Mechanized Metatheory Model-Checking

James Cheney
University of Edinburgh

Abstract

Full, machine-checkable formal verification appears to be the highest attainable standard of evidence for the correctness of a system. However, such tools have high learning curves, require considerable effort even from knowledgeable users, and are unhelpful in the (common) case that there is a shallow syntactic bug in the system or its specification.

In this note, I outline an alternative approach to analyzing a formal system called *stress-testing*, inspired by the model checking approach to verification of finite-state systems. This approach searches exhaustively for counterexamples to proposed properties of a system. Thus, it cannot guarantee that the system is correct, but is capable of identifying flaws in a system. Moreover, it is more immediately usable by non-experts, takes the user's brain out of the inner loop, and produces concrete counterexamples illustrating bugs. This approach complements other traditional techniques for gaining confidence in the correctness of a system, such as unit testing and informal proof sketches.

Stress-testing can already be programmed directly in a logic programming setting using generate-and-test and negation-as-failure. However, writing useful stress-tests as logic programs is painstaking, yet tedious. Therefore, I have developed a lightweight extension to the αProlog system that reads in a program and specification and automatically searches for counterexamples up to a given resource bound. I will present the approach using an example, then explain an implementation, discuss some additional experience with it, and sketch how it might be improved.

1 Introduction

A large amount of modern programming languages research is founded on proving consistency and correctness properties by syntactic methods (e.g., cut elimination, substitution lemmas, strong normalization, type soundness). Convincing syntactic proofs are challenging to perform on paper for several reasons, including the presence of variable binding, substitution, and structural congruences (as in the pi-calculus), the need to perform reasoning by simultaneous induction on independent structural definitions, and the often large number of cases that must be considered. Paper proofs are believed not to be reliable due in part to the fact that they usually sketch only the essential part of the argument, while leaving out verification of the many 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 has been focused on the problem of formally verifying such properties using computational tools, (see, among others, [1]). Formal, machine-checkable proof is widely agreed to provide the highest possible standard of evidence for believing a system is correct. However, all extant verification tools 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 theorem-proving is a brain-bound, not CPU-bound, process. Moreover, verification provides little assistance in the case of an incorrect 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 verification system.

These observations about formal verification are not new. They have been used to justify an alternative approaches to verifying a system called *model-checking* [7]. In a model-checking approach, the user specifies the system and describes properties which it should satisfy; it is the computer's job to search for counterexamples or to report that none exist. Although it was impractical 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 [10]) have made it feasible to verify industrial hardware designs. As a result, model checking has gained widespread acceptance in industry.

The thesis of this paper is that proof is not the only, or 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. Instead, the investigation of alternative techniques with different performance characteristics seems well-motivated. 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 guarantee that either the design is correct, or a concrete counterexample is produced. Moreover, model-checking tools are CPU-bound, so throwing hardware at the problem makes it easier. Model-checkers do not generally require as much expertise as theorem provers, so once the model specification and formula languages have been learned, an engineer can formalize a design, write a desired property, and "push the button". 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 separates concerns and modularizes the problem in a way that seems conducive to progress.

I think that it is worth investigating mechanized approaches to model-checking metatheory as a useful complement to formal verification, manual testing, and paper proofs. For infinite state systems, model checking is necessarily partial in general: we cannot guarantee that the system can be automatically verified in finite time, but we can at least automate the search for counterexamples. Thus, partial 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. However, this approach shares the other advantages of model-checking over formal proof: it is CPU-bound, not brain-bound; provides a high-level specification interface that isolates non-expert users from orthogonal issues of efficient implementation, and can provide explicit counterexamples. Thus, partial model checking seems more useful than verification typically is during the development of a system. It does not (and cannot) provide absolute assurance, but can be used to check for "shallow" bugs in a system, thus easing the task of experimenting with designs and weeding out wrong attempts.

In this paper, I will describe an approach to defining and specifying desired properties of formal systems in αProlog, a logic programming language which provides nominal abstract syntax techniques for programming with names and binding. Our approach could also be used in other settings such as λProlog [11], Twelf [13], Delphin [17], FreshML [18], αCaml [15], or using QuickCheck [6] and FreshLib [3] in Haskell. However, nominal logic programming seems particularly convenient because it provides a uniform and simple language to both define relations and state their desired properties. In the rest of this abstract, I will illustrate the approach via a tutorial example, discuss some other preliminary experiments I have performed, and discuss next steps and some related approaches.

2 Tutorial Example

I will take for granted some basic knowledge about logic programming. In αProlog, there are a few extra built-in functions and relations that have special semantics. First of all, αProlog includes "name types", which are each 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, there is a built-in binary relation `a # u` called *freshness*, which 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`. Further information about αProlog can be found in [2].

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 (though arguably, not as convenient as in Twelf or λProlog because capture-avoiding substitution is not built-in). As an initial example, we consider the lambda calculus with units, pairs, and function types. Since there is no built-in substitution operation in αProlog, we have to define it explicitly. 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.

2.1 Terms and substitution

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

var  : id -> tm.
app  : (tm,tm) -> tm.
lam  : id\tm -> tm.
b : 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(b,Y,N)            = b.
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

After the definition of the `sub` function, I have added some annotations that state desired properties of substitution. First, the `sub_fun` property states that the result of substitution is uniquely defined. Since *sub* is really a relation under the hood (and since αProlog currently doesn't try to do mode or determinism analysis), this is something we ought to check. Second, we check 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 is a standard substitution lemma involving changing the order of substitutions.

```
base : 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 # (T,G), wf_ctx(G).

pred tc(ctx,tm, ty).
tc(Gamma,b,base).
tc([(V,T)|Gamma],var V, T).
tc(Gamma,app(E1,E2),T)       :- tc(Gamma,E1,T ==> T0),
                                tc(Gamma,E2,T0).
tc(Gamma,lam(x\E),T1 ==> T2) :- tc ((x,T1)::Gamma, E, T2).
tc(Gamma,pair(M,N),T1 ** T2) :- tc(Gamma,M,T1), tc(Gamma,N,T2).
tc(Gamma,fst(M),T1)          :- tc(Gamma,M,T1 ** T2).
tc(Gamma,snd(M),T1)          :- tc(Gamma,M,T1 ** T2).

#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

2.2 Types and typechecking

Next we turn to types and typechecking. We define a standard syntax of types including a unit type called `base`, pairing, and function types. The typechecking judgment is standard. In addition, we check some standard properties of typechecking, including weakening and the substitution lemma. Note that since we are merely specifying, not proving, the substitution lemma, we do not have to state its most general form. However, 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 accomplished using the `wf_ctx` predicate.

```
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).

#check "tc_pres" 5 : tc([],M,T), step(M,M') => tc([],M',T).

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

#check "tc_sound" 10 : tc([],E,T) => not_stuck(E).
```

Figure 3: Reduction, type preservation, non-stuckness, and soundness

2.3 Evaluation and soundness

Now we arrive at the main point of the exercise, namely showing that the overall system is sound. We first define values and introduce a one-step call-by-value reduction relation and show that it is type-preserving for ground terms. A traditional definition of "stuck" or "wrong" state for a term is that the term is neither a value nor able to take a step. Thus, we check that every closed, well-formed term is not stuck.

The alert reader may have noticed several errors in the program above. In fact, every specification I have ascribed to it is violated. I deliberately introduced some of the bugs, but to my amusement discovered others in my original implementation while debugging the specification using the tool I will now describe. Before proceeding, the reader may wish to try to find all of the errors using his or her preferred method.

3 Specification checking

I will now describe the results of my prototype specification checker, along the way giving some intuition as to how it works. We first consider the substitution specifications.

```
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(V10589))
N = var(V10589)
--------
Checking for counterexamples to
sub_id: sub(M,x,var(x)) = M
Checking depth 1 2
Counterexample found:
M = var(V1)
x # V1
--------
Checking for counterexamples to
sub_fresh: x # M => sub(M,x,N) = M
Checking depth 1 2
Counterexample found:
M = var(V1)
N = var(V2)
x # V1
--------
Checking for counterexamples to
sub_comm: x # N' => sub(sub(M,x,N),y,N') = sub(sub(M,y,N'),x,sub(N,y,N'))
Checking depth 1 2
Counterexample found:
M = var(x)
N = var(y)
N' = var(V3)
x # V3
```

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 good program analysis.

The remaining errors appear 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, `sub_fun` passes, 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 counterexample found 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(V191)
G = [(V191,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(x1714\fst(var(x1714))),b)
M' = b
T = base ** T24840
```

For `tc_weak`, of course we have to change

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

to

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

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

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

which is completely wrong and should be

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

The remaining buggy clauses generate further counterexamples. Without further ado, they are

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

which should be

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

After making these corrections, all of the specifications check up to the limits of my patience.

4 Implementation

As currently implemented the stress-tester considers only specification formulas of the form

$$\text{И}\vec{a}.\forall\vec{X}.H_1 \wedge \cdots \wedge H_n \supset A \tag{1}$$

where $H_1, \ldots, H_n$ are atomic formulas (including equality and freshness constraints). Assuming that all of the atomic formulas involved are decidable, these are essentially Π_1 formulas. A counterexample is a closed substitution θ such that $\theta(H_1), \ldots, \theta(H_n)$ all hold (that is, are derivable), but the conclusion $\theta(A)$ fails. In principle, provided all of the atomic formulas terminate when run on ground terms, we could simply enumerate all possible valuations and test them. This method is guaranteed to find a counterexample, if one exists. However, this approach is prohibitively expensive, in part of the presence of infinite base types (such as names).

Another intuitively appealing, but incorrect approach is to avoid ever instantiating variables to concrete valuations, and instead to view the search for counterexamples as ordinary logic programming proof search. The (classical) negation of a formula of form (1) is

$$\text{И}\vec{a}.\exists\vec{X}.H_1 \wedge \cdots \wedge H_n \wedge \neg A \tag{2}$$

and since the above is a legal logic programming query, we can simply execute it. Of course, if any of the predicates involved are recursive, proof search typically diverges in a way that does not ensure that the search space is covered. Moreover, even if we successfully solve the hypotheses, in αProlog (as in Prolog) negation-as-failure is does not provide the behavior we want when the negated formula contains free variables. For example, the goal

```
?- not(X = Y)
```

fails, because $X = Y$ succeeds by unifying X and Y. Technically, this goal is equivalent to $\neg(\exists X, Y.X = Y)$, so this behavior is correct; however, this is not what we want in solving the goal (2). To get the right behavior, we need to ensure that all of the free variables $\vec{X}$ are ground before solving `not(A)`.

We work around this problem by combining the two approaches. We first search symbolically for solutions that make the hypotheses true, then instantiate all free variables in the conclusion, and finally test the conclusion using negation-as-failure. In order to ensure termination, we bound the number of resolution steps that can be used in each derivation. This also ensures that the search space is explored in a fair way.

The stress-testing algorithm proceeds in three stages.

1. Search for "finished" derivations of the hypotheses up to a given bound b on the number of resolutions performed in deriving each hypothesis. That is, search for solutions to the αProlog query $\text{И}\vec{\mathsf{a}}.\exists\vec{X}.H_1 \wedge \cdots \wedge H_n$. This produces an answer substitution θ and freshness constraints describing the original names and variables $\vec{\mathsf{a}}, \vec{X}$, possibly in terms of some new names and variables $\vec{\mathsf{b}}, \vec{Y}$.

2. Consider the conclusion $\theta(A)$, which may mention some of the variables $\vec{Z} \subseteq \vec{X}$. Instantiate any free variables in $\vec{Y}$ reachable from $\vec{Z}$ with ground terms using a substitution σ. At the moment, our implementation only instantiates free variables with term constructors, not names; this avoids an expensive brute-force search over name-valuations, but may damage fairness. Nevertheless, this approach seems to work better than other obvious alternatives such as instantiating name-variables with completely fresh names.

3. Search for a "finished" derivation of A up to some bound b' on the number of resolutions. If such a derivation exists, then the valuation $\sigma \circ \theta$ is not a counterexample. If every derivation attempt fails in fewer than b' resolutions, then A is a counterexample. Otherwise, we can't be sure one way or another, so we report A as a possible counterexample.

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 following show all of the "finished" derivations of $tc(G, E, T)$ using at most three resolutions:

```
tc(G,E,T) --> mem((X,T),G)  | E = var(X)
          -->               | G = [(X,T)|G']

tc(G,E,T) --> mem((X,T),G)   | E  = var(X)
          --> mem((X,T),G1)  | G  = [(Y,T')|G1],
          -->                | G1 = [(X,T)|G2]

tc(G,E,T) --> tc([(x,T1)|G],M,T2)     | x # G, E = lam(x\E1), T = T1 ==> T2,
          --> mem((X,T2),[(x,T1)|G]) | E1 = var(X)
          -->                         | X = x, T1 = T2
```

Here, the constraint formulas on the right-hand side of the `|` indicate conditions under which the formula $tc(G, E, T)$ is derivable.

I implemented a solver in αProlog which 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 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 or cut-elimination). While there are certainly situations where $2 * b$ is not enough, it seems to be an acceptable tradeoff point between avoiding false positives and efficiency.

5 Further experiments and future work

In addition to the relatively straightforward lambda-calculus example considered in Section 2, I have tried the stress-tester out on two substantial examples: an implementation (written by me) of Harper and Pfenning's algorithm for deciding typechecking and equivalence in LF [9], and an implementation of the $F_{\leq}$ language described in the POPLMark Challenge written by Matthew Fairbairn [8].

For the LF example, I used the stress-tester to check several "structural" lemmas asserted but not proved in the paper, specifically lemmas 3.2 (weakening), 3.3 (determinacy), 3.4 (symmetry), and 3.5 (transitivity). In the process, I caught two minor errors in my implementation in which variables were transposed.

For the POPLMark example, I took the code written by Fairbairn and wrote specifications corresponding to the challenge problems. I did not find any mistakes in the existing code. However, it is hard to assess how comprehensive the testing was; searching was only feasible up to depths less than 10, and because of the large number of rules, it is not clear that this was large enough to generate realistic counterexamples. For example, no counterexamples were found after I changed the contravariant subtyping rule to a covariant one. However, an explicit check of the inversion lemma (Lemma 26.4.10 of TAPL [14]) catches this bug.

From these experiences, two interesting observations can be made:

1. It is not enough to just check the overall type soundness property. 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.

2. There is a lot of room for improvement in test case coverage. One thing that seems to contribute heavily to the state explosion is repeatedly visiting states that are "essentially the same", for example in which the order of the variables in the context differs only by a permutation. Adding record/row unification that could be used for contexts could help with this. Another issue is that each resolution step in a derivation has the same cost, even if it is the only possible resolution step in the current context. Instead, it could be more useful to assign costs on context-dependent basis. A third possibility is random search.

I am currently planning on evaluating the prototype implementation of the stress-tester on larger examples, and considering additional search strategies. Also of interest are the problems of obtaining fairness results for efficient search strategies such as the one in our prototype, and increasing the expressiveness of the test formulas.

6 Related work

As stated earlier, our approach draws inspiration from general trends in the model-checking approach to verification of finite-state systems [7]. Particularly relevant is the idea of *symbolic* model checking [10], 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 the idea of automated testing in functional programming languages, for example in the QuickCheck system for Haskell [6]. 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 [3]), QuickCheck could be used to implement metatheory model-checking. Laziness in Haskell means that QuickCheck is to some extent also capable of searching on a derivation-by-derivation basis rather than a substitution-by-substitution basis. QuickCheck tests run as compiled Haskell code, whereas our approach is presently interpreted, so QuickCheck might be faster. On the other hand, writing inference rule specifications seems much easier in αProlog. One interesting thing to try could be to translate αProlog programs and specifications to FreshLib programs and QuickCheck tests.

Many of the properties considered in the example (for example, that substitution is a function) correspond to standard concepts in logic programming such as mode checking and coverage checking, which play an important role in the Twelf system for metatheoretic reasoning about LF signatures [13]. Analyses for checking modes, coverage, termination, and other program properties can be used to verify program properties, not just test them. This approach is also possible (and likely to be helpful) in αProlog, but such analyses have not yet been adapted to this setting. There also seems to be no theoretical obstacle to implementing counterexample search in Twelf. Indeed, the presence of built-in substitution in Twelf seems like a significant advantage.

In addition, there is a body of related work on *declarative debugging* in logic programming languages. Generally, in declarative debugging [12], 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 part of the program. More recently, Puebla et al. [16] have considered assertion languages similar to our specifications which state desired properties of the program. Declarative debugging could be used to help with the process of finding the reason why a counterexample is produced.

7 Conclusions

I have proposed an automatic stress-testing approach to analyzing the properties of logic programs, specifically, nominal logic programs that implement various type systems and operational semantics. Stress-testing is inspired by the success of fully-automatic verification techniques such as model-checking. This approach differs from the traditional "theorem proving" paradigm in that it has a lower learning curve, provides concrete counterexamples, and factors the problem in a way that frees users and researchers to focus on their separate areas of expertise. On the negative side, stress-testing, like testing in general, can never certify the correctness of infinite-state systems, only the absence of bugs in a finite number of cases. Nevertheless, I believe that making it easier to catch such "shallow" bugs earlier in the development of a formal system complements the other practical techniques presently used to evaluate such systems, and can be a useful tool for determining whether a system is mature enough to justify the effort of full verification. It may also be of pedagogical use (e.g., for writing or grading assignments involving type systems and operational semantics).

References

[1] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark Challenge. In *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, 2005. To appear.

[2] J. Cheney and C. Urban. Alpha-Prolog: A logic programming language with names, binding and alpha-equivalence. In *Proceedings of the 20th International Conference on Logic Programming (ICLP 2004)*, number 3132 in LNCS, pages 269–283, 2004.

[3] James Cheney. Scrap your nameplate (functional pearl). In Benjamin Pierce, editor, *Proceedings of the 10th International Conference on Functional Programming (ICFP 2005)*, Tallinn, Estonia, 2005. To appear.

[4] James Cheney. The semantics of nominal logic programs. In *Proceedings of the 2006 International Conference on Logic Programming*, 2006. To appear.

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

[6] Koen Claessen and John 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.

[7] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 2000.

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

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

[10] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[11] G. Nadathur and D. Miller. Higher-order logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter 8, pages 499–590. Oxford University Press, 1998.

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

[13] Frank Pfenning and Carsten 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.

[14] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[15] François Pottier. An overview of Cαml. In *Proceedings of the 2005 ACM SIGPLAN Workshop on ML (ML 05)*, Tallinn, Estonia, September 2005.

[16] German Puebla, Francisco Bueno, and Manuel V. Hermenegildo. Combined static and dynamic assertion-based debugging of constraint logic programs. In *Logic Program Synthesis and Transformation*, pages 273–292, 1999.

[17] C. Schürmann, R. Fontana, and Y. Liao. Delphin: Functional programming with deductive systems. Draft available at `http://www.itu.dk/people/carsten/papers/delphin.pdf`, 2006.

[18] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programmming with binders made simple. In *Proc. 8th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 2003)*, pages 263–274, Uppsala, Sweden, 2003. ACM Press.