Property-Based Testing by Elaborating Proof Outlines

DALE MILLER
INRIA Saclay & LIX, École Polytechnique, France
(*e-mail:* `dale.miller@inria.fr`)

ALBERTO MOMIGLIANO
DI, Università degli Studi di Milano, Italy
(*e-mail:* `momigliano@di.unimi.it`)

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

Abstract

Property-based testing (PBT) is a technique for validating code against an executable specification by automatically generating test-data. We present a proof-theoretical reconstruction of this style of testing for relational specifications and employ the Foundational Proof Certificate framework to describe test generators. We do this by encoding certain kinds of "proof outlines" as proof certificates that can be used to describe various common generation strategies in the PBT literature, ranging from random to exhaustive, including their combination. We also address the *shrinking* of counterexamples as a first step toward their explanation. Once generation is accomplished, the testing phase is a standard logic programming search. After illustrating our techniques on simple, first-order (algebraic) data structures, we first lift it to data structures containing bindings using λ-tree syntax. The λProlog programming language is can perform both the generation and checking of tests using this approach to syntax. We then further extend PBT to specifications in a fragment of linear logic.

KEYWORDS: property based testing; relational specifications; metatheory of programming languages; λ-tree syntax; linear logic

1 Introduction

Property-based testing (PBT) is a technique for validating code that successfully combines two well-trodden ideas: *automatic* test data generation trying to refute *executable* specifications. Pioneered by *QuickCheck* for functional programming (Claessen and Hughes 2000), PBT tools are now available for most programming languages and are having a growing impact in industry (Hughes 2007). Moreover, this idea has spread to several proof assistants (Blanchette et al. 2011; Paraskevopoulou et al. 2015) to complement (interactive) theorem proving with a preliminary phase of conjecture testing. The synergy of PBT with proof assistants is so accomplished that PBT is now a part of the *Software Foundations*'s curriculum (`https://softwarefoundations.cis.upenn.edu/qc-current`).

In our opinion, this tumultuous rate of growth is characterized by a lack of common (logical) foundation. For one, PBT comes in different flavors as far as data generation is concerned: while random generation is the most common one, other tools employ exhaustive generation (Runciman et al. 2008; Cheney and Momigliano 2017) or a combination thereof (Duregård et al. 2012). At the same time, one could say that PBT is rediscovering logic and, in particular, logic programming: to begin with, QuickCheck's DSL is based on Horn clause logic; *LazySmallCheck* (Runciman et al. 2008) has adopted *narrowing* to permit less redundant testing over partial rather than ground terms; a recent version of PLT-Redex (Felleisen et al. 2009) contains a re-implementation of constraint logic programming in order to better generate well-typed λ-terms (Fetscher et al. 2015). Finally, PBT in Isabelle/HOL features the notion of *smart* test generators (Bulwahn 2012), and this is achieved by turning the functional code into logic programs and inferring through mode analysis their data-flow behavior. We refer to the Related Work (Section 8) for more examples of this phenomenon.

This paper considers the general setting of applying PBT techniques to *logic specifications*. In doing so, we also insist on the need to involve *two levels* of logic.

1. The *specification-level logic* is the logic of entailment between a logic program and a goal. In this paper, logic programs can be Horn clauses, *hereditary Harrop formulas*, or a linear logic extension of the latter. The entailment use at the specification level is classical, intuitionistic, or linear.
2. The *reasoning-level logic* is the logic where statements about the specification level entailment are made. For example, in this logic, one might try to argue that a certain specification-level entailment *does not hold*. This level of logic can also be called *arithmetic* since it involves least fixed points. In particular, our use of arithmetic fits within the $I\Sigma_1$ fragment of Peano arithmetic, which is known of coincide with Heyting arithmetic (Friedman 1978). As a result, we can consider our uses of the reasoning-level logic to be either classical or intuitionistic.

We shall use proof-theoretic techniques to deal with both of these logic levels. In particular, instead of attempting some kind of amalgamation of these two levels, we will encode into the reasoning logic inductively defined predicates that faithfully capture specification-level terms, formulas, and provability. One of the strengths of using proof theory (in particular, the sequent calculus) is that it allows for an elegant treatment of syntactic structures with bindings (such as quantified formulas) at both logic levels. As a result, our approach to PBT lifts from the conventional suite of examples to meta-programming examples without significant complications.

Property-based testing requires a flexible way to specify what tests should be generated. This flexibility arises here from our use of *foundational proof certificates* (FPC) (Chihani et al. 2017). Arising from proof-theoretic considerations, FPCs can describe proofs with varying degrees of detail: in this paper, we use FPCs to describe proofs in the specification logic. For example, an FPC can specify that a proof has a certain height, size, or satisfies a specific outline (Blanco and Miller 2015). It can also specify that all instantiations for quantifiers have a specific properties. By employing standard logic programming techniques (e.g., unification and backtracking search), the very process of *checking* that a (specification-level) sequent has a proof that satisfies an FPC is a process that *generates* such proofs, or more in general, results in an attempt to fill in the missing details. In this way, a proof certificate goes through a proof reconstruction to yield a fully *elaborated* proof that a trusted proof-checking kernel can accept. As we shall see, small localized changes in the specification of relevant FPCs allow us to account for both exhaustive and random generation. We can also use FPCs to perform *shrinking*: this is an indispensable ingredient in the random generation approach, whereby counterexamples are minimized to be more easily understandable by the user.

Throughout this paper, we use λProlog (Miller and Nadathur 2012) to specify and prototype all aspects of our PBT project. One role for λProlog is as an example of computing within the Specification Logic. However, since the kinds of queries that PBT requires of our Reasoning Logic are relatively weak, it is possible to use λProlog to implement a prover for the needed properties at the reasoning level. The typing system of λProlog will help clarify when we are using it at these two different levels: in particular, logic program clauses are used as specifications within a reasoning level prover are given the type `sl` instead of the usual type `o` of λProlog clauses.

If we are only concerned with PBT for logic specifications written using first-order Horn clauses (as is the case for Sections 4 and 5), then the λProlog specifications can be replaced with Prolog specifications without much change. However, this interchangeability between λProlog and Prolog disappears when we turn our attention to applying PBT to meta-programming or, equivalently, *meta-theory model-checking* (Cheney and Momigliano 2017). Although PBT has been used extensively with meta-theory specifications, there are many difficulties (Klein et al. 2012), mainly dealing with the *binding structures* one finds within the syntax of many programming language constructs. In that setting, λProlog's support of λ-tree syntax (not found in Prolog) allows us to be competitive with specialized tools, such as αCheck (Cheney and Momigliano 2017).

This paper is organized as follows. In the next two sections, we describe the two levels of logic—the specification level and the reasoning level—whose importance we wish to underline when applying PBT to logic programs. Section 4 gives a gentle introduction to FPCs and their use for data generation. Section 5 shows how FPCs can be used to specify many flavors of PBT flexibly. In Section 6 we lift our approach to meta-programming with applications to the issue of confluence in the λ-calculus. Section 7 extends our approach to a fragment of linear logic. We conclude with a review of related work (Section 8).

This paper significantly extends our conference paper (Blanco et al. 2019) by clarifying the relationship between specification and reasoning logic, by tackling new examples and by including logic specifications based on linear logic. The code mentioned in this paper can be found at

`https://github.com/proofcert/pbt/tree/journal`

2 The specification logic SL

Originally, logic programming was first based on relational specifications (*i.e.*, formulas in first-order predicate logic) given as Horn clauses. Such clauses can be defined as closed formulas of the form $\forall \bar{x}[G \supset A]$ where $\bar{x}$ is a list of variables (all universally quantified), A is an atomic formula, and G (a *goal* formula) is a formula built using disjunctions, conjunctions, existential quantifiers, true (the unit for conjunction), and atomic formulas. An early extension of the logic programming paradigm, called the *hereditary Harrop formulas*, allowed both universal quantification and certain restricted forms of the *intuitionistic implication* ($\supset$) in goal formulas (Miller et al. 1991). A subsequent extension of that paradigm, called Lolli (Hodas and Miller 1994), allowed universal quantifiers and certain uses of the *linear implication* ($\multimap$) from Girard's linear logic (Girard 1987).

Except for Section 7, we shall consider the following two classes of formulas.

$$\begin{array}{rcl} D & ::= & G \supset A \mid \forall x : \tau.D \\ G & ::= & A \mid tt \mid G_1 \vee G_2 \mid G_1 \wedge G_2 \mid \exists x : \tau.G \mid \forall x : \tau.G \mid A \supset G \end{array}$$

The D-formulas are also called *program clauses* and *definite clauses* while G-formulas are *goal formulas*. We will omit type information when not relevant. Here, A is a schematic variable that ranges over atomic formulas. Given the D-formula $\forall x_1 \ldots \forall x_n[G \supset A]$, we say that G is the *body* and A is the *head* of that program clause. In general, every D-formula is an *hereditary Harrop formula* (Miller et al. 1991), although the latter is a much richer set of formulas.

```
kind nat            type.
type z              nat.
type s              nat -> nat.
type nat            nat -> o.
type nlist          list nat -> o.
type append, rev    list A -> list A -> list A -> o.
type reverse        list A -> list A -> o.

nat z.
nat (s X) :- nat X.

nlist nil.
nlist (N::Ns) :- nat N, nlist Ns.

append nil K K.
append (X::L) K (X::M) :- append L K M.

reverse L K  :- rev L K nil.
rev nil A A.
rev (X::L) K A :- rev L K (X::A).
```

Fig. 1. The λProlog specification of five predicates.

We use λProlog to display logic programs in this paper. For example, Figure 1 contains the Horn clause specifications of five predicates related to natural numbers and lists. The main difference between Prolog and λProlog that appears from this example is the fact that λProlog is explicitly polymorphically typed. Another difference is that λProlog allows goal formulas to contain universal quantification and implications.

Traditionally, entailment between a logic program and a goal has been described using classical logic and a theorem proving technique called SLD-resolution refutations (Apt and van Emden 1982). As is now common when intuitionistic (and linear) logics are used within logic programming, refutations are replaced with proofs using Gentzen's sequent calculus (Gentzen 1935). Let $\mathcal{P}$ be a finite set of D-formulas and let G be a goal formula. We are interested in finding proofs of the *sequent* $\mathcal{P} \longrightarrow G$. As it has been shown in (Miller et al. 1991), a simple, two-phase proof search strategy based on *goal reduction* and *backchaining* forms a complete proof system when that logic is intuitionistic logic.

(For a survey of how Gentzen's proof theory has been applied to logic programming, see (Miller 2022).) In the next section, we will write an interpreter for such sequents: the code for that interpreter is taken directly from that two-phase proof system.

3 The reasoning logic RL

The specification logic SL we presented in the previous section is not capable of proving the negation of any atomic formula. This observation is an immediate consequence of the *monotoncity* of SL: that is, if $\mathcal{P} \subseteq \mathcal{P}'$ and A follows from $\mathcal{P}$ then A follows from $\mathcal{P}'$. If $\neg A$ is provable from $\mathcal{P}$ then both A and $\neg A$ are provable from $\mathcal{P} \cup \{A\}$. Thus, we must conclude that $\mathcal{P} \cup \{A\}$ is inconsistent, but that is not possible since the set $\mathcal{P} \cup \{A\}$ is satisfiable (interpret all the predicates to be the universally true property for their corresponding arity). For example, neither ($reverse$ (z :: nil) nil) nor its negation are provable from the clauses in Figure 1.

Clearly, any PBT setting must permit proving the negation of some formulas, since, for example, a counterexample to the claim $\forall x.[P(x) \supset Q(x)]$ is a term t such that $P(t)$ is provable and $\neg Q(t)$ is provable. At least two different approaches have been used to move to a stronger logic in which such negations are provable. The Clark completion of Horn clause programs can be used for this purpose (Clark 1978). An advantage of that approach is that it requires only using first-order classical logic (with an appropriate specification of equality) (Apt and Doets 1994; Lloyd and Topor 1984). A disadvantage is that it only seems to work for Horn clauses: this approach does not seem appropriate when working with intuitionistic and linear logics.

In this paper, we follow an approach used in both the Abella proof assistant (Baelde et al. 2014; Gacek et al. 2012) and the Hybrid (Felty and Momigliano 2012) library for Coq and Isabelle. In those systems, a *second logic*, called the *reasoning logic* (RL, for short), is used to give an *inductive definition* for the provability for a specification logic (in those cases, the specification logic is a fragment of higher-order intuitionistic logic). In particular, consider the λProlog specification in Figure 2. Here, terms of type `sl` denote formulas in SL. The predicate `<>==` is used to encode the SL-level program clauses: for example, the specification in Figure 3 encode the Horn clause programs in Figure 1. Note that we are able to simplify our specification of the `<>==` predicate; the universal quantification at the RL level can be used to encode the (implicit) quantifiers in the SL level.

Figure 4 contains the single clause specification of `interp` that corresponds to its Clark's completion. (In λProlog, the existential quantifier $\exists X$ is written as `sigma X`.) This single clause can be turned directly into the least fixed point expression displayed in Figure 4. The proof theory for RL specifications using fixed points and equality has been developing since the 1990s. Early partial steps were taken by Girard (Girard 1992a) and Schroeder-Heister (Schroeder-Heister 1993). Their approach was strengthen to include least and greatest fixed points for intuitionistic logic (McDowell and Miller 2000; Momigliano and Tiu 2012). Applications of this fixed point logic were made to both model checking (Heath and Miller 2019; McDowell et al. 2003) and to interactive theorem proving (Baelde et al. 2014).

The full RL has a proof search problem that is truly difficult to automate since proofs involving least and greatest fixed points require the proof search mechanism to invent

```
kind sl        type.            % Specification logic formulas
type tt        sl.              % True
type and, or   sl -> sl -> sl. % Conjunction and disjunction
type some      (A -> sl) -> sl. % Existential quantifier
type eq        A  -> A  -> sl. % Equality

infixr and     50.
infixr or      40.
infix  eq      60.

type   <>==    sl -> sl -> o.  % Storing sl clauses
infix  <>==    30.
type interp    sl -> o.         % Interpreter of sl goals

interp tt.
interp (T eq T).
interp (G1 and G2) :- interp G1, interp G2.
interp (G1 or G2)  :- interp G1 ; interp G2.
interp (some G)    :- interp (G T).
interp A           :- (A <>== G), interp G.
```

Fig. 2. The basic interpreter for Horn clause specifications

```
type nat                                  nat -> sl.
type nlist                           list nat -> sl.
type append, rev      list A -> list A -> list A -> sl.
type reverse                   list A -> list A -> sl.

(nat z)      <>== tt.
(nat (s X)) <>== (nat X).
(nlist nil)      <>== tt.
(nlist (N::Ns)) <>== ((nat N) and (nlist Ns)).
(append nil K K)          <>== tt.
(append (X::L) K (X::M)) <>== (append L K M).
(reverse L K)     <>== (rev L K nil).
(rev nil A A)     <>== tt.
(rev (X::L) K A) <>== (rev L K (X::A)).
```

Fig. 3. The encoding of the Horn clauses in Figure 1 as atomic formulas in RL.

```
interp G :- (G = tt);  (sigma T\ G = (T eq T));
 (sigma H\ sigma K\ G = (H and K),  interp H, interp K);
 (sigma H\ sigma K\ G = (H or  K),  interp H; interp K);
 (sigma H\ sigma T\ G = (some H),   interp (H T)) ;
 (sigma B\ G <>== B), interp B).
```

Fig. 4. An equivalent specification of `interp` as one clause.

$$
\begin{aligned}
\mathcal{I} = \mu\lambda I\lambda g\ [g = \texttt{tt} \vee\ & (\exists t.\ g = (t\ \texttt{eq}\ t)) \\
\vee\ & (\exists h \exists k.\ g = (h\ \texttt{and}\ k) \wedge (I\ h) \wedge (I\ k)) \\
\vee\ & (\exists h \exists k.\ g = (h\ \texttt{or}\ k) \wedge (I\ h) \vee (I\ k)) \\
\vee\ & (\exists h \exists t.\ g = (\texttt{some}\ h) \wedge (I\ (h\ t))) \\
\vee\ & (\exists b.\ (g\ \texttt{<>==} b) \wedge (I\ b))]
\end{aligned}
$$

Fig. 5. The least fixed point expression for `interp`.

invariants (pre- and post-fixed points), a problem that is far outside the usual logic programming search paradigm. Fortunately, for our purposes here, we will be attempting to prove simple theorems in RL that are strictly related to queries about SL formulas. In particular, we consider only the following kinds of theorems in RL. Let A and $\mathcal{P}$ be, respectively, an atomic formula and a finite set of D-formulas in SL. Also, let $\hat{A}$ be the direct encodings of A into a term of type `sl`, and let $\hat{\mathcal{P}}$ be a set of RL atomic formulas using the `<>==` predicate that encodes the Horn clauses in $\mathcal{P}$.

1. $(\mathcal{I}\ \hat{A})$ is RL-provable from $\hat{\mathcal{P}}$ if and only if A is SL-provable from $\mathcal{P}$. In addition, these are also equivalent to the fact that (`interp` $\hat{A}$) is provable from $\hat{\mathcal{P}}$ using the logic program for the interpreter in Figure 2.
2. If λProlog's negation-as-finite-failure procedure (NAF) succeeds for the goal (`interp` $\hat{A}$) with respect to the program $\mathcal{P}$ (using the logic program for the interpreter in Figure 2), then $\neg(\mathcal{I}\ \hat{A})$ is RL-provable from $\hat{\mathcal{P}}$ (Hallnäs and Schroeder-Heister 1991). In this case, there is no SL-proof of A from $\mathcal{P}$.

Note that the second statement above is not an equivalence: that is, there may be proofs of $\neg(\mathcal{I}\ \hat{A})$ in RL, which will not be captured by negation-as-finite-failure. For example, if p is an atomic SL formula and $\mathcal{P}$ is the set containing just $p \supset p$, then the usual notion of negation-as-finite-failure will not succeed with the goal $\hat{p}$ and logic program containing just $\hat{p}$ `<>==` $\hat{p}$, while there would be a proof (using induction) that $\neg(\mathcal{I}\ \hat{p})$.

Thus, we can use λProlog as follows. If λProlog proves (`interp` $\hat{A}$) then we know that A is provable from $\mathcal{P}$ in SL. Also, if λProlog's negation-as-finite-failure proves that (`interp` $\hat{A}$) does not hold, then we know that A is not provable from $\mathcal{P}$ in SL. In conclusion, although λProlog has limited abilities to prove formulas in RL, it can still be used in the context of PBT where we require limited forms of inference in RL.

4 Controlling the generation of tests

4.1 Generate-and-test as a proof-search strategy

Imagine that we wish to write a relational specification for reversing lists. There are, of course, many ways to write such a specification, say $\mathcal{P}$, but in every case it should be the case that if $\mathcal{P} \vdash (\mathit{reverse}\ L\ R)$ then $\mathcal{P} \vdash (\mathit{reverse}\ R\ L)$: that is, *reverse* is symmetric.

In the RL setting that we have described in the last section, this property can be written as the formula

$$\forall L \forall K.(\texttt{interp}\ (\mathit{reverse}\ L\ K)) \supset (\texttt{interp}\ (\mathit{reverse}\ K\ L))$$

where we forgo the $(\hat{\_})$ notation. If a formula like this is provable, it is likely that the only such proofs would involve finding an appropriate induction invariant (and possibly additional lemmas). In the property-based testing setting, we are willing to look, instead, for counterexamples to a proposed property. In other words, we are willing to consider searching for proofs of the negation of this formula, namely

$$\exists L \exists K.(\texttt{interp}\ (\mathit{reverse}\ L\ K)) \wedge \neg(\texttt{interp}\ (\mathit{reverse}\ K\ L)).$$

This formula might be easier to prove since it involves only standard logic programming search mechanisms. Of course, proving this second formula would mean that the first formula is not provable.

More generally, we might wish to prove a number of formulas of the form

$$\forall x\colon \tau\ [(\texttt{interp}\ (P(x))) \supset (\texttt{interp}\ (Q(x)))]$$

where both P and Q are SL-level relations (predicates) of a single argument (it is an easy matter to deal with more than one argument). Often, it can be important in this setting to move the type judgment $x\colon \tau$ into the logic by turning the type into a predicate: $\forall x[(\texttt{interp}\ (\tau(x) \wedge P(x))) \supset (\texttt{interp}\ (Q(x)))]$. As we mentioned above, it can be valuable to first attempt to find counterexamples to such formulas prior to pursuing a proof. That is, we might try to prove formulas of the form

$$\exists x[(\texttt{interp}\ (\tau(x) \wedge P(x))) \wedge \neg(\texttt{interp}\ (Q(x)))] \tag{*}$$

instead. If a term t of type τ can be discovered such that $P(t)$ holds while $Q(t)$ does not, then one can return to the specifications in P and Q and revise them using the concrete evidence in t as a witness of how the specifications are wrong. The process of writing and revising relational specifications should be smoother if such counterexamples are discovered quickly and automatically.

The search for a proof of ($*$) can be seen as essentially the *generate-and-test* paradigm that is behind much of property-based testing. In particular, a proof of ($*$) contain the information needed to prove

$$\exists x[(\texttt{interp}\ (\tau(x) \wedge P(x)))] \tag{**}$$

Conversely, not every proof of ($**$) yields, in fact, a proof of ($*$). However, if we are willing to generate proofs of ($**$), each such proof yields a term t such that $[\tau(t) \wedge P(t)]$ is provable. In order to achieve a proof of ($*$), we only need to prove $\neg Q(t)$. In the proof systems used for RL, such a proof involves only negation-as-finite-failure: that is, all possible paths for proving $Q(t)$ must be attempted and all of these must end in failures.

Of course, there can be many possible terms t that are generated in the first steps of this process. We next show how the notion of a *proof certificate* can be used to flexibly constraint the space of terms that can generated for consideration against the testing phase.

4.2 Proof certificate checking with expert predicates

Recall that we assume that the SL is limited so that D-formulas are Horn clauses. We shall consider the full range of D and G-formulas in Section 6 when we consider PBT for metaprogramming.

Figure 6 contains a simple proof system for Horn clause provability that is augmented with *proof certificate* (using the schematic variable Ξ) and additional premises involving *expert predicates*. The intended meaning of these augmentations are the following: proof certificates contain some description of a proof. That outline might be detailed or it might just provide some hints or constraints on the proofs it describes. The expert predicates provide additional premises that are used to "extract" information from proof certificates (in the case of $\vee$ and $\exists$) and to provide continuation certificates where needed.

Figure 7 contains the λProlog implementation of the inference rules in Figure 6: here the infix turnstile $\vdash$ symbol is replaced by the `check` predicate and the predicates `ttE`, `andE`, `orE`, `someE`, `eqE`, and `backchainE` name the predicates $tt_e(\cdot)$, $\wedge_e(\cdot,\cdot,\cdot)$,

$$\frac{tt_e(\Xi)}{\Xi \vdash tt} \qquad \frac{\Xi_1 \vdash G_1 \qquad \Xi_2 \vdash G_2 \qquad \wedge_e(\Xi, \Xi_1, \Xi_2)}{\Xi \vdash G_1 \wedge G_2} \qquad \frac{\Xi' \vdash G_i \qquad \vee_e(\Xi, \Xi', i)}{\Xi \vdash G_1 \vee G_2}$$

$$\frac{\Xi' \vdash G[t/x] \qquad \exists_e(\Xi, \Xi', t)}{\Xi \vdash \exists x : \tau.G}\ (1) \qquad \frac{=_e(\Xi)}{\Xi \vdash t = t} \qquad \frac{\Xi' \vdash G \quad backchain_e(A, \Xi, \Xi')}{\Xi \vdash A}\ (2)$$

The two provisos (1) and (2) are:

1. The term t is of type τ.
2. There is a program clause $\forall \bar{x}(G' \supset A') \in \mathcal{P}$ and a substitution for the variables $\bar{x}$ such that A is $A'\theta$ and G is $G'\theta$.

Fig. 6. A proof system augmented with proof certificates and "expert" predicates.

```
% Certificates
kind cert           type.
kind choice         type.
type left, right    choice.

% The types for the expert predicates
type ttE, eqE                     cert -> o.
type backchainE          sl -> cert -> cert -> o.
type someE          cert -> cert -> A -> o.
type andE        cert -> cert -> cert -> o.
type orE       cert -> cert -> choice -> o.

% Certificate checker
type   check    cert -> sl -> o.

check Cert tt          :- ttE Cert.
check Cert (T eq T)    :- eqE Cert.
check Cert (G1 and G2) :- andE Cert Cert1 Cert2,
   check Cert1 G1, check Cert2 G2.
check Cert (G1 or G2) :- orE Cert Cert' LR,
   ((LR = left,  check Cert' G1);
    (LR = right, check Cert' G2)).
check Cert (some G) :- someE Cert Cert1 T, check Cert1 (G T).
check Cert A :-
          backchainE  A Cert Cert', (A <>== G), check Cert' G.
```

Fig. 7. A simple proof checking kernel.

$\vee_e(\cdot,\cdot,\cdot)$, $\exists_e(\cdot,\cdot,\cdot)$, $=_e(\cdot)$, and $backchain_e(\cdot,\cdot,\cdot)$ used as premises in the inference rules in Figure 6. The intended meaning of the predicate `check Cert G` is that there exists a proof of `G` from the Horn clauses in $\mathcal{P}$ that fits the outline prescribed by the proof certificate `Cert`. Note that it is easy to show that no matter how the expert predicates are defined, if the goal `check Cert G` is provable in λProlog then the goal `interp G` is provable in λProlog and, therefore, `G` is a consequence of the program clauses stored in $\mathcal{P}$.

A proof certificate will be any term of type `cert` (see Figure 7) and an FPC is a logic program that specifies the meaning of the (six) expert predicates. For example, Figure 8 declares two constructors for certificates and displays two FPCs, i.e., the clauses describ-

```
type   height   int -> cert.
type   sze      int -> int -> cert.

ttE     (height _).
eqE     (height _).
orE     (height H) (height H) _.
someE   (height H) (height H) _.
andE    (height H) (height H) (height H).
backchainE _ (height H) (height H') :- H  > 0, H' is H  - 1.

ttE     (sze In In).
eqE     (sze In In).
orE     (sze In Out) (sze In Out) _.
someE   (sze In Out) (sze In Out) _.
andE    (sze In Out) (sze In Mid) (sze Mid Out).
backchainE _  (sze In Out) (sze In' Out) :-
         In > 0, In' is In - 1.
```

Fig. 8. Two FPCs that describe proofs that are limited in either height or in size.

```
kind max      type.
type max      max    -> cert.
type binary   max    -> max -> max.
type choose   choice -> max -> max.
type term     A      -> max -> max.
type empty    max.

ttE     (max empty).
eqE     (max empty).
orE     (max (choose C M)) (max M) C.
someE   (max (term   T M)) (max M) T.
andE    (max (binary M N)) (max M) (max N).
backchainE _ (max M) (max M).
```

Fig. 9. The `max` FPC

ing how the expert clauses treat certificates based on these two constructors. Specifically, the first FPC defines the experts for treating certificates that are constructed using the `height` constructor. As is easy to verify, the query `(check (height 5) G)` (for the encoding `G` of a goal formula) is provable in λProlog using the clauses in Figures 7 and 8 if and only if the height of that proof is 5 or less. Similarly, the second FPC uses the constructor `sze`[1] (with two integers) and can be used to bound the total number of instances of backchaining steps in a proof. In particular, the query `(sigma H\ check (sze 5 H) G)` is provable if and only if the total number is 5 or less.

Figure 9 contains the FPC based on the constructor `max` that is used to record explicitly all information within a proof, not unlike a proof-term in type theory: in particular, all disjunctive choices and all substitution instances for existential quantifiers are collected into a binary tree structure of type `max`. In this sense, proof certificates built with

[1] This spelling is used since “size” is a reserved word in the *Teyjus* compiler for λProlog (Qi et al. 2015).

this constructor are *maximally* explicit. Such proof certificates are used, for example, in (Blanco et al. 2017); it is important to note that proof checking with such maximally explicit certificates can be done with much simpler proof-checkers than those used in logic programming since backtracking search and unification are not needed.

A characteristic of the FPCs that we have presented here is that none contain the substitution terms used in backchaining. At the same time, they may choose to explicitly store substitution information for the existential quantifiers in goals (see the `max` FPC above). While there is no problem in reorganizing our setting so that the substitution instances used in the backchaining inference are stored explicitly (see, for example, (Chihani et al. 2017)), we find our particular design convenient. Furthermore, if we wish to record all the substitution instances used in a proof, we can write logic programs in the Clark completion style (Clark 1978). In that case, all substitutions used in specifying backchaining are also captured by explicit existential quantifiers in the body of those clauses.

If we view a particular FPC as a means of *restricting* proofs, it is possible to build an FPC that restricts proofs satisfying two FPCs simultaneously. In particular, Figure 10 defines an FPC based on the (infix) constructor `<c>`, which *pairs* two terms of type `cert`. The pairing experts for the certificate `Cert1 <c> Cert2` simply request that the corresponding experts also succeed for both `Cert1` and `Cert2` and, in the case of the `orE` and `someE`, also return the same choice and substitution term, respectively. Thus, the query

```
?- check ((height 4) <c> (sze 10 H)) G
```

will succeed if there is a proof of `G` that has a height less than or equal to 4 while also being of size less than or equal to 10. A related use of the pairing of two proof certificates is to *distill* or *elaborate* proof certificates. For example, the proof certificate `(sze 5 0)` is rather implicit since it will match any proof that used backchain exactly 5 times. However, the query

```
?- check ((sze 5 0) <c> (max Max)) G.
```

will store into the λProlog variable `Max` more complete details of any proof that satisfies the `(sze 5 0)` constraint. These maximal certificates are an appropriate starting point for documenting both the counterexample and why it serves as such. In particular, this forms the infrastructure of an *explanation* tool for attributing "blame" for the origin of a counterexample.

Various additional examples and experiments using the pairing of FPCs can be found in (Blanco et al. 2017). Using similar techniques, it is possible to define FPCs that target specific types for special treatment: for example, when generating integers, only (user-defined) small integers can be inserted into counterexamples.

5 PBT as proof elaboration in the reasoning logic

The two-level logic approach resembles the use of meta-interpreters in logic programming. Particularly strong versions of such interpreters have been formalized in (McDowell and Miller 2002; Gacek et al. 2012) and exploited in (Baelde et al. 2014; Felty and Momigliano 2012). In our generate-and-test approach to PBT, the generation phase is controlled by

```
type   <c>     cert ->  cert ->  cert.
infixr <c>     5.

ttE      (A <c> B) :- ttE A, ttE B.
eqE      (A <c> B) :- eqE A, eqE B.
someE    (A <c> B) (C <c> D) T          :- someE A C T, someE B D T.
orE      (A <c> B) (C <c> D) E          :- orE A C E, orE B D E.
andE     (A <c> B) (C <c> D) (E <c> F)  :- andE A C E, andE B D F.
backchainE At (A <c> B) (C <c> D)  :-
         backchainE At A C, backchainE At B D.
```

Fig. 10. FPC for pairing

using appropriate FPCs, and the testing phase is performed by the standard, vanilla meta-interpreter (such as the one in Figure 2).

To illustrate this division between generation and testing, consider the following two simple examples. Suppose we want to falsify the assertion that the reversal of a list is equal to itself. The generation phase is steered by the predicate `check`, which uses a certificate (its first argument) to produce candidate lists according to a generation strategy. The testing phase performs the list reversal computation using the meta-interpreter `interp`, and then negates the conclusion using negation-as-finite-failure, yielding the clause:

```
prop_rev_id Gen Xs :- check Gen (nlist Xs),
                      interp (rev Xs Ys),
                      not (interp (Xs eq Ys)).
```

If we set `Gen` to be say `height 3`, the logic programming engine will return, among others, the answer `Xs = s z :: z :: nil`. Note that the call to `not` is safe since, by the totality of `rev`, `Ys` will be ground at calling time.

As a second simple example, the testing of the symmetry of *reverse* can be written as:

```
prop_rev_sym Gen Xs :- check Gen (nlist Xs),
                       interp (rev Xs Ys),
                       not (interp (rev Ys Xs)).
```

Unless one's implementation of *reverse* is grossly mistaken, the engine should complete its search (according to the prescriptions of the generator `Gen`) without finding a counterexamples.

We now illustrate how we can capture in our framework various flavors of PBT.

5.1 Exhaustive generation

While PBT is traditionally associated with random generation, several tools rely on exhaustive data generation up to a bound (Sullivan et al. 2004) — in fact, such strategy is now the default in Isabelle/HOL's PBT suite (Blanchette et al. 2011). In particular,

1. (Lazy)SmallCheck (Runciman et al. 2008) views the bound as the nesting depth of constructors of algebraic data types.
2. αCheck (Cheney et al. 2016) employs the derivation height.

Our `sze` and `height` FPCs in Figure 8, respectively, match (1) and (2) and, therefore, can accommodate both.

One minor limitation of our method is that, although `check Gen P` will generate terms up to the specified bound when using either `sze` or `height` for `Gen`, the logic programming engine will enumerate these terms in reverse order, starting from the bound and going downwards, during a PBT query. For example, a query such as `?- prop_rev_id (height 10) Xs` will return larger counterexamples first, starting here with a list of nine 0 and a 1. This means that if we do not have a good estimate of the dimension of our counterexample, our query may take an unnecessary long time or even loop.

A first fix, as we have seen, is certificate pairing. The query

```
?- prop_rev_id ((height 10) <c> (sze 6 _)) Xs
```

will converge quickly, quicker in fact that with the separate bounds, to the usual minimal answer. However, we still ought to have some idea about the dimension of the counterexample beforehand and this is not realistic. Yet, it is easy, thanks to logic programming, to implement a simple-minded form of *iterative deepening*, where we backtrack over an increasing list of bounds:

```
prop_rev_sym_it Start End Xs :-
  mk_list Start End Range, member H Range,
  check (height H) (nlist Xs),
  interp (rev Xs Ys),
  not (interp (Xs eq Ys)).
```

Here, `mk_list Start End Range` holds when `Start,End` are positive integers and `Range` is the of list natural numbers `[Start,...,End]`. In addition, we can choose to express size as a function of height — of course this can be tuned by the user, depending on the data they are working with:

```
prop_rev_sym_it Start End Xs :-
  mk_list Start End Range, member H Range, Sh is H * 3,
  check ((height H) <c> (sze Sh _)) (nlist Xs),
  interp (rev Xs Ys),
  not (interp (Xs eq Ys)).
```

While these test generators are easy to construct, they have the drawback of recomputing candidates at each level. A better approach is to introduce an FPC for a form of iterative deepening for *exact* bounds, where we output only those candidates requiring that precise bound. This has some similarity with the approach in *Feat* (Duregård et al. 2012). We will not pursue this avenue here.

5.2 Random generation

The FPC setup can be extended to support random generation of candidates. The idea is to implement a form of *randomization* of choice points: when a choice appears, we flip a coin to decide on which case to choose. There are two major sources of choice in running a logic program: which disjunct in a disjunction to pick and which clause to on which to backchain. In this subsection, we will assume that there is only one clause for backchaining: this can be done by putting clauses into their Clark-completion format (Clark 1978). Thus, both forms of choice are represented as the selection of a disjunct within a disjunction. For example, we shall write the definitions of the `nat` and `nlist` predicates from Figure 3 as follows.

```
type   random   cert.

ttE random.
eqE random.
orE random random Choice :- next_bit I,
  ((I = 0, Choice = left); (I = 1, Choice = right)).
someE random random _.
andE  random random random.
backchainE _ random random.
```

Fig. 11. FPC for random generation

```
(nat N)    <>== (N eq z) or
                (some N'\ N  eq (s N') and (nat N')) or ff.
(nlist L) <>== (L eq nil) or
                (some N\ some Ns\ L eq (N::Ns) and (nat N)
                                         and (nlist Ns)) or ff.
```

In these two examples, the body of clauses is written as a *list* of disjunctions: that is, the body of such clauses is written as

$$D_1 \texttt{ or } D_2 \texttt{ or } \cdots D_n \texttt{ or ff},$$

where $n \geq 1$ and $D_1, \ldots, D_n$ are formulas that are not disjunctions. (Here, false, written as `ff`, represents the empty list of disjunctions.) This choice of writing the body of clauses will make it easier to specify a probability distribution to the disjunctions $D_1, \ldots, D_n$ (see the FPC defined in Figure 12).

A simple FPC given by the constructor `random` is described in Figure 11. Here, we assume that the predicate `next_bit` can access a stream of random bits.

A more useful random test generator is based on a certificate instructing the kernel to select disjunctions according to certain probability distributions. The user can specify such a distribution in a list of weights assigned to each disjunction. In the examples we consider here, these disjuncts appear only at the top level of the body of the clause defining a given predicate. When the kernel encounters an atomic formula, the backchain expert `backchainE` is responsible for expanding that atomic formula into its definition, which is why the expert is indexed by an atom. At this stage, it is necessary to consider the list of weights assigned to individual predicates.

Consider the FPC specification in Figure 12. This certificate has two constructors. The constant `noweight` indicates that no weights are in force at this part of the certificate. The other certificate is of the form `cases Rnd Ws Acc`, where `Rnd` is a random number (between 0 and 127 inclusively), `Ws` are the remaining weights for the remaining disjunctions, and `Acc` is the accumulation of the weights that have been skipped at this point in the proof-checking process. The value of this certificate is initialized (by the `backchainE` expert) to be `cases Rnd Ws 0` using the a random 7-bit number `Rnd` (which can be computed by calling `next_bit` seven times) and a list of weights `Ws` stored in the `weights` predicate associated to the atomic formula that is being unfolded.

The weights (in Figure 12) used here for `nat`-atoms selects the first disjunction (for zero) one time out of four and select the successor clause in the remaining cases. Thus,

```
type noweight      cert.
type cases         int -> list int -> int -> cert.

ttE     noweight.
eqE     noweight.
andE    noweight noweight noweight.
someE   noweight noweight _.

backchainE Atom noweight (cases Rnd Ws 0) :-
   weights Atom Ws, read_7_bits Rnd.

orE (cases Rnd (W::Ws) Acc) Cert Choice :- Acc' is Acc + W,
  ((Acc'  > Rnd, Choice = left,  Cert = noweight) ;
   (Acc' =< Rnd, Choice = right, Cert = (cases Rnd Ws Acc'))).

weights (nat _)    [32,96].
weights (nlist _)  [32,96].

iterate N :- N > 0.
iterate N :- N > 0, N' is N - 1, iterate N'.
```

Fig. 12. An FPC that selects randomly from a weighted disjunct.

this weighting scheme favors selecting small natural numbers. The weighting scheme for `nlist` similarly favors short lists. For example, the query[2]

```
iterate 5, check noweight (nlist L).
```

would then generate the following stream of five lists of natural numbers (depending, of course, on the random stream of bits provided).

```
L = nil
L = nil
L = s (s (s (s (s (s z))))) :: s (s (s (s (s (s (s z)))))) :: z ::
    s (s z) :: s (s (s z)) :: s z :: s z ::
    s (s (s (s (s (s (s (s (s (s (s (s (s z))))))))))))) ::
    s z :: z :: nil
L = s z :: z :: z :: s z :: s (s (s (s (s (s (s (s z))))))) :: nil
L = s z :: s (s (s (s (s (s z))))) :: nil
```

As an example of using such randomize test case generation, the query

```
iterate 10, check noweight (nlist L),
            interp (reverse L R) ,
            not (interp (reverse R L))) .
```

will test the property that `reverse` is a symmetric relation on 10 randomly selected short lists of small numbers.

As we mention in Section 8, this is but one strategy for random generation and quite possibly not the most efficient one, as the experiments in (Blanco et al. 2019) indicate.

[2] You can only execute this and the next query with an implementation that can access a stream of random bits. This is not shown here.

```
kind item                        type.
type c_nat               nat -> item.
type c_list_nat     list nat -> item.

type subterm        item -> item -> o.
type collect        list item -> list item -> cert.

ttE    (collect In In).
eqE    (collect In In).
orE    (collect In Out)
       (collect In Out) C.
andE   (collect In Out) (collect In Mid) (collect Mid Out).
backchainE _ (collect In Out) (collect In Out).
someE  (collect [(c_nat T) | In] Out)
       (collect In Out) (T : nat).
someE  (collect [(c_list_nat T)|In] Out)
       (collect In Out) (T : list nat).

subterm Item Item.
subterm Item (c_nat (succ M)) :- subterm Item (c_nat M).
subterm Item (c_list_nat (Nat::L)) :- subterm Item (c_nat Nat) ;
                                      subterm Item (c_list_nat L).
```

Fig. 13. An FPC for collecting substitution terms from proof and a predicate to compute subterm.

In fact, programming random generators is an art (Hritcu et al. 2013; Fetscher et al. 2015; Lampropoulos et al. 2018) in every PBT approach. We can, of course, use the pairing of FPCs (Figure 10) to help filter and fully elaborate structures generated using the randomization techniques mentioned above.

5.3 Shrinking

Randomly generated data that raise counter-examples may be too large to be the basis for the often frustrating process of bug-fixing. For a compelling example, look no further than the run of the information-flow abstract machine described in (Hritcu et al. 2013). For our much simpler example, there is certainly a smaller counter-example than the above for our running property, say `z :: s z :: nil`.

Clearly, it is desirable to find automatically such smaller counterexamples. This phase is known as *shrinking* and consists of creating a number of smaller variants of the bug-triggering data. These variants are then tested to determine if they induce a failure. If that is the case, the shrinking process can be repeated until we get to a local minimum. In the QuickCheck tradition, shrinkers, as well as custom generators, are the user's responsibility, in the sense that PBT tools offer little support for their formulation. This is particularly painful when we need to shrink *modulo* some invariant, e.g., well-typed terms or meaningful sequences of machine instructions.

One way to describe shrinking using FPCs is to follow the following outline.

```
type huniv   (item -> o) -> cert.

ttE     (huniv _).
eqE     (huniv _).
orE     (huniv Pred) (huniv Pred) _.
andE    (huniv Pred) (huniv Pred) (huniv Pred).
backchainE _ (huniv Pred) (huniv Pred).
someE (huniv Pred) (huniv Pred) (T:nat) :-
  Pred (c_nat T).
someE (huniv Pred) (huniv Pred) (T:list nat) :-
  Pred (c_list_nat T).
```

Fig. 14. An FPC for restricting existential choices.

Step 1: Collect all substitution terms in an existing proof. Given a successful proof that a counterexample exists, use the `collect` FPC in Figure 13 to extract the list of terms instantiating the existentials in that proof. Note that this FPC formally collects a list of terms of different types, in our running example `nat` and `list nat`: we accommodate such a collection by providing constructors (e.g., `c_nat` and `c_list_nat`) that map each of these types into the type `item`. Since the third argument of the `someE` expert predicate can be of any type, we use the *ad hoc polymorphism* available in λProlog (Nadathur and Pfenning 1992) to specify different clauses for this expert depending on the type of the term in that position: this allows us to chose different coercion constructors to inject all these terms into the one type `item`.

For the purposes of the next step, it might also be useful to remove from this list any item that is a subterm of another item in that list. (The definition of the subterm relation is given also in Figure 13.)

Step 2: Search again restricting substitution instances. Search again for the proof of a counterexample but this time use the `huniv` FPC (Figure 14) that restricts the existential quantifiers to use subterms of terms collected in the first pass. (The name `huniv` is mnemonic for “Herbrand universe”: that is, its argument is a predicate that describes the set of allowed substitution terms within the certificate.) Replacing the subterm relation with the proper-subterm relation can further constrain the search for proofs. For example, consider the following λProlog query, where `G` is a formula that encodes the generator, `Is` is the list of terms (items) collected from the proof of a counterexample, and `H` is the height determined for that proof.

```
check ((huniv (T\ sigma I\ member I Is, subterm T I)) <c>
       (height H) <c> (max Max)) B.
```

In this case, the variable `Max` will record the details of a proof that satisfies the height bound as well as instantiates the existential quantifiers with terms that were subterms of the original proof. One can also rerun this query with a lower height bound and by replacing the implemented notion of subterm with “proper subterm”. In this way, the search for proofs involving smaller but related instantiations can be used to shrink a counterexample.

6 PBT for metaprogramming

We now describe how we can move PBT into the setting of metaprogramming. For example, we would like to find counterexamples to claims that might be made about the evaluation of a certain functional program or about the type preservation of a programming language. The main difficulty in treating entities such as programs as data structures within a (meta) programming settings is the treatment of bound variables. There have been many approaches to the treatment of term-level bindings within symbolic systems: they include nameless dummies (de Bruijn 1972), higher-order abstract syntax (HOAS) (Pfenning and Elliott 1988), nominal logic (Pitts 2003), parametric HOAS (Chlipala 2008), and locally nameless (Charguéraud 2011). The approach used in λProlog, called the *λ-tree syntax* approach (Miller 2019), is based on the movement of binders from term-level abstractions to formula-level abstractions (*i.e.*, quantifiers) to proof-level abstract variables (called *eigenvariables* in (Gentzen 1935)). This approach to bindings is probably the oldest one, since it appears naturally when organizing Church's Simple Theory of Types (Church 1940) within Gentzen's sequent calculus (Gentzen 1935). As we illustrate in this section, the λ-tree syntax approach to bindings allows us to lift PBT to the meta-programming setting in a simple and modular manner. In what follows, we assume that the reader has a passing understanding of how λ-tree syntax is supported in frameworks such as λProlog or Twelf (Pfenning and Schürmann 1999).

The treatment of bindings in λProlog is intimately related to including into G-formulas universal quantification and implications. While we restricted SL in the previous two sections to Horn clauses, we now allow the full set of D and G-formulas that were defined in Section 2. To that end, we now replace the interpreter code given in Figure 2 with the specification in Figure 15. Here, the goal `interp Ctx G` is intended to capture the fact that `G` follows (in SL) from the union of the atomic formulas in `Ctx` and the logic programs defined by the `<>==` predicate.

Similar to the extensions made to `interp`, we need to extend the notion of FPC and the `check` program: this is given in Figure 16. Three new predicates—`initE`, `impC`, and `allC`—have been added to FPCs. Using the terminology of (Chihani et al. 2017), the last two of these predicates are referred to as *clerks* instead of *experts*. This distinction arises from the fact that no essential information is extracted from a certificate by these predicate, whereas experts often need to make such extractions. In order to use a previously defined FPC in this setting, we simply need to provide the definition of these three definitions for the constructors used in that FPC. For example, the `max` and `sze` FPCs (see Section 4.2) are accommodated by the additional clauses listed in Figure 17.

To showcase the ease with which we handle searching for counterexamples in binding signatures, we go back in history and explore a tiny bit of the classical theory of the lambda-calculus, namely the Church-Rosser theorem and related notions. We recall two basic definitions, for a binary relation R and its Kleene closure R^*:

```
type all      (A -> sl) -> sl.     % Universal quantifier
type =o       sl  -> sl -> sl.     % Intuitionistic implication
infixr =o     30.
type interp    list sl -> sl -> o. % The new type for interp

interp Ctx tt.
interp Ctx (T eq T).
interp Ctx (G1 and G2) :- interp Ctx G1, interp Ctx G2.
interp Ctx (G1 or G2)  :- interp Ctx G1; interp Ctx G2.
interp Ctx (some G)    :- interp Ctx (G T).
interp Ctx (A =o G)    :- interp (A::Ctx) G.
interp Ctx (all G)     :- pi x\ interp Ctx (G x).
interp Ctx A           :- member A Ctx;
                          (A <>== G), interp Ctx G.
```

Fig. 15. A re-implementation of the interpreter code in Figure 2 that treats implications and universal quantifiers in G-formulas.

```
type check   cert -> list sl -> sl -> o. % New type for check
type initE   cert -> o.                  % Expert for initial rule
type impC    cert -> cert -> o.          % Clerk for implication
type allC    cert -> (A -> cert) -> o.   % Clerk for universal

check Cert Ctx tt          :- ttE Cert.
check Cert Ctx (T eq T)    :- eqE Cert.
check Cert Ctx (G1 and G2) :- andE Cert Cert1 Cert2,
                              check Cert1 Ctx G1,
                              check Cert2 Ctx G2.
check Cert Ctx (G1 or G2)  :- orE Cert Cert' LR,
                              ((LR = left,  check Cert' Ctx G1);
                               (LR = right, check Cert' Ctx G2)).
check Cert Ctx (some G)    :- someE Cert Cert1 T,
                              check Cert1 Ctx (G T).
check Cert Gamma (D =o G ) :- impC Cert Cert',
                              check Cert' (D::Gamma) G.
check Cert Gamma (all G)   :- allC Cert Cert',
                              pi x\ check (Cert' x) Gamma (G x).
check Cert Ctx A           :- initE Cert, member A Ctx.
check Cert Ctx A           :- backchainE Cert Cert',
                              (A <>== G), check Cert' Ctx G.
```

Fig. 16. A re-implementation of the FPC checker in Figure 7 that treats implications and universal quantifiers in G-formulas.

```
initE (max empty).
allC  (max C) (x\ max C).
impC  (max C) (max C)
initE (sze In In') :- In > 0, In' is In - 1.
allC  (sze In Out) (x\ sze In Out).
impC  (sze In Out) (sze In Out).
```

Fig. 17. Additional clauses for two FPCs.

$$\frac{}{(\lambda x.\ M)\ N \longrightarrow_\beta [x \mapsto N]M}\ \mathtt{B}-\beta \qquad \frac{M \longrightarrow_\beta M'}{\lambda x.M \longrightarrow_\beta \lambda x.M'}\ \mathtt{B}-\xi$$

$$\frac{M_1 \longrightarrow_\beta M_1'}{M_1\ M_2 \longrightarrow_\beta M_1'\ M_2}\ \mathtt{B-APP1} \qquad \frac{M_2 \longrightarrow_\beta M_2'}{M_1\ M_2 \longrightarrow_\beta M_1\ M_2'}\ \mathtt{B-APP2}$$

$$\frac{x \in \Gamma}{\Gamma \vdash x : \mathtt{exp}} \qquad \frac{\Gamma \vdash M : \mathtt{exp} \quad \Gamma \vdash M' : \mathtt{exp}}{\Gamma \vdash M\ M' : \mathtt{exp}} \qquad \frac{\Gamma, x \vdash M : \mathtt{exp}}{\Gamma \vdash \lambda x.M : \mathtt{exp}}$$

Fig. 18. Specifications of beta reduction and well-formed terms.

```
kind   exp       type.
type   app       exp -> exp -> exp.
type   lam       (exp -> exp) -> exp.

type   beta      exp -> exp -> sl.
type   is_exp           exp -> sl.

beta (app (lam M) N) (M N)     <>== tt.
beta (app N1 N2) (app N11 N2) <>== beta N1 N11.
beta (app N1 N2) (app N1 N22) <>== beta N2 N22.
beta (lam M)     (lam N)      <>== all x\ beta (M x) (N x).

is_exp (app E1 E2) <>== is_exp E1 and is_exp E2.
is_exp (lam E)     <>== all x\ is_exp x =o is_exp (E x).
```

Fig. 19. The λProlog specification of the inference rules in Figure 18.

$$\begin{tikzcd} \cdot \arrow[r, "\mathrm{R}"] \arrow[d, "\mathrm{R}"'] & \cdot \arrow[d, dashed, "\mathrm{R}"] \\ \cdot \arrow[r, dashed, "\mathrm{R}"'] & \cdot \end{tikzcd} \qquad \begin{tikzcd} \cdot \arrow[r, "R^*"] \arrow[d, "R^*"'] & \cdot \arrow[d, dashed, "R^*"] \\ \cdot \arrow[r, dashed, "R^*"'] & \cdot \end{tikzcd}$$

diamond for R confluence for R

Given the syntax of the untyped lambda-calculus:

$$\text{Terms} \quad M \quad ::= \quad x \mid \lambda x.\ M \mid M_1\ M_2$$

in Fig. 18 we display the standard rules for beta reduction, consisting of the beta rule itself augmented by congruences.

Fig. 19 displays the encoding in λ-tree form of the syntax of the untyped lambda calculus. As it is now a staple of λProlog and similar systems, we do not comment it further. Note how in the encoding of the beta rule, substitution is realized via meta-level application and how in the $\mathtt{B}-\xi$ rule we descend into an abstraction via SL-level universal quantification. The clause for generating/checking abstractions features the combination of hypothetical and parametric al judgments.

When proving the *confluence* of a (binary) reduction relation, a key stepping stone is the *diamond property*. In fact, diamond implies confluence. It is a well-known fact, however, that beta reduction does *not* satisfy the diamond property, since redexes can be discarded or duplicated and this is why notions such as *parallel* reduction have been developed (Takahashi 1995).

To find a counterexample to the claim that beta reduction implies the diamond property, we write the following simple code, which abstracts over a binary reduction relation.

```
type joinable    (exp -> exp -> sl) -> exp -> exp -> sl.
type prop_dia    cert -> (exp -> exp -> sl) -> exp -> o.

joinable Step M M     <>== tt.
joinable Step M1 M2   <>== some P\ (Step M1 P) and (Step M2 P).

prop_dia Cert Step M :-
    check Cert nil (is_exp M),
    interp nil (Step M M1),
    interp nil (Step M M2),
    not(interp nil (joinable Step M1 M2)).
```

Note that the NAF call is safe since, when the last goal is called, all variables in it will be bound to closed terms. A minimal counterexample found by exhaustive generation is:

```
app (lam x\ app x x) (app (lam x\ x) (lam x\ x))
```

or, using the identity combinators, the term $(\lambda x.\ x\ x)(I\ I)$, which beta reduces to $(I\ I)(I\ I)$ and $(I\ I)$. Of course, the property would not be falsified had we taken `Step` to be the reflexive-transitive closure of beta reduction, or, for that matter, other relations such as parallel reduction and complete developments—see the code in the repository for their implementation. As expected, such queries do not report any counterexample.

Let us dive further by looking at η-reduction in a typed setting. Again, it is well-know (see, e.g. (Selinger 2008)) that the diamond property fails for $\beta\eta$-reduction for the simply-typed lambda calculus, once we add unit and pairs: the main culprit is the η rule for unit, which licenses any term of type unit to η-reduce to the empty pair. Verifying the existence of such counterexamples requires building-in typing obligations in the reduction semantics, following the style of (Goguen 1995). In fact, it is not enough for the generation phase to yield only well-typed terms, lest we meet false positives.

Since a counterexample manifests itself considering only η and unit, we list in Fig. 20 the η rules restricted to arrow and unit; see Fig. 21 for their implementation.

A first order of business is to ensure that the typing annotations we have inserted in the reduction semantics are consistent with the standard typing rules. In other words, we need to verify that eta reduction preserves typing. The encoding of the property

$$\Gamma \vdash M \longrightarrow_\eta M' : A \Longrightarrow \Gamma \vdash M : A \wedge \Gamma \vdash M' : A$$

follows and does not report any problem.

```
wt_pres M M' A <>== (wt M A) and (wt M' A).

prop_eta_pres Gen M M' A:-
    check Gen nil (is_exp M),
    interp nil (teta M M' A),
    not (interp nil (wt_pres M M' A)).
```

However, had we made a small mistake in the rules in Fig. 20, say forget a typing

$$\frac{\Gamma \vdash M : A \to B \quad x \notin \mathrm{FV(M)}}{\Gamma \vdash \lambda x.(M\ x) \longrightarrow_\eta M : A \to B}\ \eta \qquad \frac{\Gamma \vdash M : \mathbf{1}}{\Gamma \vdash M \longrightarrow_\eta \langle\rangle : \mathbf{1}}\ \eta - \mathbf{1}$$

$$\frac{\Gamma \vdash M \longrightarrow_\eta M' : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\,N \longrightarrow_\eta M'\,N : B}\ \texttt{E-App-L} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N \longrightarrow_\eta N' : A}{\Gamma \vdash M\,N \longrightarrow_\eta M\,N' : B}\ \texttt{E-App-R}$$

$$\frac{\Gamma, x{:}A \vdash M \longrightarrow_\eta M' : B}{\Gamma \vdash \lambda x.M \longrightarrow_\eta \lambda x.M' : A \to B}\ \texttt{E-}\xi$$

Fig. 20. Type-directed η reduction

```
type  wt            exp -> ty -> sl.
type  teta          exp -> exp -> ty ->  sl.

wt unit unitTy         <>== tt.
wt (lam M) (arTy A B) <>== all x\ wt x A =o wt (M x) B.
wt (app M N) B         <>== some A\ wt M (arTy A B) and wt N A.

teta (lam x\ app M x) M (arTy A B)  <>== wt M (arTy A B).
teta M unit unitTy                  <>== (wt M unitTy).
teta (app M N) (app M' N) B <>==
   some A\ (teta M M' (arTy A B)) and (wt N A).
teta (app M N) (app M N') B <>==
  some A\ (teta N N' A) and (wt M (arTy A B)).
teta (lam M) (lam N) (arTy A B) <>==
  all w\ wt w A =o teta (M w) (N w) B.
```

Fig. 21. The λProlog specification of the inference rules in Figure 20.

assumption in a congruence rule:

$$\frac{\Gamma \vdash M \longrightarrow_\eta M' : A \to B}{\Gamma \vdash M\,N \longrightarrow_\eta M'\,N : B}\ \texttt{E-App-L} - \texttt{BUG}$$

then type preservation would have failed with the following counterexample.

```
A = unitTy
N = app (lam (x\ unit)) (lam (x\ x))
M = app (lam (x\ x))  (lam (x\ x))
```

A failed attempt of an inductive proof of this property in a proof assistant would have eventually pointed to the missing assumption, but testing is certainly a faster way to discover this mistake.

We can now refute the diamond property for η. The harness is the obvious extension of the previous definitions, where to foster better coverage we only generate well-typed terms:

```
prop_eta_dia Cert  M A :-
    check Cert nil (wt M A and is_ty A),
    interp nil (teta M M1 A),
    interp nil (teta M M2 A),
    not(interp nil (joinable_teta M1 M2 A)).
```

One counterexample found by exhaustive generation is `lam x\ lam y\ app x y`, which at type `(unitTy -> unitTy) -> unitTy -> unitTy`, reduces to `lam x\ x` by eta at function type and `lam x\ lam y\ unit` by eta at unit.

7 Linear logic as the specification logic

One of linear logic's early promises was that it could help in specifying computational systems with side-effects, exceptions, and concurrency (Girard 1987; Girard 1992b). In support of that promise, an early application of linear logic was to enhance big-step operational semantic specifications (Kahn 1987) for programming languages that incorporated such features: see, for example, (Andreoli and Pareschi 1990; Hodas and Miller 1994; Chirimar 1995; Miller 1996; Pfenning 2000). In this section, we adapt the work in (Mantovani and Momigliano 2021) to show how PBT can be applied in the setting where the specification logic is a fragment of linear logic.

7.1 SL as a subset of linear logic

We extend the definition of SL given in Section 2 to involve the following grammar for D and G-formulas.

$$
\begin{array}{rcl}
D & ::= & G \multimap A \mid \forall x:\tau.D \\
G & ::= & A \mid tt \mid G_1 \vee G_2 \mid G_1 \wedge G_2 \mid \exists x:\tau.G \mid \forall x:\tau.G \mid A \supset G \\
 & & \mid A \multimap G \mid\ !G
\end{array}
$$

That is, we allow G-formulas to be formed using the linear implications $\multimap$ (with atomic antecedents) and the exponential $!$. As the reader might be aware, linear logic has two conjunctions ($\&$ and $\otimes$) and two disjunctions ($\invamp$ and $\oplus$). When we view G-formulas in terms of linear logic, we identify $\vee$ as $\oplus$ and $\wedge$ as $\otimes$ (and tt as the unit of $\otimes$). Note that we have also changed the top-level implication for D-formulas into a linear implication: this change is actually a refinement in the sense that the $\multimap$ is a more precise form of the top-level implications of D-formulas.

A proof system for this specification logic is given in Figure 22: here, $\mathcal{P}$ is a set of closed D-formulas. The sequent $\Gamma; \Delta \vdash G$ has a left-hand side that is divided into two zones, each of which are multisets of atomic formulas. The Γ zone is the *unbounded* zone, meaning that the atomic assumptions that it contains can be used *any number* of times in building this proof. The Δ zone is the *bounded* zone, meaning that its atomic assumptions must be used *exactly once* in building this proof. In order to support this strict accounting of formulas in the bounded zone, that zone must be empty in certain rules (the $\cdot$ is used to denote the empty zone), it must be the multiset contains exactly one formula (as in one of the two initial rules displayed in the last row of inference rules), and it must be split when dealing with a conjunctive goal. Also note that (when reading inference rules from conclusion to premises) a goal of the form $A \supset G$, places its assumption A in the unbounded zone and a goal of the form $A \multimap G$, places its assumption A in the bounded zone. Finally, the goal $!G$ can only be proved if the bounded zone is empty: this is the *promotion rule* of linear logic.

The inference rule for $\wedge$ can be expensive to realize in a proof search setting, since, when we read inference rules from conclusion to premises, it requires *splitting* the bounded zone into two multisets before proceeding with the proof. Unfortunately, at the time that this

$$\frac{}{\Gamma; \cdot \vdash tt} \qquad \frac{\Gamma; \Delta_1 \vdash G_1 \quad \Gamma; \Delta_2 \vdash G_2}{\Gamma; \Delta_1, \Delta_2 \vdash G_1 \wedge G_2} \qquad \frac{\Gamma; \Delta \vdash G_i}{\Gamma; \Delta \vdash G_1 \vee G_2}\ i \in \{1,2\}$$

$$\frac{\Gamma; \cdot \vdash G}{\Gamma; \cdot \vdash\ !G} \qquad \frac{\Gamma; \Delta \vdash G[y/x]}{\Gamma; \Delta \vdash \forall x : \tau.G}\ (3) \qquad \frac{\Gamma; \Delta \vdash G[t/x]}{\Gamma; \Delta \vdash \exists x : \tau.G}\ (1)$$

$$\frac{\Gamma, A; \Delta \vdash G}{\Gamma; \Delta \vdash A \supset G} \qquad \frac{\Gamma; \Delta, A \vdash G}{\Gamma; \Delta \vdash A \multimap G}$$

$$\frac{}{\Gamma; A \vdash A} \qquad \frac{}{\Gamma, A; \cdot \vdash A} \qquad \frac{\Gamma; \Delta \vdash G}{\Gamma; \Delta \vdash A}\ (2)$$

The three provisos (1), (2), and (3) are the standard ones. The first two are repeated from Figure 6.

1. The term t is of type τ.
2. There is a program clause $\forall \bar{x}(G' \multimap A') \in \mathcal{P}$ and a substitution for the variables $\bar{x}$ such that A is $A'\theta$ and G is $G'\theta$.
3. The eigenvariable y is not free in the formulas in the concluding sequent.

Fig. 22. A sequent calculus proof system for our linear SL.

$$\frac{}{\Delta_I \setminus \Delta_I \vdash tt} \qquad \frac{\Delta_I \setminus \Delta_M \vdash G_1 \quad \Delta_M \setminus \Delta_O \vdash G_2}{\Delta_I \setminus \Delta_O \vdash G_1 \wedge G_2}$$

$$\frac{\Delta_I, !\,A \setminus \Delta_O, !\,A \vdash G}{\Delta_I \setminus \Delta_O \vdash A \supset G} \qquad \frac{\Delta_I, A \setminus \Delta_O, \square \vdash G}{\Delta_I \setminus \Delta_O \vdash A \multimap G} \qquad \frac{\Delta_I \setminus \Delta_I \vdash G}{\Delta_I \setminus \Delta_I \vdash\ !G}$$

$$\frac{\Delta_I \setminus \Delta_O \vdash G_i}{\Delta_I \setminus \Delta_O \vdash G_1 \vee G_2}\ i \in \{1,2\} \qquad \frac{\Delta_I \setminus \Delta_O \vdash G[y/x]}{\Delta_I \setminus \Delta_O \vdash \forall x : \tau.G}\ (1) \qquad \frac{\Delta_I \setminus \Delta_O \vdash G[t/x]}{\Delta_I \setminus \Delta_O \vdash \exists x : \tau.G}\ (2)$$

$$\frac{}{\Delta_I, A, \Delta'_I \setminus \Delta_I, \square, \Delta'_I \vdash A} \qquad \frac{}{\Delta_I, !\,A, \Delta'_I \setminus \Delta_I, !\,A, \Delta'_I \vdash A} \qquad \frac{\Delta_I \setminus \Delta_O \vdash G}{\Delta_I \setminus \Delta_O \vdash A}\ (3)$$

The three proviso (1), (2), and (3) are the same as in Figure 22.

Fig. 23. The I/O proof system alternative to the proof system in Figure 22.

split is made, it might not be clear which atoms in the bounded zone will be needed to prove the left premise and which are needed to prove the right premise. If the bounded zone contains n distinct items, there are 2^n possible ways to make such a split: thus, considering all splittings is far from desirable. Figure 23 presents a different proof system that is organized around making this split in a *lazy* fashion. Here, the sequents are of the form $\Delta_I \setminus \Delta_O \vdash G$ where Δ_I and Δ_O are *lists* of items that are of the form $\square$, A, and $!\,A$ (where A is an atomic formula).

The idea behind proving the sequent $\Delta_I \setminus \Delta_O \vdash G$ is that all the formulas in Δ_I are *input* to the proof search process for finding a proof of G: in that process, atoms in Δ_I that are not marked by a ! and that are used in building that proof are then deleted (by replacing them with $\square$). That proof search method outputs Δ_O as a result of such a deletion. Thus, the process of proving $\Delta_I \setminus \Delta_O \vdash G_1 \otimes G_2$ involves sending all full context Δ_I into the process of finding a proof of G_1, which returns the output context Δ_M, which is then made into the input for the process of finding a proof of G_2.

```
type bang          sl -> sl.
type -o            sl -> sl -> sl.       % Linear implication
infixr -o          35.
kind optsl         type.                 % Option SL formulas
type del           optsl.
type bnd, ubnd     sl -> optsl.

type llinterp   list optsl -> list optsl -> sl -> o.
type pick       sl -> list optsl -> list optsl -> o.

llinterp In In   tt.
llinterp In In   (T eq T).
llinterp In Out (G1 and G2) :- llinterp In Mid G1,
                               llinterp Mid Out G2.
llinterp In Out (G1 or G2)  :- llinterp In Out G1;
                               llinterp In Out G2.
llinterp In Out (some G) :- llinterp In Out (G T).
llinterp In Out (all G)  :- pi x\ llinterp In Out (G x).
llinterp In In  (bang G) :- llinterp In In G.
llinterp In Out (A =o G) :- llinterp ((ubnd A)::In)
                                     ((ubnd A)::Out) G.
llinterp In Out (A -o G) :- llinterp ((bnd A)::In) (del::Out) G.
llinterp In Out A        :- pick A In Out;
                            (A <>== G), llinterp In Out G.

pick A (bnd A::L)  (del::L).
pick A (ubnd A::L) (ubnd A::L).
pick A (I::L)      (I::K) :- pick A L K.
```

Fig. 24. An interpreter based on the proof system in Figure 23.

The correctness and completeness of this alternative proof system follows directly from results in (Hodas and Miller 1994). A λProlog specification of this proof system appears in Figure 24. In that specification, the input and output contexts are represented by a list of *option-SL-formulas*, which are of three kinds: `del` to denote $\square$, `(ubnd A)` to denote $!A$, and `(bnd A)` to denote simply A.

Note that if we use this interpreter on the version of SL described in Section 2 (*i.e.*, without occurrences of $\multimap$ and $!$ within G formulas), then the first two arguments of `llinterp` are the always the same. If we further restriction ourselves to having only Horn clauses (*i.e.*, they have no occurrences of implication in G formulas), then those first two arguments of `llinterp` are both `nil`. Given these observations, the interpreters in Figures 2 and 15 can be derived directly from the one given in Figure 24.

It is a simple matter to modify the interpreter in Figure 23 in order to get the corresponding `llcheck` predicate that works with proof certificates. In the process of making that change, we need to add two new predicates: the clerk predicate `limpC` to treat the linear implication and the expert predicate `bangE` to treat the $!$ operator. In order to save space, we do not display the clauses for the `llcheck` predicate.

A simple linear logic program is the one that turns a switch on and off.

```
type on, off    sl.
```

```
type toggle     sl -> sl.

toggle G <>== on  and (off -o G).
toggle G <>== off and (on  -o G).
```

When attempting to prove `toggle G` when the bounded zone is $\Delta, \mathtt{on}$ should reduce to attempting to prove `G` where the bounded zone is $\Delta, \mathtt{off}$ (conversely, when the roles of `on` and `off` are switched). This operational reading of these clauses is supported by the following inference rules (following the rules in Figure 22).

$$\dfrac{\dfrac{\dfrac{}{\Gamma; \mathtt{on} \vdash \mathtt{on}} \quad \dfrac{\Gamma; \Delta, \mathtt{off} \vdash \mathtt{G}}{\Gamma; \Delta \vdash \mathtt{off} \multimap \mathtt{G}}}{\Gamma; \Delta, \mathtt{on} \vdash \mathtt{on} \wedge (\mathtt{off} \multimap \mathtt{G})}}{\Gamma; \Delta, \mathtt{on} \vdash \mathtt{toggle\ G}}$$

To illustrate a more interesting linear logic program, consider the following specification of the predicate that relates two lists if and only if they are permutations of each other.

```
type element   A -> sl.
type perm      list A -> list A -> sl.

perm (X::L) K   <>== element X -o perm L K.
perm nil (X::K) <>== element X and perm nil K.
perm nil nil    <>== tt.
```

As the reader can verify, the goal

```
?- llinterp nil nil (perm [1,2,3] K).
```

will produce six solutions for the list `K` and they will all be permutations of `[1,2,3]`. More generally, however, a call to the `perm` predicate will occur in settings where the input and output contexts are not necessarily empty, for example, in the query

```
?- llinterp In Out ((perm [1,2,3] K) and Goal).
```

where `Out` might have some entries in `In` marked as deleted and where `Goal` is some goal. In order to ensure that the permutation specification works properly in such a situation, we should invoke the following goal instead:

```
?- llinterp In Out ((bang (perm [1,2,3] K)) and Goal).
```

This invocation will ensure that all the entries that are in the bounded zone are passed to the process of building a proof of `Goal`.

7.2 The operational semantics of λ-terms with a counter

Figure 25 contains the SL specification of call-by-value and call-by-name big-step operational semantics for a version of the λ-calculus to which a single memory location (a counter) is added[3]. The untyped λ-calculus of Section 6, with its two constructors `app` and `lam`, is extended with the additional four constants.

[3] This specification can easily be generalized to finite registers or to a specification of references in functional programming languages (Chirimar 1995; Miller 1996).

```
type cst     int -> exp. % Coerce integers into expressions
type set     int -> exp. % Command to set the counter
type get     exp.        % Command to get the counter's value
type unit    exp.        % Value returned by set
```

The specification in Figure 25 uses *continuations* to provide for a sequencing of operations. A continuation is an SL goal formula that should be called once the program getting evaluated is completed. For example, the attempt to prove the goal `cbn M V K` when the bounded zone is the multiset containing only the formula *counter* C (for some integer C) reduces to an attempt to prove the goal `K` with the bounded zone consisting of just *counter* D (for some other integer D), provided `V` is the call-by-name value of `M`. This situation can be represented as the following (open) derivation (following the rules in Figure 22).

$$\dfrac{\dfrac{\mathcal{P}; \textit{counter}\ D \vdash \texttt{K}}{\vdots}}{\mathcal{P}; \textit{counter}\ C \vdash \texttt{cbn M V K}}$$

Such a goal reduction can be captured in λProlog using the following higher-order quantified expression

```
?- (pi k\ (pi I\ pi O\ llinterp I O k) =>
           (llinterp [bnd(counter 0)] _ (cbn M V k)).
```

Operationally, λProlog introduces a new eigenvariable (essentially, a local constant) `k` of type `sl` and assumes that this new SL formula can be proved no matter the values of the input and output contexts. Once this assumption is made, the linear logic interpreter is then called with the counter given the initial value of 0 and with the `cbn` evaluator asked to compute the call-by-name value of `M` to be `V` and with the final continuation being `k`. This hypothetical reasoning can be captured by the following predicate.

```
type eval   (exp -> exp ->  sl -> sl) -> exp -> exp -> o.

eval Pred M V :-
 (pi k\ (pi I\ pi O\ llinterp I O k) =>
          (llinterp [bnd(counter 0)] _ (Pred M V k))).
```

It is well known that if the call-by-name and call-by-value strategies terminate when evaluating a *pure* untyped λ-term (those without side-effects such as our counter), then those two strategies yield the same value. One might conjecture that this is also true once counters are added. To probe that conjecture, we write the following logic program.

```
prop_cbnv Cert M V U:-
  llcheck Cert nil nil (is_prog M),
  eval cbn M V, eval cbv M U,
  not(llinterp nil nil (V eq U)).
```

The query `prop_cbnv (height 3) M V U` returns the smallest counterexample to the claim that call-by-name and call-by-value produce the same values in this setting. In particular, this query instantiates `M` with `app (lam (w\ get)) (set (- 1))`: this expression has the call-by-name value of 0 while the it has a call-by-value value of -1, given the generator `is_prog` in Fig. 25.

```
type is_prog, value   exp -> sl.
type is_int, counter  int -> sl.
type cbn, cbv         exp -> exp ->  sl -> sl.

is_int (~ 1)  <>== tt.  %% some integers
is_int   0    <>== tt.
is_int  42    <>== tt.

value unit    <>== tt.
value (cst N) <>== tt.
value (lam M) <>== tt.

is_prog (cst C)      <>== is_int C.
is_prog get          <>== tt.
is_prog (set N)      <>== is_int N.
is_prog (app E1 E2)  <>== is_prog E1 and is_prog E2.
is_prog (lam E)      <>== all x\ is_prog x =o is_prog (E x).

cbn V V             K <>== value V and K.
cbn get (cst C)     K <>== counter C and (counter C -o K).
cbn (set C) unit    K <>== counter D and (counter C -o K).
cbn (app E1 E2) V K <>== some R\ cbn E1 (lam R) (cbn (R E2) V K).

cbv V V             K <>== value V and K.
cbv get (cst C)     K <>== counter C and (counter C -o K).
cbv (set C) unit    K <>== counter D and (counter C -o K).
cbv (app E1 E2) V K <>== some R\ some U\ cbv E1 (lam R)
                         (cbv E2 U (cbv (R U) V K)).
```

Fig. 25. Specifications of call-by-name (`cbn`) and call-by-value (`cbv`) evaluations.

7.3 Queries over linear λ-expressions

A slight variation to `is_exp` (Figure 19) yields the following SL specification that succeeds with a λ-term only when the bindings are used linearly.

```
type is_lexp             exp -> sl.

is_lexp (app E1 E2) <>== is_lexp E1 and is_lexp E2.
is_lexp (lam E)     <>== all x\ is_lexp x -o is_lexp (E x).
```

Using this predicate and others defined in Section 6, it is an easy matter to search for untyped λ-terms with various properties. Consider, for example, the following two predicates definitions.

```
type prop_pres1, prop_pres2
     cert -> (exp -> exp -> sl -> sl) -> exp -> exp -> o.

prop_pres1 Cert Step M N :-
  llcheck Cert nil nil (is_lexp M),
  eval Step M V,
  not(llinterp nil nil (is_lexp V)).

prop_pres2 Cert Step M V :-
```

```
  llcheck Cert nil nil (is_exp M),
  not(llinterp nil nil (is_lexp M)),
  llinterp nil nil (wt M Ty),
  eval Step M V,
  llinterp nil nil (is_lexp V).
```

The `prop_pres1` predicate is designed to search for linear λ-terms that are related by `Step` to a non-linear λ-term. When `Step` is `cbn` or `cbv`, no such term is possible. The `prop_pres2` predicate is designed to search for non-linear λ-terms that have a simple type (via the `wt` predicate) and are related by `Step` to a linear λ-term. When `Step` is `cbn` or `cbv`, the smallest such terms (using the certificate `(height 4)`) are

```
app (lam x\ lam y\ y) (app (lam x\ x) (lam x\ x))
app (lam x\ lam y\ y) (lam x\ x)
app (lam x\ lam y\ y) (lam x\ lam y\ y)
app (lam x\ lam y\ y) (lam x\ lam y\ x)
```

All of these terms evaluates (using either `cbn` or `cbv`) to the term `(lam x\ x)`.

8 Related work and conclusions

8.1 Two-level logic approach

First-order Horn clauses have a long tradition, via the Prolog language, of specifying computation. Such clauses have also been used to specify the operational semantics of other programming languages: see, for example, the early work on *natural semantics* (Kahn 1987) and the Centaur system (Borras et al. 1988). The intuitionistic logic setting of *higher-order hereditary Harrop formulas* (Miller et al. 1991)—a logical framework that significantly extends the SL logic in Section 2—has similarly been used for the past three decades to specify the static and dynamic semantics of programming language: see, for example, (Hannan and Miller 1992; Hannan 1993). Similar specifications could also be written in the dependently typed λ-calculus LF (Harper et al. 1993): see, for example, (Burstall and Honsell 1988; Michaylov and Pfenning 1992).

Linear logic has been effectively used to enrich the natural semantic framework. The Lolli subset of linear logic (Hodas and Miller 1994) as well as the Forum presentation (Miller 1996) of all of linear logic have been used to specify the operational semantics of references and concurrency (Miller 1996) as well as the behavior of the sequential and the concurrent (piped-line) operational semantics of the DLX RISC processor (Chirimar 1995). Another fruitful example is the specification of *session types* (Caires et al. 2016). Ordered logic programming (Polakow and Yi 2000; Pfenning and Simmons 2009) has also been investigated. Similar specifications are also possible in linear-logic inspired variants of LF (Cervesato and Pfenning 2002; Schack-Nielsen and Schürmann 2008; Georges et al. 2017).

The use of a separate *reasoning logic* to reason directly on the kind of logic specifications used in this paper was proposed in (McDowell and Miller 2002). That logic included certain inductive principles that could be applied to the definition of sequent calculus provability. That framework was challenged, however, by the need to treat eigenvariables in sequent calculus proof systems. The ∇-quantifier, introduced in (Miller and Tiu 2005), provided an elegant solution to the treatment of eigenvariables (Gacek et al. 2012). In this paper, our use of the reasoning logic is mainly to determine reachability and non-reachability: in those situations, the ∇-quantifier can be implemented by the universal quantifier in λProlog (see (Miller and Tiu 2005, Proposition 7.10)). If we were to investigate PBT examples that go beyond that simple class of queries, then we would have to abandon λProlog for the richer logic that underlies Abella (Gacek et al. 2011): see, for example, the specifications of bisimulation for the π-calculus in (Tiu and Miller 2010). The two-level approach extends to specifications in dependently typed λ-calculus, first via the $\mathcal{M}_\omega$ reasoning logic over LF in (Schürmann 2000) and then more extensively within the *Beluga* proof environment (Pientka and Dunfield 2010; Pientka and Cave 2015).

However, formal *verification* by reasoning over a linear logic frameworks, is still in its infancy, although the two-level approach is flexible enough to accommodate one reasoning logic over several SL. The most common case study is type soundness of MiniML with references, first checked in (McDowell and Miller 2002) with the Π proof checker and then by Martin in his dissertation (Martin 2010) using Isabelle/HOL's Hybrid library (Felty and Momigliano 2012), in several styles, including linear and ordered specifications. More extensive use of Hybrid, this time on top of Coq, includes the recent verification in a Lolli-like specification logic of type soundness of the *proto-Quipper* quantum functional programming language (Mahmoud and Felty 2019), as well as the meta-theory of sequent calculi (Felty et al. 2021).

8.2 Foundational proof certificates

The notion of *foundational proof certificates* was introduced in (Chihani et al. 2017) as a means for flexibly describing proofs to a logic programming base proof checker (Miller 2017). In that setting, proof certificates can include all or just certain details, whereby missing details can often be recreated during checking using unification and backtracking search. The pairing FPC in Section 4.2 can be used to *elaborate* a proof certificate into one including full details and to *distill* a detailed proof into a certificate containing less information (Blanco et al. 2017).

Using this style of proof elaboration, it is possible to use the ELPI plugin to Coq (Tassi 2018) (which supplies the Coq computing environment with a λProlog implementation) to elaborate proof certificates from external theorem prover into fully detailed certificates that can be checked by the Coq kernel (Manighetti et al. 2020; Manighetti 2022). This same interface between Coq and ELPI allowed Manighetti el al. to illustrate how PBT could be used to search for counterexamples to conjectures proposed to Coq.

Using an FPC as a description of how to traverse a search space bears some resemblance with principled ways to change the depth-first nature of search in logic programming implementations. An example is *Tor* (Schrijvers et al. 2014), which, however, is unable to account for random search. Similarly to *Tor*, FPCs would benefit of *partial evaluation* to remove the meta-interpretive layer.

8.3 Property based testing for meta-theory model-checking

The literature on PBT is very large and always evolving. We refer to (Cheney and Momigliano 2017) for a review with an emphasis to its interaction with proof assistants and specialized domains such as programming language meta-theory.

While Isabelle/HOL were at the forefront of combining theorem proving and refutations (Blanchette et al. 2011; Bulwahn 2012), nowadays most of the action is within Coq: *QuickChick* (Paraskevopoulou et al. 2015) started as a clone of QuickCheck, but soon evolved in a major research project involving automatic generation of custom generators (Lampropoulos et al. 2018), coverage based improvements of random generation (Lampropoulos et al. 2019), as well as going beyond the executable fragment of Coq (Paraskevopoulou et al. 2022).

Within the confine of model checking PL theory, a major player is *PLT-Redex* (Felleisen et al. 2009), an executable DSL for mechanizing semantic models built on top of *DrRacket* with support for random testing à la QuickCheck; its usefulness has been demonstrated in several impressive case studies (Klein et al. 2012). However, Redex has limited support for relational specifications and none whatsoever for binding signature. On the other hand, α*Check* (Cheney and Momigliano 2017; Cheney et al. 2016) is built on top of the nominal logic programming language αProlog and it checks relational specifications similar to those considered here. Arguably, αCheck is less flexible than the FPC-based architecture, since its generation strategy can be seen as a fixed choice of experts. The same comment applies to (Lazy)SmallCheck (Runciman et al. 2008). In both cases, those strategies are wired-in and cannot be varied, let alone combined as we can via pairing. For a small empirical comparison between our approach and αCheck on the PLT-Redex benchmark `http://docs.racket-lang.org/redex/benchmark.html`, please see (Blanco et al. 2019).

In the random case, the logic programming paradigm has some advantages over the labor-intensive QuickCheck approach of writing custom generators. Moreover, the last few years have witnessed an increasing interest in the (random) generation of data satisfying some invariants (Claessen et al. 2015; Fetscher et al. 2015; Lampropoulos et al. 2018); in particular well-typed λ-terms, with an application to testing optimizing compilers (Palka et al. 2011; Midtgaard et al. 2017; Bendkowski et al. 2018). Our approach can use *judgments* (think typing), as generators, avoiding the issue of keeping generators and predicates in sync when checking invariant-preserving properties such as type preservation (Lampropoulos et al. 2017). Further, viewing random generation as expert-driven random back-chaining opens up several possibilities: we have chosen just one simple-minded strategy, namely permuting the predicate definition at each unfolding, but we could easily follow others, such as the ones described in (Fetscher et al. 2015): permuting the definition just once at the start of the generation phase, or even changing the weights at the end of the run so as to steer the derivation towards axioms/facts. Of course, our default uniform distribution corresponds to QuickCheck's `oneOf` combinator, while the weights table to `frequency`. Moreover, pairing random and size-based FPC accounts for some of QuickCheck's configuration options, such as *StartSize* and *EndSize*.

In mainstream programming, property-based testing of stateful programs is accomplished via some form of *state machine* (Hughes 2007; de Barrio et al. 2021). The idea of linear PBT has been proposed in (Mantovani and Momigliano 2021) and applied to mechanized meta-theory model checking, although limited to first-order encodings, e.g. the compilation of a simple imperative language to a stack machine. For efficient generation of typed linear lambda terms, see (Tarau and de Paiva 2020)

8.4 Final remarks

We have described an approach that uses logic programming techniques viewed through the lens of proof-theory to design a uniform and flexible framework that accounts for many features of PBT. Given this proof-theoretic pedigree, it was immediate to extend PBT to the metaprogramming setting, inheriting the handling of λ-tree syntax, which is naturally supported by λProlog and notably absent from most other environments for meta-theory model checking. Similarly it was straightforward to apply PBT to specifications in fragments of linear logic.

While λProlog is used here to discover counterexamples, one does not actually need to trust the logical soundness of λProlog, negation-as-failure making this a complex issue. Any identified counterexample can be exported and utilized within, say, Abella. In fact, it would be a logical future task to incorporate our perspective on PBT into Abella in order to accommodate both proofs and disproofs, as many proof helpers frequently do.

The strategy we have outlined here serves more as a proof-of-concept or logical reconstruction than as a robust implementation. A natural environment to support PBT for every specification in Abella is the Bedwyr model-checker (Baelde et al. 2007), which shares the same meta-logic, but is more efficient from the point of view of proof search.

Finally, we have just hinted at ways for localizing the origin of the bugs reported by PBT. This issue can benefit from research in declarative debugging as well as in *justification* for logic programs (Pemmasani et al. 2004). Coupled with recent results in focusing (Miller and Saurin 2006) this could lead us also to a reappraisal of techniques for *repairing* (inductive) proofs (Ireland and Bundy 1996).

Acknowledgments We would like to thank Rob Blanco for his major contributions to (Blanco et al. 2019), on which the present paper is based.

References

Andreoli, J.-M. and Pareschi, R. 1990. Linear objects: Logical processes with built-in inheritance. In *Proceeding of the Seventh International Conference on Logic Programming, Jerusalem*. MIT Press.

Apt, K. R. and Doets, K. 1994. A new definition of SLDNF-resolution. *The Journal of Logic Programming 18,* 2 (Feb.), 177–190.

Apt, K. R. and van Emden, M. H. 1982. Contributions to the theory of logic programming. *JACM 29,* 3, 841–862.

Baelde, D., Chaudhuri, K., Gacek, A., Miller, D., Nadathur, G., Tiu, A., and Wang, Y. 2014. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning 7,* 2, 1–89.

Baelde, D., Gacek, A., Miller, D., Nadathur, G., and Tiu, A. 2007. The Bedwyr system for model checking over syntactic expressions. In *21th Conf. on Automated Deduction (CADE)*, F. Pfenning, Ed. Number 4603 in LNAI. Springer, New York, 391–397.

Bendkowski, M., Grygiel, K., and Tarau, P. 2018. Random generation of closed simply typed λ-terms: A synergy between logic programming and boltzmann samplers. *Theory Pract. Log. Program. 18,* 1, 97–119.

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

Blanco, R., Chihani, Z., and Miller, D. 2017. Translating between implicit and explicit versions of proof. In *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction*, L. de Moura, Ed. Lecture Notes in Computer Science, vol. 10395. Springer, 255–273.

Blanco, R. and Miller, D. 2015. Proof outlines as proof certificates: a system description. In *Proceedings First International Workshop on Focusing*, I. Cervesato and C. Schürmann, Eds. Electronic Proceedings in Theoretical Computer Science, vol. 197. Open Publishing Association, 7–14.

Blanco, R., Miller, D., and Momigliano, A. 2019. Property-based testing via proof reconstruction. In *Principles and Practice of Programming Languages 2019 (PPDP '19)*, E. Komendantskaya, Ed.

Borras, P., Clément, D., Despeyroux, T., Incerpi, J., Kahn, G., Lang, B., and Pascual, V. 1988. Centaur: the system. In *Third Annual Symposium on Software Development Environments (SDE3)*. ACM, Boston, 14–24.

Bulwahn, L. 2012. The new quickcheck for isabelle - random, exhaustive and symbolic testing under one roof. In *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, C. Hawblitzel and D. Miller, Eds. Lecture Notes in Computer Science, vol. 7679. Springer, 92–108.

Burstall, R. and Honsell, F. 1988. A natural deduction treatment of operational semantics. In *Proceedings of the 8th Conf. on Foundations of Software Technology and Theoretical Computer Science*. Number 338 in Lecture Notes in Computer Science. Springer, 250–269.

Caires, L., Pfenning, F., and Toninho, B. 2016. Linear logic propositions as session types. *Math. Struct. Comput. Sci. 26,* 3, 367–423.

Cervesato, I. and Pfenning, F. 2002. A linear logical framework. *Inf. Comput. 179,* 1, 19–75.

Charguéraud, A. 2011. The locally nameless representation. *Journal of Automated Reasoning 49*, 363–408.

Cheney, J. and Momigliano, A. 2017. αCheck: A mechanized metatheory model checker. *Theory and Practice of Logic Programming 17,* 3, 311–352.

Cheney, J., Momigliano, A., and Pessina, M. 2016. Advances in property-based testing for αProlog. In *Tests and Proofs - 10th International Conference, TAP 2016, Vienna, Austria, July 5-7, 2016, Proceedings*, B. K. Aichernig and C. A. Furia, Eds. Lecture Notes in Computer Science, vol. 9762. Springer, 37–56.

Chihani, Z., Miller, D., and Renaud, F. 2017. A semantic framework for proof evidence. *J. of Automated Reasoning 59,* 3, 287–330.

Chirimar, J. 1995. Proof theoretic approach to specification languages. Ph.D. thesis, University of Pennsylvania.

Chlipala, A. 2008. Parametric higher-order abstract syntax for mechanized semantics. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, J. Hook and P. Thiemann, Eds. ACM, 143–156.

Church, A. 1940. A formulation of the Simple Theory of Types. *J. of Symbolic Logic 5*, 56–68.

Claessen, K., Duregård, J., and Palka, M. H. 2015. Generating constrained random data with uniform distribution. *J. Funct. Program. 25*, e8:1–e8:31.

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

Clark, K. L. 1978. Negation as failure. In *Logic and Data Bases*, J. Gallaire and J. Minker, Eds. Plenum Press, New York, 293–322.

de Barrio, L. E. B., Fredlund, L., Herranz, Á., Earle, C. B., and Mariño, J. 2021. Makina: a new quickcheck state machine library. In *Proceedings of the 20th ACM SIGPLAN International Workshop on Erlang, Erlang@ICFP 2021, Virtual Event, Korea, August 26, 2021*, S. Aronis and A. Bieniusa, Eds. ACM, 41–53.

de Bruijn, N. G. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with an application to the Church-Rosser theorem. *Indagationes Mathematicae 34,* 5, 381–392.

Duregård, J., Jansson, P., and Wang, M. 2012. Feat: functional enumeration of algebraic types. In *Haskell Workshop*, J. Voigtländer, Ed. ACM, 61–72.

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

Felty, A. P. and Momigliano, A. 2012. Hybrid - A definitional two-level approach to reasoning with higher-order abstract syntax. *J. Autom. Reasoning 48,* 1, 43–105.

Felty, A. P., Olarte, C., and Xavier, B. 2021. A focused linear logical framework and its application to metatheory of object logics. *Math. Struct. Comput. Sci. 31,* 3, 312–340.

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

Friedman, H. M. 1978. Classically and intuitionistically provably recursive functions. In *Higher Order Set Theory*, G. H. Müller and D. S. Scott, Eds. Springer Verlag, Berlin, 21–27.

Gacek, A., Miller, D., and Nadathur, G. 2011. Nominal abstraction. *Information and Computation 209,* 1, 48–73.

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

Gentzen, G. 1935. Investigations into logical deduction. In *The Collected Papers of Gerhard Gentzen*, M. E. Szabo, Ed. North-Holland, Amsterdam, 68–131.

Georges, A. L., Murawska, A., Otis, S., and Pientka, B. 2017. LINCX: A linear logical framework with first-class contexts. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, H. Yang, Ed. Lecture Notes in Computer Science, vol. 10201. Springer, 530–555.

Girard, J.-Y. 1987. Linear logic. *Theoretical Computer Science 50,* 1, 1–102.

Girard, J.-Y. 1992a. A fixpoint theorem in linear logic. An email posting to the mailing list linear@cs.stanford.edu.

Girard, J.-Y. 1992b. Logic and Exceptions: A Few Remarks. *Journal of Logic and Computation 2,* 2 (05), 111–118.

Goguen, H. 1995. Typed operational semantics. In *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA '95, Edinburgh, UK, April 10-12, 1995, Proceedings*, M. Dezani-Ciancaglini and G. D. Plotkin, Eds. Lecture Notes in Computer Science, vol. 902. Springer, 186–200.

Hallnäs, L. and Schroeder-Heister, P. 1991. A proof-theoretic approach to logic programming. II. programs as definitions. *J. Log. Comput. 1,* 5, 635–660.

Hannan, J. 1993. Extended natural semantics. *J. of Functional Programming 3,* 2 (Apr.), 123–152.

Hannan, J. and Miller, D. 1992. From operational semantics to abstract machines. *Mathematical Structures in Computer Science 2,* 4, 415–459.

Harper, R., Honsell, F., and Plotkin, G. 1993. A framework for defining logics. *Journal of the ACM 40,* 1, 143–184.

Heath, Q. and Miller, D. 2019. A proof theory for model checking. *J. of Automated Reasoning 63,* 4, 857–885.

Hodas, J. and Miller, D. 1994. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation 110,* 2, 327–365.

Hritcu, C., Hughes, J., Pierce, B. C., Spector-Zabusky, A., Vytiniotis, D., Azevedo de Amorim, A., and Lampropoulos, L. 2013. Testing noninterference, quickly. In *Proceedings* *of the 18th ACM SIGPLAN International Conference on Functional Programming*. ICFP '13. ACM, New York, NY, USA, 455–468.

Hughes, J. 2007. Quickcheck testing for fun and profit. In *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007*, M. Hanus, Ed. Lecture Notes in Computer Science, vol. 4354. Springer, 1–32.

Ireland, A. and Bundy, A. 1996. Productive use of failure in inductive proof. *Journal of Automated Reasoning 16*, 79–111.

Kahn, G. 1987. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Eds. Lecture Notes in Computer Science, vol. 247. Springer, 22–39.

Klein, C., Clements, J., Dimoulas, C., Eastlund, C., Felleisen, M., Flatt, M., McCarthy, J. A., Rafkind, J., Tobin-Hochstadt, S., and Findler, R. B. 2012. Run your research: on the effectiveness of lightweight mechanization. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '12. ACM, New York, NY, USA, 285–296.

Lampropoulos, L., Gallois-Wong, D., Hriţcu, C., Hughes, J., Pierce, B. C., and Xia, L. 2017. Beginner's Luck: A language for random generators. In *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*.

Lampropoulos, L., Hicks, M., and Pierce, B. C. 2019. Coverage guided, property based testing. *Proc. ACM Program. Lang. 3,* OOPSLA, 181:1–181:29.

Lampropoulos, L., Paraskevopoulou, Z., and Pierce, B. C. 2018. Generating good generators for inductive relations. *Proc. ACM Program. Lang. 2,* POPL, 45:1–45:30.

Lloyd, J. and Topor, R. 1984. Making Prolog more expressive. *Journal of Logic Programming 1,* 3 (Oct.), 225–240.

Mahmoud, M. Y. and Felty, A. P. 2019. Formalization of metatheory of the quipper quantum programming language in a linear logic. *J. Autom. Reason. 63,* 4, 967–1002.

Manighetti, M. 2022. Developing proof theory for proof exchange. Ph.D. thesis, Institut Polytechnique de Paris.

Manighetti, M., Miller, D., and Momigliano, A. 2020. Two applications of logic programming to coq. In *26th International Conference on Types for Proofs and Programs, TYPES 2020, March 2-5, 2020, University of Turin, Italy*, U. de'Liguoro, S. Berardi, and T. Altenkirch, Eds. LIPIcs, vol. 188. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:19.

Mantovani, M. and Momigliano, A. 2021. Towards substructural property-based testing. In *Logic-Based Program Synthesis and Transformation - 31st International Symposium, LOPSTR 2021, Tallinn, Estonia, September 7-8, 2021, Proceedings*, E. D. Angelis and W. Vanhoof, Eds. Lecture Notes in Computer Science, vol. 13290. Springer, 92–112.

Martin, A. 2010. Reasoning using higher-order abstract syntax in a higher-order logic proof environment: Improvements to hybrid and a case study. Ph.D. thesis, University of Ottawa. https://ruor.uottawa.ca/handle/10393/19711.

McDowell, R. and Miller, D. 2000. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science 232*, 91–119.

McDowell, R. and Miller, D. 2002. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic 3,* 1, 80–136.

McDowell, R., Miller, D., and Palamidessi, C. 2003. Encoding transition systems in sequent calculus. *Theoretical Computer Science 294,* 3, 411–437.

Michaylov, S. and Pfenning, F. 1992. Natural semantics and some of its meta-theory in Elf. In *Extensions of Logic Programming*, L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, Eds. Lecture Notes in Computer Science. Springer.

Midtgaard, J., Justesen, M. N., Kasting, P., Nielson, F., and Nielson, H. R. 2017. Effect-driven quickchecking of compilers. *PACMPL 1,* ICFP, 15:1–15:23.

Miller, D. 1996. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science 165,* 1, 201–232.

Miller, D. 2017. Proof checking and logic programming. *Formal Aspects of Computing 29,* 3, 383–399.

Miller, D. 2019. Mechanized metatheory revisited. *Journal of Automated Reasoning 63,* 3, 625–665.

Miller, D. 2022. A survey of the proof-theoretic foundations of logic programming. *Theory and Practice of Logic Programming 22,* 6 (Oct.), 859–904. Published online November 2021.

Miller, D. and Nadathur, G. 2012. *Programming with Higher-Order Logic.* Cambridge University Press.

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

Miller, D. and Saurin, A. 2006. A game semantics for proof search: Preliminary results. *Electr. Notes Theor. Comput. Sci. 155*, 543–563.

Miller, D. and Tiu, A. 2005. A proof theory for generic judgments. *ACM Trans. on Computational Logic 6,* 4 (Oct.), 749–783.

Momigliano, A. and Tiu, A. 2012. Induction and co-induction in sequent calculus. *Journal of Applied Logic 10*, 330–367.

Nadathur, G. and Pfenning, F. 1992. The type system of a higher-order logic programming language. In *Types in Logic Programming*, F. Pfenning, Ed. MIT Press, 245–283.

Palka, M. H., Claessen, K., Russo, A., and Hughes, J. 2011. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test, AST 2011, Waikiki, Honolulu, HI, USA, May 23-24, 2011*, A. Bertolino, H. Foster, and J. J. Li, Eds. ACM, 91–97.

Paraskevopoulou, Z., Eline, A., and Lampropoulos, L. 2022. Computing correctly with inductive relations. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, R. Jhala and I. Dillig, Eds. ACM, 966–980.

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

Pemmasani, G., Guo, H.-F., Dong, Y., Ramakrishnan, C. R., and Ramakrishnan, I. V. 2004. Online justification for tabled logic programs. In *Functional and Logic Programming*, Y. Kameyama and P. J. Stuckey, Eds. Springer Berlin Heidelberg, Berlin, Heidelberg, 24–38.

Pfenning, F. 2000. Structural cut elimination I. intuitionistic and classical logic. *Information and Computation 157,* 1/2 (Mar.), 84–141.

Pfenning, F. and Elliott, C. 1988. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation.* ACM Press, 199–208.

Pfenning, F. and Schürmann, C. 1999. System description: Twelf — A meta-logical framework for deductive systems. In *16th Conf. on Automated Deduction (CADE)*, H. Ganzinger, Ed. Number 1632 in LNAI. Springer, Trento, 202–206.

Pfenning, F. and Simmons, R. J. 2009. Substructural operational semantics as ordered logic programming. In *LICS*. IEEE Computer Society, 101–110.

Pientka, B. and Cave, A. 2015. Inductive beluga: Programming proofs. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, A. P. Felty and A. Middeldorp, Eds. Lecture Notes in Computer Science, vol. 9195. Springer, 272–281.

Pientka, B. and Dunfield, J. 2010. Beluga: A framework for programming and reasoning with deductive systems (system description). In *Fifth International Joint Conference on Automated*

Reasoning, J. Giesl and R. Hähnle, Eds. Number 6173 in Lecture Notes in Computer Science. 15–21.

Pitts, A. M. 2003. Nominal logic, A first order theory of names and binding. *Information and Computation 186,* 2, 165–193.

Polakow, J. and Yi, K. 2000. Proving syntactic properties of exceptions in an ordered logical framework. In *The First Asian Workshop on Programming Languages and Systems, APLAS 2000, National University of Singapore, Singapore, December 18-20, 2000, Proceedings*. 23–32.

Qi, X., Gacek, A., Holte, S., Nadathur, G., and Snow, Z. 2015. The Teyjus system – version 2. http://teyjus.cs.umn.edu/.

Runciman, C., Naylor, M., and Lindblad, F. 2008. Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values. In *Haskell*. ACM, 37–48.

Schack-Nielsen, A. and Schürmann, C. 2008. Celf - A logical framework for deductive and concurrent systems (system description). In *IJCAR*. Lecture Notes in Computer Science, vol. 5195. Springer, 320–326.

Schrijvers, T., Demoen, B., Triska, M., and Desouter, B. 2014. Tor: Modular search with hookable disjunction. *Sci. Comput. Program. 84*, 101–120.

Schroeder-Heister, P. 1993. Rules of definitional reflection. In *8th Symp. on Logic in Computer Science*, M. Vardi, Ed. IEEE Computer Society Press, IEEE, 222–232.

Schürmann, C. 2000. Automating the meta theory of deductive systems. Ph.D. thesis, Carnegie Mellon University. CMU-CS-00-146.

Selinger, P. 2008. Lecture notes on the lambda calculus. https://arxiv.org/abs/0804.3434.

Sullivan, K., Yang, J., Coppit, D., Khurshid, S., and Jackson, D. 2004. Software assurance by bounded exhaustive testing. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '04. ACM, New York, NY, USA, 133–142.

Takahashi, M. 1995. Parallel reductions in lambda-calculus. *Inf. Comput. 118,* 1, 120–127.

Tarau, P. and de Paiva, V. 2020. Deriving theorems in implicational linear logic, declaratively. In *ICLP Technical Communications*. EPTCS, vol. 325. 110–123.

Tassi, E. 2018. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect). working paper or preprint.

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