($s \ \#_\tau \ u$). In nominal logic programming, both are treated as constraints, and unification involves freshness constraint solving. The meaning of ground freshness constraints $\mathsf{a} \ \#_\tau \ u$, where a is a name and u is a ground term of type τ, is defined using the following inference rules, where $f : \tau \to \delta \in \Sigma_F$:

$$\frac{\mathsf{a} \neq \mathsf{b}}{\mathsf{a} \ \#_\nu \ \mathsf{b}} \quad \frac{}{\mathsf{a} \ \#_{\mathbf{1}} \ \langle\rangle} \quad \frac{\mathsf{a} \ \#_\tau \ t}{\mathsf{a} \ \#_\delta \ f(t)} \quad \frac{\mathsf{a} \ \#_{\tau_1} \ t_1 \quad \mathsf{a} \ \#_{\tau_2} \ t_2}{\mathsf{a} \ \#_{\tau_1 \times \tau_2} \ \langle t_1, t_2 \rangle}$$

$$\frac{\mathsf{a} \ \#_{\nu'} \ \mathsf{b} \quad \mathsf{a} \ \#_\tau \ t}{\mathsf{a} \ \#_{\langle \nu' \rangle \tau} \ \langle \mathsf{b} \rangle t} \quad \frac{}{\mathsf{a} \ \#_{\langle \nu' \rangle \tau} \ \langle \mathsf{a} \rangle t}$$

We define similarly the equality relation, which identifies abstractions up to "safe" renaming:

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

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

We adopt the convention to leave out the type subscript when it is clear from the context.

The Gabbay-Pitts *fresh-name* quantifier И, which, intuitively, quantifies over names not appearing in the formula (or in the values of its variables) can be defined in terms of freshness; that is, provided the free variables and name of ϕ are $\{\mathsf{a}, \vec{X}\}$, the formula $\text{И}\mathsf{a}{:}\nu.\phi(\mathsf{a})$ is logically equivalent to $\exists A{:}\nu. A \# \vec{X} \wedge \phi(A)$ (or, dually, $\forall A{:}\nu. A \# \vec{X} \supset \phi(A)$). However, as explained by Cheney and Urban (2008), we use И-quantified names directly instead of variables because they fit better with the nominal terms and unification algorithm of Urban et al. (2004). In αProlog programs, the И-quantifier is written `new`.

Given a signature, we consider *goal* and *(definite) program clause* formulas G and D, respectively, defined by the following grammar:

$$\begin{array}{rcl} E & ::= & t \approx u \mid t \# u \\ G & ::= & \bot \mid \top \mid E \mid p(t) \mid G \wedge G' \mid G \vee G' \mid \exists X{:}\tau.\, G \mid \text{И}\mathsf{a}{:}\nu.\, G \\ D & ::= & \top \mid p(t) \mid G \supset D \mid D \wedge D' \mid \forall X{:}\tau.\, D \end{array}$$

This fragment of nominal logic known as И-goal clauses, which disallows the И quantifier in the head of clauses, has been introduced in previous work (Cheney and Urban 2008) and resolution based on nominal unification has been shown sound and complete for proof search for this fragment. This is in contrast to the general case where the more complicated (and NP-hard) *equivariant unification* problem must be solved (Cheney 2010). For example, the clause

```
tc(G,lam(x\M),T ==> U) :- x # G, tc([(x,T)|G],M,U).
```

can be equivalently expressed as the following И-goal clause:

```
tc(G,lam(M),T ==> U) :- new x. \exists N. N = x\M, tc([(x,T)|G],N,U).
```

Although we permit programs to be defined using arbitrary (sets of) definite clauses Δ in И-goal form, we take advantage of the fact that such programs can always be *elaborated* (see discussion in Section 5.2 of Cheney and Urban (2008)) to sets of clauses of the form $\forall \vec{X}.\, G \supset p(t)$. It is also useful to single out in an elaborated program Δ all the clauses that belong to the definition of a predicate, $\mathrm{def}(p, \Delta) = \{D \mid D \in \Delta, D = \forall \vec{X}.\, G \supset p(t)\}$.

We define *contexts* Γ to be sequences of bindings of names or of variables:

$$\Gamma ::= \cdot \mid \Gamma, X{:}\tau \mid \Gamma \# \mathsf{a}{:}\nu$$

Note that names in closed formulas are always introduced using the И-quantifier; as such, names in a context are always intended to be fresh with respect to the values of variables and other names already in scope when introduced. For this reason, we write name-bindings as $\Gamma \# \mathsf{a}{:}\nu$, where the $\#$ symbol is a syntactic reminder that a must be fresh for other names and variables in Γ.

Terms are typed according to the following rules:

$$\frac{}{\Gamma \vdash \langle\rangle : \mathbf{1}} \quad \frac{\mathsf{a} : \nu \in \Gamma}{\Gamma \vdash \mathsf{a} : \nu} \quad \frac{X : \tau \in \Gamma \quad \Gamma \vdash \pi : \mathsf{perm}}{\Gamma \vdash \pi \cdot X : \tau} \quad \frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \langle t_1, t_2 \rangle : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash \mathsf{a} : \nu \quad \Gamma \vdash t : \tau}{\Gamma \vdash \langle \mathsf{a} \rangle t : \langle \nu \rangle \tau} \quad \frac{f : \tau \to \delta \in \Sigma \quad \Gamma \vdash t : \tau}{\Gamma \vdash f(t) : \delta}$$

The judgment $\Gamma \vdash \pi : \mathsf{perm}$ simply checks that all swappings in π involve names of the same type. The typing rules for goals and definite clauses are straightforward.

We write $\mathsf{T}^{\Sigma}_{\Gamma}[\![\tau]\!]$ for the set of all well-formed terms of type τ in signature Σ with variables assigned types as in Γ and likewise we write $\mathsf{G}^{\Sigma}_{\Gamma}$ and $\mathsf{D}^{\Sigma}_{\Gamma}$ for the sets of goals and respectively definite clauses formed with constants from Σ and variables from Γ.

We define *constraints* to be 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. Constraint-solving is modeled by the satisfiability judgment $\Gamma; \mathcal{K} \models C$. Let θ be a valuation, *i.e.* a function from variables to ground terms. We say that θ matches Γ (notation $\theta : \Gamma$) if $\theta(X) : \Gamma(X)$ for each X, and all of the freshness constraints implicit in Γ are satisfied, that is, if $\Gamma = \Gamma_1, X{:}\tau, \Gamma_2 \# \mathsf{a}{:}\nu, \Gamma_3$ then $\mathsf{a} \mathrel{\#} \theta(X)$, as formalized by the following three rules:

$$\frac{}{\theta : \cdot} \qquad \frac{\theta : \Gamma \quad \cdot \vdash \theta(X) : \tau}{\theta : \Gamma, X{:}\tau} \qquad \frac{\theta : \Gamma \quad \forall X \in \Gamma . \mathsf{a} \mathrel{\#} \theta(X)}{\theta : \Gamma \# \mathsf{a}{:}\nu}$$

Define satisfiability for valuations as follows:

$$\begin{array}{rcl} \theta \models \top & & \\ \theta \models t \approx u & \Leftrightarrow & \theta(t) \approx \theta(u) \\ \theta \models t \mathrel{\#} u & \Leftrightarrow & \theta(t) \mathrel{\#} \theta(u) \\ \theta \models C \wedge C' & \Leftrightarrow & \theta \models C \text{ and } \theta \models C' \\ \theta \models \exists X{:}\tau.\, C & \Leftrightarrow & \text{for some } t : \tau,\ \theta[X := t] \models C \\ \theta \models \text{И}\mathsf{a}{:}\nu.\, C & \Leftrightarrow & \text{for some } \mathsf{b} \mathrel{\#} (\theta, C),\ \theta \models C[\mathsf{b}/\mathsf{a}] \end{array}$$

Then we say that $\Gamma; \mathcal{K} \models C$ holds if for all $\theta : \Gamma$ such that $\theta \models \mathcal{K}$, we have $\theta \models C$.

Efficient algorithms for constraint solving and unification for nominal terms of the above form and for freshness constraints of the form $\mathsf{a} \mathrel{\#} t$ were studied by Urban et al. (2004). Note, however, that we also consider freshness constraints of the form $\pi{\cdot}X \mathrel{\#} \pi'{\cdot}Y$. These constraints are needed to express the α-inequality predicate *neq* (see Figure 10 in Section 5.2). Constraint solving and satisfiability become NP-hard in the presence of these constraints (Cheney 2010). In the current implementation of αProlog, such constraints are delayed until the end of proof search, and any remaining ones of the form $\pi \cdot X \mathrel{\#} \pi' \cdot X$ are checked for consistency by brute force, as these are essentially finite domain constraints. Any remaining constraint $\pi \cdot X \mathrel{\#} \pi' \cdot Y$, where X and Y are distinct variables, is always satisfiable.

We adapt here the "amalgamated" proof-theoretic semantics of αProlog programs, introduced in (Cheney and Urban 2008), based on previous techniques stemming from CLP (Leach et al. 2001) — see Figure 5. This semantics allows us to focus on the high-level proof search issues, without requiring us to introduce or manage low-level operational details concerning constraint solving. Differently from the cited paper, we use a single backchaining-based judgment $\Gamma; \Delta; \mathcal{K} \Rightarrow G$, where Δ is our (fixed and elaborated) program and $\mathcal{K}$ a set of constraints, rather than the partitioning of goal-directed or *uniform* proof search, and program clause-directed or *focused* proof search (Miller et al. 1991). This style of judgment conforms

$$\dfrac{}{\Gamma;\Delta;\mathcal{K} \Rightarrow \top}\,\top R \qquad \dfrac{\Gamma;\mathcal{K} \models E}{\Gamma;\Delta;\mathcal{K} \Rightarrow E}\,\mathit{con} \qquad \dfrac{\Gamma;\Delta;\mathcal{K} \Rightarrow G_1 \quad \Gamma;\Delta;\mathcal{K} \Rightarrow G_2}{\Gamma;\Delta;\mathcal{K} \Rightarrow G_1 \wedge G_2}\,\wedge R$$

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

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

$$\dfrac{\Gamma;\mathcal{K} \models \exists \vec{X}{:}\vec{\tau}.\, \vec{C} \wedge t \approx u \quad \Gamma, \vec{X}{:}\vec{\tau};\Delta;\mathcal{K}, \vec{C} \Rightarrow G \quad (\forall \vec{X}{:}\vec{\tau}.\, G \supset p(t)) \in \Delta}{\Gamma;\Delta;\mathcal{K} \Rightarrow p(u)}\,\mathit{back}$$

Fig. 5. Proof search semantics of αProlog programs with backchaining

$$\dfrac{J_1 \quad \dfrac{J_2 \quad \dfrac{J_3 \quad \dfrac{\dfrac{\Gamma_3; C_3 \models M \approx \langle \mathsf{y} \rangle N}{\Gamma_3; \Delta; C_3 \Rightarrow M \approx \langle \mathsf{y} \rangle N}\,\mathit{con} \quad \dfrac{J_4 \quad \dfrac{}{\Gamma_4; \Delta; C_4 \Rightarrow \top}\,\top R}{\Gamma_3; \Delta; C_3 \Rightarrow \mathit{tc}((y, T) :: G, N, U)}\,\mathit{back}}{\Gamma_3; \Delta; C_3 \Rightarrow M \approx \langle \mathsf{y} \rangle N \wedge \mathit{tc}((y, T) :: G, N, U)}\,\wedge R}{\Gamma_2; \Delta; C_2 \Rightarrow \exists N.M \approx \langle \mathsf{y} \rangle N \wedge \mathit{tc}((y, T) :: G, N, U)}\,\exists R}{\Gamma_1; \Delta; C_1 \Rightarrow \mathsf{И}\mathsf{y}.\exists N.M \approx \langle \mathsf{y} \rangle N \wedge \mathit{tc}((y, T) :: G, N, U)}\,\mathsf{И}R}{\cdot; \Delta; \cdot \Rightarrow \mathit{tc}([], \mathit{lam}(\langle \mathsf{x} \rangle \mathit{var}(\mathsf{x})), A \Rightarrow A)}\,\mathit{back}$$

where:

$$\begin{array}{lcl}
J_1 & = & \cdot;\cdot \models \exists G, M, T, U.C_1 \wedge E_1 \\
\Gamma_1 & = & G : \mathtt{ctx}, M : \mathtt{tm}, T : \mathtt{ty}, U : \mathtt{ty} \\
E_1 & = & \mathit{tc}(G, \mathit{lam}(M), T \Rightarrow U) \approx \mathit{tc}([], \mathit{lam}(\langle \mathsf{x} \rangle \mathit{var}(\mathsf{x})), A \Rightarrow A) \\
C_1 & = & G = [] \wedge M = \langle \mathsf{x} \rangle \mathit{var}(\mathsf{x}) \wedge T = A \wedge U = A \\
J_2 & = & \Gamma_1; C_1 \models \mathsf{И}\mathsf{y}.\top \\
\Gamma_2 & = & \Gamma_1\#\mathsf{y} \\
C_2 & = & C_1, \top \\
J_3 & = & \Gamma_1; C_1 \models \exists N.N = \mathit{var}(\mathsf{y}) \\
\Gamma_3 & = & \Gamma_2, N : \mathtt{tm} \\
C_3 & = & C_2, N = \mathit{var}(\mathsf{y}) \\
J_4 & = & \Gamma_3; C_1, C_2 \models \exists G', X, T'.C_3 \wedge E_2 \\
\Gamma_4 & = & \Gamma_3, G' : \mathtt{ctx}, X : \mathtt{id}, T' : \mathtt{tm} \\
C_4 & = & X = \mathsf{y} \wedge T' = U \\
E_4 & = & \mathit{tc}((X, T') :: G', \mathit{var}(X), T') \approx \mathit{tc}((\mathsf{y}, T) :: G, N, U)
\end{array}$$

Fig. 6. Partial derivation of goal $\mathit{tc}([], \mathit{lam}(\langle \mathsf{x} \rangle \mathit{var}(\mathsf{x})), A \Rightarrow A)$

better to the proof techniques required to proving the correctness of the negation elimination transformation (see Section 5).

Figure 6 shows the derivation of the goal $\mathit{tc}([], \mathit{lam}(\langle \mathsf{x} \rangle \mathit{var}(\mathsf{x})), A \Rightarrow A)$, illustrating how the rules in Figure 5 work. These rules are highly nondeterministic, requiring choices of constraints in the $\exists R$, $\mathsf{И}R$ and backchaining rules. The choice of constraint in the backchaining rule typically corresponds to the unifier, while constraints introduced in the $\exists R$ and $\mathsf{И}R$ rules correspond to witnessing substitutions or freshness assumptions. These choices are operationalized in αProlog using

$$
\begin{array}{rcl}
gen\llbracket\tau\rrbracket & : & \mathsf{T}^{\Sigma}_{\Gamma}\llbracket\tau\rrbracket \to \mathsf{G}^{\Sigma}_{\Gamma} \\
gen\llbracket\mathbf{1}\rrbracket(t) & = & t \approx \langle\rangle \\
gen\llbracket\tau_1 \times \tau_2\rrbracket(t) & = & \exists X_1{:}\tau_1, X_2{:}\tau_2.\, t \approx \langle X_1, X_2\rangle \wedge gen\llbracket\tau_1\rrbracket(X_1) \wedge gen\llbracket\tau_2\rrbracket(X_2) \\
gen\llbracket\delta\rrbracket(t) & = & gen_{\delta}(t) \\
gen\llbracket\langle\nu\rangle\tau\rrbracket(t) & = & \text{И}\mathsf{a}{:}\nu.\exists X{:}\tau.\, t \approx \langle\mathsf{a}\rangle X \wedge gen\llbracket\tau\rrbracket(X) \\
gen\llbracket\nu\rrbracket(t) & = & \top \\
gen_{\delta}(t) & :- & \bigvee\{\exists X{:}\tau.\, t \approx f(X) \wedge gen\llbracket\tau\rrbracket(X) \mid f : \tau \to \delta \in \Sigma\}
\end{array}
$$

Fig. 7. Term-generator predicates

nominal unification and resolution in the operational semantics given by Cheney and Urban (2008), to which we refer for more explanation.

4 Specification checking *via* negation-as-failure

The `#check` specifications correspond to specification formulas of the form

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

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 a formula (1) is of the form $\text{И}\vec{\mathsf{a}}.\exists\vec{X}.\, G \wedge \neg A$. A *(finite) counterexample* is a closed substitution θ providing values for $\vec{X}$ that satisfy this formula using negation-as-failure: that is, such that $\theta(G)$ is derivable, but the conclusion $\theta(A)$ finitely fails.

We define the *bounded model checking* problem for such programs and properties as follows: given a resource bound (*e.g.* a bound on the sizes of counterexamples or number of inference steps needed), decide whether a counterexample can be derived using the given resources, and if so, compute such a counterexample.

To begin with, we consider two approaches to solving this problem using *negation-as-failure* (*NAF*). First, we could simply enumerate all possible *valuations* and test them using *NAF* . More precisely, given predicates $gen\llbracket\tau\rrbracket : \tau \to o$ for each type τ (see Figure 7), which generate all possible values of type τ, we may translate a specification of the form (1) to a goal

$$\text{И}\vec{\mathsf{a}}.\exists\vec{X}{:}\vec{\tau}.\, gen\llbracket\tau_1\rrbracket(X_1) \wedge \cdots \wedge gen\llbracket\tau_m\rrbracket(X_m) \wedge G \wedge not(A) \tag{2}$$

where $not(A)$ is the ordinary negation-as-failure familiar from Prolog. In fact, we only need to generate ground values for the free variables of A, to ensure that negation-as-failure is well-behaved, since we can push the existential quantifiers of any variables mentioned only in G into G. Such a goal can simply be executed 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. However, this is clearly wasteful, as it involves premature commitment to ground instantiations. For example, if we have

$$gen\llbracket\tau\rrbracket(X), gen\llbracket\tau\rrbracket(Y), bar(Y), foo(X), not(baz(X, Y))$$

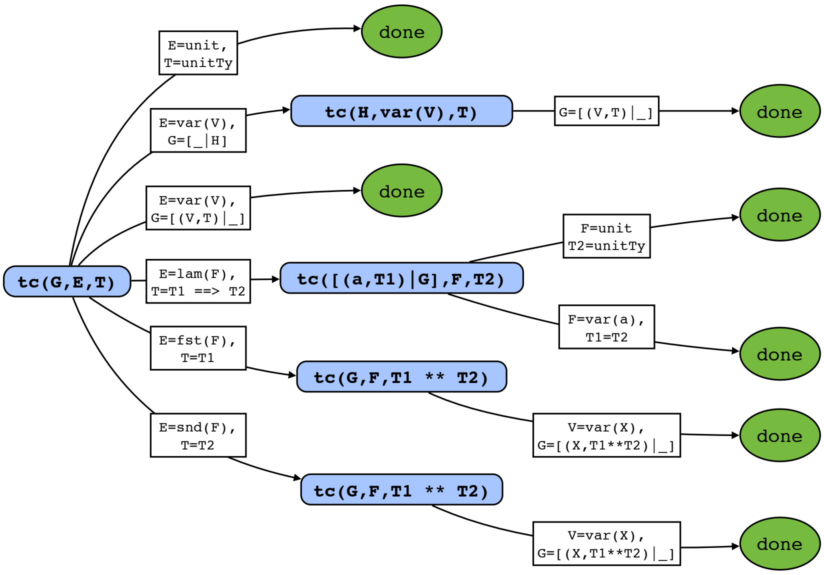

Fig. 8. “Finished” derivations for `tc(G,E,T)` up to depth 3

and we happen to generate an X that just does not satisfy $foo(X)$, we will still search all of the possible instantiations of Y and derivations of $bar(Y)$ up to the depth bound before trying a different instantiation of X. Instead, it is more efficient to use the *definitions* of *foo* and *bar* to guide search towards suitable instantiations of X and Y. Therefore we consider an approach that first enumerates *derivations* of the hypotheses and then tests whether the negated conclusion is satisfiable under the resulting answer constraint. Compared with the ground substitution enumeration technique above, this *derivation-first* approach simply delays the *gen* predicates until after the hypotheses:

$$\reflectbox{N}\vec{a}.\exists\vec{X}{:}\vec{\tau}.\, G \wedge gen[\![\tau]\!](X_1) \wedge \cdots \wedge gen[\![\tau]\!](X_n) \wedge not(A) \tag{3}$$

Of course, if G is a complex goal, the order in which we solve its subgoals can also affect search speed, but we leave this choice in the hands of the user in the current implementation.

In essence, this derivation-first approach generates all “finished” derivations of the hypothesis G up to a given depth, considers all sufficiently ground instantiations of variables in each up to the depth bound, and finally tests whether the conclusion finitely fails for the resulting substitution. A finished derivation is the result of performing a finite number of resolution steps on a goal formula in order to obtain a goal containing only equations and freshness constraints. For example, the proof search tree in Figure 8 shows all of the finished derivations of $tc(G, E, T)$ using at most 3 resolution steps. Here, the conjunction of constraint formulas along a path through the tree describes the solution corresponding to the path.

We note in passing that the dichotomy between the two approaches above corresponds to the well-known problem that property-based systems such as QuickCheck face when trying to test conjectures with hard-to satisfy premises — and this is especially acute when random testing is used. The derivation-first approach is a very simple rendering of the idea of *smart generators* (Bulwahn 2012a), thanks to the fact that we are already living in a logic programming world — we discuss this further in Section 7.

The $gen[\![\tau]\!]$ predicates are implemented as a built-in generic function in αProlog: given a `#check` directive $\text{И}\vec{\mathsf{a}}.\forall\vec{X}.\, G \supset A$, the interpreter generates predicates gen_δ for the (user-defined) datatypes δ over which the free variables of A range. Note that we do not exhaustively instantiate base types such as name-types; instead, we just use a fresh variable to represent an unknown name. This appears to behave correctly, but we do not have a proof of correctness.

The implementation of counterexample search using negation-as-failure described in this section still has several disadvantages:

- Negation-as-failure is unsound for non-ground goals, so we must sooner or later blindly instantiate all free variables before solving the negated conclusion[2]. This may be expensive, as we have argued before, and prevents optimizations by goal reordering. For an analogy, *NE* is to *NAF* as symbolic evaluation is to standard (ground) testing in property-based testing, see Section 7.2.
- Proving soundness (and completeness) of counterexample search, particularly with respect to names, requires proving properties of negation-as-failure in αProlog that have not yet been studied.
- Nested negations are not well-behaved, so we cannot use negation (nor, of course, if-then-else) in "pure" programs or specifications we wish to check.

Notwithstanding years of research, *NAF* (and an unsound version of it, by the way) is the negation operator offered by Prolog. However, we are not interested in programming, but in disproving conjectures and therefore relying on an operational interpretation of negation seems sub-optimal.

We therefore consider an alternative approach, which, almost paradoxically, addresses the issue of negation in logic programming by *eliminating* it.

5 Specification checking *via* negation elimination

Negation elimination (*NE*) (Barbuti et al. 1990; Momigliano 2000; Muñoz-Hernández et al. 2004) is a source-to-source transformation aimed at replacing negated subgoals with calls to equivalent positively defined predicates. *NE* by-passes complex semantic and implementation issues arising for *NAF* since, in the absence of local (existential) variables, it yields an ordinary (α)Prolog program, whose success set is included in the complement of the success set of the original predicate that occurred negatively. In other terms, a predicate and its complement are mutually *exclusive*. Moreover, for terminating programs we also expect *exhaustivity*: that

[2] As well known, this can be soundly weakened to checking for bindings of the free variables of the goal upon a successful derivation of the latter.

is, either the original predicate or its negation will succeed on a given input — of course, we cannot expect this for arbitrary programs that may denote sets whose complement is not recursively enumerable. When local variables are present, the derived program will also feature a form of *extensional* universal quantification, as we detail in Section 5.2.

We begin by summarizing how negation elimination works at a high level. Replacing occurrences of negated predicates with positive ones that are operationally equivalent entails two phases:

- *Complementing (nominal) terms.* One reason an atom can fail is when its arguments do not unify with any clause head in its definition. To exploit this observation, we pre-compute the complement of the term structure in each clause head by constructing a set of terms that differ in at least one position. This is known as the *(relative) complement* problem (Lassez and Marriott 1987), which we describe next in Section 5.1.
- *Complementing (nominal) clauses.* 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 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 clause complementation algorithm is described in Section 5.2.

5.1 Term complement

An open term t in a given signature can be seen as the intensional representation of the set of its ground instances. Accordingly, the *complement* of t is the set of ground terms which are *not* instances of t.

A complement operation satisfies the following desiderata: for fixed t, and all ground terms s

1. Exclusivity: it is not the case that s is both a ground instance of t and of its complement.
2. Exhaustivity: s is a ground instance of t or s is a ground instance of its complement.

As it was initially observed in (Lassez and Marriott 1987), this cannot be achieved unless we restrict to *linear* terms, *viz.* such that they have no repeated occurrences of the same logic variables. However, this restriction is immaterial for our intended application, thanks to *left-linearization*, a simple source to source transformation, where we replace repeated occurrence of the same variable in a clause head with fresh variables that are then constrained in the body by $\approx$.

Complementing nominal terms, however, introduces new and more significant issues, similarly to the higher-order case. There, in fact, even restricting to patterns, (intuitionistic) lambda calculi are not closed under complementation, due

$$\begin{array}{rcl} \mathit{not}[\![\tau]\!] & : & \mathsf{T}^{\Sigma}_{\Gamma}[\![\tau]\!] \to \mathcal{P}(\mathsf{T}^{\Sigma}_{\Gamma}[\![\tau]\!]) \\ \mathit{not}[\![\tau]\!](t) & = & \emptyset \qquad\qquad\qquad \text{when } \tau \in \{\mathbf{1}, \nu, \langle\nu\rangle\tau\} \text{ or } t \text{ is a variable} \\ \mathit{not}[\![\tau_1 \times \tau_2]\!](t_1, t_2) & = & \{(s_1, \_) \mid s_1 \in \mathit{not}[\![\tau_1]\!](t_1)\} \cup \{(\_, s_2) \mid s_s \in \mathit{not}[\![\tau_2]\!](t_2)\} \\ \mathit{not}[\![\delta]\!](f(t)) & = & \{g(\_) \mid g \in \Sigma, g : \sigma \to \delta, f \;\neq g\} \cup \{f(s) \mid s \in \mathit{not}[\![\tau]\!](t)\} \end{array}$$

Fig. 9. Term complement

the presence of *partially applied* lambda terms. Consider a higher-order pattern `(lam [x] E)` in Twelf's concrete syntax, where the logic variable `E` *does not* depend on `x`. Its complement contains all the functions that *must* depend on `x`, but this is not directly expressible with a finite set of patterns. This problem is solved by developing a strict lambda calculus, where we can directly express whether a function depends on its argument (Momigliano and Pfenning 2003). Although we do not consider logical variables at function types in αProlog, the presence of names, abstractions, and swappings leads to a similar problem. Indeed, consider the complement of say `lam(x\var(x))`: it would contain terms of the form `lam(x\var(Y))` such that `x # Y`. This means that the complement of a term (containing free or bound names) cannot again be represented by a finite set of nominal terms. A possible solution is to embrace the (constraint) disunification route and this means dealing (at least) with equivariant unification; this is not an attractive option since equivariant unification has high computational complexity as shown in (Cheney 2010). As far as negation elimination is concerned, it is simpler to further restrict И-goal clauses to a fragment that is term complement-closed: require terms in the heads of source program clauses to be linear and also forbid occurrence of names (including swapping and abstractions) in clause heads. These are replaced by logic variables appropriately constrained in the clause body by a *concretion* to a fresh name. A concretion, written $t \mathbin{@} \mathsf{a}$, is the elimination form for abstraction. Concretions need not be taken as primitives, since they can be implemented by translating $G[t \mathbin{@} \mathsf{a}]$ to $\exists X. t \approx \langle\mathsf{a}\rangle X \wedge G[X]$. However, we will *not* expand their definition during negation elimination — this would introduce pointless existential variables that would be turned into extensional universal quantifiers as we explain in the next Section 5.2.

For example, the И-goal clause:

```
tc(G,lam(M),T ==> U) :- new x. exists Y. M = x\Y, tc([(x,T)|G],Y,U).
```

can instead be written as follows:

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

avoiding an explicit existential quantifier in the body of the clause. Thus, we can simply use a type directed functional version of the standard rules for first-order term complementation, listed in Figure 9, where $f : \tau \to \delta$.

The correctness of the algorithm, analogously to previous results (Barbuti et al. 1990; Momigliano and Pfenning 2003), can be stated in the following constraint-conscious way, as required by the proof of the main Theorem 1:

$$
\begin{array}{rcl}
neq[\![\tau]\!] & : & \mathsf{T}^{\Sigma}_{\Gamma}[\![\tau]\!] \times \mathsf{T}^{\Sigma}_{\Gamma}[\![\tau]\!] \to \mathsf{G}^{\Sigma}_{\Gamma} \\
neq[\![\mathbf{1}]\!](t,u) & = & \bot \\
neq[\![\tau_1 \times \tau_2]\!](t,u) & = & neq[\![\tau_1]\!](\pi_1(t),\pi_1(u)) \vee neq[\![\tau_2]\!](\pi_2(t),\pi_2(u)) \\
neq[\![\delta]\!](t,u) & = & neq_{\delta}(t,u) \\
neq[\![\langle\nu\rangle\tau]\!](t,u) & = & \text{И}\mathsf{a}{:}\nu.\, neq[\![\tau]\!](t \mathbin{@} \mathsf{a}, u \mathbin{@} \mathsf{a}) \\
neq[\![\nu]\!](t,u) & = & t \mathrel{\#} u \\
neq_{\delta}(t,u) & :- & \bigvee\{\exists X, Y{:}\tau.\, t \approx f(X) \wedge u \approx f(Y) \wedge neq[\![\tau]\!](X,Y) \\
 & & \qquad \mid f : \tau \to \delta \in \Sigma\} \\
 & & \vee \bigvee\{\exists X{:}\tau, Y{:}\tau'.\, t \approx f(X) \wedge u \approx g(Y) \\
 & & \qquad \mid f : \tau \to \delta, g : \tau' \to \delta \in \Sigma, f \neq g\}
\end{array}
$$

$$
\begin{array}{rcl}
nfr[\![\nu,\tau]\!] & : & \mathsf{T}^{\Sigma}_{\Gamma}[\![\nu]\!] \times \mathsf{T}^{\Sigma}_{\Gamma}[\![\tau]\!] \to \mathsf{G}^{\Sigma}_{\Gamma} \\
nfr[\![\nu,\mathbf{1}]\!](a,t) & = & \bot \\
nfr[\![\nu,\tau_1 \times \tau_2]\!](a,t) & = & nfr[\![\nu,\tau_1]\!](a,\pi_1(t)) \vee nfr[\![\nu,\tau_2]\!](a,\pi_2(t)) \\
nfr[\![\nu,\delta]\!](a,t) & = & nfr_{\nu,\delta}(a,t) \\
nfr[\![\nu,\langle\nu'\rangle\tau]\!](a,t) & = & \text{И}\mathsf{b}{:}\nu'.\, nfr[\![\tau]\!](a, t \mathbin{@} \mathsf{b}) \\
nfr[\![\nu,\nu]\!](a,b) & = & a \approx b \\
nfr[\![\nu,\nu']\!](a,b) & = & \bot \quad (\nu \neq \nu') \\
nfr_{\nu,\delta}(a,t) & :- & \bigvee\{\exists X{:}\tau.\, t \approx f(X) \wedge nfr[\![\nu,\tau]\!](a,X) \mid f : \tau \to \delta \in \Sigma\}
\end{array}
$$

Fig. 10. Inequality and non-freshness

Lemma 1 (*Term Exclusivity*)
Let $\mathcal{K}$ be consistent, $s \in not[\![\tau]\!](t)$, $FV(u) \subseteq \Gamma$ and $FV(s,t) \subseteq \vec{X}$. It is not the case that both $\Gamma;\mathcal{K} \models \exists\vec{X}{:}\vec{\tau}.u \approx t$ and $\Gamma;\mathcal{K} \models \exists\vec{X}{:}\vec{\tau}.u \approx s$.

5.2 Clause complementation via generic operations

Clause complementation is usually described in terms of the contraposition of the *only-if* part of the completion of a predicate (Barbuti et al. 1990; Bruscoli et al. 1994; Muñoz-Hernández et al. 2004). We instead present a judgmental, syntax-directed approach. To complement atomic constraints such as equality and freshness, we need (α-)inequality and non-freshness; we implemented these using type-directed code generation within the αProlog interpreter. We write neq_{δ}, $nfr_{\nu,\delta}$, *etc.* as the names of the generated clauses (*cf.* analogous notions in (Fernández and Gabbay 2005)). Each of these clauses is defined as shown in Figure 10, together with mutually recursive auxiliary type-indexed functions $neq[\![\tau]\!]$, $nfr[\![\nu,\tau]\!]$, *etc.* which are used to construct appropriate subgoals for each type.

Complementing goals, as achieved *via* the not^{G} function (Figure 11), is quite intuitive: we just put goals in negation normal form, respecting the operational semantics of failure. Note that the self-duality of the И-quantifier (*cf.* (Pitts 2003; Gabbay and Pitts 2002)) allows goal negation to be applied recursively. The exis-

$$\begin{array}{rcl}
not^G(\top) & = & \bot \\
not^G(\bot) & = & \top \\
not^G(p(t)) & = & p^{\neg}(t) \\
not^G(t \approx_\tau u) & = & neq[\![\tau]\!](t,u) \\
not^G(a \#_\tau u) & = & nfr[\![\nu,\tau]\!](a,u) \\
not^G(G \wedge G') & = & not^G(G) \vee not^G(G') \\
not^G(G \vee G') & = & not^G(G) \wedge not^G(G') \\
not^G(\exists X{:}\tau.\, G) & = & \forall^* X{:}\tau. not^G(G) \\
not^G(\text{И}\mathsf{a}{:}\nu.\, G) & = & \text{И}\mathsf{a}{:}\nu. not^G(G)
\end{array}$$

Fig. 11. Negation of a goal

$$\begin{array}{rcl}
not_i^D(\forall \vec{X}{:}\vec{\tau}.p(t) :- \ G) & = & \bigwedge\{\forall \vec{X}{:}\vec{\tau}.p_i^{\neg}(u) \mid u \in not[\![\tau]\!](t)\} \wedge \\
 & & \forall \vec{X}{:}\vec{\tau}.p_i^{\neg}(t) :- \ not^G(G)
\end{array}$$

Fig. 12. Negation of a single clause

$$\begin{array}{rcl}
not^D(\mathrm{def}(p,\Delta)) & = & \bigwedge_{i=1}^{n} not_i^D(\forall \vec{X}{:}\vec{\tau}.p(t_i) :- \ G_i) \wedge \forall X.\ p^{\neg}(X) :- \ \bigwedge_{i=1}^{n} p_i^{\neg}(X) \\
 & & \text{where } \Delta_p = \{p(t_1) :- \ G_1, \ldots, p(t_n) :- \ G_n\} \\
 & & \text{is the set of all clauses in } \Delta \text{ with head } p.
\end{array}$$

Fig. 13. Negation of $\mathrm{def}(p,\Delta)$

tential case is instead more delicate: a well known difficulty in the theory of negation elimination is that in general Horn programs are not closed under complementation, as first observed in (Mancarella and Pedreschi 1988); if a clause contains an existential variable, *i.e.* a variable that does not appear in the head of the clause, the complemented clause will contain a universally quantified goal, call it $\forall^* X{:}\tau.\ G$. Moreover, this quantification cannot be directly realized by the standard *generic* search operation familiar from uniform proofs (Miller et al. 1991). In the latter case $\forall X : A.\ G$ succeeds iff so does $G[a/X]$, for a new eigenvariable a, while the $\forall^*$ quantification refers to *every* term in the domain, *viz.* $\forall^* X{:}\tau.\ G$ holds iff so does $G[t/X]$ for every (ground) term of type τ. We call this *extensional* universal quantification.

We add to the rules in Figure 5 the following ω-rule for extensional universal quantification in the sense of Gentzen and others:

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

This rule says that a universally quantified formula $\forall^* X{:}\tau.G$ 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 operationalize this rule in our implementation, similarly to (Muñoz-

$$\frac{\Gamma, X{:}\tau; \Delta; \mathcal{K} \Rightarrow G}{\Gamma; \Delta; \mathcal{K} \Rightarrow \forall^* X{:}\tau.\, G}\ \forall^*\forall \qquad \frac{\Gamma; \Delta; \mathcal{K} \Rightarrow G[\langle\rangle/X]}{\Gamma; \Delta; \mathcal{K} \Rightarrow \forall^* X{:}\mathbf{1}.\, G}\ \forall^*\mathbf{1}$$

$$\frac{\Gamma; \Delta; \mathcal{K} \Rightarrow \forall^* X_1{:}\tau_1.\forall^* X_2{:}\tau_2.\, G[\langle X_1, X_2\rangle/X]}{\Gamma; \Delta; \mathcal{K} \Rightarrow \forall^* X{:}\tau_1 \times \tau_2.\, G}\ \forall^*\times$$

$$\frac{\Gamma; \Delta; \mathcal{K} \Rightarrow \reflectbox{N}\mathsf{a}{:}\nu.\forall^* Y{:}\tau.\, G[\langle \mathsf{a}\rangle Y/X]}{\Gamma; \Delta; \mathcal{K} \Rightarrow \forall^* X{:}\langle\nu\rangle\tau.\, G}\ \forall^*\mathsf{abs}$$

$$\frac{\Gamma; \Delta; \mathcal{K} \Rightarrow \bigwedge\{\forall^* Y{:}\tau.\, G[f(Y)/X] \mid f : \tau \to \delta \in \Sigma\}}{\Gamma; \Delta; \mathcal{K} \Rightarrow \forall^* X{:}\delta.\, G}\ \forall^*\delta$$

Fig. 14. Proof search rules for extensional universal quantification

Hernández et al. 2004), by alternating between using the traditional ∀R rule and type-directed expansion of the quantified variable, as shown in Figure 14: 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 viewing $\forall^*$ as generic quantification. The latter is available in the (first-order) Hereditary Harrop formulæ extension of αProlog. This procedure is sound but may not be complete w.r.t. $\forall^*\omega$.

We now move to *clause* complementation, which is carried out definition-wise: if $\forall(p(t) \leftarrow G)$ is the i-th clause in $\mathrm{def}(p, \Delta)$, $i \in 1 \dots n$, its complement must contain a "factual" part motivating failure due to clash with (some term in) the head; the remainder $not^G(G)$ expresses failure in the body, if any. This is accomplished in Figure 12 by the not_i^D function, where a set of negative facts is built *via term* complementation $not(t)$; moreover the negative counterpart of the source clause is obtained *via* complementation of the body. Finally all the contributions from each source clause in a definition are merged by conjoining the above in the body of a clause for another new predicate symbol, say $p^\neg(X)$, which calls all the $p_i^\neg$ (Figure 13).

We list in Figure 15 the complement of the typechecking predicate from Section 2, which we have simplified by renaming and inlining the definitions of the $p_i^\neg$.[3] As expected, local variables in the application and projection cases yield the corresponding $\forall^*$-quantified bodies.

The most important property for our intended application is soundness, which we state in terms of exclusivity of clause complementation. Extend the signature Σ_p as follows: for every p add a new symbol $p^\neg$ and for every clause $p_i \in (\mathrm{def}(p, \Delta))$ add new $p_i^\neg$. Let $\Delta^- = not^D(\mathrm{def}(p, \Delta))$ for all p in Σ_P.

Theorem 1 (*Exclusivity*)
Let $\mathcal{K}$ be consistent. It is not the case that $\Gamma; \Delta; \mathcal{K} \Rightarrow G$ and $\Gamma; \Delta^-; \mathcal{K} \Rightarrow not^G(G)$.

Completeness (exhaustivity) can be stated as follows: if a goal G finitely fails from

[3] The un-simplified definition consists of more than 40 clauses.

```
pred not_tc ([(id,ty)],exp,ty).
not_tc([],var(_),_).
not_tc([(_,_)| G],var(X),T) :- not_tc(G,var(X),T).
not_tc(G,app(M,N),U)        :- forall* T. (not_tc(G,M,arr(T,U));
                                            not_tc(G,N,T)).
not_tc(G,lam(M),T ==> U)    :- new x. not_tc([(x,T)|G],M@x,U).
not_tc(G,pair(M,N),T ** U)  :- not_tc(G,M,T) ; not_tc(G,N,U).
not_tc(G,fst(M),T)          :- forall* U. not_tc(G,M,T ** U).
not_tc(G,snd(M),U)          :- forall* T. not_tc(G,M,T ** U).
not_tc(_,lam(_),unitTy).
not_tc(_,lam(_),_ ** _).
not_tc(_,unit,_ ==> _).
not_tc(_,unit,_ ** _).
not_tc(_,pair(_,_),unitTy).
not_tc(_,pair(_,_),_ ==> _).
```

Fig. 15. Negation of typechecking predicate (with manual simplification)

Δ, then its complement $not^G(G)$ should be provable from Δ^-. In a model checking context, this is a desirable, though not vital property. Logic programs in fact may define recursively enumerable relations, and the complement of such a program will not capture the complement of the corresponding relation — consider for a simple example, a Δ that defines the r.e. predicate *halts* that recognizes Turing machines halting on their inputs; it is obvious that the predicate $\neg$*halts* cannot define the exact complement of *halts*. We therefore cannot expect true completeness results unless we restrict to recursive programs, and determining whether a logic program defines a recursive relation is an orthogonal well-studied issue, see, *e.g.* the termination analysis approach taken in the Twelf system (Pientka 2005). In any case, we do not believe completeness is necessary for our approach to be useful, since we are mostly interested in testing systems with undecidable predicates such as first-order sequent calculi or undecidable typing/evaluation relations.

6 Experimental evaluation

We implemented counterexample search in the αProlog interpreter using both (grounded) negation-as-failure and negation-elimination, as described in the previous section. In this section, we present performance results comparing these approaches. We first measure the time needed by each approach to find counterexamples (TFCE). Then we measure the amount of time it takes for a given approach to exhaust its search space up to a given depth bound (TESS).

For negation-elimination, we considered two variants, one (called *NE*) in which the $\forall^*$ quantifier is implemented fully as a primitive in the interpreter, and a second in which $\forall^*$ is interpreted as ordinary intensional $\forall$. The second approach, which we call NE^-, is incomplete relative to the first; some counterexamples found by *NE* may be missed by NE^-. Nevertheless, NE^- is potentially faster since it avoids the overhead of run-time dispatch based on type information (and since it searches a smaller number of counterexample derivations).

Table 1. *TFCE and relative depths for code with bugs*

	NAF	*NE*	*NE*$^{-}$
`tc_weak`	<0.01, 3	<0.01, 2	<0.01, 2
`tc_subst`	<0.01, 3	0.17, 3	0.15, 3
`tc_pres`	<0.01, 4	<0.01, 4	<0.01, 4
`tc_prog`	<0.01, 4	<0.01, 5	<0.01, 5
`tc_sound`	3.76, 5	2.79, 5	2.14, 5

All test have been performed under Ubuntu 15.04 on an Intel Core i7 CPU 870, 2.93GHz with 8GB RAM. We, somewhat arbitrarily, time-out the computation when it exceeds 40 seconds.

6.1 The λ-calculus with pairs

We first go back to the examples in Section 2, using both the "buggy" version we have presented and the debugged version in the electronic appendix.

Time to find counterexamples For checks involving substitution, all counterexamples were found by all approaches in less than 0.01 seconds. Table 1 shows the times needed for checks involving *typechecking*, in seconds. The first column shows the name of the checked property and the others the time taken together with the search depth where the counterexample has been found by each technique.

In this benchmark, the three approaches *NAF*, *NE*, and *NE*$^{-}$ are basically equivalent, despite the fact that the latter two potentially cover more of the search space within a given depth bound. This is not always the case, as some of the other case studies mentioned in Section 6.3 showcase. In fact, axiomatizing what holds to be true is intrinsically more economical than stating what is false. This is one reason why techniques such as *NAF*, which gives an operational rather than logical solution to the frame problem, have been so empirically successful. These results also indicate that pragmatically speaking the faster *NE*$^{-}$ approach can be used first, with *NE* as a backup if no counterexamples are found using *NE*$^{-}$.

When using the derivation-first approach, the counterexamples found by *NAF* (and discussed in Section 2) are in all cases but one (`tc_prog`) *ground instances* of the ones found by NE. In this benchmark there is not a significant difference in the depth bound, but in general *NAF* tends to find the counterexample at a smaller bound than *NE* (and *NE*$^{-}$).

Time to exhaust a finite search space For each technique and test, we measured TESS for $n = 1, 2, \ldots$ up to the point where we time-out. The experimental results are shown in Table 2. For each test, we used the largest n for which *all* three approaches were successful within the time-out. Note that we report the results according to the "best" ordering of subgoals that we have experimented with.

Table 2. *Time to search up to bound n for debugged code*

	n	*NAF*	*NE*	*NE*$^-$
`sub_fun`	5	1.38	0.25	same as *NE*
`sub_id`	7	9.85	0.82	same as *NE*
`sub_fresh`	4	3.93	0.75	same as *NE*
`sub_comm`	4	39.39	5.96	same as *NE*
`tc_weak`	5	2.14	6.58	3.33
`tc_subst`	4	6.15	33.56	26.86
`tc_pres`	6	0.27	1.04	same as *NE*
`tc_prog`	8	6.84	8.18	same as *NE*
`tc_sound`	7	6.15	29.4	6.01

These results are mixed. In some cases, particularly those involving substitution, *NE* and *NE*$^-$ are clearly much more efficient (up to 10 times faster) than the *NAF* approach. In others, particularly key results such as substitution and type soundness, *NE* often takes significantly longer, up to five times, with *NE*$^-$ usually doing better. On the other hand, for the `tc_prog` checks, both *NE*-based techniques are competitive.

However, it is important to note that the search spaces considered by each of the approaches for a given depth bound are not equivalent. Thus, it is not meaningful to compare the different approaches directly based on the search bounds. Indeed, it is not clear how we should report the sizes of the search spaces, since even a simple unifier $X = f(c, Z)$ represents an infinite (but clearly incomplete) subset of the search space. We can, however, get an idea of the relationship between the search spaces based on the depths at which counterexamples are found.

The translation of negated clauses in *NE* and *NE*$^-$ (Section 5) is a conjunction of disjunctions. This causes our algorithm to do inefficient backtracking. This can probably be improved using standard optimization techniques which are not implemented in the current αProlog prototype. An alternative is changing the clause complementation algorithm to obtain a more "succinct" negative program: some initial results are presented in (Cheney et al. 2016).

A second major source of inefficiency, which accounts for the difference between *NE*$^-$ and *NE*, is the extensional quantifier; in fact, *NE*$^-$ outperforms *NE* significantly for checks `tc_weak, tc_subst, tc_sound` involving extensional quantifiers in the negation of `tc`. The culprit is likely the implementation of extensional quantification as a built-in proof search operation, which dispatches on type information at run-time. This is obviously inefficient and we believe it could be improved. However, doing so appears to require significant alterations to the implementation.

Variations We performed also some limited experiments comparing the two approaches based on negation-as-failure, and by changing the order of subgoals (Table 3) *w.r.t.* TFCE and TESS. Not surprisingly, we found that placing the generator predicates at the end of the list of hypotheses, and giving preference to most con-

Table 3. *TFCE and TESS with* NAF *and different orderings on* `tc_prog` *and* `tc_sound`

`check`	TFCE	TESS
`tc([],E,T),gen_exp(E) => progress(E)`	<0.01, 4	6.84, 8
`gen_exp(E),tc([],E,T) => progress(E)`	<0.01, 4	31.2, 8
`tc([],E,T),steps(E,E'),gen_ty(T),gen_exp(E') => tc([],E',T)`	3.74, 5	6.07, 7
`tc([],E,T),steps(E,E'),gen_exp(E'),gen_ty(T) => tc([],E',T)`	3.98, 5	6.17, 7
`steps(E,E'),tc([],E,T),gen_ty(T),gen_exp(E') => tc([],E',T)`	5.62, 5	7.38, 7
`gen_ty(T),tc([],E,T),gen_exp(E'),steps(E,E') => tc([],E',T)`	1.11, 5	t.o., 7
`gen_ty(T),tc([],E,T),steps(E,E'),gen_exp(E') => tc([],E',T)`	0.36, 5	18.9, 7
`gen_ty(T),gen_exp(E'),tc([],E,T),steps(E,E') => tc([],E',T)`	9.82, 5	t.o., 7
`gen_exp(E'),gen_ty(T),tc([],E,T),steps(E,E') => tc([],E',T)`	t.o.	t.o., 7

strained predicates (in terms of least number of clauses), generators included, can make some difference, especially in terms of TESS. In fact, time-outs in this case are more frequent. However, type-driven search, that is, putting the type generator first, seems in this case the most successful strategy in terms of TFCE.

The most constrained goal first heuristic can be applied to *NE* and NE^- as well. We will not report the experimental evidence, but point out the in the *NE* case we definitely want to give precedence to predicates that do *not* use extensional quantification. In both cases, by the very fact that negated predicates are now positivized, they can be re-ordered as appropriate. This in contrast with *NAF*, where negated predicates must occur after grounding. Finally, we remark that those orderings are not hard-coded but stay in the hands of the user, as she writes her `#check` directives. This is important, as general heuristics cannot replace the user understanding of the SUT.

6.2 Security type systems

For another test, we selected a variant of a case study mentioned in (Blanchette and Nipkow 2010): an encoding of the security type system of Volpano et al. (1996), whereby the basic imperative language *IMP* is endowed with a type system that prevents information flow from private to public variables. Given a fixed assignment *sec* of security levels (naturals) 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 variables lower then l and only safe flows to variables $\geq l$. We inserted two mutations in the typing rule, one (`bug1`) suggested by Nipkow and Klein (2014), which forgets an assumption in the sequence rule; the other (`bug2`), inverting the first disequality in the assignment rule — the latter slipped in during encoding. We show in Figure 6.2 the typing rules, where the over-strike and the box signal the inserted mutations.

The properties that are influenced by those mutations relate states that agree

$$\frac{\mathit{sec}\ a \leq \mathit{sec}\ x \qquad l \leq \mathit{sec}\ x}{l \vdash x := a} \qquad \frac{\boxed{\mathit{sec}\ x \leq \mathit{sec}\ a} \qquad l \leq \mathit{sec}\ x}{l \vdash x := a}\ \texttt{bug2}$$

$$\frac{l \vdash c_1 \qquad \cancel{l \vdash c_2}}{l \vdash c_1; c_2}\ \texttt{bug1} \qquad \frac{\max\ (\mathit{sec}\ b)\ l \vdash c}{l \vdash \mathit{WHILE}\ b\ \mathit{DO}\ c}$$

$$\frac{}{l \vdash \mathit{SKIP}} \qquad \frac{\max\ (\mathit{sec}\ b)\ l \vdash c_1 \qquad \max\ (\mathit{sec}\ b)\ l \vdash c_2}{l \vdash \mathit{IF}\ b\ \mathit{THEN}\ c_1\ \mathit{ELSE}\ c_2}$$

Fig. 16. Bugged rules for the Volpano et al. type system

Table 4. *TFCE on Volpano benchmark*

		NAF	*NE*	*NE*$^-$
`bug1`	Confinement	0.03, 5	0.76, 7	t.o.
	Non-interference	10.32, 8	8.13, 8	t.o.
`bug2`	Non-interference	3.91, 8	3.61, 8	t.o.

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 $\forall x.\ \mathit{sec}\ x \leq l \to \sigma_1(x) = \sigma_2(x)$ (resp. $<$). Given a standard big-step evaluation semantics for IMP (Winskel 1993), relating an initial state σ and a command c to a final state τ ($\langle c, \sigma\rangle \downarrow \tau$):

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

Our encoding is fully relational, where, for example, states and security assignments are reified in association lists. We cannot rely on built-in types such as integers and booleans, which αCheck does not handle yet, but we have to employ hand-written (inefficient) datatypes for unary natural numbers and booleans. Finally, this case study does not exercise binders intensely, as nominal techniques have a role in representing program variables as names and using freshness to guarantee well-formedness of states and of variable security settings.

We sum up the results in Table 4 and 5. A first thing to note is that NE is doing fairly well *w.r.t. NAF* catching the non-interference counterexamples, notwithstanding having essentially to rely on extensional quantification: *NE*$^-$ in fact shows

Table 5. *TESS on Volpano benchmark*

	n	*NAF*	*NE*
Confinement	8	9.74	4.31
Non-interference	8	13.14	6.94

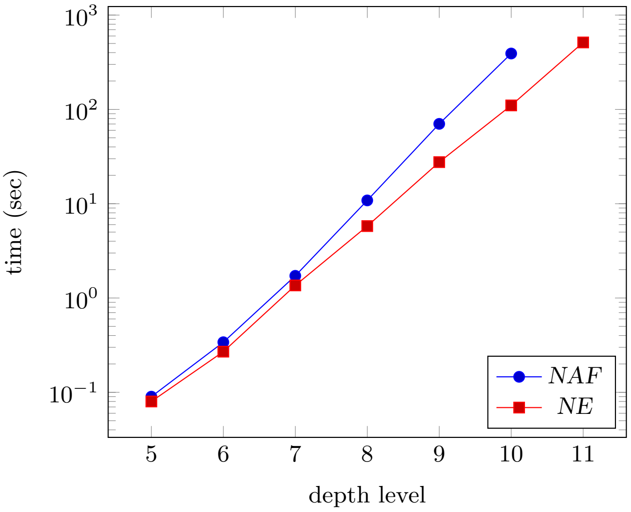

Fig. 17. Loglinear-plot comparing *NAF* with *NE* in TESS on non-interference

its incompleteness here, failing to find any counterexample — this is why we do not even bother to measure its TESS-behavior. *NE*'s TESS behavior is also quite pleasing and more so asymptotically, as we show in Figure 17.

For `bug1` *NE* finds this counterexample to confinement: c is $(\mathit{SKIP}\ ; x := 0)$, $\mathit{sec}\ x = 0$, $l > 0$, σ maps x to a non-zero level and τ 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 $(\mathit{SKIP}\ ; x := y)$, $\mathit{sec}\ x = 0, \sec y > 0$, $l = 0$ and σ maps y to $n > 1$ and x unconstrained (i.e. to a logic variable), while τ maps y to > 0 and keeps x unconstrained. *NAF* finds ground instances of the above, for example in the first case $l = 4$. We omit the details of the counterexample to `bug2`.

6.3 Further experience

In addition to the examples discussed above, we have used the checker in several more substantial examples. In this section we briefly summarize some additional experimental results and experiences with larger examples.

First we discuss three case studies in which we defined object languages and specified some of their desired properties from extant research papers:

- LF equivalence algorithms and their structural properties (Harper and Pfenning 2005), which were formally verified in Nominal Isabelle by Urban et al. (2011), with three mutations inserted.
- λ^{zap}, a "faulty lambda calculus" (Walker et al. 2006)
- The example based on "Causal commutative arrows and their optimization" (Liu et al. 2009), also used as a case study for PLT Redex by Klein et al. (2012).

Table 6 summarizes TFCE and TESS measurements for these examples on representative tests using *NAF*, *NE* and *NE*$^{-}$.

Table 6. *TFCE and TESS for additional experiments*

			NAF	*NE*	*NE*$^-$
LFEquiv	lem3.2(1)	[TFCE]	0.1, 7	t.o.	same as *NE*
	lem3.4(1)	[TFCE]	0.1, 7	0.1, 7	same as *NE*
	lem3.4(2)	[TFCE]	0.1, 7	t.o.	same as *NE*
	lem3.5(2)	[TFCE]	0.1, 7	t.o.	same as *NE*
Zap	fstep_det	[TFCE]	0.1, 3	0, 2	same as *NE*
	2fault	[TFCE]	0, 3	0, 3	same as *NE*
CCA	exists_norm	[TESS]	0.3, 6	36,6	0.1, 6
	red_equiv	[TESS]	0.5, 4	0.6, 4	same as *NE*

We have also performed some additional case studies, for which we do not report experimental results — some results about the last case study can be found in (Cheney et al. 2016), together with some additional comparison to other tools such as Isabelle's Nitpick and QuickCheck.

- A (type-unsafe) mini-ML language with polymorphism and references.
- The exercises in the *Types.v* and *StlcProp.v* chapters of *Software Foundations* (Pierce et al. 2016), which ask whether properties such as type preservation hold under variations of the given calculi.
- A λ-calculus with lists, from the PLT-Redex benchmarks suite (Findler et al. 2015).

We did not find previously unknown errors in these systems, nor did we expect to; however, αCheck gives us some confidence that there are no obvious typos or transcription errors in *our implementations* of the systems. In some cases, we were able to confirm known, desired properties of the systems *via* counterexample search. For example, in λ^{zap}, the type soundness theorem applies as long as at most one fault occurs during execution; we confirmed that two faults can lead to unsoundness. Similarly, it is well-known that the naive combination of ML-style references and let-bound polymorphism is unsound; we are able to confirm this by guiding the counterexample search, but the smallest counterexample (that we know of) cannot be found automatically in interactive time. Further, while re-encoding some of the benchmarks proposed in the relevant literature, we have been successful in catching almost all the inserted mutations (Cheney et al. 2016).

Our subjective experiences with the implementations have been positive. Writing specifications for programs requires little added effort and also seems helpful for documentation purposes.

From these experiences, several observations can be made:

1. Checking properties of published, well-understood systems does confirm that αCheck avoids false positives, but does not necessarily show that it is helpful during the development of a system. Our personal experience strongly points in this direction, but further study would be needed to establish this, perhaps *via* usability studies.

2. It is not advisable to just check the main properties such as type soundness, since the system may be flawed in such a way that soundness holds trivially, but other properties such as inversion or substitution fail. In fact, just checking `tc_sound` on our buggy λ-calculus will miss 80% of the bugs. Moreover, of the bugs found, not only they are found at deeper levels and hence more likely to be timed out, but they are more difficult to interpret, as, e.g. an issue with reduction must be located to a bug in the substitution function. Instead, it is generally worthwhile to enumerate all of the desired properties of the system (including auxiliary properties that might arise during a proof). This could be especially helpful when one wishes to make a change to the system, since the checks can serve as regression tests.
3. The ordering of subgoals often has a significant effect on performance and we have informally adopted the "most constrained goal first" heuristic. Many alternative search strategies and optimizations (e.g. random search, coroutining, tabling), could be considered to improve performance.

7 Related work

7.1 Nominal abstract syntax

Our work builds on the nominal approach to abstract syntax initiated by Gabbay and Pitts (2002), which has led to a great deal of research on unification, rewriting, algebraic and logical foundations of languages with name-binding. Since the conference version of this paper was published, there has been considerable work on nominal techniques, particularly regarding unification and rewriting of nominal terms. We do not have space to provide a comprehensive survey of this work; in this section we place our work in context, and point to other work that complements or could be combined with our approach.

Nominal terms, rewriting, and unification There has been great progress on algorithms for nominal unification and other algorithms and theory for nominal terms. For example, αProlog uses the naive, asymptotically exponential algorithm for nominal unification presented by Urban et al. (2004), but subsequent work has led to more efficient algorithms (Calvès and Fernández 2008; Levy and Villaret 2010). Implementing such techniques in αProlog may lead to faster specification checking. It has also been shown that nominal terms and unification are closely related to higher-order patterns and higher-order pattern unification (Cheney 2005a; Levy and Villaret 2012). This suggests that one could perform nominal term complementation by mapping nominal terms to higher-order patterns, and using existing techniques for higher-order pattern complement (Momigliano and Pfenning 2003); however, there would be little benefit to doing so, because the latter problem requires further extensions to the type system to deal with binding, whereas our approach avoids these complications by complementing first-order terms only and using the predicates *neq* and *nfr* to deal with names and binding.

In αProlog, functions such as substitution can be defined, but they are implemented by translation to relations ("flattening"). In αML (Lakin and Pitts 2009), functional and logic programming styles are combined, using a variant of nominal abstract syntax and unification that avoids the use of constant names. Rewriting techniques (Fernández and Gabbay 2005), particularly *nominal narrowing* (Ayala-Rincón et al. 2016), could be incorporated into αProlog and might improve the performance of specification checking in the presence of function definitions.

Nominal logic and logic programming Nominal logic was initially defined as a Hilbert-style first-order theory axiomatizing names and name-binding by Pitts (2003). As with "first-order" or "higher-order" logic, however, we regard "nominal logic" as a name for a family of systems, not just the influential initial proposal by Pitts. As a foundation for logic programming, Pitts' system had two drawbacks: it did not allow for constant names, and its Hilbert-style presentation made it difficult to develop proof-theoretic semantics following Miller et al. (1991). Name constants are required to use the nominal unification algorithm, and Cheney (2006) showed how to incorporate name constants into nominal logic and established completeness and Herbrand theorems relevant to logic programming. To address the second problem, Gabbay (2007) proposed natural deduction system Fresh Logic (FL) and Gabbay and Cheney (2004) proposed a related sequent calculus $FL^{\Rightarrow}$. The system used as a basis for αProlog by Cheney and Urban (2008) is the $NL^{\Rightarrow}$ system of Cheney (2016), which avoids some of the technical complications of earlier systems and is proved conservative with respect to Pitts' original axiomatization.

Nominal automata and model-checking Intriguing connections between nominal techniques and automata theory have also come to light (Bojańczyk et al. 2013; Pitts 2016). In particular, Gadducci et al. (2006) have established interesting connections between nominal sets and *history-dependent automata* (Montanari and Pistore 2005), which can be used to model-check processes in calculi such as CCS or the pi-calculus. Although we are not aware of any work on automata that could be used to model-check properties of relations over nominal terms, it may be fruitful to investigate the relationship between our work and other directions that draw upon the classical automata-theoretic approaches to model checking.

7.2 Testing, model checking, and mechanized metatheory

As stated earlier, our approach draws inspiration from the success of finite state model-checking systems. Particularly relevant is the idea of *symbolic* model checking, in which data structures such as Boolean decision diagrams represent large numbers of similar states; in our approach, answer substitutions and constraints with free variables play a similar role.

Testing Another major inspiration comes from *property-based testing* in functional programming languages, as first realized by QuickCheck for Haskell (Claessen and Hughes 2000). QuickCheck provides type class libraries for generator functions to construct *random* test data for user-defined types, as well as to monitor and customize data distribution, and a logical specification language, basically coinciding with Horn clauses, to describe the properties the program should satisfy. The QuickCheck approach has been widely successful — so much that there are now versions for many other programming languages, including imperative ones. A major feature/drawback of QuickCheck is that the user has to *program* possibly fairly sophisticated test generators to obtain a suitable distribution of values. Further, random testing is notoriously inefficient in checking *conditional* properties. Both issues are tricky, linked as they are to the well known problem of the quality of test coverage. There are at least two versions of QuickCheck for Prolog, see `https://github.com/mndrix/quickcheck` and (Amaral et al. 2014). Both essentially implement the *NAF* approach and struggle with types. On the other hand, they are quite efficient being built on top, respectively, of SWI-Prolog and Yap.

An alternative to QuickCheck is SmallCheck (Runciman et al. 2008), which, although conceived independently from our approach, shares with us the idea of *exhaustive* testing of properties for all finitely many values up to some depth. It enriches QuickCheck's specification language with existential quantification and, in *Lazy* SmallCheck, with *parallel* conjunction, which abstracts over the order of atoms in conditions. Lazy SmallCheck can also generate and evaluate partially-defined inputs, by using a form of *needed narrowing*. In conjunction with an implementation of nominal abstract syntax (such as FreshLib (Cheney 2005b) or Binders Unbound (Weirich et al. 2011)), Quick/SmallCheck could be used to implement metatheory model-checking, although this would build several levels of indirectness that may make counter-example search rather problematic. Compared to us, QuickCheck is a widely used library for general purpose programming, while we have so far put little effort into making our counter-example search more efficient. However, by the very fact that we use (nominal) logic programming, our specification language tends to be more expressive. Further, the idea of negation elimination goes well beyond Lazy SmallCheck's partially defined inputs, as it allows us to test open conditions without further ado. Finally, so far, we have used as test generator the built-in $gen[\![\tau]\!]$ function without feeling the need to provide an API to write *custom* generators; this may also be due to the fact that we do not generate tests at function types, which are not available in αProlog.

The success of QuickCheck has lead many theorem proving systems to adopt random testing, among them PVS (Owre 2006), Agda (Dybjer et al. 2004) and very recently Coq with the *QuickChick* tool (Paraskevopoulou et al. 2015). The system where proofs and disproofs are best integrated is arguably Isabelle/HOL (Blanchette et al. 2011), which offers a combination of random, exhaustive and symbolic testing (Bulwahn 2012a). Random testing has been present in the system for a decade; it is executed directly *via* Isabelle/HOL's code generation and has been recently enriched with a notion of *smart* test generators to improve its success rate w.r.t. conditional properties. This is achieved by turning the functional code into logic programs and inferring through mode analysis their data-flow behavior. Interestingly, generators for inductive types are automatically inferred and user input is required only for HOL-style type definitions. Exhaustive and symbolic testing follow the SmallCheck approach, where narrowing is simulated with a refinement algorithm that has several similarities with our extensional quantifier. We note that exhaustive checking is the default setting for Isabelle/HOL. Notwithstanding all these improvements, QuickCheck requires all code and specs to be *executable* in the underlying functional language, while many of the specifications that we are interested in are best seen as *partial* and *not terminating*. For the latter, a valuable alternative is *Nitpick* in (Blanchette and Nipkow 2010), a higher-order model finder in the *Alloy* lineage supporting (co)inductive definitions. It 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 evaluated by means of mutation testing of the metatheory of type-inference in MiniML, the POPLMark challenge, and type safety proofs for multiple inheritance in C++. Nitpick in these reported experiments finds out roughly a third of the mutants, but it also signals a certain number of potential false positives without any easy way to tell which is which. It would be natural to couple Isabelle/HOL's QuickCheck and/or Nitpick's capabilities with *Nominal* Isabelle (Urban and Kaliszyk 2012), but this would require strengthening the latter's support for computation with names, permutations and abstract syntax modulo α-conversion.

Environments for programming language descriptions. The main players are *PLT-Redex* (Felleisen et al. 2009) and the *K* framework (Roşu and Şerbănuţă 2010). In both, several large-scale language descriptions have been specified. We concentrate on the former as *K*, while providing many tools needed to execute and analyze programs written in an object language, is not geared towards metatheory model checking, nor does it support binding syntax. PLT-Redex is an executable DSL for mechanizing semantic models built on top of *DrRacket*. It supports the formalization of the syntax and the semantics of an object language, with special support for small-step semantics with evaluation contexts. It provides visualization tools for animating those models as well as automatic type-setting facilities. The most notable feature for our purpose is Redex's support for random testing á la QuickCheck, whose usefulness has been demonstrated in several impressive case studies (Findler et al. 2015; Klein et al. 2012; Klein et al. 2012), some of which we have started replicating with our tool (Cheney et al. 2016). The main drawback is again the lack of support for binders: variables are just another non-terminal and they are handled in an ad hoc way. A generic substitution (meta)function is provided but it has to be tweaked to respect binding occurrences. The tool provides naive test generators stemming from grammar definitions, but they tend to offer very little coverage, especially when dealing with typed languages and non-algorithmic relations. However, in a very recent paper (Fetscher et al. 2015) the authors build a form of constraint logic programming on top of PLT-Redex to obtain *random* typing derivations; the motivation here is overcoming the problem that well-typed terms are rather sparse in the space of pre-terms and as such random generation of them tends to be wasteful. Hence they construct partial type derivations by flipping a coin when several typing rules can be selected. Clearly, our setup enjoys an *exhaustive* version of this notion of generation for free and as we comment further in the Conclusion, it would not be hard to incorporate the random angle.

Ott (Sewell et al. 2010) is a highly engineered tool for "working semanticists", allowing them to write programming language definitions in a style very close to paper-and-pen specifications; the system then performs some sanity checks on those specs, compiles them into LaTeX, and, more interestingly, into proof assistant code, currently supporting Coq, Isabelle/HOL and HOL. Ott's metalanguage is endowed with a rich theory of binders, but the current implementation favors the "concrete" (non α-quotiented) representation, while providing support for the nameless representation for a single binder. Since Ott tends to be used mostly as a documentation system, it would make sense to pair it with a lightweight validation tool such as ours, so as to catch (shallow) bugs early in the design phase of some piece of PL theory. In fact, most mainstream systems for static and dynamic semantics appear easy to translate into αProlog clauses, we claim more naturally and of course more adequately *w.r.t.* any concrete syntax for binders. In this sense, a plug-in for Ott to produce αProlog code as well would be a valuable future work to pursue.

Other more specific approaches include (Roberson et al. 2008), where the authors extend their previous work on using a software model checker for data structure properties to the realm of ASTs and type soundness. The idea is to exhaustively generate all possible program states, that is, well typed expressions in an object PL, execute one step and check that types are preserved and execution does not get stuck. The crucial contribution is in the taming of the search space, whereby ASTs that roughly exercise the same SOS rules are pruned away. This yields a dramatic reduction of the generated states. SOS and typing rules must be encoded in Java; thus no support for binders etc. is provided. More importantly, the system is wired to check *only* progress and preservation properties and a user would need to re-program it to test any other property. The authors mention experimental results about mutation testing of an extension of Featherweight Java with imperative features and ownership types, but no additional description is available, preventing us from trying to replicate the experience.

Negation and logic programming There is an extremely large literature on negation as failure, constructive/intensional negation, and disunification; we restrict attention only to closely related work.

Negation elimination (a.k.a. *intensional* negation) has a long history in logic programming dating back the late 80's (Barbuti et al. 1990) and later extended to constraint logic programming languages (Bruscoli et al. 1994), although no concrete implementation has been reported until Muñoz-Hernández's thesis and subsequent papers (Moreno-Navarro and Muñoz-Hernández 2000; Muñoz-Hernández et al. 2004). In all these papers, negative predicates are schematically synthesized by applying several non-deterministic (classical) manipulations to the *completion*, whose correctness is formulated in terms of Kunen's three-valued semantics. Our approach, instead, is based on a judgmental and syntax-directed translation, which is straightforward and directly implementable. Our presentation of negation elimination can also be applied to ordinary typed first-order logic programming; it is closely related to (Momigliano 2000), where the target language is a fragment of λProlog, namely (monomorphic) third-order hereditary Harrop formulae, although the main focus (and challenge) there is complementing hypothetical clauses, an issue that does not occur in αProlog.

A related approach is *constructive negation*, in particular as formulated by Stuckey (1995), in which negated existential subgoals are handled *via* a combination of case analysis and disunification.

Proof search in the presence of an *extensional* universal quantifier has been studied in several settings; our approach is inspired by ω-rules such as the one in the proof-theory of arithmetic. A principle of "proof by case analysis" was first proposed in (Barbuti et al. 1990) and then refined in (Muñoz-Hernández et al. 2004). The related proof-theory of success and failure of existential goals has been investigated in (Harland 1993) in the context of uniform proofs.

Model checking and logic programming The Logic-Programming-Based Model Checking project at Stony Brook implements the model checker XMC for value-passing CCS and a fragment of the mu-calculus on top of the XSB tabled logic programming system (Ramakrishnan et al. 2000), which extends *SLD* resolution with tabled resolution. As the latter terminates on programs having finite models and avoids redundant sub-computations, it can be used as a fixed-point engine for implementing local model checkers. Similarly, in the paradigm of Answer Set Programming (Niemelä 2006) a program is devised such that the solutions of the problem can be retrieved constructing a collection of models of the program. To achieve this, the language is essentially function-free disjunctive logic programming, although its expressivity has been consistently expanded in the ensuing years. These two paradigms do not readily provide support for the binding syntax that is essential for formalizing and checking meta-theoretic properties. On the other hand, optimizations such as tabling could certainly be useful, for example to improve $\forall^*$ performance.

The Bedwyr system (Baelde et al. 2007) instead is based on proof-search in a fragment of the $\mathcal{G}$ logic of Gacek et al. (2012), which allows a form of model checking directly on syntactic expressions possibly containing binding. This is supported by term-level λ-binders, a fresh name ∇-quantifier, higher-order pattern unification and tabling. The relationship of (a fragment of) this framework with nominal logic has been investigated elsewhere (Gabbay and Cheney 2004; Schöpp 2007; Gacek 2010). As a model checker, Bedwyr views the proof of a statement $\forall x.\ p(x) \to G(x)$ as the attempted verification that $G(t)$ holds for all the t s.t. $p(t)$ (the "model" that is enumerated). Since Bedwyr uses depth-first search, checking properties for infinite domains can be approximated by writing logic programs encoding generators for a finite portion of that model. Recent work about "augmented focusing systems" (Heath and Miller 2015) could make this automatic. Loop checking implemented with a limited form of tabling is added to handle (co)inductive specifications, whereby a loop over an inductive (resp. coinductive) predicate is interpreted as failure (resp. success). However, this interpretation is not yet supported by any metatheory. Bedwyr captures finite failure by seeing $\Gamma \vdash \neg A$ as $\Gamma, A \vdash \bot$ and solved as above. However, this treatment seems to be sound only *w.r.t.* the Horn+∇ fragment of the logic, hence checks involving hypothetical judgments as typical of higher-order abstract syntax need to be expressed moving to an explicit

"2-levels" approach (Gacek et al. 2012), and this may be too indirect to be effective. Nevertheless, nothing prevents the user to write (binding) specifications and checks in the Horn+∇ fragment, similarly to what we do in αProlog, although no experiment in this sense has yet been carried out.

Analyses for checking modes, coverage, termination, and other (logic) program properties can be used to verify program properties, playing an important role in the Twelf system (Schürmann 2009). This approach is also possible (and seems likely to be helpful) in αProlog, but such analyses have not yet been adapted to the setting of nominal logic programming. Conversely, it may also be possible to implement counterexample search in Twelf *via* negation elimination along the lines of (Momigliano 2000).

8 Conclusions and future work

A great deal of modern research in programming languages involves proving metatheoretic properties of formal systems, such as type soundness. Although the problem of specifying such systems and verifying their properties has received a lot of attention recently, verification tools still require substantial effort to learn and use successfully. We have presented a complementary approach that we call *metatheory model-checking* and a tool, αCheck, which address the dual problem of identifying flaws in specified systems (that is, counterexamples to desired properties). We introduced several possible implementation strategies based on different approaches to negation in nominal logic programming including *negation-as-failure* and *negation elimination*. We have detailed how to accommodate negation elimination in nominal logic programs and discussed experimental results that show that both techniques have encouraging performance. We plan to address several obvious performance issues in *NE* in future work. From a pragmatical standpoint in fact, the implementation of universal quantification currently involves analyzing type information in the run-time system. This appears to be one source of inefficiency in predicates such as `not_tc` that involve local variables. We are looking into ways to pre-compile this information, in order to avoid this expensive run-time type analysis.

In this article, we have restricted attention to a particularly well-behaved fragment of nominal logic programs in which И-quantification and names may only be used in goal formulas. This suffices for many examples, but some phenomena (such as name-generation) cannot be modeled naturally in this sub-language. We would like to investigate the general theory of elimination of negation in nominal logic, in particular complementing clause heads containing free names. This may also be useful for extending Twelf-like static analysis to αProlog; in fact *coverage* analysis can be stated as a relative complement problem.

Property-based testing in systems such as PLT-Redex and Isabelle/HOL is, in a sense, rediscovering logic programming (Bulwahn 2012b; Fetscher et al. 2015). The notion of *random typing derivation* in the latter paper, in particular, seems just a special case of having random rather then exhaustive backchaining in a logic programming interpreter. Whether this is effective in catching deeper bugs is an empirical issue, but we are certainly well placed to explore this idea.

One pressing question is the relationship between the different forms of negation: *NAF*, *NE* and *NE*$^-$. We have used *NAF* pragmatically without worrying too much about its correctness, and the semantics of negation-as-failure have yet to be formalized for αProlog; we have stronger evidence for the (partial) correctness of *NE*, but we do not know, for example, whether *NE* (or *NE*$^-$) is complete relative to *NAF* on ground goals or vice versa. Soundness and completeness have been investigated in the context of pure Prolog (Barbuti et al. 1990), but in a way that is hard to generalize to nominal logic programming. A better (proof-theoretic) way could be to relate *NE* to the completion by viewing logic programs as fixed points (Schroeder-Heister 1993). This view could also open the road to handle specifications that are *coinductive* in nature, as in concurrent calculi (Tiu and Miller 2010) or studies about program equivalence (Momigliano 2012). Our main contribution is showing empirically that both *NAF* and *NE*/*NE*$^-$ can be *useful* as a basis for mechanized model-checking, and the lack of answers to these questions does not detract from this contribution, but we think it would be worthwhile to study them in more detail.

Another direction for future work is to investigate automatic support for identifying the culprit when a check fails. One might naively expect this to be straightforward, for example using a similar approach to *declarative debugging* (Naish 1997); however, in the presence of negation (whether *NAF* or *NE*), it is not at all clear how to concisely explain the reason why a goal succeeds or fails. Indeed, the reason for the failure could be the absence of a needed rule, or an error in a rule that means it can never be used.

In conclusion, we have presented two approaches to mechanized metatheory model-checking in αProlog, one based on negation-as-failure and the other based on negation elimination. They have complementary strengths: negation-as-failure is conceptually simple and appears efficient in practice, but currently lacks a solid theoretical foundation, while negation elimination has been proved correct but may be slower on some examples. Our experiments also suggest that further optimizations would be valuable, but these two techniques are already useful for debugging language specifications formalized using αProlog.

The sources for αProlog and αCheck, including all the examples mentioned here, can be found at `http://github.com/aprolog-lang`.

Acknowledgments We wish to thank Matteo Pessina for his contribution to the usability of the implementation of the checker, and Frank Pfenning for still-ongoing discussion about negation elimination.

References

Amaral, C., Florido, M., and Santos Costa, V. 2014. PrologCheck: Property-based testing in Prolog. In *Functional and Logic Programming*, M. Codish and E. Sumii, Eds. Lecture Notes in Computer Science, vol. 8475. Springer International Publishing, 1–17.

Apt, K. R. and Bol, R. N. 1994. Logic programming and negation: A survey. *The Journal of Logic Programming 19*, 9 – 71.

Ayala-Rincón, M., Fernández, M., and Nantes-Sobrinho, D. 2016. Nominal narrowing. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal*. 11:1–11:17.

Aydemir, B. E., Bohannon, A., Fairbairn, M., Foster, J. N., Pierce, B. C., Sewell, P., Vytiniotis, D., Washburn, G., Weirich, S., and Zdancewic, S. 2005. Mechanized metatheory for the masses: The PoplMark Challenge. In *TPHOLs*, J. Hurd and T. F. Melham, Eds. Lecture Notes in Computer Science, vol. 3603. Springer, 50–65.

Baelde, D., Gacek, A., Miller, D., Nadathur, G., and Tiu, A. 2007. The Bedwyr system for model checking over syntactic expressions. In *CADE*, F. Pfenning, Ed. Lecture Notes in Computer Science, vol. 4603. Springer, 391–397.

Barbuti, R., Mancarella, P., Pedreschi, D., and Turini, F. 1990. A transformational approach to negation in logic programming. *J. of Log. Program. 8*, 201–228.

Blanchette, J. C., Bulwahn, L., and Nipkow, T. 2011. Automatic proof and disproof in Isabelle/HOL. In *FroCoS*, C. Tinelli and V. Sofronie-Stokkermans, Eds. Lecture Notes in Computer Science, vol. 6989. Springer, 12–27.

Blanchette, J. C. and Nipkow, T. 2010. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *ITP 2010*, M. Kaufmann and L. Paulson, Eds. LNCS, vol. 6172. Springer, 131–146.

Bojańczyk, M., Klin, B., Kurz, A., and Pitts, A. M. 2013. Nominal computation theory (Dagstuhl Seminar 13422). *Dagstuhl Reports 3,* 10, 58–71.

Bruscoli, P., Levi, F., Levi, G., and Meo, M. C. 1994. Compilative constructive negation in constraint logic programs. In Proc. Trees in Algebra and Programming - CAAP'94, 19th International Colloquium, S. Tison, Ed. *Lecture Notes in Computer Science 787*, 52–76.

Bulwahn, L. 2012a. The new Quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. In *CPP*, C. Hawblitzel and D. Miller, Eds. Lecture Notes in Computer Science, vol. 7679. Springer, 92–108.

Bulwahn, L. 2012b. Smart testing of functional programs in Isabelle. In *LPAR*, N. Bjørner and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 7180. Springer, 153–167.

Calvès, C. and Fernández, M. 2008. A polynomial nominal unification algorithm. *Theor. Comput. Sci. 403,* 2-3, 285–306.

Cheney, J. 2005a. Relating nominal and higher-order pattern unification. In *Proceedings of the 19th International Workshop on Unification (UNIF 2005)*. 104–119.

Cheney, J. 2005b. Scrap your nameplate (functional pearl). In *Proceedings of the 10th International Conference on Functional Programming (ICFP 2005)*, B. Pierce, Ed. ACM, Tallinn, Estonia, 180–191.

Cheney, J. 2006. Completeness and Herbrand theorems for nominal logic. *Journal of Symbolic Logic 71,* 1, 299–320.

Cheney, J. 2010. Equivariant unification. *Journal of Automated Reasoning 45,* 3, 267–300.

Cheney, J. 2016. A simple sequent calculus for nominal logic. *Journal of Logic and Computation 26,* 2, 699–726.

Cheney, J. and Momigliano, A. 2007. Mechanized metatheory model-checking. In *PPDP*, M. Leuschel and A. Podelski, Eds. ACM, 75–86.

Cheney, J., Momigliano, A., and Pessina, M. 2016. Advances in property-based testing for αProlog. In *Proceedings of the 10th International Conference on Tests and Proofs (TAP 2016)*, B. K. Aichernig and C. A. Furia, Eds. Lecture Notes in Computer Science, vol. 9762. Springer, 37–56.

Cheney, J. and Urban, C. 2008. Nominal logic programming. *ACM Transactions on Programming Languages and Systems 30,* 5 (August), 26.

Claessen, K. and Hughes, J. 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, 268–279.

Clarke, E. M., Grumberg, O., and Peled, D. A. 2000. *Model Checking*. MIT Press.

Dybjer, P., Haiyan, Q., and Takeyama, M. 2004. Verifying Haskell programs by combining testing, model checking and interactive theorem proving. *Information & Software Technology 46,* 15, 1011–1025.

Felleisen, M., Findler, R. B., and Flatt, M. 2009. *Semantics Engineering with PLT Redex*. The MIT Press.

Fernández, M. and Gabbay, M. 2005. Nominal rewriting with name generation: abstraction vs. locality. In *PPDP*, P. Barahona and A. P. Felty, Eds. ACM, 47–58.

Fetscher, B., Claessen, K., Palka, M. H., Hughes, J., and Findler, R. B. 2015. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. In *Proceedings of ESOP 2015*, J. Vitek, Ed. Lecture Notes in Computer Science, vol. 9032. Springer, 383–405.

Findler, R. B., Klein, C., and Fetscher, B. 2015. Redex: Practical semantics engineering. Online at http://docs.racket-lang.org/redex.

Gabbay, M. 2007. Fresh logic: proof-theory and semantics for FM and nominal techniques. *J. Applied Logic 5,* 2, 356–387.

Gabbay, M. J. 2011. Foundations of nominal techniques: logic and semantics of variables in abstract syntax. *Bulletin of Symbolic Logic 17,* 2, 161–229.

Gabbay, M. J. and Cheney, J. 2004. A sequent calculus for nominal logic. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS 2004)*. IEEE Computer Society, Turku, Finland, 139–148.

Gabbay, M. J. and Pitts, A. M. 2002. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing 13*, 341–363.

Gacek, A. 2010. Relating nominal and higher-order abstract syntax specifications. In *PPDP*, T. Kutsia, W. Schreiner, and M. Fernández, Eds. ACM, 177–186.

Gacek, A., Miller, D., and Nadathur, G. 2012. A two-level logic approach to reasoning about computations. *Journal of Automated Reasoning 49,* 2, 241–273.

Gadducci, F., Miculan, M., and Montanari, U. 2006. About permutation algebras, (pre)sheaves and named sets. *Higher-Order and Symbolic Computation 19,* 2-3, 283–304.

Harland, J. 1993. Success and failure for hereditary Harrop formulae. *J. Log. Program. 17,* 1, 1–29.

Harper, R. and Pfenning, F. 2005. On equivalence and canonical forms in the LF type theory. *ACM Trans. Comput. Logic 6,* 1, 61–101.

Heath, Q. and Miller, D. 2015. A framework for proof certificates in finite state exploration. In *Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, August 2-3, 2015.*, C. Kaliszyk and A. Paskevich, Eds. EPTCS, vol. 186. 11–26.

Klein, C., Clements, J., Dimoulas, C., Eastlund, C., Felleisen, M., Flatt, M., McCarthy, J. A., Rafkind, J., Tobin-Hochstadt, S., and Findler, R. B. 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, New York, NY, USA, 285–296.

Klein, C., Flatt, M., and Findler, R. B. 2012. The Racket virtual machine and randomized testing. *Higher-Order and Symbolic Computation 25,* 2-4, 209–253.

Lakin, M. R. and Pitts, A. M. 2009. Resolving inductive definitions with binders in higher-order typed functional programming. In *Proceedings of the 18th European Symposium on Programming (ESOP 2009)*. 47–61.

Lassez, J.-L. and Marriott, K. 1987. Explicit representation of terms defined by counter examples. *Journal of Automated Reasoning 3,* 3 (Sept.), 301–318.

Leach, J., Nieva, S., and Rodríguez-Artalejo, M. 2001. Constraint logic programming with hereditary Harrop formulas. *TPLP 1,* 4 (July), 409–445.

Levy, J. and Villaret, M. 2010. An efficient nominal unification algorithm. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scottland, UK*. 209–226.

Levy, J. and Villaret, M. 2012. Nominal unification from a higher-order perspective. *ACM Trans. Comput. Logic 13,* 2 (Apr.), 10:1–10:31.

Liu, H., Cheng, E., and Hudak, P. 2009. Causal commutative arrows and their optimization. *SIGPLAN Not. 44,* 9 (Aug.), 35–46.

Mancarella, P. and Pedreschi, D. 1988. An algebra of logic programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, R. A. Kowalski and K. A. Bowen, Eds. ALP, IEEE, The MIT Press, Seatle, 1006–1023.

Miller, D., Nadathur, G., Pfenning, F., and Scedrov, A. 1991. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic 51*, 125–157.

Momigliano, A. 2000. Elimination of negation in a logical framework. In *CSL*, P. Clote and H. Schwichtenberg, Eds. Lecture Notes in Computer Science, vol. 1862. Springer, 411–426.

Momigliano, A. 2012. A supposedly fun thing I may have to do again: A HOAS encoding of Howe's method. In *Proceedings of the Seventh International Workshop on Logical Frameworks and Meta-languages, Theory and Practice*. LFMTP '12. ACM, New York, NY, USA, 33–42.

Momigliano, A. and Pfenning, F. 2003. Higher-order pattern complement and the strict lambda-calculus. *ACM Trans. Comput. Log. 4,* 4, 493–529.

Montanari, U. and Pistore, M. 2005. History-dependent automata: An introduction. In *Advanced Lectures of the 5th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-Moby 2005)*. 1–28.

Moreno-Navarro, J. J. and Muñoz-Hernández, S. 2000. How to incorporate negation in a Prolog compiler. In *PADL 2000*, E. Pontelli and V. S. Costa, Eds. LNCS, vol. 1753. Springer, 124–140.

Muñoz-Hernández, S., Mariño, J., and Moreno-Navarro, J. J. 2004. Constructive intensional negation. In *FLOPS*, Y. Kameyama and P. J. Stuckey, Eds. Lecture Notes in Computer Science, vol. 2998. Springer, 39–54.

Naish, L. 1997. A declarative debugging scheme. *Journal of Functional and Logic Programming 1997,* 3, 3–29.

Niemelä, I. 2006. Answer set programming: A declarative approach to solving search problems. In *JELIA*, M. Fisher, W. van der Hoek, B. Konev, and A. Lisitsa, Eds. Lecture Notes in Computer Science, vol. 4160. Springer, 15–18.

Nipkow, T. and Klein, G. 2014. *Concrete Semantics - With Isabelle/HOL*. Springer.

Owre, S. 2006. Random testing in PVS. In *Workshop on Automated Formal Methods (AFM)*.

Paraskevopoulou, Z., Hritcu, C., Dénès, M., Lampropoulos, L., and Pierce, B. C. 2015. Foundational property-based testing. In *Proceedings of the 6th International Conference on Interactive Theorem Proving (ITP 2015)*, C. Urban and X. Zhang, Eds. Lecture Notes in Computer Science, vol. 9236. Springer, 325–343.

Pientka, B. 2005. Verifying termination and reduction properties about higher-order logic programs. *J. Autom. Reasoning 34,* 2, 179–207.

Pierce, B. C. 2002. *Types and Programming Languages.* MIT Press.

Pierce, B. C., de Amorim, A. A., Casinghino, C., Gaboardi, M., Greenberg, M., Hriţcu, C., Sjöberg, V., and Yorgey, B. 2016. *Software Foundations.* Electronic textbook. Version 4.0. `http://www.cis.upenn.edu/~bcpierce/sf`.

Pitts, A. 2016. Nominal techniques. *ACM SIGLOG News 3,* 1 (Feb.), 57–72.

Pitts, A. M. 2003. Nominal logic, a first order theory of names and binding. *Information and Computation 183*, 165–193.

Pitts, A. M. 2013. *Nominal Sets: Names and symmetry in computer science.* Cambridge University Press.

Ramakrishnan, C. R., Ramakrishnan, I. V., Smolka, S. A., Dong, Y., Du, X., Roychoudhury, A., and Venkatakrishnan, V. N. 2000. XMC: A logic-programming-based verification toolset. In *CAV 2000.* Springer-Verlag, London, UK, 576–580.

Roberson, M., Harries, M., Darga, P. T., and Boyapati, C. 2008. Efficient software model checking of soundness of type systems. In *OOPSLA*, G. E. Harris, Ed. ACM, 493–504.

Roşu, G. and Şerbănuţă, T. F. 2010. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming 79,* 6, 397–434.

Runciman, C., Naylor, M., and Lindblad, F. 2008. Smallcheck and lazy SmallCheck: automatic exhaustive testing for small values. In *Haskell Workshop*, A. Gill, Ed. ACM, 37–48.

Schöpp, U. 2007. Modelling generic judgements. *Electronic Notes in Theoretical Computer Science 174,* 5, 19–35.

Schroeder-Heister, P. 1993. Definitional reflection and the completion. In *Proceedings of the 4th International Workshop on Extensions of Logic Programming (ELP'93)*, R. Dyckhoff, Ed. Lecture Notes in Computer Science, vol. 798. Springer, 333–347.

Schürmann, C. 2009. The Twelf proof assistant. In *TPHOLs*, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds. Lecture Notes in Computer Science, vol. 5674. Springer, 79–83.

Sewell, P., Nardelli, F. Z., Owens, S., Peskine, G., Ridge, T., Sarkar, S., and Strnisa, R. 2010. Ott: Effective tool support for the working semanticist. *J. Funct. Program. 20,* 1, 71–122.

Stuckey, P. J. 1995. Negation and constraint logic programming. *Information and Computation 118,* 1, 12–33.

Tiu, A. and Miller, D. 2010. Proof search specifications of bisimulation and modal logics for the π-calculus. *ACM Trans. Comput. Logic 11,* 2 (Jan.), 13:1–13:35.

Urban, C., Cheney, J., and Berghofer, S. 2011. Mechanizing the metatheory of LF. *ACM Transactions on Computational Logic 12,* 2, 15. 42 pages.

Urban, C. and Kaliszyk, C. 2012. General bindings and alpha-equivalence in Nominal Isabelle. *Logical Methods in Computer Science 8,* 2, 1–35.

Urban, C., Pitts, A. M., and Gabbay, M. J. 2004. Nominal unification. *Theoretical Computer Science 323,* 1–3, 473–497.

Volpano, D., Irvine, C., and Smith, G. 1996. A sound type system for secure flow analysis. *J. Comput. Secur. 4,* 2-3 (Jan.), 167–187.

Walker, D., Mackey, L., Ligatti, J., Reis, G. A., and August, D. I. 2006. Static typing for a faulty lambda calculus. In *ICFP '06: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming.* ACM Press, New York, NY, USA, 38–49.

Weirich, S., Yorgey, B. A., and Sheard, T. 2011. Binders unbound. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming.* ICFP '11. ACM, New York, NY, USA, 333–345.

Winskel, G. 1993. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, Cambridge, MA, USA.