Integrating Functional Logic Programming with Constraint Solving for Random Generation of Structured Data

Li-yao Xia, supervised by Cătălin Hriţcu (INRIA Paris)

August 21, 2015

General context

Property-based testing is a principled testing technique, made popular by the tool QuickCheck for Haskell [3]. Users specify executable properties that can be checked on a large amount of random test cases. The confidence in the correctness of such properties increases as more tests are run. When a counter-example is found, it is shrinked in order to reduce the noise originating from random generation.

Many functions assume their inputs to be structured in some way; for instance, an implementation of search trees would expose functions expecting well-formed search trees. Thus, properties generally have the shape $\forall x : t, P(x) \Rightarrow Q(x)$, where the main property Q is guarded by a precondition P. A naive way of testing such a property is to generate and filter, generating inputs of type t and only applying Q on those inputs that happen to satisfy P. However, this wastes a lot of effort if P is sparse—i.e., the proportion of satisfying values is small. Writing a specialized random *generator* that produces only values satisfying P greatly improves testing in such cases. When P is a complex and evolving property, an efficient generator becomes challenging to write, and to keep in sync with a boolean predicate–i.e., a *checker*—for property P. The checker is often useful to keep in the codebase as part of input validation, or in the specification of an invariant $\forall x, P(x) \Rightarrow P(f(x))$.

Research problem

The issue we aim to address is that a checker for a property P and a generator of values satisfying P often have symmetric implementations. Lampropoulos et al. [6] have recently proposed to exploit this similarity by designing a domain-specific language, named Luck, for generators written as decorated predicates. Luck is strongly influenced by functional logic programming [4, 8, 5], as it is designed around the principle of *narrowing* [1, 7, 9]: in Luck, a generator for P solves the equation $P(x) = \mathit{True}$ by incrementally instantiating the unknown x so that the evaluation of $P(x)$ progresses. Sometimes, it is not practical to instantiate unknowns randomly. Luck allows to locally generate and solve constraints in order to better control the effect of narrowing.

The semantics of Luck has been formalized, leading to the proofs of two important properties: a generator produces only values which satisfy the underlying predicate (*soundness*) and conversely any satisfying value can be generated by it (*completeness*). However, I have identified some serious issues with the semantics and the implementation affecting expressiveness and performance, which constitute the subject of this report.

Contribution

I have significantly contributed to the design and implementation of Luck, the first language for generators to put together instantiation and constraint solving while providing control over the

obtained probability distribution.

I have first helped with the implementation and evaluation of a prototype interpreter for Luck [6], with contributions to both the front end to the back end. This contribution amounts to over 2000 lines of code, or about 20% of the codebase. I have had the opportunity of writing about a part of the translation of the surface language of Luck to its interpretable core language in a POPL submission [6], in particular about the desugaring of nested patterns (section 4).

Becoming more familiar with the project made me aware of the issues I mention. In order to address them, I propose a new semantics for Luck targeted at improving both expressiveness and efficiency (sections 2 and 3). My new semantics is closer to those of more standard functional languages. As opposed to the original presentation which starts with simple generators and progressively extends the language with more functional features, I try to upgrade the semantics of a regular functional program to support narrowing and constraint solving. An implementation of this new semantics is work in progress.

Arguments supporting its validity

The new semantics cleanly isolates different aspects in a more modular presentation: narrowing, constraint solving, backtracking. Existing programs can be interpreted with the new semantics, and some can be refactored to a more natural programming style. Unfortunately I haven't managed to finish implementing an interpreter for this new semantics yet, so practical performance evaluation is future work that I hope to conclude in the upcoming weeks.

On the theory side I have identified the formal soundness and completeness properties I expect the new semantics to satisfy (subsection 2.6), as well as some of the auxiliary lemmas necessary for proving these properties, however, complete proofs of these properties remain future work.

Conclusion and future work

Luck shows that instantiation and constraint solving can be put together into an effective and usable language for generators, but, more work on this is needed.

Due to the close deadline for the POPL submission, the scope of the changes I discuss here could not have been integrated in time, hence I did not immediately have the opportunity of exploring the ideas I've had while working on the more practical aspects of Luck—implementation and evaluation. Thus the main preliminary results remain to be proved.

On the longer term, generalizing the constraint solving component of Luck, hopefully with a more flexible interface, would enable support for existing off-the-shelf constraint solvers. We also wish to produce a certified implementation which compiles Luck programs efficiently to a general-purpose programming language such as Haskell.

```
case (x, x) of
| (a, b) -> [|a| 0 < a && a < 10000 |]
          && [|b| 0 < b && b < 10000 |]
end
```

Figure 1: With the original semantics this program would instantiate `a` and `b` independently.

1 Overview of Luck

Luck programs are predicates—functions returning a boolean value—decorated with annotations to guide the derivation of a generator. The starting idea is narrowing, originating from functional logic programming: to generate a value satisfying a predicate $f : X \to Bool$, we may try to incrementally instantiate an unknown x in order to make progress in the computation of $f(x)$, while keeping in mind the goals of satisfying f and obtaining a value x which is random. Pattern matching naturally specifies values with which x may be sampled from.

On the other hand, choosing a random integer when encountering a condition $0 < x$ is overly eager: for instance to solve a conjunction: `f x = 0 < x && x < 42`, if we sampled a positive value for `x` when encountering the first condition, it would have to be from an arbitrary distribution which is likely to produce values which are either too small, unfairly skewing the distribution, or too large, making the condition fail often. In this situation, Luck provides *delay brackets*, such that `f x = [|x| 0 < x && x < 42 |]` prevents the instantiation of `x` during the evaluation of the inner expression, and constraints are emitted instead. A constraint solver can then tell that `x` should lie between 0 and 42, and instantiate it with that knowledge.

Delay brackets are part of the surface language (Outer) Luck, which is translated to Core Luck before being interpreted. Core Luck is a minimal language where each constructs have simple meanings, and Outer Luck constructs thus correspond to a combination Core constructs. In particular, whereas the Outer Luck comparison operators are eager to instantiate variables unless they are protected by delay brackets, Core Luck operators only produce constraints and there are explicit constructs to use these constraints to refine the domains of unknown values, and to instantiate them within these domains.

1.1 Issues

1.1.1 Lack of sharing

The original semantics reduces expressions in the context of an environment ρ which maps variables to sets of values, this environment itself can be seen as a representation of the cartesian product of these sets, i.e., the set of maps ρ_0 such that $\rho_0(x) \in \rho(x)$ for all x in the support of ρ. However, with algebraic datatypes such a representation can not accurately capture the dependency between variables. In Fig. 1, assuming `x` starts bound to $\mathbb{Z}$, pattern matching introduces two variables `a` and `b` also bound to $\mathbb{Z}$ but the information that they should refer to the same value as `x` is lost! Consequently, evaluating the body of the case expression instantiates them separately and a necessary unification step will cause the program to fail. Furthermore, with deeply nested structures such unification in itself is expensive, for that reason the current implementation does it only partially and is thus unsound: generators may produce values which do not satisfy the predicate.

This motivates semantics which capture the sharing of data between variables, which are already standard in narrowing based or lazy functional languages.

1.1.2 Boolean-only semantics

The main parts of the semantics are defined only for boolean expressions, describing the tranformation of the environment when trying to evaluate the expression to *True*. Of course, the definition of a predicate often depends on non-boolean computations, and we must rely on a "standard interpretation" to obtain the values of case scrutinees and function arguments, which also assumes that other values are "sufficiently defined" in order for evaluation to succeed. For example, `head x` has no meaning if the outermost constructor of `x` is still unknown.

One workaround is to use tautologies as a way to force the instantiation of a variable sufficiently before applying a non-boolean function to it. But that defeats the very purpose for which Luck was designed: to reduce redundancy.

We wish to define the non-deterministic evaluation of an arbitrary expression, and although we could tie it to the original semantics, it is odd to have two ways of evaluating expressions. The main benefit of a special definition for boolean expressions is that the result guides the evaluation by eliminating *False* branches without them affecting the probability of (immediate) success. We argue that the branches which could be removed that way only cost us a single additional step of backtracking in our semantics.

2 New semantics for Core Luck

Luck programs are expressions with two interpretations. In a *fully defined* context, expressions evaluate as standard functional programs, whereas in a *partially defined* context, the interpreter tries to refine the context by instantiating unknowns with randomly generated values in order to obtain a successful execution.

2.1 Syntax and notations

Syntax of expressions We start with a standard lambda calculus with integers and datatypes. Expressions are identified up to α-renaming. Variables are bound by lambdas, case patterns, and let constructs.

- Variables: $f, x, y \in \mathcal{X}$
- Integer literals: $\mathbf{m}, \mathbf{n} \in \mathbb{Z}$
- Constructors: $C, D \in \mathcal{C}$
- Expressions (in A-normal form):

$$
\begin{aligned}
e ::= \; & x \\
\mid \; & \mathbf{n} \\
\mid \; & C(x_1, ..., x_n) \\
\mid \; & \lambda x.e \\
\mid \; & \mu f x.e \\
\mid \; & x = y \mid x \neq y \mid x \leq y \mid x < y \mid x \geq y \mid x > y \\
\mid \; & \text{let}\, x = e' \,\text{in}\, e \\
\mid \; & f \; y \\
\mid \; & \text{case}\, x \; (..., C(x_1, ..., x_n) \rightarrow e, ...)
\end{aligned}
$$

Execution contexts The context in which an expression is evaluated contains an *environment* binding variables to *heap indices*, and a *heap* binding these indices to *heap values*. Heaps allow the semantics to capture sharing of data between variables, which is useful in general to reduce memory usage, but it is especially important here in order to give a sound semantics that also admits an efficient implementation.

- Environments: $\rho \in (\mathcal{X} \rightharpoonup \mathcal{I})$
- Heap indices: $\alpha, \beta \in \mathcal{I}$
- Heaps: $h \in (\mathcal{I} \rightharpoonup \mathcal{H})$
- Heap values: $\eta \in \mathcal{H}$

Heap values are integers, applied constructors, and closures.

$$\eta ::= n \mid C(\alpha_1, \ldots, \alpha_n) \mid (\rho, \lambda x.e)$$

The *support* of a partial map m (heap h or environment ρ) is denoted by $\mathrm{supp}(m)$. We use the following standard operations on partial maps:

Update When $\alpha \in \mathrm{supp}(m)$, it can be remapped to another value η: $m[\alpha \mapsto \eta]$.

Union When two maps m and m' have disjoint supports, we denote their union by $m \cdot m'$. In particular, if $\alpha \notin h$, we may extend h with a value η, the result is denoted by $h \cdot \{\alpha \mapsto \eta\}$.

Restriction When $A \subseteq \mathrm{supp}(m)$, the restriction of m to A is denoted $m\restriction_A$.

Well-formedness The triple e, h, ρ is *well-scoped* if the set of free variables of e is a subset of the support of ρ and the image of ρ is a subset of the support of h.

A heap h is *closed* if for any value of the form $C(\alpha_1, \ldots, \alpha_n)$ in its image, $\alpha_1, \ldots, \alpha_n$ are in its support, and for any closure $(\rho, \lambda x.e)$, the triple $\lambda x.e, h, \rho$ is well-scoped. In other words, the heap does not contain any undefined reference.

The triple e, h, ρ is *well-formed* if it is well-scoped and h is closed.

2.2 Evaluation rules

We present a non-deterministic big-step evaluation relation $e, h, \rho \Downarrow \alpha, h', \kappa$ defining the semantics of an expression e in the context of an environment ρ and heap h, to be a heap index α in a new heap h', with the output of a constraint κ whose definition and structure shall be introduced in a later subsection. The fragment of this subsection is deterministic, but the relation will become non-deterministic with the introduction of instantiations of unknowns.

$$\frac{}{x, h, \rho \Downarrow \rho(x), h, \epsilon} \quad \text{(Variable)}$$

Values are added to the heap before being returned (α is a *fresh* address, not appearing in h).

$$\frac{}{\mathbf{n}, h, \rho \Downarrow \alpha, h \cdot \{\alpha \mapsto \mathbf{n}\}, \epsilon} \quad \text{(Literal)}$$

$$\frac{}{(\lambda x.e), h, \rho \Downarrow \alpha, h \cdot \{\alpha \mapsto (\rho, \lambda x.e)\}, \epsilon} \quad \text{(Lambda)}$$

$$\frac{}{(\mu f x.e), h, \rho \Downarrow \alpha, h \cdot \{\alpha \mapsto (\rho \cdot \{f \mapsto \alpha\}, \lambda x.e)\}, \epsilon} \quad \text{(Mu)}$$

$$\frac{}{C(x_1,\ldots,x_k), h, \rho \Downarrow \alpha, h \cdot \{\alpha \mapsto C(\rho(x_1),\ldots,\rho(x_k))\}, \epsilon} \quad \text{(Constructor)}$$

For operators, $\triangle \in \{=, \neq, \leq, <, \geq, >\}$,

$$\frac{h(\rho(x)) = \mathbf{m} \qquad h(\rho(y)) = \mathbf{n}}{x \triangle y, h, \rho \Downarrow \alpha, h \cdot \{\alpha \mapsto \mathbb{B}(\mathbf{m} \triangle \mathbf{n})\}, \epsilon} \quad (\triangle\text{-Operation})$$

where $\mathbb{B}$ maps truth values to the corresponding constructors:

$$\mathbb{B}(\mathit{true}) = \mathit{True} \in \mathcal{C},$$
$$\mathbb{B}(\mathit{false}) = \mathit{False} \in \mathcal{C}.$$

$$\frac{h(\rho(f)) = (\rho', \lambda x.e) \qquad e, h, \rho' \cdot \{x \mapsto \rho(y)\} \Downarrow \alpha, h', \kappa}{f\ y, h, \rho \Downarrow \alpha, h', \kappa} \quad \text{(Function)}$$

Let-bindings allow to sequence computations.

$$\frac{e, h, \rho \Downarrow \alpha, h', \kappa \qquad e', h', \rho \cdot \{x \mapsto \alpha\} \Downarrow \beta, h'', \kappa'}{(\mathrm{let}\, x = e\, \mathrm{in}\, e'), h, \rho \Downarrow \beta, h'', \kappa \wedge \kappa'} \quad \text{(Let)}$$

$$\frac{h(\rho(x)) = C(\alpha_1,\ldots,\alpha_n) \qquad e, h, \rho \cdot \{x_1 \mapsto \alpha_1,\ldots,x_n \mapsto \alpha_n\} \Downarrow \alpha, h', \kappa}{(\mathrm{case}\, x\ (\ldots, C(x_1,\ldots,x_n) \to e, \ldots)), h, \rho \Downarrow \alpha, h', \kappa} \quad \text{(Case)}$$

2.3 Unknowns and instantiation

The goal is to interpret a boolean function $f : X \to \mathit{Bool}$ as a generator $g : \mathit{Gen}\ X$, which can be seen as *solving* the equation $f(x) = \mathit{True}$ for the unknown x. The natural thing to do is to try to reduce $f(x)$ and proceed by case analysis when it pattern matches on x for instance.

To interpret expressions as generators, we allow the heap to contain *unknowns* and generate data by gradually building satisfying instantiations for these unknowns. Formally, we extend the definition of heap values to include unknowns, denoted by $\star$:

$$\eta ::= \ldots \mid \star$$

We leave the treatment of integers for the next subsection and focus our attention on the language fragment with datatypes. When pattern-matching on an unknown $\star$, we *non-deterministically* choose one of the patterns and partially instantiate the value of the scrutinee so as to satisfy this pattern. We leave the rest of the scrutinee's value uninstantiated by introducing more unknowns as its subterms. Formally, this is captured by the following additional evaluation rule for case that triggers when the scrutinee is an unknown:

$$\frac{h(\rho(x)) = \star \qquad e, h', \rho' \Downarrow \beta, h'', \kappa}{(\mathrm{case}\, x\ (\ldots, C(x_1,\ldots,x_n) \to e, \ldots)), h, \rho \Downarrow \beta, h'', \kappa} \quad \text{(CaseNarrow)}$$

where

$$h' := h[\rho(x) \mapsto C(\alpha_1,\ldots,\alpha_n)] \cdot \{\alpha_1 \mapsto \star,\ldots,\alpha_n \mapsto \star\},$$
$$\rho' := \rho \cdot \{x_1 \mapsto \alpha_1,\ldots,x_n \mapsto \alpha_n\}.$$

2.4 Constraint solving for integers

Before instantiating a variable, it may be necessary to first reduce the set of values it could take. Pattern matching naturally specifies a finite set of constructors. For integers, instantiating the operands as soon as a primitive operation is encountered is too eager a strategy. If x is bound to an unknown, a simple comparison $0 < x$ does not give a meaningful distribution to sample x from, and it is likely to break later constraints.

Instead, comparisons emit constraints between unknowns without instantiating them to a single integer value. We rely on a constraint solver to refine the sets of values from which we sample unknowns. Instantiating an unbounded integer will still be allowed, to remain consistent with the idea that an unbounded unknown $\star$ represents any value. It shall be an implicit expectation that unknowns should be bounded before being instantiated in order to obtain more adequate samples.

The constraint solver is the most variable component of this language, thus to avoid tying ourselves to a single implementation, we shall give a modular presentation, by first describing a generic interface. We shall detail our implementation, but proofs will only rely on the interface requirements.

The heap now may contain *bounded unknowns*, which are sets of integers $\mathfrak{z} \subseteq \mathbb{Z}$. Only some sets may be representable.

$$\eta ::= \ldots \mid \star \mid \mathfrak{z}$$

A *fully defined heap* is a heap containing no unknowns ($\star$ or $\mathfrak{z}$), otherwise it is *partially defined*.

Heap ordering We define a strict ordering $\prec$ on heap elements so that: $\eta \prec \star$ for all $\eta \neq \star$; $\mathfrak{z} \prec \mathfrak{z}'$ if $\mathfrak{z} \subset \mathfrak{z}'$; and $\mathbf{n} \prec \mathfrak{z}$ if $\mathbf{n} \in \mathfrak{z}$.

Let h and h' be two heaps, let $A \subseteq \mathcal{I}$ be a set of heap indices. We say that h' is an *instantiation* of h on A, denoted $h' \sqsubseteq_A h$ if $A \subseteq \mathrm{supp}(h) \cap \mathrm{supp}(h')$ and for all $\alpha \in A$, $h'(\alpha) \preccurlyeq h(\alpha)$.

We say simply that h' is an instantiation of h, denoted $h' \sqsubseteq h$, if $h' \sqsubseteq_{\mathrm{supp}(h)} h$.

This ordering is quite strong, as it requires the indices of heap elements to match exactly, but weakening it to allow some amount of renaming would be problematic because external environments ρ and constraints κ contain references to it.

Constraints Naturally, the constraint solver comes with a set of *constraints* $\mathcal{K}$, containing at least *basic constraints* of the form $\alpha \triangle \beta$, for $\alpha, \beta \in \mathcal{I}$, and $\triangle \in \{=, \neq, <, \leq, >, \geq\}$.

Operations The set of constraints is equipped with a *conjunction* operation and an empty constraint, making a monoid: $\wedge \in \mathcal{K} \times \mathcal{K} \to \mathcal{K}$, $\epsilon \in \mathcal{K}$, as well as a *refining* partial function $\phi \in (\mathcal{I} \rightharpoonup \mathcal{H}) \times \mathcal{K} \rightharpoonup (\mathcal{I} \rightharpoonup \mathcal{H}) \times \mathcal{K}$. Informally, ϕ *refines* the sets in the heap by removing values which do not satisfy the given constraint, and simplifies the constraint.

Constraint satisfaction A heap h *satisfies* a basic constraint $\alpha \triangle \beta$ if $h(\alpha) = \mathbf{m}$, $h(\beta) = \mathbf{n}$, and $\mathbf{m} \triangle \mathbf{n}$ holds.

A heap h is *consistent* with κ if there exists some instantiation of h which satisfies κ.

The constraint solver should extend the satisfaction relation such that:

1. h satisfies $\kappa \wedge \kappa'$ if and only if h satisfies both κ and κ';
2. if partial function ϕ is not defined for input pair (h, κ), then h is not consistent with κ;[1]

[1] The converse does not have to hold, and it is indeed computationally expensive in general.

3. if $\phi(h, \kappa) = (h', \kappa')$, then:
   (a) h' is an instantiation of h (ϕ is *decreasing*),
   (b) any instantiation of h' satisfying κ' also satisfies κ (ϕ is *sound*), and
   (c) any instantiation of h satisfying κ is an instantiation of h' satisfying κ' (ϕ is *complete*).

One implementation As a concrete instance, our implementation uses the following:

1. Bounded unknowns $\mathfrak{z}$ are (finite) unions of intervals.
2. Constraints κ are lists of basic constraints $\alpha \vartriangle \beta$, conjunction is concatenation.
3. A basic constraint $\alpha \vartriangle \beta$ is applied to the heap h by removing from $h(\alpha)$ any value $\mathbf{m} \in h(\alpha)$ such that there is no $\mathbf{n} \in h(\beta)$ with $\mathbf{m} \vartriangle \mathbf{n}$, and similarly for $h(\beta)$. Furthermore, if for all $\mathbf{m} \in h(\alpha)$ and $\mathbf{n} \in h(\beta)$, $\mathbf{m} \vartriangle \mathbf{n}$ holds, then the basic constraint can be dropped. $\phi(h, \kappa)$ applies all basic constraints in its argument to the heap until a fixed point is reached (here it turns out non-termination implies inconsistency, and can thus be denoted by undefinedness).

This simple constraint solver may be extended in the future to support arithmetic operations by keeping symbolic representations of operations on unknowns on the heap and extending the operation ϕ to handle such values.

2.5 Rules for constraint solving

We introduce two new constructs in the core language: constraints are applied to the heap using refine and integer unknowns are instantiated with inst.

$$\begin{aligned} e ::= & \ldots \\ & \mid \ \text{refine}(e) \\ & \mid \ \text{inst}(x) \end{aligned}$$

Let us look at constraints in the previous rules. Those which directly return a value emit the empty constraint—(Variable), (Literal), (Lambda), (Mu), (Constructor). Other rules just propagate constraints—(Function), (Case), (CaseNarrow)—and since (Let) sequences computations, it joins the emitted constraints. By not including constraints on the left hand side of the $\Downarrow$ relation, an expression cannot access constraints emitted before its evaluation; that limits the scope of constraints, which can only be applied by an enclosing refine.

We replace the (△-Operation) rule by two complementary rules. A rule is chosen non-deterministically to either assert that a condition is true or that it is false.

$$\frac{}{x \vartriangle y, h, \rho \Downarrow \alpha, h' \cdot \{\alpha \mapsto \mathit{True}\}, (\rho(x) \vartriangle \rho(y))} \quad (\vartriangle\text{-True-Operation})$$

$$\frac{}{x \vartriangle y, h, \rho \Downarrow \alpha, h' \cdot \{\alpha \mapsto \mathit{False}\}, (\rho(x) \blacktriangledown \rho(y))} \quad (\vartriangle\text{-False-Operation})$$

where $\blacktriangledown$ is the negation of $\vartriangle$: $\{\vartriangle, \blacktriangledown\} \in \{\{=, \neq\}, \{\leq, >\}, \{\geq, <\}\}$.

The (Refine) rule modifies the heap to exclude instantiations which do not satisfy the constraints, and simplifies the constraints emitted by the enclosed computation.

$$\frac{e, h, \rho \Downarrow \alpha, h', \kappa \qquad \phi(h', \kappa) = (h'_0, \kappa_0)}{\text{refine}(e), h, \rho \Downarrow \alpha, h'_0, \kappa_0} \quad (\text{Refine})$$

The (Instantiate) rule replaces an unknown with a non-deterministically chosen integer. A small abuse of notation handles the unbounded unknown, identifying $\star$ with the whole set $\mathbb{Z}$ when a set $\mathfrak{z}$ is expected.

$$\frac{h(\rho(x)) = \mathfrak{z} \qquad \mathbf{n} \in \mathfrak{z}}{\mathrm{inst}(x), h, \rho \Downarrow \rho(x), h[\rho(x) \mapsto \mathbf{n}], \epsilon} \qquad \text{(Instantiate)}$$

Another rule for inst is used in order to handle already instantiated variables transparently.

$$\frac{h(\rho(x)) = \mathbf{n}}{\mathrm{inst}(x), h, \rho \Downarrow \rho(x), h, \epsilon} \qquad \text{(Already instantiated)}$$

2.6 Theoretical Properties

Reachability Our reduction relation clutters the heap with many uninteresting elements. The notion of reachable indices is useful to focus our attention the part of the heap that is actually used. Given a closed heap h and a subset A of its support, the set of heap indices which are *reachable* from A in h is the smallest subset $R(A, h)$ of the support of h such that $A \subseteq R(A, h)$ and the following holds:

1. if $\alpha \in R(A, h)$ and $h(\alpha) = C(\alpha_1, \ldots, \alpha_n)$, then $\alpha_1, \ldots, \alpha_n \in R(A, h)$;
2. if $\alpha \in R(A, h)$ and $h(\alpha) = (\rho, \lambda x.e)$, then $\rho(\mathrm{supp}(\rho)) \subseteq \mathrm{supp}(h)$.

Lemmas and conjectures Clearly, the computations either extend the heap with new bindings, or replace unknowns with smaller (i.e., more defined) values.

Lemma 1 (Decrease)**.** *If $e, h, \rho \Downarrow \alpha, h', \kappa$, then $h' \leq h$.*

Lemma 2 (Closure preservation)**.** *If e, h, ρ is well-formed and $e, h, \rho \Downarrow \alpha, h', \kappa$, then h' is closed and $\alpha \in \mathrm{supp}(h)$.*

Heap indices that cannot be referenced by the evaluated expression can be erased with few consequences.

Lemma 3 (Garbage collection)**.** *Let e, h, ρ be a well-formed triple. Let $X \subseteq \mathrm{supp}(\rho)$, $A \subseteq \mathrm{supp}(h)$, such that $e, h\restriction_A, \rho\restriction_X$ is well-formed.*
If $e, h, \rho \Downarrow \alpha, h', \kappa$, then $e, h\restriction_A, \rho\restriction_X \Downarrow \alpha, (h'\restriction_{A \cup (\mathrm{supp}(h') \setminus \mathrm{supp}(h))}), \kappa$.

Interpreting an expression under a heap containing unknowns can be seen as simultaneously interpreting that expression in the context of every smaller heap. When the execution paths split, *narrowing* chooses one branch and discards other possibilities.

We claim that a reduction with a partially defined heap simulates a reduction with any heap obtained by instantiating the output heap (which itself is an instantiation of the original heap) while satisfying the emitted constraints, and conversely that any reduction with an instantiation[2] of the input heap h can be simulated with h.

Due to lack of time, these are still conjectures. We believe them to hold under the assumption that the high-level requirements on the constraint solver are met.

Conjecture 1 (Soundness)**.** *If e, h, ρ is well-formed and $e, h, \rho \Downarrow \alpha, h', \kappa$, then for any closed heap h_0 such that*

1. $\mathrm{supp}(h_0) \cap \mathrm{supp}(h') \subseteq R(\mathrm{supp}(h), h')$,

[2] not necessarily fully defined

2. $h_0 \sqsubseteq_{R(\mathrm{supp}(h),h')} h'$,

3. h_0 *satisfies* κ,

there exists h'_0, κ_0 *such that*

1. $e, h_0, \rho \Downarrow \alpha, h'_0, \kappa_0$,

2. h'_0 *satisfies* κ_0,

3. $h'_0 \sqsubseteq h'$.

The first two conditions on h_0 above should be sufficient to clear the locations allocated by the reduction of e, h, ρ so that they can be reused by the reduction of e, h_0, ρ. The conclusion then concisely expresses the fact that the value returned through α in h'_0 is equal to or more defined than in h'.

Conjecture 2 (Completeness)**.** *If* e, h, ρ *is well-formed,* $e, h, \rho \Downarrow \alpha, h', \kappa$ *and* h' *satisfies* κ*, then for any heap* h_0 *such that* $h \sqsubseteq h_0$*, and* e, h_0, ρ *is well-formed, there exists* h'_0, κ_0 *such that:*

1. $e, h_0, \rho \Downarrow \alpha, h'_0, \kappa_0$,

2. $h' \sqsubseteq h'_0$,

3. h' *satisfies* κ_0.

Additional properties of constraint solvers With the semantics above, expressions can be interpreted as fairly standard functional programs in the context of a fully defined heap, or as generators of values with a partially defined heap instead, by searching for successful reductions. In particular, we obtain a generator out of a predicate, i.e., a boolean-typed expression e, by searching for derivations of reductions of the form $e, h, \rho \Downarrow \alpha, h', \kappa$, with $h'(\alpha) = \mathit{True}$.[3] With an appropriate constraint solver and disciplined use of it via refine, it is possible to guarantee that $\kappa = \epsilon$ in any case. Consequently, we may obtain a fully defined heap leading to the same execution path by instantiating the remaining unknowns independently.

Even though the main requirements of constraint solvers are supposedly sufficient to imply soundness and completeness, they are too weak to imply that the constraint solver is doing any work. Indeed, simply gathering constraints and doing nothing with them—i.e., $\phi(h, \kappa) = (h, \kappa)$—define a valid solver.

The following property forces the solver to simplify constraints to some extent: $\phi(h, \kappa \wedge \kappa') = \phi(h, \kappa')$ if h satisfies κ. Furthermore, the minimal requirements of the solver do not force it to check whether constraints are in fact satisfied on fully defined heaps, as shown by the inactive solver $\phi(h, \kappa) = (h, \kappa)$. This property excludes such behavior: if κ_0 is a basic constraint, $\phi(h, \kappa_0 \wedge \kappa)$ is undefined if no instantiation of h satisfies κ_0. Both of these additional properties are satisfied by our simple solver implementation.

3 Backtracking and probabilities

3.1 Motivation

The semantics presented above only gives the possible values of a program; richer semantics shall express the probabilistic aspects of Luck. The original semantics of Lampropoulos et al. [6] was

[3] Alternatively, the $h'(\alpha) = \mathit{True}$ restriction can be specified within our language, with partial pattern matching: let $x = e$ in case x ($\mathit{True} \to \mathit{Unit}$).

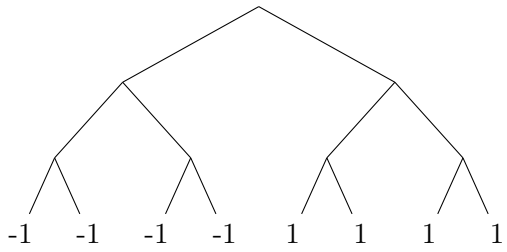

Figure 2: A decision tree whose answers depend on the first decision.

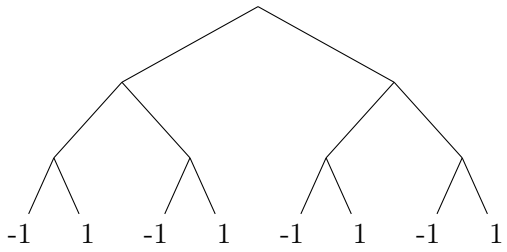

Figure 3: A decision tree whose answers depend on the last decision.

essentially subprobability distributions of heaps, where the missing probability is the probability of a random execution to fail. However Luck programs make a lot of choices blindly—in particular due to the rules (△-True-Operation) and (△-False-Operation)—therefore programs are in fact highly likely to fail on the first try, and we rely on constraint solving in order to detect mistakes early and on backtracking to correct recent choices instead of restarting from the beginning. It turns out that a simple probability distribution is insufficient to even distinguish programs with widely different backtracking behaviors.

The decision trees in Fig. 2 and Fig. 3 illustrate this. We consider randomized depth-first search: at every node, starting from the root, first go either left or right with probability $\frac{1}{2}$, and if no positive answer was found, go the other way. In both cases, the first answer met is positive with probability $\frac{1}{2}$. However if it is negative, the search in the first decision tree ends up exploring the whole of the left subtree before going in the right one, whereas in the second tree by backtracking a single step, it is guaranteed to find a positive answer next. One way to quantify this difference is to calculate the expected number of negative answers encountered before a positive answer is found: 2 for the first tree and $\frac{1}{2}$ for the second.

Thus for an accurate semantics, it seems necessary to concretely represent the *tree* of all possible executions of a Luck program, where branching nodes correspond to instantiation points (case or inst on an unknown) or random decisions on the outcome of comparisons $x \triangle y$. We use such trees here to provide a more precise probabilistic semantics for Luck. We may then specify separately a search algorithm these trees.

3.2 Controlling probabilities

In order to control the final probability distribution of results, non-deterministic constructs of the language are extended to contain weight annotations. Intuitively, the probability of choosing one of the possible branches —i.e., choosing a pattern for (CaseNarrow), or choosing a boolean value for (△-True-Operation) and (△-False-Operation)—is calculated by dividing the annotated weight by the sum of all weights in the construct. For the sake of simplicity, weights are constants here, but dynamic weights can also be used with few complications.

$$
\begin{aligned}
e ::= {} & \dots \\
\mid {} & \operatorname{case} x\ (\dots, \mathbf{n} : C(x_1, \dots, x_n) \to e, \dots) \\
\mid {} & (\mathbf{m} \| \mathbf{n}) : x \triangle y
\end{aligned}
$$

For instantiation inst, we rely on the constraint solver to associate a probability distribution to every possible unknown. In the case of our union of intervals implementation, if the set is finite we can propose a uniform distribution; in other cases a distribution should still be available for completeness, although we implicitly expect unknowns to be bounded before instantiation.

3.3 Probabilistic tree semantics

One way to formally define the execution tree of a Core Luck expression would be to give a probabilistic small-step semantics for this language, from which we would easily define the tree of states reachable from a given initial state. Here we take a different approach, and derive the tree associated to expressions by re-interpreting the big-step semantics above in monadic style. We have written a version of this in Haskell, and most, if not all, of the code appears easily reusable for the implementation of a working interpreter by replacing the underlying monad.

Trees Given a set V we define $T(V)$ the set of *V-trees* coinductively as follows: $t \in T(V)$ is either a *leaf* containing a value $v \in V$, an *empty node* ($\emptyset$), a *unary node* $\mathrm{Step}(t)$ containing a single sub-tree t, a *finitary node* with a finite number of weighted children, or a *wide node* with a family of children indexed by integers of a given set—a probability distribution on the children $\tau \in (\mathfrak{Z} \to T(V))$ is implicitly provided by the constraint solver.

$$\begin{aligned} t ::= & \ \mathrm{Leaf}(v) \\ | & \ \emptyset \\ | & \ \mathrm{Step}(t) \\ | & \ \mathrm{FNode}(\mathbf{n}_1 : t_1, \ldots, \mathbf{n}_k : t_k) \\ | & \ \mathrm{WNode}(\mathfrak{Z}, \tau) \end{aligned}$$

The type of trees is a monad, with return = Leaf and a bind operation which substitutes leaves by trees; $(\gg\!=) \in T(U) \to (U \to T(V)) \to T(V)$:

$$\begin{aligned} \mathrm{Leaf}(v) \gg\!= f &= f(v), \\ \emptyset \gg\!= f &= \emptyset, \\ \mathrm{Step}(t) \gg\!= f &= \mathrm{Step}(t \gg\!= f), \\ \mathrm{FNode}(\mathbf{n}_1 : t_1, \ldots, \mathbf{n}_k : t_k) \gg\!= f &= \mathrm{FNode}(\mathbf{n}_1 : (t_1 \gg\!= f), \ldots, \mathbf{n}_k : (t_k \gg\!= f)), \\ \mathrm{WNode}(\mathfrak{Z}, \tau) \gg\!= f &= \mathrm{WNode}(\mathfrak{Z}, \lambda\mathbf{n}.(\tau(\mathbf{n}) \gg\!= f)). \end{aligned}$$

The *tree of evaluations* of a well-formed triple e, h, ρ, denoted $\mathrm{tree}(e, h, \rho)$, is a tree of triples α, h', κ, that is to say in $T(\mathcal{I} \times (\mathcal{I} \rightharpoonup \mathcal{H}) \times \mathcal{K})$, and is defined as a cofixpoint. The remaining paragraphs detail the cases of this definition.

Simple values When e is a variable x, a literal $\mathbf{n}$, a lambda abstraction λ, a fixpoint operator μ, or a constructor $C(x_1, \ldots, x_n)$, only one rule applies, with no premise, and the right hand side of the conclusion is a function of the left hand side:

$$\frac{}{e, h, \rho \Downarrow f(e, h, \rho)}$$

In all these cases, we define $\mathrm{tree}(e, h, \rho) = \mathrm{Leaf}(f(e, h, \rho))$.

Other deterministic constructs When e is a function application $f\ y$, or a case expression $\mathrm{case}\, x\ (\ldots, C(x_1, \ldots, x_n) \to e', \ldots)$ with a matching value $h(\rho(x)) = C(\alpha_1, \ldots, \alpha_n)$, the only applicable rule takes the following form:

$$\frac{f(e, h, \rho) \Downarrow \alpha, h', \kappa}{e, h, \rho \Downarrow \alpha, h', \kappa}$$

then $\text{tree}(e, h, \rho) = \text{Step}(\text{tree}(f(e, h, \rho)))$.

For a case expression, when no constructor matches, then $\text{tree}(e, h, \rho) = \emptyset$.

When e is a let expression, we use the following definition for tree:

$$\text{tree}(\text{let}\, x = e\, \text{in}\, e', h, \rho) = \\ \text{tree}(e, h, \rho) \gg\!= \lambda(\alpha, h', \kappa).\ (\text{tree}(e', h', \rho \cdot \{x \mapsto \alpha\}) \gg\!= \lambda(\beta, h'', \kappa').\ \text{Leaf}(\beta, h'', \kappa \wedge \kappa')).$$

In this case we mix co-recursion with recursion in a way that is known to be well-defined [2].

Narrowing and constraint solving When $e = \text{case}\, x\ (\dots)$ matches an unknown, i.e., $h(\rho(x)) = \star$, we use a finitary node with one child for every alternative. The variants of the (CaseNarrow) rule all have a shape similar to (Case):

$$\frac{f_i(e, h, \rho) \Downarrow \alpha, h', \kappa}{e, h, \rho \Downarrow \alpha, h', \kappa}$$

where f_i instantiates the unknown with the i-th pattern. Let k be the total number of patterns and $\mathbf{n}_i$ be the weight of the i-th alternative in e, then

$$\text{tree}(e, h, \rho) = \text{FNode}(\mathbf{n}_1 : \text{tree}(f_1(e, h, \rho)), \dots, \mathbf{n}_k : \text{tree}(f_k(e, h, \rho))).$$

When $e = (\mathbf{m} \| \mathbf{n}) x \ \triangle\ y$, with the rules (△-True-Operation) and (△-False-Operation), we branch again with FNode and return leaves similarly to the first case above.

$$\text{tree}(e, h, \rho) = \text{FNode}(\mathbf{m} : \text{Leaf}(\alpha, h' \cdot \{\alpha \mapsto \mathit{True}\}, (\rho(x) \ \triangle\ \rho(y))), \\ \mathbf{n} : \text{Leaf}(\alpha, h' \cdot \{\alpha \mapsto \mathit{False}\}, (\rho(x) \blacktriangledown \rho(y)))).$$

When $e = \text{refine}(e')$, the rule (Refine) introduces a guard at the end of a computation:

$$\text{tree}(e, h, \rho) = \text{tree}(e', h, \rho) \gg\!= f,$$

where

$$f(\alpha, h', \rho) = \begin{cases} \text{Leaf}(\alpha, h'_0, \rho_0) & \text{if } \phi(h', \rho) = (h'_0, \rho'_0), \\ \emptyset & \text{if } \phi(h', \rho) \text{ is undefined.} \end{cases}$$

Finally, when $e = \text{inst}(x)$, if x is already instantiated, we do nothing:

$$\text{tree}(e, h, \rho) = \text{Leaf}(\rho(x), h, \epsilon),$$

otherwise, $h(\rho(x)) = \mathfrak{z}$, and thus

$$\text{tree}(e, h, \rho) = \text{WNode}(\mathfrak{z}, \lambda\mathbf{n}.\, \text{Leaf}(\rho(x), h[\rho(x) \mapsto \mathbf{n}], \epsilon)).$$

We may verify that the tree semantics are consistent with the $\Downarrow$ semantics.

Lemma 4. *$e, h, \rho \Downarrow \alpha, h', \kappa$ if and only if α, h', κ is in a leaf of* $\text{tree}(e, h, \rho)$.

```
isSorted xxs =
  case xxs of
    x1 : xs@(x2 : _) -> x1 < x2 && isSorted xs
    _ -> True
```

Figure 4: A definition featuring nested pattern matching in Haskell.

```
isSorted xxs =
  case xxs of
    [] -> True
    x1 : xs ->
      case xs of
        [] -> True
        x2 : _ -> x1 < x2 && isSorted xs
```

Figure 5: Expansion of Fig. 4 to simple pattern matches.

Search algorithm We use a variant of depth-first search (DFS) in order to find a Leaf in the tree, possibly with other conditions on the resulting value. Because our trees are potentially infinite, DFS may not terminate. In fact DFS never backtracks out of infinite subtrees.

One cause of infinite trees is that instantiating a variable bound to an infinite set creates a node with infinitely many children. Furthermore, the wide nodes generally have a quite high branching factor even when it is finite. Thus we will search only one of the randomly chosen children. This strategy is efficient when the program does not rely on integers at a high precision, or when the generated value is unlikely to be the main cause of backtracking, for instance in generators of search trees. Conversely, in generators of low-level machine instructions, the control flow is highly sensitive to the values of pointers; however we have not had the opportunity to investigate this particular issue in more detail.

Sometimes it is preferable to restart generation from scratch, two reasons being that some early choices might lead to conditions which are hard to satisfy, and that backtracking skews the distribution of outcomes in unpredictable ways. Therefore, we allow to set a bound on the number of dead ends encountered by the search.

4 Expanding nested patterns

Nested pattern matching is a ubiquitous feature of languages using algebraic datatypes. They can easily be implemented as syntactic sugar for successive matches with simple patterns with a single constructor; see Fig. 4 and Fig. 5 for instance. Patterns containing wildcards or variable patterns may be *expanded* into multiple patterns, such that patterns do not overlap anymore. Thus some branches are duplicated, such as the `True` branches in Fig. 5.

The previous section presented the semantics of an untyped core language. Here, we consider nested patterns as a feature of a typed outer language where algebraic datatypes each have a finite number of constructors, such that it makes sense to expand patterns in the way we mentioned.

```
data T = A T T | B T | C

fun foo t =
    case t of
    | 3 % A _ C -> A C C
    | 2 % B C -> B C
    | 1 % _ -> C
    end
```

Figure 6: A Luck definition with nested patterns.

4.1 Nested patterns in Luck

In Luck, patterns are annotated with weights. As pattern expansion may result in more branches than in the original program, it is not clear what the weights of the expanded branches should be.

One property which seems intuitive to expect and consequently which we wish to guarantee is that the sum of the weights of the *clones* of a branch is equal to the weight of the original branch. What remains to be determined is how to distribute the weight among these branches which come from the same original one.

We investigate two candidates for a default reweighing strategy.

The most obvious strategy is to simply share the weight equally among all branch clones. The weight of a single branch then depends on the total number of clones: that can be hard for users to determine, and may vary widely even between sets of patterns that appear similar.

For instance in Fig. 8, adding a pattern with an `A` constructor at the root causes the weights in branches associated with the other constructors to change as well.

The alternative strategy we propose makes variations of the original patterns affect the weight distribution more locally. The expanded expression has the shape of a tree, with case expressions as nodes. For every original branch B, and every node N, the sums of the weights of the clones of B in the immediate subtrees of N which contain at least one such clone are all equal.

Backtracking may also depend on the way nested patterns are expanded.

References

[1] Sergio Antoy. A needed narrowing strategy. In *Journal of the ACM*, number 4, pages 776–822. ACM Press, 2000. URL: https://www.informatik.uni-kiel.de/~mh/papers/JACM00.pdf.

[2] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Foundational extensible corecursion. *CoRR*, abs/1501.05425, 2015. URL: http://arxiv.org/abs/1501.05425.

[3] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279. ACM, 2000. URL: http://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/quick.pdf.

[4] Sebastian Fischer and Herbert Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 63–74. ACM, 2007. URL: http://www-ps.informatik.uni-kiel.de/~sebf/pub/ppdp07.html.

```
case t of
| (3 + 2/5) % A _ r ->
    case r of
    | (1/5) % A _ _ -> C
    | (1/5) % B _ -> C
    | 3 % C -> A C C
    end
| (2 + 2/5) % B b ->
    case b of
    | (1/5) % A _ _ -> C
    | (1/5) % B _ -> C
    | 2 % C -> B C
    end
| (1/5) % C -> C
end
```

Figure 7: The expanded body of `foo`, with equally distributed weights.

```
fun bar t =
    case t of
    | 3 % A _ C -> A C C
    | 3 % A (B _) _ -> B (B C)
    | 2 % B C -> B C
    | 1 % _ -> C
    end
```

Figure 8: A similar definition with an additional case.

```
case t of
| (3 + 3 + 4/7) % A l r ->
    case l of
    | (3/2 + 2/7) % A _ _ ->
        case r of
        | (1/7) % A _ _ -> C
        | (1/7) % B _ -> C
        | (3/2) % C -> A C C
        end
    | 3 % B _ -> B (B C)
    | (3/2 + 2/7) % C ->
        case r of
        | (1/7) % A _ _ -> C
        | (1/7) % B _ -> C
        | (3/2) % C -> A C C
        end
    end
| (2 + 2/7) % B b ->
    case b of
    | (1/7) % A _ _ -> C
    | (1/7) % B _ -> C
    | 2 % C -> B C
    end
| (1/7) % C -> C
end
```

Figure 9: The expanded body of `bar`, with equally distributed weights.

```
case t of
| (3 + 1/3) % A _ r ->
    case r of
    | (1/6) % A _ _ -> C
    | (1/6) % B _ -> C
    | 3 % C -> A C C
    end
| (2 + 1/3) % B b ->
    case b of
    | (1/6) % A _ _ -> C
    | (1/6) % B _ -> C
    | 2 % C -> B C
    end
| (1/3) % C -> C
end
```

Figure 10: The expanded body of `foo`, with locally uniformly distributed weights.

```
case t of
| (3 + 3 + 1/3) % A l r ->
    case l of
    | (3/2 + 1/6) % A _ _ ->
        case r of
        | (1/12) % A _ _ -> C
        | (1/12) % B _ -> C
        | (3/2) % C -> A C C
        end
    | 3 % B _ -> B (B C)
    | (3/2 + 1/6) % C ->
        case r of
        | (1/12) % A _ _ -> C
        | (1/12) % B _ -> C
        | (3/2) % C -> A C C
        end
    end
| (2 + 1/3) % B b ->
    case b of
    | (1/6) % A _ _ -> C
    | (1/6) % B _ -> C
    | 2 % C -> B C
    end
| (1/3) % C -> C
end
```

Figure 11: The expanded body of `bar`, with locally uniformly distributed weights. Notice that the weights in the cases `B b` and `C` at the root have the same weights as for `foo`.

[5] Michael Hanus. A unified computation model for functional and logic programming. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 80–93. ACM Press, 1997. URL: http://www.informatik.uni-kiel.de/~mh/papers/POPL97.pdf.

[6] Leonidas Lampropoulos, Benjamin C. Pierce, Cătălin Hriţcu, John Hughes, Zoe Paraskevopoulou, and Li-yao Xia. Making our own Luck: A language for random generators. Draft, July 2015. URL: http://prosecco.gforge.inria.fr/personal/hritcu/publications/luck.pdf.

[7] Fredrik Lindblad. Property directed generation of first-order test data. In *8th Symposium on Trends in Functional Programming (TFP)*, pages 105–123, 2007. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.116.2439&rep=rep1&type=pdf.

[8] Zoltan Somogyi, Fergus Henderson, and Thomas C. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, October-December 1996. URL: http://www.mercurylang.org/information/papers/jlp.ps.gz.

[9] Andrew P. Tolmach and Sergio Antoy. A monadic semantics for core Curry. *Electr. Notes Theor. Comput. Sci.*, 86(3):16–34, 2003. URL: http://dx.doi.org/10.1016/S1571-0661(04)80691-1, doi:10.1016/S1571-0661(04)80691-1.