Mechanized Metatheory Model-Checking

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

Alberto Momigliano
LFCS, University of Edinburgh/DSI, University of Milan
amomigl1@inf.ed.ac.uk

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 received comparatively little attention. In this paper, we consider the problem of *bounded model-checking* 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.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Constraint and logic languages; D.2.4 [*Software Engineering*]: Program Verification—Model checking; Assertion checkers ; D.2.5 [*Software Engineering*]: Testing and Debugging—Testing tools; F.3.1 [*Logics and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Logic and constraint programming

General Terms Experimentation, Verification

Keywords nominal logic, model checking, counterexample search

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

A great deal of attention, recently reinvigorated by the POPLMark Challenge [2], 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 extant theorem proving systems that have been employed in metatheory verification[1] have high learning curves, and using them to verify the properties of a nontrivial system requires a great deal of 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 fail due to either flaws in the system or mistakes on the user's part. Determining which 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* [15]. 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 25 years ago, improved techniques for searching the state space efficiently (such as *symbolic model checking* using Boolean decision diagrams [28]) 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. 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 are CPU-bound; thus, they can often leverage hardware advances more readily than theorem provers. 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

PPDP'07 July 14–16, 2007, Wroclaw, Poland.
Copyright © 2007 ACM 978-1-59593-769-8/07/0007. . . $5.00

[1] principally theorem provers such as Twelf, Coq, Isabelle/HOL, HOL

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 facilitated great progress.

We advocate *mechanized metatheory model-checking* as a useful complement to established techniques for analyzing programming languages and related systems. Of course, such systems are usually infinite-state, so cannot be automatically verified in finite time, 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 paper we describe an approach to checking desired properties of formal systems implemented in αProlog, a logic programming language which supports programming with "concrete" names and binding modulo α-equivalence [11]. Our work is the first to show how to find bugs in high-level specifications of programming languages and other calculi *automatically* and *effectively*. We explore techniques based on both negation-as-failure and negation elimination, along with encouraging, though preliminary, experimental results. Our experience 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 or typos.

The structure of the remainder of the paper is as follows. Following a brief introduction to αProlog, Section 2 presents our approach 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 paper. Section 4 discusses a simple implementation of metatheory model-checking in αProlog based on *negation-as-failure* (*NF*). Section 5 discusses a more sophisticated implementation based on *negation elimination* (*NE*), including a discussion of the modifications needed to adapt existing negation elimination algorithms to αProlog and a sketch of the soundness proof for the technique. Section 6 presents experimental results that show that the negation elimination approach may offers performance improvements over the other approaches we considered. Sections 7–9 discuss related and future work and conclude.

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. αProlog is a logic programming language based on *nominal logic*, a first-order theory axiomatizing names and name-binding introduced by Pitts [40] and based on Gabbay and Pitts' swapping-based approach to binding syntax [20]. 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 a detailed semantics, can be found in [11, 12].

In αProlog, there are several built-in types, functions and relations with special behavior. There are distinguished *name types* which are are populated with infinitely many *name constants*. In program text, a name constant is generally a lower-case symbol

```
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')).
```

Figure 1. Terms and substitution

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 : A1, ..., An => C.
```

where `spec` is a label naming the property, `n` is a parameter that bounds the search space, and `A1` through `An` and `C` 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 units, pairs, and function types together with appropriate specifications of properties that one usually wishes to verify.

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. However, substitution can be defined declaratively in αProlog; see Figure 1. For convenience, αProlog provides a function-definition syntax, but this is presently just translated to an equivalent (but far more verbose) relational implementation, using *flattening* (more sophisticated functional logic programming techniques, such as *narrowing* [21], would require the development of novel nominal equational unification algorithms).

After the definition of the `sub` function, we have added some directives that state desired properties of substitution that we wish

```
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) :- tc ([(x,T1)|G], E, T2).
tc(G,app(E1,E2),T)       :- tc(G,E1,T ==> T0),
                            tc(G,E2,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)
                   => 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').
```

Figure 2. Types, contexts, and well-formedness

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

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

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(empty,E,T), steps(E,E')
                    => tc(empty,E',T).
```

Figure 3. Reduction, type preservation, progress, and soundness

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 (and since mode and determinism analysis are not yet available in αProlog), this is something we ought to check. 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, Figure 2. We introduce constructors for simple types including unit, pairing, and function types. The typechecking judgment is standard. In addition, we check some 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, 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, Figure 3. 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 program above. In fact, *every* specification we have ascribed to it is violated. Some of the bugs were introduced deliberately, others were discovered while debugging the specification using an early version of the checking tool. Before proceeding, the reader may wish to try to find all of the errors using his or her preferred formal verification method.

We now describe the results of the specification checker on the above program, along the way giving intuition concerning the counterexample search strategy. First, consider the substitution specifications. The specification checker produces the following typical (though slightly sanitized) output for the first two:

```
Checking for counterexamples to
sub_fun: sub(M,x,N) = M1, sub(M,x,N) = M2 => M1 = M2
Checking depth 1 2 3
Counterexample found:
M = fst(var(x)), M1 = fst(var(x)),
M2 = snd(var(V)), N = var(V)
--------
Checking for counterexamples to
sub_id: sub(M,x,var(x)) = M
Checking depth 1 2
Counterexample found:
M = var(V1)
x # V1
```

The first error is due to the following bug:

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

should be

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

Of course, this kind of problem could also be detected by a mode analysis (which has, however, not yet been developed for αProlog).

The second error 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 4
Counterexample found:
```

```
M = pair(var(x),b)
```

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 4
Counterexample found:
M = fst(lam(x4877\var(x)))
```

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 4
Counterexample found:
E = var(V)
G = [(V,base)]
T = base
T' = base ==> base
--------
Checking for counterexamples to
tc_pres: tc([],M,T), step(M,M') => tc([],M',T)
Checking depth 1 2 3 4 5 6
Counterexample found:
M = app(lam(x\fst(var(x))),b)
M' = b
T = base ** T
```

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

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

to

```
tc(G,var V, T) :- mem((V,T),G).
```

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

```
tc(G,app(E1,E2),T)          :- tc(G,E1,T ==> T0),
                               tc(G,E2,T0).
```

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

```
tc(G,app(E1,E2),T)          :- tc(G,E1,T0 ==> T),
                               tc(G,E2,T0).
```

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

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

should be changed to

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

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

3. Core language

The implementation of αProlog includes a number of high-level conveniences including parameterized types such as lists, polymorphism, function definition notation (as used in $subst$ above), 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 [12, 11, 10, 48]. In particular, to simplify the presentation we consider only monomorphic, non-parametric types.

A *signature* $\Sigma = (\Sigma_D, \Sigma_N, \Sigma_F)$ consists of sets Σ_D and Σ_N of base data types δ and name types ν, respectively, together with a collection 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 $\delta \in \Sigma_D$ and $\nu \in \Sigma_N$. We consider constants of type δ to be function symbols of arity $\mathbf{1} \to \delta$.

Given a signature, the language of *terms* over sets V of variables $x, y, z, \ldots$ and A of names $\mathsf{a}, \mathsf{b}, \ldots$ is defined by the following grammar:

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

We say that a term is *ground* if it has no variables (but possibly does contain names). Suspended permutations appearing before variables are often omitted, that is, we write X for $\mathsf{id} \cdot X$. The abstract syntax $\langle\mathsf{a}\rangle t$ is synonymous with the concrete syntax `a\t` for name-abstraction.

The swapping operation is defined as follows on ground terms:

$$\pi \cdot \langle\rangle = \langle\rangle \quad \pi \cdot f(t) = f(\pi \cdot t) \quad \pi \cdot \langle t, u\rangle = \langle \pi \cdot t, \pi \cdot u\rangle$$
$$\pi \cdot \mathsf{a} = \pi(\mathsf{a}) \quad \pi \cdot \langle\mathsf{a}\rangle t = \langle \pi \cdot \mathsf{a}\rangle \pi \cdot t$$

where $\pi(\mathsf{a})$ denotes the result of applying the permutation π (considered as a function) to a.

We consider atomic formulae for equality ($t \approx u$) and freshness ($t \ \# \ u$), where t is a term of some name type. The freshness relation is defined on ground terms using the following inference rules:

$$\frac{(\mathsf{a} \neq \mathsf{b})}{\mathsf{a} \# \mathsf{b}} \quad \frac{}{\mathsf{a} \# \langle\rangle} \quad \frac{\mathsf{a} \# t}{\mathsf{a} \# f(t)} \quad \frac{\mathsf{a} \# t \quad \mathsf{a} \# u}{\mathsf{a} \# \langle t, u\rangle}$$
$$\frac{\mathsf{a} \# \mathsf{b} \quad \mathsf{a} \# t}{\mathsf{a} \# \langle\mathsf{b}\rangle t} \quad \frac{}{\mathsf{a} \# \langle\mathsf{a}\rangle t}$$

Similarly, the equality relation, which identifies abstractions up to "safe" renaming, is defined on ground terms as follows:

$$\frac{}{\mathsf{a} \approx \mathsf{a}} \quad \frac{}{c \approx c} \quad \frac{t_1 \approx u_2 \quad t_2 \approx u_2}{\langle t_1, t_2\rangle \approx \langle u_1, u_2\rangle} \quad \frac{t \approx u}{f(t) \approx f(u)}$$
$$\frac{\mathsf{a} \approx \mathsf{b} \quad t \approx u}{\langle\mathsf{a}\rangle t \approx \langle\mathsf{b}\rangle u} \quad \frac{\mathsf{a} \# (\mathsf{b}, u) \quad t \approx (\mathsf{a}\ \mathsf{b}) \cdot u}{\langle\mathsf{a}\rangle t \approx \langle\mathsf{b}\rangle u}$$

Given a signature which includes a distinguished base type o of *propositions* along with *predicate symbols* $p : \tau \to o$, we consider *goal* and *(definite) program clause* formulas G and D, respectively, defined by the following grammar:

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

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

$$\frac{\Gamma : \nabla \models \forall X{:}\tau.\, C \quad \Gamma, X{:}\tau : \Delta; \nabla, C \Rightarrow G}{\Gamma : \Delta; \nabla \Rightarrow \forall X{:}\tau.\, G}\ \forall R \quad \frac{\Gamma : \nabla \models \text{И}\mathsf{a}{:}\nu.\, C \quad \Gamma\#\mathsf{a}{:}\nu : \Delta; \nabla, C \Rightarrow G}{\Gamma : \Delta; \nabla \Rightarrow \text{И}\mathsf{a}{:}\nu.\, G}\ \text{И}R \quad \frac{\Gamma : \Delta; \nabla \xrightarrow{D} A \quad D \in \Delta}{\Gamma : \Delta; \nabla \Rightarrow A}\ sel$$

$$\frac{t \approx u}{\Gamma : \Delta; \nabla \xrightarrow{p(t)} p(u)}\ hyp \quad \frac{\Gamma : \Delta; \nabla \xrightarrow{D_i} A}{\Gamma : \Delta; \nabla \xrightarrow{D_1 \wedge D_2} A}\ \wedge L_i \quad \frac{\Gamma : \Delta; \nabla \xrightarrow{D} A \quad \Gamma : \Delta; \nabla \Rightarrow G}{\Gamma : \Delta; \nabla \xrightarrow{G \supset D} A}\ \supset L \quad \frac{\Gamma : \nabla \models \exists X{:}\tau.\, C \quad \Gamma, X{:}\tau : \Delta; \nabla, C \xrightarrow{D} A}{\Gamma : \Delta; \nabla \xrightarrow{\forall X{:}\tau.\, D} A}\ \forall L$$

Figure 4. Proof search semantics of αProlog programs

The novel features of nominal logic programs consist of the freshness and equality constraints ($t \# u, t \approx u$) described above and the Gabbay-Pitts fresh-name quantifier И. The latter, intuitively, quantifies over names not appearing in the formula (or in the values of its variables); that is, provided the free variables and names of ϕ are $\{\mathsf{a}, \vec{X}\}$, the formula $\text{И}\mathsf{a}{:}\nu.\ \phi$ is logically equivalent to $\exists A{:}\nu.\ A \# \vec{X} \wedge \phi$.

Although we permit programs to be defined using arbitrary (sets of) definite clauses Δ, we take advantage of the fact that such programs can always be normalized to sets of clauses of the form $\forall\vec{X}.\ G \supset p(t)$. It is useful to single out in a normalized program Δ all the clauses that belong to the definition of a predicate; thus, we define $\mathrm{def}(p, \Delta) = \{D \mid D \in \Delta, D = \forall\vec{X}.\ G \supset p(t)\}$.

As mentioned earlier, the implementation of αProlog includes a notationally convenient syntax for defining functions. Functions f of n arguments can be defined using clauses of the form $f(\vec{t}) = u :\!-\ G$; these are translated to $n+1$-ary predicate clauses $p_f(\vec{t}, u) :\!-\ G$. Uses of f within goals such as $G[f(\vec{t})]$ are translated to subgoals of the form $\exists X.\ p_f(\vec{t}, X) \wedge G[X]$.

A similar technique can be used to provide more convenient *elimination forms* for pair and abstraction types. The projection functions $\pi_i : \tau_1 \times \tau_2 \to \tau_i$ can be defined directly using the equation $\pi_i(X_1, X_2) = X_i$. The elimination form for abstraction, called *concretion* and written $t \,@\, \mathsf{a}$, can be implemented by translating $G[t \,@\, \mathsf{a}]$ to $\exists X.\ t \approx \langle\mathsf{a}\rangle X \wedge G[X]$. We use these elimination forms freely in the rest of the paper, subject to the proviso that they must be expanded before any program transformations are applied.

Finally, concrete-syntax program clauses containing free names and variables such as

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

are viewed as equivalent to closed formulas such as:

$$\begin{array}{l}\forall G, F, T, U.\\ (\text{И}\mathsf{x}.\ F = \langle\mathsf{x}\rangle M \wedge \mathsf{x} \# G \wedge tc([(\mathsf{x}, T)|G], M, U))\\ \supset tc(G, lam(F), T \Longrightarrow U)\end{array}$$

Note that this transformation yields a proper definite clause in which И is used only in the subgoal. Such clauses have been studied in previous work [12, 48] and it is known that resolution based on nominal unification is sound and complete for proof search for this case, in contrast to the general case where a more complicated *equivariant unification* problem must be solved [7].

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 $\Gamma\#\mathsf{a}{:}\nu$.

We define *constraints* to be G-formulas of the following form:

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

We write ∇ for a set of constraints. Constraint-solving is modeled by the satisfiability judgment $\Gamma : \nabla \models C$. For constraints ∇, C containing only free variables/names from Γ, this judgment means that for every ground instantiation θ of the variables consistent with freshness assertions in Γ, if $\bigwedge \theta(\nabla)$ holds, then $\theta(C)$ holds.

Efficient algorithms for constraint solving and unification for nominal terms of the above form and for freshness constraints of the form $\mathsf{a} \# t$ were studied by Urban, Pitts, and Gabbay [49]. Note, however, that we also consider freshness constraints of the form $\pi \cdot X \# \pi' \cdot Y$. These constraints are needed to express the alpha-inequality predicate neq (see Section 5.2). Constraint solving and satisfiability become NP-hard in the presence of these constraints [13, Ch. 7]. In the current implementation, such constraints are delayed until the end of proof search, and any remaining constraints of the form $\pi \cdot X \# \pi' \cdot X$ are checked for consistency by brute force; these are essentially finite domain constraints. Any remaining constraints $\pi \cdot X \# \pi' \cdot Y$ where X and Y are not necessarily equal are always satisfiable.

We use here a proof-theoretic model of the semantics of αProlog programs, introduced in [10, 12], and shown here in Figure 4. This semantics allows us to focus on the high-level proof search issues relevant to proving the correctness of negation elimination, without requiring us to introduce or manage low-level operational details concerning constraint solving. The only novelty is the presence of rules for universally quantified goals, which are handled here as in λProlog. The semantics defines two judgments: goal-directed or *uniform* proof search $\Gamma : \Delta; \nabla \Rightarrow G$, and program clause-directed or *focused* proof search $\Gamma : \Delta; \nabla \xrightarrow{D} A$, where Δ is a set of clauses and ∇ a set of constraints.

4. Implementation using negation-as-failure

We consider informal `#check` specifications to correspond to specification formulas of the form

$$\text{И}\vec{\mathsf{a}}.\forall\vec{X}.\ H_1 \wedge \cdots \wedge H_n \supset A \qquad (1)$$

where $H_1, \ldots, H_n, A$ are atomic formulas (including equality and freshness constraints). A *(finite) counterexample* is a closed substitution θ such that $\theta(H_1), \ldots, \theta(H_n)$ all hold (that is, are derivable), but the conclusion $\theta(A)$ finitely fails (that is, is not derivable in finitely many steps).

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.

We consider two approaches to solving this problem using negation-as-failure. First, in principle, we could simply enumerate all possible *valuations* and test them, using negation-as-failure.

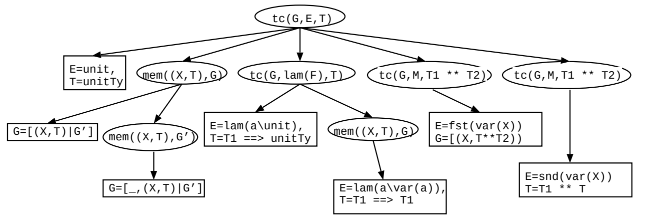

Figure 5. "Finished" derivations for `tc(G,E,T)` up to depth 3

More precisely, given predicates $gen[\![\tau]\!] : \tau \to o$ for each type τ (see Figure 6), which generate all possible values of type τ, we may translate a specification of the form (1) to a goal

$$\text{И}\vec{\mathsf{a}}.\exists\vec{X}{:}\vec{\tau}.\ gen[\![\tau]\!](X_1) \wedge \cdots \wedge gen[\![\tau]\!](X_n) \wedge \vec{H} \wedge not(A) \quad (2)$$

where, here, $not(A)$ is the ordinary negation-as-failure familiar from Prolog. 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 iterative deepening should find a counterexample, if one exists.

Second, we consider an approach that enumerates *derivations* of the hypotheses and then tests whether the negated conclusion is satisfiable under the resulting answer constraint. This avoids wasteful backtracking due to premature commitment to a ground instantiation. For example, if we have

$$gen[\![\tau]\!](X), gen[\![\tau]\!](Y), bar(Y), foo(X), not(baz(X,Y))$$

and we happen to generate an X that simply 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 seems more efficient to use the *definitions* of foo and bar to guide search towards suitable instantiations of X and Y. Compared with the ground substitution enumeration technique above, this derivation-first approach simply delays the gen predicates until after the hypotheses:

$$\text{И}\vec{\mathsf{a}}.\exists\vec{X}{:}\vec{\tau}.\ \vec{H} \wedge gen[\![\tau]\!](X_1) \wedge \cdots \wedge gen[\![\tau]\!](X_n) \wedge not(A) \quad (3)$$

In fact, we only need to generate instantiations for the variables X_i actually appearing in A, since only they need to be ground to ensure that negation-as-failure is well-behaved. Of course, the order of the other hypotheses $\vec{H}$ 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 hypotheses $\vec{H}$ 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 5 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 implemented a variant of the αProlog interpreter that limits the number of resolution steps taken when trying to prove a hypothesis to the bound b provided as part of the `#check` declaration, and uses the *ad hoc* limit $2 * b$ when solving $not(A)$. The reason for the larger bound on the conclusion is that often the derivation of the conclusion combines the derivations of the hypotheses (consider, for example, the derivation transformations involved in proofs of substitution lemmas). We only report a counterexample when the

$$\begin{array}{rcl}
gen[\![\tau]\!] & : & \tau \to o \\
gen[\![\mathbf{1}]\!](t) & = & t \approx \langle\rangle \\
gen[\![\tau_1 \times \tau_2]\!](t) & = & gen[\![\tau_1]\!](\pi_1(t)) \wedge gen[\![\tau_2]\!](\pi_2(t)) \\
gen[\![\delta]\!](t) & = & gen_\delta(t) \\
gen[\![\langle\nu\rangle\tau]\!](t) & = & \text{И}\mathsf{a}{:}\nu.\ gen[\![\tau]\!](t @ \mathsf{a}) \\
gen[\![\nu]\!](t) & = & \top \\
gen_\delta(t) & :- & \bigvee\{\exists X{:}\tau.\ t = f(X) \wedge gen[\![\tau]\!](X) \\
 & & \quad \mid f : \tau \to \delta \in \Sigma\}
\end{array}$$

Figure 6. Term-generator predicates

negated goal fails in fewer than $2 * b$ steps, not because it runs out of resources; the latter case could result in false positives.

The $gen[\![\tau]\!]$ predicates are currently implemented as a built-in generic functions in αProlog. On user request, the interpreter generates predicates gen_δ for all user-defined datatypes δ; these can be used within `#check` directives. Note that we do not exhaustively instantiate base types such as name-types; instead, we just use a fresh variable to represent all possible names.

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

- Negation-as-failure is unsound for non-ground goals, so we must blindly instantiate all free variables before solving the negated conclusion[2]. This is expensive and prevents optimizations by reordering the negated conclusion.
- Proving soundness and completeness of counterexample search requires proving properties of negation-as-failure in αProlog which have not yet been studied.
- Nested negations are not well-behaved, so we cannot use negation in "pure" programs or specifications we wish to check.

We therefore consider another approach, based on *negation elimination*.

5. Negation Elimination

Logical frameworks with a logic programming interpretation such as αProlog or hereditary Harrop formulae cannot directly express negative information, although negation is a useful specification tool. The issue of negation in logic programming has been extensively researched, though negation-as-failure has proven to be universally accepted from a practical standpoint, in spite of its notorious difficulties, namely the lack of a unique intuitive semantics. However, when we are interested in using a logic programming language as a *logical* framework, where adequacy of the encoding of object logics is paramount, we have to be more picky and select a sound notion of negation.

Negation elimination (*NE*) is a source-to-source transformation aimed at replacing negated subgoals with calls to "equivalent" positively defined predicates. *NE* bypasses complex semantic and implementation issues arising for *NF* since, in the absence of local variables, it yields an ordinary (α)Prolog program whose success set is included (or coincides in the case of terminating programs) with the complement of the success set of the original negated predicate. In the context of specification checking, negation elimination has several advantages: it avoids the expensive term generation step

[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.

needed to ground free variables, it is more transparently correct, and it may open up other opportunities for optimization.

Replacing occurrences of negated predicates with positive ones which are operationally equivalent entails two phases:

- *Complementing (nominal) terms/patterns*. Certainly, one reason a goal 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 [25], which we formally define in Section 5.1.
- *Complementing clauses*. This can be seen as a negation normal form procedure which is consistent with the operational semantics of the language. The idea of the clause complementation algorithm is thus 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. The contribution of each of the original clauses are finally merged via unification.

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, notation $\|t\|$. According to this interpretation, the *complement* of t ($\mathrm{Not}(t)$) is the set of ground terms which are *not* instances of t, *i.e.* are in the set-theoretic complement of $\|t\|$. It is natural to generalize this to the notion of *relative* complement, a suitable representation of all the ground instances of a given (finite) set of terms which are not instances of another given one, in symbols:

$$\|t_1, \dots, t_n\| - \|u_1, \dots, u_m\|$$

where dots represent (set theoretic) union. Let $FV(t_1, \dots, t_n) = \vec{x}$ disjoint from $FV(u_1, \dots, u_m) = \vec{y}$. Then the relative complement problem can be also expressed by the following (restricted) form of *equational problem* [16], where the z_i's are free variables.

$$\exists\vec{x}\forall\vec{y}.\ \bigwedge_{i=1}^{n} z_i = t_i \wedge \bigwedge_{i=1}^{m} z_i \neq u_i$$

A complement operator must satisfy 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 $\mathrm{Not}(t)$.
2. Exhaustivity: s is a ground instance of t or s is a ground instance of $\mathrm{Not}(t)$.

As it was initially remarked in [25], 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 which are then constrained in the body by $\approx$.

However, complementing nominal terms, similarly to the higher-order case, introduces new issues. In fact, even restricting to patterns, (intuitionistic) lambda calculi are not closed under complementation, due the presence of *partially applied* lambda terms. Consider a higher-order pattern `(lam [x] E)`, 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 may be solved by developing a strict lambda calculus, where we can directly express whether a function depends on its argument [31]. This is not an issue for nominal terms as presented in αProlog, since we do not

$$\text{no rule for } \mathbf{1}, \nu, \langle\nu\rangle\tau$$

$$\frac{g \in \Sigma, g : \tau \to \delta, f \not\equiv g}{\mathrm{Not}(f(t)) \Rightarrow g(X) : \delta}\ \mathrm{Not}\_f^1$$

$$\frac{\mathrm{Not}(t) \Rightarrow s : \tau}{\mathrm{Not}(f(t)) \Rightarrow f(s) : \delta}\ \mathrm{Not}\_f^2$$

$$\frac{\mathrm{Not}(t_1) \Rightarrow s_1 : \tau_1}{\mathrm{Not}(\langle t_1, t_2\rangle) \Rightarrow \langle s_1, X_2\rangle : \tau_1 \times \tau_2}\ \mathrm{Not}\_p^1$$

$$\frac{\mathrm{Not}(t_2) \Rightarrow s_2 : \tau_2}{\mathrm{Not}(\langle t_1, t_2\rangle) \Rightarrow \langle X_1, s_2\rangle : \tau_1 \times \tau_2}\ \mathrm{Not}\_p^2$$

Figure 7. Term complement

consider logical variables at function types. However, 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 would yield not only some non-determinism *w.r.t.* the rules for complementing abstractions, but it also means that the complement of a term (containing free or bound names) cannot again be represented by a finite set of nominal terms. This would not be problematic if we took the (constraint) disunification route; however, as far as negation elimination is concerned, it is simpler to restrict to a core language that is 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 clauses heads. These are replaced by logic variables appropriately constrained in the clause body by a concretion to a fresh name. Note also that the syntax of D-formulas in Section 3 already enforces this, as И quantification is not allowed over clause heads.

Thus, we can simply use a nondeterministic, type directed, version of the standard rules for first-order term complementation (listed in Figure 7), where the X's are fresh logic variables and $f : \tau \to \delta$. Define $\mathrm{Not}(t) = \mathcal{N} : \tau$ iff $\mathcal{N} = \{n \mid \mathrm{Not}(t) \Rightarrow n : \tau\}$. The correctness of the algorithm follows from previous results [3, 31]. We intend to address the general case of nominal term complement in future work (Section 8).

5.2 Clause Complementation *via* generic operations

Clause complementation is usually described in terms of contraposition of the *only-if* part of the completion [3, 6, 33]. We instead present a more direct, 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_δ, *etc.* as the names of the generated clauses (*cf.* analogous notions in [18]). Each of these clauses is defined as shown in Figure 8, together with auxiliary, type-indexed functions $neq[\![\tau]\!]$, $nfr[\![\tau]\!]$, *etc.* which are used to construct appropriate subgoals for each type. In particular, note that recursion through data types δ is broken through calling the predicates neq_δ, $nfr_{\nu,\delta}$.

Complementing goals (Figure 10) is intuitive: we just put the latter in negation normal form, respecting the operational semantics of failure. Note that the self-duality of the И-quantifier (*cf.* [40, 20]) allows goal negation to be applied recursively. The existential 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 [26]; if a clause contains a existential variable (more commonly known as a *local* variable) *i.e.* a

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

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

Figure 8. Inequality and non-freshness

logic 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. In the latter $\forall X : A.\, G$ succeeds iff $G[a/X]$ succeeds, for a new eigenvariable a, while extensional 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. Since this is hardly practical to implement from the logic programming standpoint, extensional quantification has been interpreted by case analysis [5] and *SLD*-derivations have been extended with such a step. Figure 9 shows the proof search semantics of the $\forall^*$-quantifier.

Clause complementation is now unsurprising: given a rule $\forall(q(t) \leftarrow G)$, its complement must contain a 'factual' part motivating failure due to clash with the head; the remainder $\mathrm{Not}_{\mathrm{G}}(G)$ expresses failure in the body, if any. This is accomplished in Figure 11 by the $\mathrm{Not}_{\mathrm{i}}^{\mathrm{D}}$ function, where a set of negative facts is built via term complementation $\mathrm{Not}(t)$; 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 with a clause for a new predicate symbol, say $p^\neg(X)$, which calls all the $p_i^\neg$ (Figure 12).

We list in Figure 13 the complement of the typechecking predicate from Section 2. This results from a final (not yet fully implemented) optimization pass consisting of renaming and inlining the definitions of the $p_i^\neg$. As expected, local variables in the application and projection cases yield the corresponding $\forall^*$-quantified bodies.

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

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

$$
\frac{\Gamma : \Delta; \nabla \Rightarrow \text{И}\mathsf{a}{:}\nu.\forall^* Y{:}\tau.\ G[\langle\mathsf{a}\rangle Y/X]}{\Gamma : \Delta; \nabla \Rightarrow \forall^* X{:}\langle\nu\rangle\tau.\ G}\ \forall^*\mathsf{abs}
$$

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

Figure 9. Proof rules for extensional universal quantification

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

Figure 10. Negation of a goal

$$
\begin{array}{rcl}
\mathrm{Not}_{\mathrm{i}}^{\mathrm{D}}(\forall(p(t) :- G))) & = & \bigwedge\{\forall(p_i^\neg(u)) \mid \mathrm{Not}(t) \Rightarrow u : \tau\} \\
 & & \land\forall(p_i^\neg(t) :- \mathrm{Not}_{\mathrm{G}}(G))
\end{array}
$$

Figure 11. Negation of a single clause

$$
\begin{array}{l}
\mathrm{Not}^{\mathrm{D}}(\forall(p(t_1) :- G_1, \ldots, p(t_n) :- G_n)) = \\
\quad \forall(\bigwedge_i \mathrm{Not}_{\mathrm{i}}^{\mathrm{D}}(p(t_i) :- G_i)) \\
\quad \land\ \forall X.\ (p^\neg(X) :- \bigwedge_i p_i^\neg(X))
\end{array}
$$

Figure 12. Negation of all clauses defining p

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

Figure 13. Negation of typechecking predicate

The most important property for our intended application is soundness, which we state in terms of exclusivity of clause complementation. Let $\Delta^- = \Delta \cup \mathrm{Not}^{\mathrm{D}}(\Delta)$.

THEOREM 1 (Exclusivity). *1. It is not the case that* $\Gamma : \Delta; \nabla \Rightarrow G$ *and* $\Gamma : \Delta^-; \nabla \Rightarrow \mathrm{Not}_{\mathrm{G}}(G)$.

2. It is not the case that $\Gamma : \Delta; \nabla \xrightarrow{D} p(t)$ *and* $\Gamma : \Delta^-; \nabla \xrightarrow{\mathrm{Not}^{\mathrm{D}}(D)} \neg p(t)$

The proof, by mutual induction on the derivation of $\Gamma : \Delta; \nabla \Rightarrow G$ and $\Gamma : \Delta; \nabla \xrightarrow{D} p(t)$, follows the lines in [30].

Completeness can be stated as follows: if a goal G finitely fails from a program D, then its complement $\mathrm{Not}_{\mathrm{G}}(G)$ should be provable from $\mathrm{Not}^{\mathrm{D}}(D)$. In a model checking context, this is is a desirable, though not vital property. In fact, it is well known [19] that the semantics of *NF* has a very high degree of unsolvability and completeness results have been proven *w.r.t.* a three-valued semantics, due to Kunen [24]. Logic programs define recursively enumerable relations, and it is only possible to define the complement of an r.e. relation if and only if it is recursive. We therefore cannot expect true completeness results unless we restrict to recursive programs. As a simple example, if Δ defines the r.e. predicate $halts$ which recognizes Turing machines that halt on their inputs, it is obvious that the predicate $\neg halts$ defined by $\mathrm{Not}^{\mathrm{D}}(\Delta)$ cannot define the exact complement of $halts$. Determining whether a logic program defines a recursive relation is an orthogonal issue, but see, *e.g.* the termination analysis approach taken in the Twelf system [38]. Nevertheless, we do not believe completeness is necessary for our approach to be useful, even for systems with undecidable predicates such as first-order sequent calculi or undecidable typing or subtyping relations.

6. Experimental results

We implemented counterexample search in the αProlog interpreter using both negation-as-failure and negation-elimination, as outlined in the previous section. In this section, we present performance results comparing these approaches. We consider the examples presented in Section 2 as test data, using both the "buggy" version we have presented and a version with all bugs that were detected by the checker fixed.

We performed informal experiments comparing the two approaches based on negation-as-failure. Not surprisingly, we found that placing the generator predicates at the end of the list of hypotheses rather than at the beginning made a dramatic difference—as did the ordering of subgoals in certain cases. We omit experimental results for the naive (generators-first) *NF* approach since it appears obvious that the derivation-first approach is always preferable. The machine used was an AMD Athlon 3000+ running Ubuntu Linux v6.10 with 1GB RAM.

For the negation-elimination approach, 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).

We attempted to identify the best orderings for the subgoals in each approach. For example, in the *NE*-based approach, some subgoals (such as context well-formedness predicates `wf_ctx`) can be delayed past the negation predicate to improve performance.

Time to find counterexamples We first measured the time needed by each approach to find counterexamples. The counterexamples found were not necessarily the same. For the checks involving substitution, all counterexamples were found in less than 0.01 seconds. Table 1 shows the times needed for the checks involving typechecking, in seconds. One counterexample could not be found by *NF* within five seconds.

	NF	*NE*	*NE*$^-$
`tc_weak`	0	3.36	0.13
`tc_subst`	0	3.06	0.32
`tc_pres`	0.02	0.01	0.02
`tc_prog`	0.08	0.03	0.1
`tc_sound`	N/A	0.19	0.22

Table 1. Time to find counterexamples

These results are encouraging, because they suggest that both *NE* and *NE*$^-$ remain competitive with *NF*, despite the fact that they potentially cover much more of the search space within a given depth bound. Moreover, in at least one case (`tc_sound`) the *NE* approaches were able to find a counterexample that *NF* was not able to within a reasonable interactive time. Finally, these results suggest that in many cases (at least for finding "shallow" bugs), the relatively incomplete search strategy of *NE*$^-$ can be used in place of *NE*. The faster *NE*$^-$ approach can be used first, with *NE* used if no counterexamples are found using *NE*$^-$.

Time to exhaust a finite search space We next measured the amount of time it takes for a given approach to exhaust its search space up to a given depth bound n. For each technique and test, we measured this time for $n = 1, 2, \ldots$ up until the point where search time exceeded a "reasonable" few seconds. The experimental results are shown in Table 2. The first column shows the name of the checked property; the second shows the search depth n used. For each test, we used the largest n for which all three approaches were successful in a reasonable period of time.

These results are mixed. In some cases, particularly those involving substitution, *NE* and *NE*$^-$ are clearly much more efficient than the *NF* approach. In others, particularly key lemmas such as substitution and weakening, *NE* often takes significantly (up to a factor of 4) longer, with *NE*$^-$ usually in between. For `tc_sound` the situation is particularly bad, with *NE* 15-20 times slower than *NF* or *NE*$^-$. On the other hand, for the `tc_prog` checks, both *NE*-based techniques are faster.

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, the comparisons among the approaches up to a given bound may be apples-to-oranges. 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, say that the search space considered by *NE* for a given bound n contains the spaces considered by the other two; thus, in cases where *NE* does better than another approach (as with `tc_prog`), it is clear that this is a significant improvement.

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.

A second source of inefficiency, which accounts for the difference between *NE*$^-$ and *NE*, is the extensional quantifier $\forall^*$, and in particular its implementation 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

	n	*NF*	*NE*	*NE*$^-$
`sub_fun`	3	4.5	2.42	3.02
`sub_id`	4	0.09	0.25	0.27
`sub_fresh`	4	13.98	0.15	0.16
`sub_comm`	3	2.39	6.01	5.09
`tc_weak`	4	5.31	21.19	16.95
`tc_subst`	4	12.43	98.44	64.7
`tc_pres`	6	0.73	1.04	0.91
`tc_prog`	7	8.89	3.75	4.51
`tc_sound`	6	1.49	36.44	1.98

Table 2. Time to search up to bound n

so appears to require significant alterations to the implementation, such as support for higher-order programming.

Further experience In addition to the relatively straightforward lambda-calculus example discussed above, we have used the *NF*-based implementation to check for errors in several substantial examples, including:

- LF typechecking and equivalence algorithms [23]
- The $F_{\leq}$ language described in the POPLMark Challenge (implemented in αProlog by Matthew Fairbairn [17])
- λ^{zap}, a "faulty lambda calculus" [50]
- A (type-unsafe) mini-ML language with polymorphism and references

We did not expect to find previously unknown errors in these systems; however, the checker 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 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.

Our experiences with the *NF*-based implementation have been positive (though this is our opinion and obviously subject to bias). Most mainstream type systems appear easy to translate into αProlog clauses, and 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 the checker avoids false positives, but does not necessarily show that the checker is helpful during the development of a system. Further study would be needed to establish this, perhaps via usability studies.
2. It is not enough 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. 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. Many alternative search strategies and optimizations (e.g. random search, coroutining, "most constrained goal first", tabling), could be considered to improve performance.

7. Related work

As stated earlier, our approach draws inspiration from the success of finite state model-checking systems [15]. Particularly relevant is the idea of *symbolic* model checking [28], in which data structures such as Boolean decision diagrams are used to represent large numbers of similar states; in our approach, answer substitutions and constraints with free variables play a similar role.

Another related technique is automated testing in functional programming languages, for example in the QuickCheck system for Haskell [14]. QuickCheck provides type class libraries for "generator" functions that construct test data, and a logical specification language to describe the properties the program should satisfy. In conjunction with an implementation of nominal abstract syntax (such as FreshLib [8]), QuickCheck could be used to implement metatheory model-checking. Using Haskell's lazy evaluation strategy, QuickCheck is also capable of searching for partially-instantiated counterexamples. Random testing and counterexample generation has also been considered in theorem proving systems such as Isabelle/HOL [4].

Analyses for checking modes, coverage, termination, and other program properties can be used to verify program properties; this technique plays an important role in the Twelf system [37, 38]. 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 [30].

In addition, there is a body of related work on *declarative debugging* in logic programming languages. Generally, in declarative debugging [34], the programmer identifies a bug (e.g. an undesired answer that is produced or a desired answer that is not produced) and the debugger analyzes the proof search process (using feedback from an oracle, such as the programmer) to help identify the responsible clause(s). More recently, Puebla *et al.* [41] have considered automated debugging using assertions which state desired properties of constraint logic programs. In our setting, declarative debugging could be helpful in explaining why a counterexample is produced.

Negation and extensional quantification There is a large literature on negation as failure, constructive/intensional negation, and disunification; we restrict attention only to closely related work.

Negation elimination (aka *intensional* negation) has a long parallel history dating back the late 80's [3] and later extended to CLP's [6], although no concrete implementation has been reported until Muñoz-Hernández's thesis and subsequent papers [33, 32]. In all these papers, negative predicates are *schematically* synthesized by various manipulations of their *completion*; moreover, existential goals (that is, goals with local variables) are handled by a somewhat opaque case-analysis technique. Our approach, instead, adapts the judgmental techniques in [36, 30]; we have presented a judgmental and syntax-directed translation, modeled closely on our implementation, which shows how to translate programs with negation to programs without negation (but possibly involving $\forall^*$ to handle existential quantification).

Proof search via case analysis, as in the extensional universal quantifier $\forall^*$, has been studied in several settings. A principle of "proof by case analysis" was studied in [3, 5] and then somewhat refined in [33]. The proof-theory of success and failure of existential goals has been investigated in [22], although not in the presence of generic (intensional) universal quantification. A related approach is *constructive negation*, in particular as formulated by Stuckey [46], in which negated existential subgoals are handled via a combination of case analysis and disunification. In logic programming, the correctness of this form of reasoning has been formulated in terms of Kunen's three-valued semantics [24].

When logic programs are seen as inductive relations, the case-analysis reasoning performed by $\forall^*$ is reminiscent of an iterated use of case analysis on an (algebraic) data-type (*cf.* `case_tac` in systems such as Isabelle), or better as an approximation of the ω-rule in the proof-theory of arithmetic. As such, this is connected to *definitional reflection* [45], although in our application we are allowed to reason by cases "on the right" of the sequent (as an introduction rule), as opposed to traditional meta-logics where case analysis corresponds to inversion, *i.e.* an elimination rule.

Model checking and logic programming The Bedwyr system [47, 29] is a generalization of logic programming based on *definitions* that allows 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 principles of (co)induction. The relationship of (a fragment of) this framework with nominal logic has been investigated elsewhere [9]; in particular, in Bedwyr it is possible to capture both finite success and finite failure, as a negated atom $\Gamma \vdash \neg A$ is seen as $\Gamma, A \vdash \bot$ and solved by case analysis (definitional reflection) on A. However, this treatment seems to be sound only *w.r.t.* the Horn+∇ fragment of the logic [44], hence checks involving judgments that rely on higher-order abstract syntax (hypothetical judgments) such as `tc_pres` cannot be directly expressed in Bedwyr without moving to a more intricate "2-levels" approach [27].

The Logic-Programming-Based Model Checking project at Stony Brook (`http://www.cs.sunysb.edu/~lmc/`) implements the model checker XMC for value-passing CCS and a fragment of the mu-calculus on top the XSB tabled logic programming system [42, 43], 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 [35] 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 the same as Datalog, although constraints can be used to increase expressivity. These two paradigms do not seem to provide support for binding syntax that is required for formalizing and checking metatheoretic properties. On the other hand, optimizations such as tabling could certainly be useful, for example to improve $\forall^*$ performance.

8. Future work

At present, the implementation based on negation-elimination is more fragile than the *NF* version; we plan to improve the robustness of both implementations and include them in the next release of αProlog. From a pragmatical standpoint, the implementation of universal quantification currently involves analyzing type information into 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 paper, 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 would also be useful for extending Twelf-like static analysis to αProlog; in fact *coverage* analysis can be stated as a relative complement problem.

We also are interested in comparing our approach with related logic programming-based model-checking or verification techniques such as Bedwyr [47], XMC [43], and ProVerif [1].

9. Conclusions

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 great deal of attention recently, verification tools still require a great deal of effort to learn and use successfully. We have presented an alternative approach called *metatheory model-checking* which addresses 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* (*NF*) and *negation elimination* (*NE*). We outlined the modifications needed to accommodate negation elimination in nominal logic programs and discussed experimental results that show that both techniques have reasonable performance. We plan to address several obvious performance issues in *NE* in future work. However, we believe that the current implementations based on negation-as-failure and unoptimized negation elimination are already useful for debugging languages formalized using αProlog.

Acknowledgments Cheney is supported by EPSRC grant R37476; Momigliano is partially funded by the European Project Mobius within the frame of IST 6th Framework. This paper reflects only the authors' views and the Community is not liable for any use that may be made of the information contained therein. We wish to thank the DSI, University of Milan, for supporting Cheney's visit to Milan. A preliminary version of this work was presented by Cheney at the 1st ACM SIGPLAN Workshop on Mechanizing Metatheory, Portland, Oregon, USA, in September 2006. We gratefully acknowledge comments on the work from a number of workshop participants.

References

[1] M. Abadi and B. Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. *Journal of the ACM*, 52(1):102–146, Jan. 2005.

[2] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLMARK CHALLENGE. In J. Hurd and T. F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.

[3] R. Barbuti, P. Mancarella, D. Pedreschi, and F. Turini. A transformational approach to negation in logic programming. *Journal of Logic Programming*, 8:201–228, 1990.

[4] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In *SEFM '04: Proceedings of the Software Engineering and Formal Methods, Second International Conference on (SEFM'04)*, pages 230–239, Washington, DC, USA, 2004. IEEE Computer Society.

[5] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Universal quantification by case analysis. In *Proc. ECAI-90*, pages 111–116, 1990.

[6] P. Bruscoli, F. Levi, G. Levi, and M. C. Meo. Compilative constructive negation in constraint logic programs. In S. Tison, editor, *Proc. Trees in Algebra and Programming - CAAP'94, 19th International Colloquium*, volume 787, pages 52–76. Springer, 1994.

[7] J. Cheney. Equivariant unification. In *Proceedings of the 2005 Conference on Rewriting Techniques and Applications (RTA 2005)*, number 3467 in LNCS, pages 74–89, Nara, Japan, 2005. Springer-Verlag.

[8] J. Cheney. Scrap your nameplate (functional pearl). In B. Pierce, editor, *Proceedings of the 10th International Conference on Functional*

Programming (ICFP 2005), pages 180–191, Tallinn, Estonia, 2005. ACM.

[9] J. Cheney. A simpler proof theory for nominal logic. In *FOSSACS 2005*, number 3441 in LNCS, pages 379–394. Springer-Verlag, 2005.

[10] J. Cheney. The semantics of nominal logic programs. In *ICLP 2006*, volume 4079 of *LNCS*, pages 361–375, 2006.

[11] J. Cheney and C. Urban. Alpha-Prolog: A logic programming language with names, binding and alpha-equivalence. In *ICLP 2004*, number 3132 in LNCS, pages 269–283, 2004.

[12] J. Cheney and C. Urban. Nominal logic programming. Technical Report cs.PL/0609062, arXiv e-print, 2006. Submitted.

[13] J. R. Cheney. *Nominal Logic Programming*. PhD thesis, Cornell University, Ithaca, NY, August 2004.

[14] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, pages 268–279. ACM, 2000.

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

[16] H. Comon. Disunification: a survey. In J.-L. Lassez and G.Plotkin, editors, *Computational Logic*. MIT Press, Cambridge,MA, 1991.

[17] M. Fairbairn. Solution to part 3 of the POPLMark Challenge. Available at the POPLMark Wiki, `http://fling-l.seas.upenn.edu/~plclub/cgi-bin/poplmark`.

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

[19] M. Fitting. Fixpoint semantics for logic programming: a survey. *Theor. Comput. Sci.*, 278(1-2):25–51, 2002.

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

[21] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19–20:583–628, 1994.

[22] J. Harland. Success and failure for hereditary Harrop formulae. *J. Log. Program.*, 17(1):1–29, 1993.

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

[24] K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4:289–308, 1987.

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

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

[27] R. C. McDowell and D. A. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic (TOCL)*, 3(1):80–136, 2002.

[28] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[29] D. Miller and A. Tiu. A proof theory for generic judgments. *ACM Trans. Comput. Logic*, 6(4):749–783, 2005.

[30] A. Momigliano. Elimination of negation in a logical framework. In P. Clote and H. Schwichtenberg, editors, *CSL*, volume 1862 of *Lecture Notes in Computer Science*, pages 411–426. Springer, 2000.

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

[32] J. J. Moreno-Navarro and S. Muñoz-Hernández. How to incorporate negation in a Prolog compiler. In E. Pontelli and V. S. Costa, editors, *PADL 2000*, volume 1753 of *LNCS*, pages 124–140. Springer, 2000.

[33] S. Muñoz-Hernández, J. Mariño, and J. J. Moreno-Navarro. Constructive intensional negation. In Y. Kameyama and P. J. Stuckey, editors, *FLOPS*, volume 2998 of *Lecture Notes in Computer Science*, pages 39–54. Springer, 2004.

[34] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

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

[36] F. Pfenning. Logical frameworks. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers, 2000. In preparation.

[37] F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proc. 16th Int. Conf. on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, 1999. Springer-Verlag LNAI 1632.

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

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

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

[41] G. Puebla, F. Bueno, and M. V. Hermenegildo. Combined static and dynamic assertion-based debugging of constraint logic programs. In *Logic Program Synthesis and Transformation*, pages 273–292, 1999.

[42] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *CAV 1997*, pages 143–154. Springer-Verlag, 1997.

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

[44] P. Schroeder-Heister. Definitional reflection and the completion. In R. Dyckhoff, editor, *Proceedings of the 4th International Workshop on Extensions of Logic Programming*, pages 333–347. Springer-Verlag LNAI 798, 1993.

[45] P. Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 222–232, Montreal, Canada, June 1993.

[46] P. J. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118(1):12–33, 1995.

[47] A. F. Tiu. Model checking for *pi*-calculus using proof search. In M. Abadi and L. de Alfaro, editors, *CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2005.

[48] C. Urban and J. Cheney. Avoiding equivariant unification. In *Proceedings of the 2005 Conference on Typed Lambda Calculus and Applications (TLCA 2005)*, number 3461 in LNCS, pages 74–89, Nara, Japan, 2005. Springer-Verlag.

[49] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1–3):473–497, 2004.

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