Towards Substructural Property-Based Testing

Marco Mantovani and Alberto Momigliano

Dipartimento di Informatica, Università degli Studi di Milano, Italy

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 [36] have been successfully used to represent/animate deductive systems (*λProlog*) and also to reason (*Twelf, Isabelle*) about them. The methodology of *higher-order abstract syntax* (HOAS) together with parametric-hypothetical judgments [30] 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, as 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 [27], which completely avoids the need to establish intermediate lemmas, as opposed to more standard and labor-intensive treatments [11].

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 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 (and its substructural cousins) can change the world, and in particular it 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/writing said state. In the early 90's this idea was taken up in linear logic programming and linear specification languages, viz., *Lolli* [17], *LLF* [6] and *Forum* [28].

In the following years, given the richness of linear logic and the flexibility of the proof-theoretic foundations of logic programming [31], more sophisticated languages were proposed, with additional features such as order (*Olli* [38]), subexponentials [34], bottom-up evaluation and concurrency (*Lollimon* [23], *Celf* [40]). Each extension required significant ingenuity, since it relied on appropriate notions of canonical forms, resource management, unification etc. At the same time tools for *reasoning* over such substructural specifications did not materialize, as opposed to the development of dedicated intuitionistic proof assistants such as *Abella* [1] and *Beluga* [37]. Meta-reasoning over such frameworks, in fact, asks for formulating appropriate meta-logical tools, which, again, is far from trivial. Schürmann et al. [25] have designed a linear meta-logics and Pientka et al. [16] have introduced a linear version of contextual modal type theory to be used within Beluga, but no implementations have appeared. The case for the concurrent logical framework is particularly striking, where, notwithstanding the wide range of applications [7], the only meta-theoretic analysis available in *Celf* is checking that a program is well-moded.

If 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* [19] (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 [3] gives a proof-theoretic reconstruction of PBT in terms of focusing and Foundational Proof Certificates (FPC) [8], 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 is no verification, but as by now common in mainstream proof assistants, e.g., [4, 35], we may resort to testing not only *in lieu of* proving, but *before* proving.

In fact, the two-level architecture [15] underlying the *Abella* system and the *Hybrid* library [13] seems a good match for the combination of testing and proving over substructural specifications. The approach keeps the meta-logic fixed, while making substructural the specification logic. Indeed, some case studies have been already carried out, the more extensive being the verification of type soundness of quantum programming languages in a Lolli-like specification logic [24].

In this paper we move the first steps in this programme by 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.

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

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

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

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

That is, we are not arguing (yet) that linear PBT is "better" than traditional ones based on 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 does make a difference in terms of drastically simplifying the infrastructure one needs to put in place towards proving the main result: one of the original success stories of linear specifications, namely type preservation of MiniMLR [26,6], 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], you need literally dozens of preliminary lemmas.

In the following, we assume a passing familiarity with linear logic programming and its proof-theory, as introduced in [17,29].

1.1 A motivating example

To preview our methodology, we present a self-contained example where we use PBT as a form of *model-based* testing: we evaluate 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 arrow (in both directions) is linear implication, `x` is multiplicative conjunction (tensor), `&` is additive conjunction and `erase` its unit $\top$.

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

As shown originally in [17], we can encode provability with a predicate `pv` that uses a linear context of propositions `hyp` for assumptions at the left of the turnstile, as shown in the first clause encoding the implication right rule $R_{\rightarrow}$ via the embedded implication `hyp(A) ->pv(B)`. In the left rules, the major premise is consumed by means of the tensor and the new assumptions (re)asserted. Note that in rule $L^{i}_{\rightarrow}$, the context Γ is duplicated via 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 $\mathrm{LJF}_{\rightarrow}$.

Taking inspiration from Tarau's [41], we consider next an optimization where we factor the two left rule for implication in one:

```
...  % similar to before
pvb(C) <- hypb(imp(A,B)) x pvb_imp(A,B) x
          (hypb(B) -> pvb(C)).

pvb_imp(imp(C,D),B) <- (hypb(imp(D,B)) -> pvb(imp(C,D))).
pvb_imp(A,_)        <-  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.

2 The proof-theory of PBT

While PBT originated in a functional programming setting [10], 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 since the seminal work of Miller et al. [31], this means proof-search in a focused sequent calculus, 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.

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

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

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

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

$$\frac{\Xi' : \Delta_I \setminus \Delta_O \vdash G \quad unfold_e(\Xi, \Xi', A, G)}{\Xi : \Delta_I \setminus \Delta_O \vdash A}$$

Fig. 2. FPC presentation of the IO system for second order Lolli

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* [8] (FPC) comes in. For the type-theoretically inclined, FPCs 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* to underline their different functionalities) 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.

2.1 Linear logic programming

Although focusing and FPC apply to most sequent calculi [22], we find convenient to stay close to the traditional semantics of uniform proofs [31]. The language that we adopt here (shown below) is a minor restriction of linear Hereditary Harrop formulæ which underlay the linear logic programming language Lolli [17]. We consider implications with atomic premises only and a first-order term language, thus making universal goals essentially irrelevant. The rationale of this is mirroring our Prolog implementation, but we could easily account for the whole

of Lolli.

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

This language induces an abstract logic programming language in the sense of [31], and as such can be given a uniform proof system with a judgment of the form $\Delta \Rightarrow G$, for which we refer once more to [17]: intuitionistic implication $A \to B$ is considered defined by $!A \multimap B$ and therefore the intuitionistic context is omitted.

However, the uniform proofs discipline does not address the question of how to perform proof search in the presence of linear assumptions, a.k.a. the *resource management problem* [5]. The problem is caused by multiplicative connectives that, under a goal-oriented strategy, require a potentially exponential partitioning of the given linear context.

One solution, based on *lazy* context splitting and known as the *IO system*, was introduced in [17], and further refined in [5]: when we need to split a context (here 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 2 contains a version of the IO system for our language as described by the judgment $\Xi : \Delta_I \setminus \Delta_O \vdash G$, where $\setminus$ is just a suggestive notation to separate input and output context. We will explain the role of Ξ and the predicates $op_e(\Xi, \ldots)$ in the next paragraphs. We overload "," to denote multi-set union and adding a formula to a context. Following on the literature and our implementation, we will signal that a resource has been consumed in the input context by replacing it with the placeholder "$\square$".

The IO system (without certificates) is known to be sound and complete w.r.t. uniform provability: $\Delta_I \setminus \Delta_O \vdash G$ iff $\Delta_I - \Delta_O \Rightarrow G$, where "$-$" is a notion of context difference modulo $\square$ (see [17] for details). 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 [5], this could be addressed by moving to a system with *slack*. However, given the preferred style of our encodings (see Section 3), where additive unit is called only as a last step, this has so far not proved necessary.

Building on the original system and in accord with the FPC approach, each inference rule is augmented 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.

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

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

The first certificate is just a natural number and it used to bound a derivation according to its *height*. Similarly, the second consists of a pair of naturals that bounds the number of clauses used in a derivation (*size*): typically n will be input and m output, so the *size* will be $n - m$. 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 provide random generation bounded by the height of the derivation; pairing is a simple, but surprisingly effective combinator [2].

Each certificate format is accompanied by the implementation of the experts 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.

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

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

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

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 $\mathbf{1}$, 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 does it decrease 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 over the grounding of the program. This very expert is also the hook for implementing random data generation via random back-chaining, where we replace chronological with randomized 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 [14]: 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 [3], more sophisticated FPCs capture other features of PBT, such as δ-debugging (shrinking) and bug-provenance, and will not be repeated here.

We are now ready to account for the soundness property from the example in Section 1.1. 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 implication formulæ, which we use as a generator. Testing the property becomes 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 [3].

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

3 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*). Here we follow Leroy's account [21], 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`	*type of integers*
|	`tbool`	*type of Boolean's*

`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:

$\sigma \vdash m \Downarrow v$ 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;
$\Gamma \vdash m : \tau$ well-typed expressions and $v : \tau$ well-typed values;
$\Gamma \vdash c$ well-typed commands and $\Gamma : \sigma$ well-typed states;

3.1 On linear encodings

In traditional accounts, a state σ is a (finite) map between variables and values. Linear logic takes a "distributed" view and represent 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 expressions and 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): 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. CPS fixes a given evaluation order, which is crucial 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, at least in our setup, it leads 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 [6][2].

Mixing additive and multiplicative connectives needs a more sophisticated resource management system; this is a concern, given the efficiency requirements that testing brings to the table — it is not called "QuickCheck" for nothing. We

[2] This can be circumvented by switching to a more expressive logic, either by internalizing the continuation as an ordered context [38] or by changing representation via forward chaining (*destination-passing style*) [23].

therefore use the notion of *logical* continuation advocated by Chirimar [9], 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 encoding:

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

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \quad \sigma \vdash e_2 \Downarrow v_2 \quad \text{plus } \mathrm{v}_1\ \mathrm{v}_2\ \mathrm{v}}{\sigma \vdash e_1 + e_2 \Downarrow v}\ e/p$$

```
eval(v(X),N,K)              <- var(X,N) x (var(X,N) -> K).
eval(i(N),vi(N),K)          <- K.
eval(plus(E1,E2),vi(V),K) <-
   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 $\ulcorner\sigma\urcorner \Rightarrow \texttt{eval}(\ulcorner m\urcorner, \ulcorner v\urcorner, \top)$ has a uniform proof, where the initial continuation $\top$ cleans up σ 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 an additive encoding as well:

```
ev(v(X),V)              <- var(X,V) x erase.
ev(i(N),vi(N))          <- erase.
ev(plus(E1,E2),vi(V)) <- 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 encoding:

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

```
ceval(asn(X,E),K) <- eval(E,V,(var(X,_) x (var(X,V) -> 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 via & 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) <- 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:

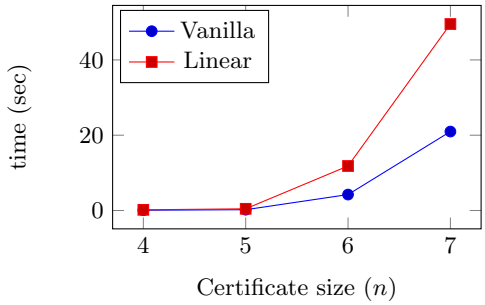
srv subject reduction for evaluation: $\Gamma \vdash m : \tau \longrightarrow \sigma \vdash m \Downarrow v \longrightarrow \Gamma : \sigma \longrightarrow v : \tau$;
dtx determinism of execution: $(\sigma, c) \Downarrow \sigma_1 \longrightarrow (\sigma, c) \Downarrow \sigma_2 \longrightarrow \sigma_1 \approx \sigma_2$;
srx preservation of state under 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. We have added a simple type discipline for the assembly language in the spirit of TAL [33] 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.

3.2 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 [42] with a very minor massage. Compare this with the amount of ingenuity poured in writing generators in [18] or with the model-checking techniques of [39]. 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.

To establish a fair baseline, we have also implemented a "vanilla" version of our benchmarks, that is *state-passing* ones, driven by a FPC-lead vanilla meta-interpreter. We have run the experiments on a laptop with an Intel i7–7500U CPU and 16GB 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. We list here only a few experiments with no pretense of completeness. In particular, we choose a fixed exhaustive



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

Fig. 3. 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. 4. Mutation testing

generation strategy (*size*), then pair it with *height* for mutation analysis. We use consistently certain bounds that experimentally have shown to be effective in generating enough interesting data.

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. 3 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. 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 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* [20], 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[3]

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

of a comparable tool such as *PLT-Redex* [12]. The mutations are described in Appendix A.

Table 4 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 cex 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 loose. Consider for example mutation M3: being a type-preserving operation swap in the evaluation of expressions, this will 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 and completeness of the (top-level) judgments where mutations occur w.r.t. their bug-free versions executed under the vanilla interpreter. This is reported in Table 5.

	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. 5. Model-based testing of IMP mutations

4 Conclusions

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 com-

pilation, with a rudimentary mutation analysis. With all the caution that our setup entails, results are encouraging.

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.

A Appendix

List of mutations We describe a selection of the mutations that we have implemented, together with a categorization, borrowed from the classification of mutations for Prolog-like languages in [32]. 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.

M1 (eval, C) tag 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.

References

1. D. Baelde, K. Chaudhuri, A. Gacek, D. Miller, G. Nadathur, A. Tiu, and Y. Wang. Abella: A system for reasoning about relational specifications. *J. Formaliz. Reason.*, 7(2):1–89, 2014.

2. R. Blanco, Z. Chihani, and D. Miller. Translating between implicit and explicit versions of proof. In L. de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction*, volume 10395 of *LNCS*, pages 255–273. Springer, 2017.
3. R. Blanco, D. Miller, and A. Momigliano. Property-based testing via proof reconstruction. In *PPDP*, pages 5:1–5:13. ACM, 2019.
4. L. Bulwahn. The new quickcheck for isabelle - random, exhaustive and symbolic testing under one roof. In C. Hawblitzel and D. Miller, editors, *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, volume 7679 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2012.
5. I. Cervesato, J. S. Hodas, and F. Pfenning. Efficient resource management for linear logic proof search. *Theor. Comput. Sci.*, 232(1-2):133–163, 2000.
6. I. Cervesato and F. Pfenning. A linear logical framework. *Inf. Comput.*, 179(1):19–75, 2002.
7. I. Cervesato, F. Pfenning, D. Walker, and K. Watkins. A concurrent logical framework ii: Examples and applications. Technical report, CMU, 2002.
8. Z. Chihani, D. Miller, and F. Renaud. A semantic framework for proof evidence. *J. of Automated Reasoning*, 59(3):287–330, 2017.
9. J. Chirimar. *Proof Theoretic Approach to Specification Languages*. PhD thesis, University of Pennsylvania, Feb. 1995.
10. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 2000 ACM SIGPLAN International Conference on Functional Programming (ICFP 2000)*, pages 268–279. ACM, 2000.
11. C. Dubois. Proving ML type soundness within coq. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings*, volume 1869 of *Lecture Notes in Computer Science*, pages 126–144. Springer, 2000.
12. M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.
13. A. P. Felty and A. Momigliano. Hybrid - A definitional two-level approach to reasoning with higher-order abstract syntax. *J. Autom. Reasoning*, 48(1):43–105, 2012.
14. B. Fetscher, K. Claessen, M. H. Palka, J. Hughes, and R. B. Findler. Making random judgments: Automatically generating well-typed terms from the definition of a type-system. In *ESOP*, volume 9032 of *LNCS*, pages 383–405. Springer, 2015.
15. A. Gacek, D. Miller, and G. Nadathur. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning*, 49(2):241–273, 2012.
16. A. L. Georges, A. Murawska, S. Otis, and B. Pientka. LINCX: A linear logical framework with first-class contexts. In *ESOP*, volume 10201 of *Lecture Notes in Computer Science*, pages 530–555. Springer, 2017.
17. J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.
18. C. Hritcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos. Testing noninterference, quickly. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 455–468, New York, NY, USA, 2013. ACM.
19. J. Hughes. Quickcheck testing for fun and profit. In M. Hanus, editor, *PADL 2007*, volume 4354 of *LNCS*, pages 1–32. Springer, 2007.
20. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, Sept. 2011.

21. X. Leroy. 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*, pages 195–224. IOS Press, 2010.
22. C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theor. Comput. Sci.*, 410(46):4747–4768, 2009.
23. P. López, F. Pfenning, J. Polakow, and K. Watkins. Monadic concurrent linear logic programming. In *PPDP*, pages 35–46. ACM, 2005.
24. M. Y. Mahmoud and A. P. Felty. Formalization of metatheory of the quipper quantum programming language in a linear logic. *J. Autom. Reason.*, 63(4):967–1002, 2019.
25. A. McCreight and C. Schürmann. A meta linear logical framework. *Electron. Notes Theor. Comput. Sci.*, 199:129–147, 2008.
26. R. McDowell and D. Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic*, 3(1):80–136, 2002.
27. S. Michaylov and F. Pfenning. Natural semantics and some of its meta-theory in elf. In L. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Extensions of Logic Programming, Second International Workshop, ELP'91, Stockholm, Sweden, January 27-29, 1991, Proceedings*, volume 596 of *Lecture Notes in Computer Science*, pages 299–344. Springer, 1991.
28. D. Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, Sept. 1996.
29. D. Miller. Overview of linear logic programming. In T. Ehrhard, J.-Y. Girard, P. Ruet, and P. Scott, editors, *Linear Logic in Computer Science*, volume 316 of *London Mathematical Society Lecture Note*, pages 119 – 150. Cambridge University Press, 2004.
30. D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.
31. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
32. A. Momigliano and M. Ornaghi. The blame game for property-based testing. In *CILC*, volume 2396 of *CEUR Workshop Proceedings*, pages 4–13. CEUR-WS.org, 2019.
33. J. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.
34. V. Nigam and D. Miller. Algorithmic specifications in linear logic with subexponentials. In *PPDP*, pages 129–140. ACM, 2009.
35. Z. Paraskevopoulou, C. Hritcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. In C. Urban and X. Zhang, editors, *ITP 2015*, volume 9236 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2015.
36. F. Pfenning. *Logical Frameworks*, page 1063–1147. Elsevier Science Publishers B. V., NLD, 2001.
37. B. Pientka and J. Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 15–21. Springer, 2010.
38. J. Polakow. Linear logic programming with an ordered context. In *PPDP*, pages 68–79. ACM, 2000.
39. M. Roberson, M. Harries, P. T. Darga, and C. Boyapati. Efficient software model checking of soundness of type systems. In G. E. Harris, editor, *OOPSLA*, pages 493–504. ACM, 2008.

40. A. Schack-Nielsen and C. Schürmann. Celf - A logical framework for deductive and concurrent systems (system description). In *IJCAR*, volume 5195 of *Lecture Notes in Computer Science*, pages 320–326. Springer, 2008.
41. P. Tarau. A combinatorial testing framework for intuitionistic propositional theorem provers. In *PADL*, volume 11372 of *Lecture Notes in Computer Science*, pages 115–132. Springer, 2019.
42. E. Yardeni and E. Shapiro. A type system for logic programs. *The Journal of Logic Programming*, 10(2):125–153, 1991.