Towards Substructural Property-Based Testing

Marco Mantovani and Alberto Momigliano(✉)

Dipartimento di Informatica, Università degli Studi di Milano, Milan, Italy
momigliano@di.unimi.it

Abstract. We propose to extend property-based testing to substructural logics to overcome the current lack of reasoning tools in the field. We take the first step by implementing a property-based testing system for specifications written in the linear logic programming language Lolli. We employ the foundational proof certificates architecture to model various data generation strategies. We validate our approach by encoding a model of a simple imperative programming language and its compilation and by testing its meta-theory via mutation analysis.

Keywords: Linear logic · Property-based testing · Focusing · Semantics of programming languages

1 Introduction

Since their inception in the late 80's, logical frameworks based on intuitionistic logic [43] have been successfully used to represent and animate deductive systems (*λProlog*) as well as to reason (*Twelf*) about them. The methodology of *higher-order abstract syntax* (HOAS) together with parametric-hypothetical judgments yields elegant encodings that lead to elegant proofs, since it delegates to the meta-logic the handling of many common notions, in particular the representation of *contexts*. For example, when modeling a typing system, we represent the typing context as a set of parametric (atomic) assumptions: this tends to simplify the meta-theory since properties such as weakening and context substitution come for free: in fact, they are inherited from the logical framework, and do not need to be proved on a case-by-case basis. For an early example, see the proof of subject reduction for MiniML in [35], which completely avoids the need to establish intermediate lemmas, as opposed to more standard and labor-intensive treatments [15].

However, this identification of meta and object level contexts turns out to be problematic in *state-passing* specifications. To fix ideas, consider specifying the operational semantics of an imperative programming language: evaluating

This work has been partially supported by the National Group of Computing Science (GNCS-INdAM) within the project "Estensioni del *Property-based Testing* di e con linguaggi di programmazione dichiarativa".

E. De Angelis and W. Vanhoof (Eds.): LOPSTR 2021, LNCS 13290, pp. 92–112, 2022.
https://doi.org/10.1007/978-3-030-98869-2_6

an assignment requires taking an input state, modifying it and finally returning it. A state (and related notions such as heaps, stacks, etc.) cannot be adequately encoded as a set of intuitionistic assumptions, since it is intrinsically ephemeral. The standard solution of reifing the state into a data structure, while doable, betrays the whole HOAS approach.

Luckily, linear logic can change the world [51]—Linear logic being of course the main example of *substructural* logics, i.e., those non-classical logics characterized by the absence of some structural rules [41]. Linearity provides a notion of context which has an immediate reading in terms of *resources*. A state can be seen as a set of linear assumptions and the linear connectives can be used to model in a declarative way reading and writing said state. In the early 90's this idea was taken up in linear logic programming and specification languages such as *Lolli* [22], *LLF* [7] and *Forum* [36].

In the ensuing years, given the richness of linear logic and the flexibility of the proof-theoretic foundations of logic programming [37], more sophisticated languages emerged, with additional features such as order (*Olli* [46]), subexponentials [40], bottom-up evaluation and concurrency (*Lollimon* [29], *Celf* [49]). Each extension required significant ingenuity, since it relied on the development of appropriate notions of canonical forms, resource management, unification etc. At the same time, tools for *reasoning* over such substructural specifications did not materialize. Meta-reasoning over a logical framework, in fact, asks for formulating appropriate meta-logics, which, again, is far from trivial, the more when the framework is substructural; in fact no implementation of the latter have appeared. The case for the concurrent logical framework CLF [8] is particularly striking, where, notwithstanding a wide and promising range of applications, the only meta-theoretic analysis available in *Celf* is checking that a program is well-moded. Compare this with the successful deployment of dedicated HOAS-based intuitionistic proof assistants such as *Beluga* [45] and *Abella* [1].

If linear verification is too hard, or just while we wait for the field to catch up, this paper suggests *validation* as a useful alternative, in particular in the form of *property-based testing* [24] (PBT). This is a lightweight validation technique whereby the user specifies executable properties that the code should satisfy and the system tries to refute them via automatic (typically random) data generation.

Previous work [4] gave a proof-theoretic reconstruction of PBT in terms of focusing and *foundational proof certificates* (FPC) [12], which, in theory, applies to all the languages mentioned above. The promise of the approach is that we can state and check properties in the very logic where we specify them, without resorting to a further meta-logic. Of course, validation falls rather short of verification, but as by now common in mainstream proof assistants, e.g., [5,42], we may resort to testing not only *in lieu of* proving, but *before* proving, so as to avoid pointless effort in trying to prove false theorems or true properties over bugged models.

In fact, the two-level architecture [19] underlying the *Abella* system and the *Hybrid* library [17] seems a good match for the combination of testing and proving over substructural specifications. In this architecture we keep the meta-logic

fixed, while making substructural the specification logic. Indeed, some case studies have been already carried out, as we detail in Sect. 6.

In this paper we move the first steps in this programme implementing PBT for Lolli and evaluating its capability in catching bugs by applying it to a mid-size case study: we give a linear encoding of the static and dynamic semantic of an imperative programming language and its compilation into a stack machine and validate several properties, among which type preservation and soundness of compilation. We have tried to test properties in the way they would be stated and hopefully proved in a linear proof assistant based on the two-level architecture. That is, we are not arguing (yet) that linear PBT is "better" than traditional ones for state-passing specifications. Besides, in the case studies we have carried out so far, we generate only *persistent* data (expressions, programs) under a given linear context. Rather, we advocate the coupling of validation and (eventually) verification for those encoding where linearity makes a difference in terms of drastically simplifying the infrastructure needed to prove the main result: one of the original success stories of linear specifications, namely type preservation of MiniML with *references* [7,34], still stands and nicely extends the cited one for MiniML: linearly, the theorem can be proven from first principles, while with a standard encoding, for example the Coq formalization in *Software foundations*[1], one needs literally dozens of preliminary lemmas.

The rest of the paper is organized as follows: we start in the next Sect. 2 with a short example of model-based testing of a linear specification. Section 3 gives a short introduction to the proof-theory of intuitionistic linear logic programming, while Sect. 4 applies the notion of FPC to our reconstruction of PBT. Next (Sect. 5), we validate our approach with a case study concerning the meta-theory of a basic imperative language including an experimental evaluation (Sect. 5.2). We conclude in Sect. 6 with a short review of related and future work.

We assume in the following a passing familiarity with linear logic and with the proof-theoretic foundations of logic programming [37].

2 A Motivating Example

To preview our methodology, we present a self-contained example where we use PBT in the guise of *model-based* testing: we test an implementation against a *trusted* version. We choose as trusted model the linear encoding of the implicational fragment of the *contraction-free* calculus for propositional intuitionistic logic, popularized by Dyckhoff. Figure 1 lists the rules for the judgment $\Gamma \vdash C$, together with a Lolli implementation. Here, and in the following, we will use Lolli's concrete syntax, where the lollipop (in both directions) is linear implication, `x` is multiplicative conjunction (tensor), `&` is additive conjunction and `erase` its unit $\top$. The of-course modality is `bang`.

As shown originally in [22], we can encode provability with a predicate `pv` that uses a linear context of propositions `hyp` for assumptions, that is occurring

[1] https://softwarefoundations.cis.upenn.edu/plf-current/References.html.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ R_{\to} \qquad \frac{}{\Gamma, A \vdash A}\ \mathit{init}$$

$$\frac{\Gamma, B, a \vdash C}{\Gamma, a \to B, a \vdash C}\ L^{a}_{\to} \qquad \frac{\Gamma, A_2 \to B \vdash A_1 \to A_2 \quad \Gamma, B \vdash C}{\Gamma, (A_1 \to A_2) \to B \vdash C}\ L^{i}_{\to}$$

```
pv(imp(A,B)) o- (hyp(A) -o pv(B)).
pv(A) o- hyp(A) x erase.
pv(C) o- hyp(imp(A,B)) x bang(atom(A)) x hyp(A) x
         (hyp(B) -o hyp(A) -o pv(C)).
pv(C) o- hyp(imp(imp(A1,A2),B)) x
         (hyp(imp(A2,B) -o pv(imp(A1,A2))) &
         (hyp(B) -o pv(C))).
```

Fig. 1. Rules for contraction free LJF$_{\to}$ and their Lolli encoding

at the left of the turnstile; this is shown in the first clause encoding the implication right rule $R_{\to}$ via the embedded implication `hyp(A) -o pv(B)`. In the left rules, the premises are consumed by means of the tensor and new assumptions (re)asserted. The fact `atom(A)` lives on thanks to the bang, since it may need to be reused. Note how in the encoding of rule $L^{i}_{\to}$, the context Γ is duplicated through additive conjunction. The *init* rule disposes via `erase` of any remaining assumption since the object logic enjoys weakening. By construction, the above code is a decision procedure for LJF$_{\to}$.

Taking inspiration from Tarau's [50], we consider next an optimization where we factor the two left rules for implication in one:

```
... % similar to before
pvb(C) o- hypb(imp(A,B)) x pvbi(A,B) x
          (hypb(B) -o pvb(C)).

pvbi(imp(C,D),B) o- hypb(imp(D,B)) -o pvb(imp(C,D)).
pvbi(A,_)        o- hypb(A).
```

Does the optimization preserve provability? Formally, the conjecture is $\forall A\colon \mathit{form}.\ \mathit{pv}(A) \supset \mathit{pvb}(A)$. We could try to prove it, although, for the reasons alluded to in the introduction, it is not clear in which (formalized) metalogic we would carry out such proof. Instead, it is simpler to test, that is to search for a counter-example. And the answer is no, the (encoding of the) optimization is faulty, as witnessed by the (pretty printed) counterexample `A => ((A => (A => B)) => B)`: this intuitionistic tautology fails to be provable in the purported optimization. We leave the fix to the reader.

3 A Primer on Linear Logic Programming

In this section we introduce some basic notions concerning the proof-theoretic foundations of intuitionistic linear logic programming. We follow quite closely

$$\frac{}{B \vdash B}\ \texttt{id} \qquad \frac{}{\Delta \vdash \top}\ \top\texttt{-R}$$

$$\frac{\Delta, B_i \vdash C}{\Delta, B_1 \,\&\, B_2 \vdash C}\ \&\texttt{-L}_i\ (i = 1, 2) \qquad \frac{\Delta \vdash B \quad \Delta \vdash C}{\Delta \vdash B \,\&\, C}\ \&\texttt{-R}$$

$$\frac{\Delta, B_1, B_2 \vdash C}{\Delta, B_1 \otimes B_2 \vdash C}\ \otimes\texttt{-L}_i \qquad \frac{\Delta_1 \vdash B \quad \Delta_2 \vdash C}{\Delta_1, \Delta_2 \vdash B \otimes C}\ \otimes\texttt{-R}$$

$$\frac{\Delta_1 \vdash B \quad \Delta_2, C \vdash E}{\Delta_1, \Delta_2, B \multimap C \vdash E}\ \multimap\texttt{-L} \qquad \frac{\Delta, B \vdash C}{\Delta \vdash B \multimap C}\ \multimap\texttt{-R}$$

$$\frac{\Gamma, B \vdash C}{\Delta, !B \vdash C}\ !\texttt{-D} \qquad \frac{!\Delta \vdash B}{!\Delta \vdash !B}\ !\texttt{-R}$$

$$\frac{\Delta \vdash E}{\Delta, !B \vdash E}\ !\texttt{-W} \qquad \frac{\Delta, !B, !B \vdash C}{\Delta, !B \vdash C}\ !\texttt{-C}$$

$$\frac{\Delta, B[t/x] \vdash C}{\Delta, \forall x.B \vdash C}\ \forall\texttt{-L} \qquad \frac{\Delta \vdash B[y/x]}{\Delta \vdash \forall x.B}\ \forall\texttt{-R}$$

provided that y is not free in the lower sequent.

Fig. 2. A sequent calculus for a fragment of linear logic.

the account by Miller and Hodas [22], to which we refer for more details and motivations. It is possible, although slightly more technically involved, to give a more general and modern treatment of the proof-theory in terms of *focusing* [28].

A *substructural* logic differs from classical and intuitionistic logic by restricting or even dropping from its proof-theory one of the usual structural rules, namely weakening, contraction, and exchange. *Linear* logic [21] is probably the most well-known: by controlling the use of contraction and weakening we can view logical deduction no longer as an ever-expanding collection of persistent "truths", but as a way of manipulating *resources* that cannot be arbitrarily duplicated or thrown away.

A linear logic programming language such as Lolli extends conservatively the logic behind λProlog, that is (first-order) Hereditary Harrop formulæ (HHF), which can be seen as the language freely generated by $\top$, $\wedge$, $\Rightarrow$ and $\forall$. Therefore it is natural to refine HHF via the connectives $\top, \&, \otimes, \multimap, !, \forall$. We present the proof-theory of this language as a two-sided sequent calculus (Fig. 2) based on the judgment $\Delta \vdash B$[2], where B is a formula over the above connectives and Δ is a multi-set of formulas. We use "," to denote both multi-set union and adding a formula to a context; further, with $!\Delta$ we mean the multiset $\{!B \mid B \in \Delta\}$. Contraction and weakening are allowed only on unrestricted assumptions (rules !-W and !-C). Linear logic induces a related distinction between connectives, which now come in two flavors: additive and multiplicative. The former duplicate the context, e.g., additive conjunction (&-R), the latter split it, e.g., multiplicative conjunction (⊗-R).

[2] We overload "⊢" to denote provability for all the sequent systems in this paper, counting on the structure of antecedent and consequent to disambiguate.

$$\frac{\Delta \vdash G_1 \quad \Delta \vdash G_2}{\Delta \vdash G_1 \,\&\, G_2} \qquad \frac{}{\cdot \vdash \mathbf{1}} \qquad \frac{\Delta_1 \vdash G_1 \quad \Delta_2 \vdash G_2}{\Delta_1, \Delta_2 \vdash G_1 \otimes G_2} \qquad \frac{}{\Delta \vdash \top}$$

$$\frac{\Delta, \alpha \vdash G}{\Delta \vdash \alpha \multimap G} \qquad \frac{\cdot \vdash G}{\cdot \vdash\, !G} \qquad \frac{\Delta, !A, A \vdash G}{\Delta, !A \vdash G}$$

$$\frac{}{!\Delta, A \vdash A} \qquad \frac{\Delta \vdash G \quad G \multimap A \in grnd(\mathcal{P})}{\Delta \vdash A}$$

Fig. 3. Uniform proofs for second order LHHF

While this calculus is well-understood, it cannot be seen as an abstract logic programming language in the sense of [37], since it does not enjoy the uniform proof property: the latter allows one to see a cut-free sequent derivation $\Gamma \vdash G$ as the state of an interpreter trying to establish if G follows from Γ. More technically, a proof is *uniform* if every occurrence of a sequent with non-atomic succedent is the conclusion of a right introduction rule.

For the fragment in Fig. 2, the problem boils down to the non-permutability of the right rules for tensor and of-course over the left rules. Miller & Hodas' solution was to limit the occurrences of those troublesome operators. We go a little bit further, following large part of the literature [4,17,19], and adopt an additional minor restriction of linear Hereditary Harrop formulæ: we limit ourselves to implications with *atomic*, possibly banged, premises. We also drop universal goals, since our term language is first-order (as opposed to λProlog), making universal goals essentially useless. On the other hand, we introduce as goals (not as first class connectives) the tensor and the of-course modality. This allows us to view, as usual, intuitionistic implication as defined: $A \Rightarrow B$ is mapped to $!A \multimap B$. Having both forms (linear and unrestricted) of hypothetical judgments is an essential ingredient in the art of linear logic specifications. Programs are sets of the universal closure of clauses of the form $G \multimap A$, which are fixed and implicitly banged, since they can be used as many times as needed. The grammar of second-order LHHF follows:

$$\begin{array}{lll}
\text{Goals} & G & ::= A \mid \top \mid \mathbf{1} \mid \alpha \multimap G \mid\, !G \mid G_1 \otimes G_2 \mid G_1 \,\&\, G_2 \\
\text{Clauses} & D & ::= \forall(G \multimap A) \\
\text{Programs} & \mathcal{P} & ::= \cdot \mid \mathcal{P}, D \\
\text{Assumption} & \alpha & ::= A \mid\, !A \\
\text{Context} & \Delta & ::= \cdot \mid \Delta, \alpha \\
\text{Atoms} & A & ::= \ldots
\end{array}$$

This reformulation of LHHF leads to the calculus in Fig. 3, which is closer to our intuition of a logic programming interpreter since the left rules have been replaced by *backchaining* (last rule) over all the ground instances (*grnd*) of a program. Note how the additive unit $\top$ allows one to discard any remaining assumption, while $\mathbf{1}$ holds only if all resources have been consumed. Similarly,

$$\frac{\Delta_I \setminus \Delta_O \vdash G_1 \quad \Delta_I \setminus \Delta_O \vdash G_2}{\Delta_I \setminus \Delta_O \vdash G_1 \,\&\, G_2} \qquad \frac{}{\Delta_I \setminus \Delta_I \vdash \mathbf{1}}$$

$$\frac{\Delta_I \setminus \Delta_M \vdash G_1 \quad \Delta_M \setminus \Delta_O \vdash G_2}{\Delta_I \setminus \Delta_O \vdash G_1 \otimes G_2} \qquad \frac{\Delta_O \subseteq \Delta_I}{\Delta_I \setminus \Delta_O \vdash \top}$$

$$\frac{\Delta_I, \alpha \setminus \Delta_O, \Box \vdash G}{\Delta_I \setminus \Delta_O \vdash \alpha \multimap G} \qquad \frac{\Delta_I \setminus \Delta_I \vdash G}{\Delta_I \setminus \Delta_I \vdash !G}$$

$$\frac{}{\Delta_I, A \setminus \Delta_I, \Box \vdash A} \qquad \frac{}{\Delta_I, !A \setminus \Delta_O, !A \vdash A}$$

$$\frac{\Delta_I \setminus \Delta_O \vdash G \quad G \multimap A \in grnd(\mathcal{P})}{\Delta_I \setminus \Delta_O \vdash A}$$

Fig. 4. The IO system for second order LHHF.

a bang can hold only if it does not depend on any resource. In the axiom rule, A is the only ephemeral assumption. Unrestricted assumptions can be copied at will.

By adapting the techniques in [22], we can show that second-order LHHF has the uniform proof property. However, the latter does not address the question of how to perform proof search in the presence of linear assumptions, a.k.a. the *resource management problem* [6]. The problem is firstly caused by multiplicative connectives that, under a goal-oriented strategy, require a potentially exponential partitioning of the given linear context, case in point the tensor right rule. Another source of non-determinism is the rule for $\top$, since it puts no constraint on the required context.

A solution to the first issue is based on *lazy* context splitting and it is known as the *IO system*: it was introduced in [22], and further refined in [6]: when we need to split a context (in our fragment only in the tensor case), we give to one of the sub-goal the whole input context (Δ_I): some of it will be consumed and the leftovers (Δ_O) returned to be used by the other sub-goal.

Figure 4 contains a version of the IO system for our language as described by the judgment $\Delta_I \setminus \Delta_O \vdash G$, where $\setminus$ is just a suggestive notation to separate input and output context. Following the literature and our implementation, we will signal that a resource has been consumed in the input context by replacing it with the placeholder "$\Box$".

The IO system is known to be sound and complete w.r.t. uniform provability: $\Delta_I \setminus \Delta_O \vdash G$ iff $\Delta_I - \Delta_O \vdash G$, where "$-$" is context difference modulo $\Box$ (see [22] for the definition). Given this relationship, the requirement for the linear context to be empty in the right rules for $\mathbf{1}$ and ! is realized by the notation $\Delta_I \setminus \Delta_I$. In particular, in the linear axiom rule, A is the only available resource, while in the intuitionistic case, $!A$ is not consumed. The tensor rule showcases lazy context splitting, while additive conjunction duplicates the linear context.

The handling of $\top$ is sub-optimal, since it succeeds with any subset of the input context. As well known, this could be addressed by using the notion of

slack [6] to remove $\top$-non determinism. However, given the preferred style of our encodings (see Sect. 5), where additive unit is called only as a last step, this has so far not proved necessary.

4 The Proof-Theory of PBT

While PBT originated in a functional programming setting [14], at least two factors make a proof-theoretic reconstruction fruitful:

1. it fits nicely with a (co)inductive reading of rule-based presentations of a system-under-test;
2. it easily generalizes to richer logics.

If we view a property as a logical formula $\forall x[(\tau(x) \wedge P(x)) \supset Q(x)]$ where τ is a typing predicate, providing a counter-example consists of negating the property, and therefore searching for a proof of $\exists x[(\tau(x) \wedge P(x)) \wedge \neg Q(x)]$.

Stated in this way the problem points to a logic programming solution, and this means uniform proofs or more generally, proof-search in a *focused* sequent calculus [28], where the specification is a set of assumptions (typically sets of clauses) and the negated property is the query.

The connection of PBT with focused proof search is that in such a query the *positive phase* is represented by $\exists x$ and $(\tau(x) \wedge P(x))$. This corresponds to the generation of possible counter-examples under precondition P. That is followed by the *negative phase* (which corresponds to counter-example testing) and is represented by $\neg Q(x)$. This formalizes the intuition that generation may be arbitrarily hard, while testing is just a deterministic computation.

How do we supply external information to the positive phase? In particular, how do we steer data generation? This is where the theory of *foundational proof certificates* [12] (FPC) comes in. For the type-theoretically inclined, FPC can be understood as a generalization of proof-terms in the Curry-Howard tradition. They have been introduced to define and share a range of proof structures used in various theorem provers (e.g., resolution refutations, Herbrand disjuncts, tableaux, etc.). A FPC implementation consists of

1. a generic proof-checking kernel,
2. the specification of a certificate format, and
3. a set of predicates (called *clerks and experts*) that decorate the sequent rules used in the kernel and help to process the certificate.

In our setting, we can view those predicates as simple logic programs that guide the search for potential counter-examples using different generation strategies. The following special case may clarify the idea: consider two variations of the beloved Prolog vanilla meta-interpreter, where in the left-hand side we bound the derivation by its *height* and in the right-hand side we limit the number of clauses used (*size*): for the latter, N is input and M output, so the size will be $N - M$. For convenience we use numerals.

$$\frac{\Xi_1 : \Delta_I \setminus \Delta_O \vdash G_1 \quad \Xi_2 : \Delta_I \setminus \Delta_O \vdash G_2 \quad \&_e(\Xi, \Xi_1, \Xi_2)}{\Xi : \Delta_I \setminus \Delta_O \vdash G_1 \,\&\, G_2} \qquad \frac{\mathbf{1}_e(\Xi)}{\Xi : \Delta_I \setminus \Delta_I \vdash \mathbf{1}}$$

$$\frac{\Xi_1 : \Delta_I \setminus \Delta_M \vdash G_1 \quad \Xi_2 : \Delta_M \setminus \Delta_O \vdash G_2 \quad \otimes_e(\Xi, \Xi_1, \Xi_2)}{\Xi : \Delta_I \setminus \Delta_O \vdash G_1 \otimes G_2} \qquad \frac{\Delta_I \supseteq \Delta_O \quad \top_e(\Xi)}{\Xi : \Delta_I \setminus \Delta_O \vdash \top}$$

$$\frac{\Xi' : \Delta_I, \alpha \setminus \Delta_O, \square \vdash G \quad \multimap_e(\Xi, \Xi')}{\Xi : \Delta_I \setminus \Delta_O \vdash \alpha \multimap G} \qquad \frac{\Xi' : \Delta_I \setminus \Delta_I \vdash G \quad !_e(\Xi, \Xi')}{\Xi : \Delta_I \setminus \Delta_I \vdash !G}$$

$$\frac{init_e(\Xi)}{\Xi : \Delta_I, A \setminus \Delta_I, \square \vdash A} \qquad \frac{init!_e(\Xi)}{\Xi : \Delta_I, !A \setminus \Delta_O, !A \vdash A}$$

$$\frac{\Xi' : \Delta_I \setminus \Delta_O \vdash G \quad (G \multimap A) \in grnd(\mathcal{P}) \quad unfold_e(\Xi, \Xi', A, G)}{\Xi : \Delta_I \setminus \Delta_O \vdash A}$$

Fig. 5. FPC presentation of the IO system for second order Lolli

```
demo(_,true).
demo(H,(G1,G2)) :-
 demo(H,G1),demo(H,G2).
demo(s(H),A)    :-
 clause(A,G),demo(H,G).
```

```
demo(N,N,true).
demo(N,M,(G1,G2)) :-
 demo(N,T,G1),demo(T,M,G2).
demo(s(N),M,A)    :-
 clause(A,G),demo(N,M,G).
```

Not only is this code repetitious, but it reflects just two specific derivations strategies. We can abstract the pattern by replacing the concrete bounds with a variable to be instantiated with a specific certificate format and add for each case/rule a predicate that will direct the search according to the given certificate.

```
demo(Cert,true)    :-
  trueE(Cert).
demo(Cert,(G1,G2)) :-
 andE(Cert,Cert1,Cert2),
 demo(Cert1,G1),demo(Cert2,G2).
demo(Cert,A)       :-
 unfoldE(Cert,Cert1),
 clause(A,G),demo(Cert1,G).
```

Then, it is just a matter to provide the predicates, implicitly fixing the certificate format:

```
trueE(height(_)).
trueE(size(N,N)).
andE(height(H),height(H),height(H)).
andE(size(N,M),size(N,T),size(T,M)).
unfoldE(height(s(H)),height(H)).
unfoldE(size(s(N),M),size(N,M)).
```

With this intuition in place, we can take the final step by augmenting each inference rule of the system in Fig. 4 with an additional premise involving an

expert predicate, a certificate Ξ, and possibly resulting certificates (Ξ', Ξ_1, Ξ_2), reading the rules from conclusion to premises. Operationally, the certificate Ξ is an input in the conclusion of a rule and the continuations are computed by the expert to be handed over to the premises, if any. We sum up the rules in Fig. 5.

As we have said, the FPC methodology requires to describe a format for the certificate. Since in this paper we use FPC only to guide proof-search, we fix the following three formats and we allow their composition, known as *pairing*:

$$\text{Certificates } \Xi ::= n \mid \langle n, m\rangle \mid d \mid (\Xi, \Xi)$$

Following on the examples above, the first certificate is just a natural number (*height*), while the second consists of a pair of naturals (*size*). In the third case, d stands for a *distribution* of weights to clauses in a predicate definition, to be used for random generation; if none is given, we assume a uniform distribution. Crucially, we can compose certificates, so that for example we can offer random generation bounded by the height of the derivation; pairing is a simple, but surprisingly effective combinator [3].

Each certificate format is accompanied by the implementation of the predicates that process the certificate in question. We exemplify the FPC discipline with a selection of rules instantiated with the *size* certificates. If we run the judgment $\langle n, m\rangle : \Delta_I \setminus \Delta_O \vdash G$, the inputs are n, Δ_I and G, while Δ_O and m will be output.

$$\frac{\langle n-1, m\rangle : \Delta_I \setminus \Delta_O \vdash G \quad (A \leftarrow G) \in grnd(\mathcal{P}) \quad n > 0}{\langle n, m\rangle : \Delta_I \setminus \Delta_O \vdash A} \qquad \frac{}{\langle n, n\rangle : \Delta_I \setminus \Delta_I \vdash \mathbf{1}}$$

$$\frac{\langle i, m\rangle : \Delta_I \setminus \Delta_M \vdash G_1 \quad \langle m, o\rangle : \Delta_M \setminus \Delta_O \vdash G_2}{\langle i, o\rangle : \Delta_I \setminus \Delta_O \vdash G_1 \otimes G_2}$$

$$\frac{\langle n, m\rangle : \Delta_I \setminus \Delta_O \vdash G_1 \quad \langle n, m\rangle : \Delta_I \setminus \Delta_O \vdash G_2}{\langle n, m\rangle : \Delta_I \setminus \Delta_O \vdash G_1 \,\&\, G_2}$$

Here (as in all the formats considered in this paper), most experts are rather simple; they basically hand over the certificate according to the connective. This is the case of & and **1**, where the expert copies the bound and its action is implicit in the instantiation of the certificates in the premises. In the tensor rule, the certificate mimics context splitting. The *unfold* expert, instead, is more interesting: not only it decreases the bound, provided we have not maxed out on the latter, but it is also in charge of selecting the next goal: for bounded search via chronological backtracking, for random data generation via random backtracking: every time the derivation reaches an atom, we permute its definition and pick a matching clause according to the distribution described by the certificate. Other strategies are possible, as suggested in [18]: for example, permuting the definition just once at the beginning of generation, or even randomizing the conjunctions in the body of a clause.

Note that we have elected *not* to delegate to the experts resource management: while possible, it would force us to pair such certificate with any other one. As detailed in [4], more sophisticated FPC capture other features of PBT, such as shrinking and bug-provenance, and will not be repeated here.

We are now ready to account for the soundness property from the example in Sect. 2. By analogy, this applies to certificate-driven PBT with a liner IO kernel in general. Let Ξ be here the height certificate with bound 4 and `form(_)` a unary predicate describing the syntax of implicational formulæ, which we use as a generator. Testing the property amounts to the following query in a host language that implements the kernel:

$$\exists F.\ (\Xi : \cdot \setminus \cdot \vdash \texttt{form(F)}) \wedge (\Xi : \cdot \setminus \cdot \vdash \texttt{pv(F)}) \wedge \neg(\Xi : \cdot \setminus \cdot \vdash \texttt{pvb(F)})$$

In our case, the meta-language is simply Prolog, where we encode the kernel with a predicate `prove/4` and to check for un-provability negation-as-failure suffices, as argued in [4].

```
C = height(4),prove(C,[],[],form(F)),
    prove(C,[],[],pv(F)),\+ prove(C,[],[],pvb(F)).
```

5 Case Study

IMP is a model of a minimalist Turing-complete imperative programming language, featuring instructions for assignment, sequencing, conditional and loop. It has been extensively used in teaching and in mechanizations (viz. formalized textbooks such as *Software Foundations* and *Concrete Semantics*[3]). Here we follow Leroy's account [27], but add a basic type system to distinguish arithmetical from Boolean expressions.

IMP is a good candidate for a linear logic encoding, since its operational semantics is, of course, state-based, while its syntax (see below) is simple enough not to require a sophisticated treatment of binders.

`expr` ::=	`var`	*variable*
|	`i`	*integer constant*
|	`b`	*Boolean constant*
|	`expr` + `expr`	*addition*
|	`expr` − `expr`	*subtraction*
|	`expr` ∗ `expr`	*multiplication*
|	`expr` ∧ `expr`	*conjunction*
|	`expr` ∨ `expr`	*disjunction*
|	¬`expr`	*negation*
|	`expr` == `expr`	*equality*

`val` ::=		
|	`vi`	*integer value*
|	`vb`	*Boolean value*

`ty` ::=		
|	`tint`	*integers type*
|	`tbool`	*Bool type*

[3] softwarefoundations.cis.upenn.edu and concrete-semantics.org.

```
cmd ::= skip                          no op
      | cmd ; cmd                     sequence
      | if expr then cmd else cmd     conditional
      | while expr do cmd             loop
      | var = expr                    assignment
```

The relevant judgments describing the dynamic and static semantics of IMP are:

$\sigma \vdash e \Downarrow v$ big step evaluation of expressions;
$(c, \sigma) \Downarrow \sigma'$ big step execution of commands;
$(c, \sigma) \rightsquigarrow (c', \sigma')$ small step execution of commands and its Kleene closure;
$\Gamma \vdash e \ : \ \tau$ well-typed expressions and $v : \tau$ well-typed values;
$\Gamma \vdash c$ well-typed commands and $\Gamma \ : \ \sigma$ well-typed states;

5.1 On Linear Encodings

In traditional accounts, a state σ is a (finite) map between variables and values. Linear logic takes a "distributed" view and represents a state as a multi-set of linear assumptions. Since this is central to our approach, we make explicit the (overloaded) encoding function $\ulcorner \cdot \urcorner$ on states. Its action on values is as expected and therefore omitted:

$$\begin{aligned} \sigma &::= \cdot \mid \sigma,\ x \mapsto v \\ \ulcorner \cdot \urcorner &= \emptyset \\ \ulcorner \sigma, x \mapsto v \urcorner &= \ulcorner \sigma \urcorner, var(x, \ulcorner v \urcorner) \end{aligned}$$

When encoding state-based computations such as evaluation and execution in a Lolli-like language, it is almost forced on us to use a *continuation-passing style* (CPS [47]): by sequencing the computation, we get a handle on how to express "what to compute next", and this turns out to be the right tool to encode the operational semantics of state update, the more when the modeled semantics has side-effects, lest adequacy is lost.

Yet, even under the CPS-umbrella, there are choices: e.g., whether to adopt an encoding that privileges *additive* connectives, in particular when using the state in a non-destructive way. In the additive style, the state is duplicated with & and then eventually disposed of via $\top$ at the leaves of the derivation.

This is well-understood, but it would lead to the reification of the continuation as a data structure and the introduction of an additional layer of instructions to manage the continuation: for an example, see the static and dynamic semantics of MiniMLR in [7][4].

Mixing additive and multiplicative connectives asks for a more sophisticated resource management system; this is a concern, given the efficiency requirements that testing brings to the table: the idea behind "QuickCheck", and hence its name, is that an outcome should be produced quickly.

[4] This can be circumvented by switching to a more expressive logic, either by internalizing the continuation as an ordered context [46] or by changing representation via forward chaining (*destination-passing style*) [29].

A solution comes from the notion of *logical* continuation advocated by Chirimar [13], which affords us the luxury to never duplicate the state. Logical continuations need higher-order logic (or can be simulated in an un-typed setting such as Prolog). Informally, the idea is to transform every atom A of type $(\tau_1 * \cdots * \tau_n) \to o$ into a new one $\hat{A}$ of type $(\tau_1 * \cdots * \tau_n * o) \to o$ where we accumulate in the additional argument the body of the definition of A as a nested goal. Facts are transformed so that the continuation becomes the precondition.

For example, consider a fragment of the rules for the evaluation judgment $\sigma \vdash m \Downarrow v$ and its CPS-encoding:

$$\frac{x \mapsto v \in \sigma}{\sigma \vdash x \Downarrow v}\ e/v \qquad \frac{}{\sigma \vdash n \Downarrow n}\ e/n$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \quad \sigma \vdash e_2 \Downarrow v_2 \quad \text{plus } v_1\ v_2\ v}{\sigma \vdash e_1 + e_2 \Downarrow v}\ e/p$$

```
eval(v(X),N,K)              o- var(X,N) x (var(X,N) -o K).
eval(i(N),vi(N),K)          o- K.
eval(plus(E1,E2),vi(V),K)  o-
   eval(E1,vi(V1),eval(E2,vi(V2),bang(sum(V1,V2,V,K)))).
```

In the variable case, the value for `X` is read (and consumed) in the linear context and consequently reasserted; then we call the continuation in the restored state. Evaluating a constant `i(N)` will have the side-effect of instantiating `N` in `K`. The clause for addition showcases the sequencing of goals inside the logical continuation, where the `sum` predicate is "banged" as a computation that does not need the state.

The *adequacy* statement for CPS-evaluation reads: $\sigma \vdash m \Downarrow v$ iff the sequent $\ulcorner\sigma\urcorner \vdash \texttt{eval}(\ulcorner m\urcorner, \ulcorner v\urcorner, \top)$ has a uniform proof, where the initial continuation $\top$ cleans up $\ulcorner\sigma\urcorner$ upon success. As well-know, we need to generalize the statement to arbitrary continuations for the proof to go through.

It is instructive to look at a direct additive encoding as well:

```
ev(v(X),V)              o- var(X,V) x erase.
ev(i(N),vi(N))          o- erase.
ev(plus(E1,E2),vi(V))   o- ev(E1,vi(V1)) &
                           ev(E2,vi(V2)) &
                           bang(sum(V1,V2,V)).
```

While this seems appealingly simpler, it breaks down when the state is updated and not just read; consider the operational semantics of assignment and its CPS-encoding:

$$\frac{\sigma \vdash m \Downarrow v}{(\sigma, x := m) \Downarrow \sigma \oplus \{x \mapsto v\}}$$

```
ceval(asn(X,E),K) o-
                  eval(E,V, (var(X,_) x (var(X,V) -o K))).
```

The continuation is in charge of both having something to compute after the assignment returns, but also of sequencing in the right order reading the state via evaluation, and updating via the embedded implication. An additive encoding using & would not be adequate, since the connective's commutativity is at odd with side-effects.

At the top level, we initialize the execution of programs (seen as a sequence of commands) by using as initial continuation a predicate `collect` that consumes the final state and returns it in a reified format.

```
main(P,Vars,S) o- ceval(P,collect(Vars,S)).
```

We are now in the position of addressing the meta-theory of our system-under-study via testing. We list the more important properties among those that we have considered. All statements are universally quantified:

srv subject reduction for evaluation: $\Gamma \vdash m : \tau \longrightarrow \sigma \vdash m \Downarrow v \longrightarrow \Gamma \ : \ \sigma \longrightarrow v : \tau$;
dtx determinism of execution: $(\sigma, c) \Downarrow \sigma_1 \longrightarrow (\sigma, c) \Downarrow \sigma_2 \longrightarrow \sigma_1 \approx \sigma_2$;
srx subject reduction for execution: $\Gamma \vdash c \longrightarrow \Gamma \ : \ \sigma \longrightarrow (\sigma, c) \Downarrow \sigma' \longrightarrow \Gamma \ : \ \sigma'$;
pr progress for small step execution: $\Gamma \vdash c \longrightarrow \Gamma \ : \ \sigma \longrightarrow c = \texttt{skip} \vee \exists c'\, \sigma',\ (c, \sigma) \rightsquigarrow (c', \sigma')$;
eq equivalence of small and big step execution (assuming determinism of both): $(\sigma, c) \Downarrow \sigma_1 \longrightarrow (c, \sigma_1) \rightsquigarrow^* (\texttt{skip}, \sigma_2) \longrightarrow \sigma_1 \approx \sigma_2$.

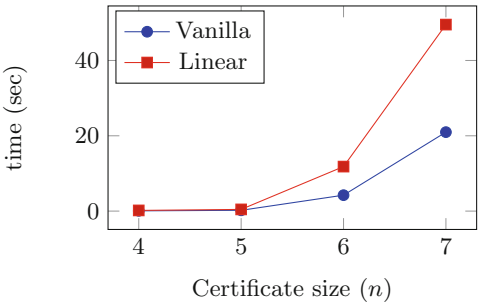
We have also encoded the compilation of IMP to a stack machine and (mutation) tested forward and backward simulation of compilation w.r.t. source and target execution [27]. We have added a simple type discipline for the assembly language in the spirit of Typed Assembly Languages [39] and tested preservation and progress, to exclude underflows in the execution of a well-typed stack machine. Details can be found in the accompanying repository[5].

5.2 Experimental Evaluation

A word of caution before discussing our experiments: first, we have spent almost no effort in crafting nor tuning custom generators; in fact, they are simply FPC-driven regular unary logic programs [52] with a very minor massage. Compare this with the amount of ingenuity poured in writing generators in [23] or with the model-checking techniques of [48]. Secondly, our interpreter is a Prolog meta-interpreter and while we have tried to exploit Prolog's indexing, there are obvious ways to improve its efficiency, from partial evaluation to better data structures for contexts.

Of the many experiments that we have run and are available in the dedicated repository, we list here only a few, with no pretense of completeness. In those, we have adopted a certain exhaustive generation strategy (*size*), then paired it

[5] https://github.com/Tovy97/Towards-Substructural-Property-Based-Testing/tree/master/Lolli/Assembly.



Size	Generated data	coverage
4	384	68%
5	1160	85%
6	15344	68%
7	63064	72%

Fig. 6. Testing property **eq** with certificate $\langle n, \_ \rangle$

	dtx	**srx**	**srv**	**pr**	**eq**	**cex**
M1	pass	pass	pass	pass	pass	
M2	found	pass	pass	pass	found	`w := 0 - 1`
M3	pass	pass	pass	pass	pass	
M4	pass	found	found	pass	pass	`x := tt /\ tt`
M5	pass	pass	pass	pass	pass	
M6	found	pass	pass	pass	found	`if x = x then {w := 0} else {w := 1}`
M7	pass	pass	pass	pass	pass	
M8	pass	pass	pass	pass	pass	
M9	pass	pass	pass	pass	pass	

Fig. 7. Mutation testing

with *height*. We have used consistently certain bounds that experimentally have shown to be effective in generating enough interesting data.

To establish a fair baseline, we have also implemented a *state-passing* version of our benchmarks driven by a FPC-lead vanilla meta-interpreter.

We have run the experiments on a laptop with an Intel i7–7500U CPU and 16 GB of RAM running WSL (Ubuntu 20.04) over Windows 10, using SWI-Prolog 8.2.4. All times are in seconds, as reported by SWI's `time/1`. They are the average of five measurements.

First we compare the time to test a sample property ("**eq**", the equivalence of big and small step execution) over a bug-free model both with linear and vanilla PBT. On the left of Fig. 6 we plot the time proportionally to the certificate size. On the right we list the number of generated programs and the percentage of those that converge within a bound given by a polynomial function over the certificate size (column "coverage"). The linear interpreter performs worse than the state-passing one, but not dramatically so. This is to be expected, since the vanilla meta-interpreter does not do any context management: in fact, it does not use logical contexts at all.

Next, to gauge the effectiveness in catching bugs, we use, as customary, *mutation analysis* [25], whereby single intentional mistakes are inserted into the system under study. A testing suite is deemed as good as its capability of detecting

those bugs (*killing a mutant*). Most of the literature about mutation analysis revolves around automatic mutant analysis for imperative code, certainly not linear logical specifications of object logics. Therefore, we resort to the *manual* design of a small number of mutants, with all the limitations entailed. Note, however, that this is the approach taken by the testing suite[6] of a leading tool such as *PLT-Redex* [16].

We list in Table 1 a selection of the mutations that we have implemented, together with a categorization, borrowed from the classification of mutations for Prolog-like languages in [38]. We also report the judgment where the mutation occurs.

Clause mutations: deletion of a predicate in the body of a clause, deleting the whole clause if a fact.
Operator mutations: arithmetic and relational operator mutation.
Variable mutations: replacing a variable with an (anonymous) variable and vice versa.
Constant mutations: replacing a constant by a constant (of the same type), or by an (anonymous) variable and vice versa.

Table 1. List of mutations

M1 (eval, C) mutation in the definition of addition
M2 (eval, Cl) added another clause to the definition of subtraction
M3 (eval, O) substitution of $-$ for $*$ in arithmetic definitions
M4 (eval, O) similar to M1 but for conjunction
M5 (exec, V) bug on assignment
M6 (exec, Cl) switch branches in if-then-else
M7 (exec, Cl) deletion of one of the `while` rule
M8 (type, C) wrong output type in rule for addition
M9 (type, C) wrong input type in rule for disjunction

Figure 7 summarizes the outcome of mutation testing, where "found" indicates that a counter-example (cex) has been found and "pass" that the bound has been exhausted. In the first case, we report counter-examples in the last column, after pretty-printing. Since this is accomplished in milliseconds, we omit the precise timing information. Note that counter-examples found by exhaustive search are minimal by construction.

The results seem at first disappointing (3 mutants out of 9 being detected), until we realize that it is not so much a question of our tool failing to kill mutants, but of the above properties being too coarse. Consider for example mutation M3: being a type-preserving operation swap in the evaluation of expressions, this will

[6] https://docs.racket-lang.org/redex/benchmark.html.

certainly not lead to a failure of subject reduction, nor invalidate determinism of evaluation. On the other hand all mutants are easily killed with model-based testing, that is taking as properties soundness ($L \to C$) and completeness ($C \to L$) of the top-level judgments (exec/type) where mutations occur w.r.t. their bug-free versions executed under the vanilla interpreter. This is reported in Fig. 8.

	exec: $C \to L$	exec: $L \to C$	cex
No Mut	pass in 2.40	pass in 6.56	
M1	found in 0.06	pass in 6.45	`w := 0 + 0`
M2	pass in 2.40	found in 0.04	`w := 0 - 1`
M3	found in 0.06	found in 0.06	`w := 0 * 1`
M4	found in 0.06	found in 0.04	`y := tt /\ tt`
M5	found in 0.00	pass in 5.15	`w := 0; w := 1`
M6	pass in 2.34	found in 0.17	`if y = y then {w := 0} else {w := 1}`
M7	found in 0.65	pass in 0.82	`while y = y /\ y = w do {y := tt}`

	type: $C \to L$	type: $L \to C$	cex
No Mut	pass in 0.89	pass in 0.87	
M8	found in 0.03	pass in 0.84	`w := 0 + 0`
M9	found in 0.04	pass in 0.71	`y := tt \/ tt`

Fig. 8. Model-based testing of IMP mutations

6 Related Work and Conclusions

The success of QuickCheck has lead many proof assistants to adopt some form of PBT or more in general of counterexamples search. The system where proofs and disproofs are best integrated is arguably Isabelle/HOL, which offers a combination of random, exhaustive and symbolic testing [5] together with a model finder [2]. A decade later *QuickChick* [42] has been added to Coq as a porting of PBT compatible with the severe constraints of constructive type theory. However, these PBT tools tend to be limited to executable total specifications, while many judgments are partial and/or non-terminating. An exception is the approach in [31], which brings relational PBT to Coq.

As far as the meta-theory of programming languages is concerned, PLT-Redex [16] is an executable DSL for mechanizing semantic models built on top of the programming environment *DrRacket*. Its usefulness has been demonstrated in several impressive case studies [26]. αCheck [10,11] is a close ancestor of the present work, since it is based on a proof-theoretic view of PBT, although it wires in a fixed generation strategy. Moreover, the system goes beyond the confine of classical or intuitionistic logic and embraces *nominal* logic as a way to give a logical account of encoding models where *binding* signatures matter [9].

While substructural logics are a recurring thread in current PL theory (see for example session types and separation logic) and while linear logic programming languages have been extensively used to represent such models [20,44,49],

formal *verification* via linear logic frameworks, as we have mentioned, is still in its infancy. Schürmann et al. [33] have designed $\mathcal{L}^{+}_{\omega}$, a linear meta-logics conservatively extending the meta-theory of Twelf and Pientka et al. [20] have introduced *LINCX*, a linear version of contextual modal type theory to be used within Beluga.

However most case studies, as elegant as they are, are still on paper, viz. type soundness of MiniML with references and cut-elimination for (object) linear logic (LLF [7,33]). Martin's dissertation [32] offers a thorough investigation of the verification of the meta-theory of MiniML with references in Isabelle/HOL's Hybrid library, in several styles, including linear and ordered specifications. A more extensive use of Hybrid, this time on top of Coq, is the recent verification of type soundness of the *proto-Quipper* quantum functional programming language in a Lolli-like specification logic [30].

In this paper we have argued for the extension of property-based testing to substructural logics to overcome the current lack of reasoning tools in the field. We have taken the first step by implementing a PBT system for specifications written in linear Hereditary Harrop formulæ, the language underlying Lolli. We have adapted the FPC architecture to model various generation strategies. We have validated our approach by encoding the meta-theory of IMP and its compilation with a dimple mutation analysis. With all the caution that our setup entails, the experiments show that linear PBT is effective w.r.t. mutations and while it under-performs vanilla PBT over bug-free models, there are immediate avenues for improvement.

There is so much future work that it is almost overwhelming: first item, from the system point of view, is abandoning the meta-interpretation approach, and then a possible integration with Abella. Theoretically, our plan is to extend our framework to richer linear logic languages, featuring ordered logic up to concurrency, as well as supporting different operational semantics, to begin with bottom-up evaluation.

Source code can be found at https://github.com/Tovy97/Towards-Substructural-Property-Based-Testing.

Acknowledgments. We are grateful to Dale Miller for many discussions and in particular for suggesting the use of logical continuations. Thanks also to Jeff Polakow for his comments on a draft version of this paper.

References

1. Baelde, D., et al.: Abella: a system for reasoning about relational specifications. J. Formaliz. Reason. **7**(2), 1–89 (2014)
2. Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic proof and disproof in Isabelle/HOL. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS (LNAI), vol. 6989, pp. 12–27. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24364-6_2
3. Blanco, R., Chihani, Z., Miller, D.: Translating between implicit and explicit versions of proof. In: de Moura, L. (ed.) CADE 2017. LNCS (LNAI), vol. 10395, pp. 255–273. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63046-5_16

4. Blanco, R., Miller, D., Momigliano, A.: Property-based testing via proof reconstruction. In: PPDP, pp. 5:1–5:13. ACM (2019)
5. Bulwahn, L.: The new Quickcheck for Isabelle. In: Hawblitzel, C., Miller, D. (eds.) CPP 2012. LNCS, vol. 7679, pp. 92–108. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35308-6_10
6. Cervesato, I., Hodas, J.S., Pfenning, F.: Efficient resource management for linear logic proof search. Theor. Comput. Sci. **232**(1–2), 133–163 (2000)
7. Cervesato, I., Pfenning, F.: A linear logical framework. In: LICS, pp. 264–275. IEEE Computer Society (1996)
8. Cervesato, I., Pfenning, F., Walker, D., Watkins, K.: A concurrent logical framework ii: examples and applications. Technical report, CMU (2002)
9. Cheney, J.: Toward a general theory of names: binding and scope. In: MERLIN, pp. 33–40. ACM (2005)
10. Cheney, J., Momigliano, A.: αCheck: a mechanized metatheory model checker. Theory Pract. Logic Program. **17**(3), 311–352 (2017)
11. Cheney, J., Momigliano, A., Pessina, M.: Advances in property-based testing for αProlog. In: Aichernig, B.K.K., Furia, C.A.A. (eds.) TAP 2016. LNCS, vol. 9762, pp. 37–56. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41135-4_3
12. Chihani, Z., Miller, D., Renaud, F.: A semantic framework for proof evidence. J. Autom. Reason. **59**(3), 287–330 (2017)
13. Chirimar, J.: Proof theoretic approach to specification languages. Ph.D. thesis. University of Pennsylvania (1995)
14. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), pp. 268–279. ACM (2000)
15. Dubois, C.: Proving ML type soundness within Coq. In: Aagaard, M., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 126–144. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44659-1_9
16. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. The MIT Press, Cambridge (2009)
17. Felty, A.P., Momigliano, A.: Hybrid - a definitional two-level approach to reasoning with higher-order abstract syntax. J. Autom. Reason. **48**(1), 43–105 (2012)
18. Fetscher, B., Claessen, K., Pałka, M., Hughes, J., Findler, R.B.: Making random judgments: automatically generating well-typed terms from the definition of a type-system. In: Vitek, J. (ed.) ESOP 2015. LNCS, vol. 9032, pp. 383–405. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46669-8_16
19. Gacek, A., Miller, D., Nadathur, G.: A two-level logic approach to reasoning about computations. J. Autom. Reason. **49**(2), 241–273 (2012)
20. Georges, A.L., Murawska, A., Otis, S., Pientka, B.: LINCX: a linear logical framework with first-class contexts. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 530–555. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_20
21. Girard, J.-Y.: Linear logic. Theor. Comput. Sci. **50**(1), 1–102 (1987)
22. Hodas, J., Miller, D.: Logic programming in a fragment of intuitionistic linear logic. Inf. Comput. **110**(2), 327–365 (1994)
23. Hritcu, C., et al.: Testing noninterference, quickly. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, pp. 455–468. ACM, New York, NY, USA (2013)
24. Hughes, J.: QuickCheck testing for fun and profit. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 1–32. Springer, Heidelberg (2006). https://doi.org/10.1007/978-3-540-69611-7_1

25. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
26. Klein, C., et al.: 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, pp. 285–296. ACM, New York, NY, USA (2012)
27. Leroy, X.: Mechanized semantics - with applications to program proof and compiler verification. In: Logics and Languages for Reliability and Security, volume 25 of NATO Science for Peace and Security Series - D: Information and Communication Security, pp. 195–224. IOS Press (2010)
28. Liang, C., Miller, D.: Focusing and polarization in linear, intuitionistic, and classical logics. Theor. Comput. Sci. **410**(46), 4747–4768 (2009)
29. López, P., Pfenning, F., Polakow, J., Watkins, K.: Monadic concurrent linear logic programming. In: PPDP, pp. 35–46. ACM (2005)
30. Mahmoud, M.Y., Felty, A.P.: Formalization of metatheory of the quipper quantum programming language in a linear logic. J. Autom. Reason. **63**(4), 967–1002 (2019)
31. Manighetti, M., Miller, D., Momigliano, A.: Two applications of logic programming to Coq. In: TYPES, volume 188 of LIPIcs, pp. 10:1–10:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
32. Martin, A.: 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 (2010). https://ruor.uottawa.ca/handle/10393/19711
33. McCreight, A., Schürmann, C.: A meta linear logical framework. Electron. Notes Theor. Comput. Sci. **199**, 129–147 (2008)
34. McDowell, R., Miller, D.: Reasoning with higher-order abstract syntax in a logical framework. ACM Trans. Comput. Log. **3**(1), 80–136 (2002)
35. Michaylov, S., Pfenning, F.: Natural semantics and some of its meta-theory in Elf. In: Eriksson, L.-H., Hallnäs, L., Schroeder-Heister, P. (eds.) ELP 1991. LNCS, vol. 596, pp. 299–344. Springer, Heidelberg (1992). https://doi.org/10.1007/BFb0013612
36. Miller, D.: Forum: a multiple-conclusion specification logic. Theor. Comput. Sci. **165**(1), 201–232 (1996)
37. Miller, D., Nadathur, G., Pfenning, F., Scedrov, A.: Uniform proofs as a foundation for logic programming. Ann. Pure Appl. Log. **51**, 125–157 (1991)
38. Momigliano, A., Ornaghi, M.: The blame game for property-based testing. In: CILC, volume 2396 of CEUR Workshop Proceedings, pp. 4–13. CEUR-WS.org (2019)
39. Morrisett, J.G., Walker, D., Crary, K., Glew, N.: From system F to typed assembly language. ACM Trans. Program. Lang. Syst. **21**(3), 527–568 (1999)
40. Nigam, V., Miller, D.: Algorithmic specifications in linear logic with subexponentials. In: PPDP, pp. 129–140. ACM (2009)
41. Paoli, F.: Substructural Logics: A Primer. Kluwer, Alphen aan den Rijn (2002)
42. Paraskevopoulou, Z., Hriţcu, C., Dénès, M., Lampropoulos, L., Pierce, B.C.: Foundational property-based testing. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 325–343. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_22
43. Pfenning, F.: Logical frameworks. In: Robinson, A., Voronkov, A. (eds.), Handbook of Automated Reasoning. Elsevier Science Publishers (1999)
44. Pfenning, F., Simmons, R.J.: Substructural operational semantics as ordered logic programming. In: LICS, pp. 101–110. IEEE Computer Society (2009)

45. Pientka, B., Dunfield, J.: Beluga: a framework for programming and reasoning with deductive systems (system description). In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 15–21. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_2
46. Polakow, J.: Linear logic programming with an ordered context. In: PPDP, pp. 68–79. ACM (2000)
47. Reynolds, J.C.: The discoveries of continuations. LISP Symb. Comput. **6**(3–4), 233–248 (1993)
48. Roberson, M., Harries, M., Darga, P.T., Boyapati, C.: Efficient software model checking of soundness of type systems. In: Harris, G.E. (ed.), OOPSLA, pp. 493–504. ACM (2008)
49. Schack-Nielsen, A., Schürmann, C.: Celf – a logical framework for deductive and concurrent systems (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 320–326. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_28
50. Tarau, P.: A combinatorial testing framework for intuitionistic propositional theorem provers. In: Alferes, J.J., Johansson, M. (eds.) PADL 2019. LNCS, vol. 11372, pp. 115–132. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-05998-9_8
51. Wadler, P.: Linear types can change the world! In: Programming Concepts and Methods, p. 561. North-Holland (1990)
52. Yardeni, E., Shapiro, E.: A type system for logic programs. J. Log. Program. **10**(2), 125–153 (1991)