ALEA IACTA EST: A Declarative Domain-Specific Language for Manually Performable Random Experiments (Draft)

Baltasar Trancón y Widemann
semantics gGmbH, Berlin, DE
Brandenburg University of Applied Sciences, DE
trancon@th-brandenburg.de

Markus Lepper
semantics gGmbH, Berlin, DE

Random experiments that are simple and clear enough to be performed by human agents feature prominently in the teaching of elementary stochastics as well as in games. We present Alea, a domain-specific language for the specification of random experiments. Alea code can either be analyzed statically to obtain and inspect probability distributions of outcomes, or be executed with a source pseudo-randomness for simulation or as a game assistant. The language is intended for ease of use by non-expert programmers, by focusing on concepts common to functional programming and basic mathematics. Both the design of the language and the implementation of runtime environments are work in progress.

1 Introduction

Human culture has devised many ways for performing controlled simple random experiments: Bones, dice and coins can be thrown, distinct items can be drawn from bags, urns or shuffled card decks, etcetera. The outcomes have been used to drive games of chance, simulations and sometimes even serious decision-making. The mathematical field of stochastics has, to a substantial degree, arisen from the analysis of such experiments [5]. Even though it has later been expanded to encompass vastly more abstract and powerful notions of uncertainty, the simple experiments still have a prominent place in its teaching.

The most basic experiments consist of isolated random operations, for example:

Toss a denarius coin. Determine whether it shows a ship or a head. [1]

More interesting experiments are created by aggregating random data, and by performing transformations and statistical calculations on them, for example:

Roll seven ten-sided dice. For every die that shows a ten, roll another and add it to the pool. Count the number of dice that show values greater than five. Compare to the number of dice that show a one. If the difference is positive, you win by that amount. If the difference is zero or negative, you lose. If there are no values greater than five but there is a one, you lose badly. [2]

Alea is being designed as a notation that gives precise syntax and semantics to such random experiments. As the examples indicate, a functional language with rich datatypes but simple control flow, orthogonally extended with randomness, is called for. Ideally, the language should be intuitive to use, without advanced programming skills, but still offer the benefits of a 'real' programming language in terms of automatic checking and interpretation. It should be expressive enough that random experiments can be specified clearly and concisely, but not too powerful for exhaustive static analysis.

[1] An ancient Roman game [1].

[2] A rule from the tabletop RPG *Vampire: The Masquerade* [6].

Submitted to:
TFPIE 2025

2 Design of Semantic Domain

The design of Alea is founded on semantic requirements that impose limits on the admissible expressive power of language features.

2.1 Probability

The semantics of Alea expressions shall be *stochastically effective*: Any well-formed and well-typed expression must be associated with a unique probability distribution that can be computed systematically, without recourse to algebraic meta-knowledge or heuristics.

A probability distribution on some set X is called *simple* if and only if it is *rational*, *discrete*, and *finite*, i.e., defined by a rational-valued, finitely supported probability mass function $P \in \mathscr{D}(X)$:

$$P : X \to \mathbb{Q} \qquad P(x) \geq 0 \qquad \sum_{x \in X} P(x) = 1 \qquad \operatorname{supp}(P) \text{ finite}$$

The inner product of a simple distribution with any other number-valued function is a finite sum, even if the function domain is infinite:

$$\sum_{x \in X} P(x) \cdot f(x) = \sum_{x \in \operatorname{supp}(P)} P(x) \cdot f(x)$$

Hence the *mean* of a simple distribution over the rationals is straightforwardly computable:

$$m(P) = \sum_{x \in \operatorname{supp}(P)} P(x) \cdot x$$

The construct $\mathscr{D}$ can be given the rich category-theoretic structure of a *commutative*, or *symmetric monoidal monad*. Namely, it gives rise to a functor that generalizes computing the distribution of random variables f from the distribution of underlying events,

$$\mathscr{D}(f : X \to Y) : \mathscr{D}(X) \to \mathscr{D}(Y) \qquad \mathscr{D}(f)(P)(y) = \sum_{x \in X} P(x) \cdot [f(x) = y]$$

with the Kronecker delta distribution as the monad unit,

$$\delta_X : X \to \mathscr{D}(X) \qquad \delta_X(x)(x') = [x = x']$$

the folding of two layers of a probability tree into one as the monad multiplication,

$$\mu_X : \mathscr{D}\big(\mathscr{D}(X)\big) \to \mathscr{D}(X) \qquad \mu_X(P)(x) = \sum_{Q \in \mathscr{D}(X)} P(Q) \cdot Q(x)$$

and the stochastically independent combination of marginal distributions,

$$\psi_{X,Y} : \mathscr{D}(X) \times \mathscr{D}(Y) \to \mathscr{D}(X \times Y) \qquad \psi_{X,Y}(P,Q)(x,y) = P(x) \cdot Q(y)$$

which are all natural and satisfy several coherence laws. These properties allow us to ‘lift’ the intuitively evident meaning of a deterministic term language to a stochastic reading in a systematic way, by means of Moggi-style [4] denotational semantics.

For example, consider an expression of the form $f(a,b)$, where a and b are subexpressions that evaluate deterministically to the values $v_a \in A$ and $v_b \in B$, respectively, and $f : A \times B \to C$ is a known binary function. Then the overall value is clearly $f(v_a, v_b) \in C$. Now let b (but not a) instead evaluate stochastically to a distribution $P_b \in \mathscr{D}(B)$, and f be also of the stochastic type $f : A \times B \to \mathscr{D}(C)$. Then the overall distribution is given canonically by $\mu_C\big(\mathscr{D}(f)\big(\psi_{A,B}(\delta_A(v_a), P_b)\big)\big) \in \mathscr{D}(C)$.

Furthermore, all distributions have finite representations and all monadic operations are inherently computable. Hence, if used together with computable random variables only, the distribution of any finitary expression is computable and the equality of distributions is decidable, in a constructive way.

On the downside, the class of simple distributions is not closed under unbounded iteration or recursion. For example, the experiment *"Repeat rolling a die until it shows a six"* does *not* have a simple distribution. We have decided to accept that limitation on the expressivity of Alea, at least in the current version of the language.

2.2 Data

The intuition underlying the data model of Alea is discussed here briefly and informally. For a more structural treatment, see section 3.2 below.

2.2.1 Primitive Data

Most primitive data involved in random experiment are either numeric (for arithmetic computation) or of a finite enumerated type (for classification). Alea supports the rational numbers and their subalgebras, the integers and naturals. Enumerated types are supported by tagging values with symbolic identifiers, in the sense of algebraic datatype constructors.

A special role is played by the Boolean truth values, which could be understood as either numeric or enumerated. Since it appears that the majority of uses of Boolean values in specifications of random experiments is for *counting*, Alea treats them as the values $\{0,1\}$, a subtype of the natural numbers. Note that for languages that treat the Booleans as non-numerical, the *Iverson bracket* $[\_]$ is a ubiquitous operation in counting computations.

The set of numeric values is extended with the special value NaN (*not-a-number*) with semantics analogous to the standard for floating-point numbers: Undefined arithmetic operations, most notably division by zero, yield NaN. If any operand is NaN, arithmetic operations also yield NaN, whereas comparison operations yield false.[3] While this convention is well-understood and works fine for purely numerical data, interesting problems arise when aggregate data are considered; see below.

2.2.2 Collections

Multiple data elements of the same type may be aggregated into collections. In addition to the most popular collection shapes in functional programming, *lists* and *sets*, Alea provides built-in support for *multisets* aka *bags*. Bags should be preferred over lists in any situation where the ordering of elements is irrelevant, and comes with two significant benefits:

1. The number of possible outcomes, and hence the size of distributions, may be reduced dramatically, by as much as $n!$, if subsequent interpretation is only up to permutation.

[3] With the notable exception of $\neq$.

2. The bulk processing with *map*, *filter* and *reduce* operations can be accelerated whenever an individual elements occurs many times in a bag; to a single step for *map* and *reduce*, and to a logarithmic number of steps for *reduce* with an associative binary operation, by means of the exponentiation-by-squaring algorithm.

In all other respects, the treatment of all three collection shapes in the language is as similar as possible.

The semantics of collections are complicated by the fact that the object-level comparison operator $\stackrel{\star}{=}$[4] for numerical values is different from the meta-level set-theoretic identity relation $=$; namely $\mathrm{NaN} \stackrel{\star}{=} \mathrm{NaN}$ is false. For sets in Alea semantics, we specify that

$$S \stackrel{\star}{\subseteq} T \iff \forall x \in S.\, \exists y \in T.\, x \stackrel{\star}{=} y$$

and derive:

$$S \stackrel{\star}{\supseteq} T \iff T \stackrel{\star}{\subseteq} S \qquad S \stackrel{\star}{\subset} T \iff S \stackrel{\star}{\subseteq} T \wedge \neg(S \stackrel{\star}{\supseteq} T) \qquad S \stackrel{\star}{=} T \iff S \stackrel{\star}{\subseteq} T \wedge S \stackrel{\star}{\supseteq} T$$

$$S \stackrel{\star}{\supset} T \iff S \stackrel{\star}{\supseteq} T \wedge \neg(S \stackrel{\star}{\subseteq} T) \qquad S \stackrel{\star}{\neq} T \iff \neg(S \stackrel{\star}{=} T)$$

For example, we find that $\{2,3\} \stackrel{\star}{\subset} \{1,2,3,\mathrm{NaN}\}$ is true, but $\{\mathrm{NaN}\} \stackrel{\star}{=} \{\mathrm{NaN}\}$ is false.

The construction can be transfered to bags by accounting for multiplicity. For lists, there are several contenders for a ‘sublist’ relationship. We currently assume that ‘contiguous sublist anywhere’ is the most useful one and define list comparison accordingly, but that design choice is contentious.

3 Notation and Concepts

3.1 Syntax

The concrete front-end syntax of Alea is not yet finalized. Hence we will not give a formal grammar here, but describe the overall style by examples. By contrast, see Section 4 for an abstract syntax with all due semantic formalities.

3.1.1 Numbers and Arithmetics

Alea features a single unified type of numbers, namely unlimited-precision rationals, with all the usual arithmetic and comparison operators.

```
3 * (x + 1) < −2/3
```

The special value *NaN* can be written as 0/0.

Integer and natural numbers are recognized as subtypes, and come with integer division operators // (*div*) and \\ (*mod*).

Booleans are just the numbers zero (*false*) and one (*true*). Hence the logical operators $\wedge$ and $\vee$ are just synonyms for *min* and *max*, respectively. Unlike in many languages where numbers and Booleans are distinct, $+$ and $*$ with one or more Boolean operands are useful for counting and numerical filtering, respectively.

[4] Written as = in the front-end notation.

3.1.2 Collections

Alea supports three shapes of collections, namely lists, bags and sets. In the syntax, they are mostly distinguished by the brace shapes [ ], ⟨ ⟩ and { }, respectively. Collections can be created by enumerating the elements or specifying ranges.

```
[1, 2, 3, 4, 5]        ⟨2, 2, 4, 3, 2⟩        {1 .. 5}
```

Collections are processed using bulk operations. The equivalent of the usual higher-order functions *map* and *filter* is expressed in first-order notation with Haskell-style comprehensions.

```
{ k∗k | k ← {1 .. n}; even(k) }
```

Unusually for a programming language with comprehensions, Alea also supports 'drawing without replacement'.

```
{ b − a | {a, b} ← {1 .. n}; a < b }
```

The equivalent of the usual higher-order function *reduce* is expressed by applying a binary function to a single collection argument. Operators can be used as functions in this sense, but must be quoted in parentheses. Furthermore, the function must be known to . . .

- be associative,
- be commutative if the collection is a bag or set,
- have a neutral element if the collection may be empty.

How to declare or check these semigroup-theoretic properties is out of scope here.

```
max{x, y, z}        (+)⟨ k ≥ 0 | k ← B ⟩
```

The latter example demonstrates how summation over a collection of Booleans, resulting in a natural number, is conveniently used for counting.

3.1.3 Named Functions and Distributions

Alea comes with a library of predefined functions, which are named but not conceptually different from operators. The argument must be given in parentheses, unless it is already some sort of bracketed expression.

```
max(a, b) + min{x, y, z}
```

Additionally, there library contains predefined distributions, wich are syntactically marked with ~, and may also take dynamic parameters. The invocation of a distribution can be thought of as an anonymous random variable. Note that Alea is not referentially transparent in this respect; every evaluation of a distribution expression is stochastically independent from the other. The expectation operator is written as the function E.

```
E(~uniform{1 .. 6} + ~bernoulli(2/3))
```

3.1.4 Records

Records, aggregate data with several fields of heterogeneous type, are written in parentheses. The fields can be identified explicitly by name or implicitly by position. Tuples are just records with only positional fields. Mixing both styles is permitted, but not recommended. Selection of fields from a record is written with the usual dot notation. As is Standard ML, positional field numbers start from one. They must be marked with # to avoid ambiguity with decimal fractions.

```
(foo: 42, bar: {})     (1, 2, 3)     r.foo     s.#3
```

3.1.5 Tagged Values

Tagged values are elements of sum types. The tag names, both as constructors and in patterns, are written like function applications but marked with @. If the argument is omitted, it defaults to the empty tuple (). Thus, tags also serve as enumerated constants. Tag can be used freely, without having to declare which sum type they inject into. Tagged data are processed using elementary pattern matching in the style of Haskell (core).

```
@good(42)     @bad     x ? { @good(n) → n; @bad → −1 }
```

Explicit matching cases must be non-overlapping; an additional default case is written with the pattern _. The same notation is also used for a C-style switch statement on numerical values.

```
n ? { 0 → @none; 1 → @one; 2, 3 → @few; _ → @many }
```

The special case of Boolean values can be abbreviated, also in a C-like style.

```
b ? { 1 → x; 0 → y }     b ? x : y
```

3.1.6 Non-Features

Alea does not have many beloved features of typical functional programming languages: No recursion, no partially defined operations such as list indexing, no higher-order functions, no lambdas. It is a deliberate design choice that the expression notation uses only graphical symbols, no keywords.

3.2 Type System

The type universe of Alea is specified by the following abstract syntax:

$$
\begin{aligned}
\mathit{Type} ::= {} & \mathit{any} \mid \mathit{none} & \mathit{Num} &::= \mathit{bool} \mid \mathit{nat} \mid \mathit{int} \mid \mathit{rat} \\
\mid {} & \mathit{num}(\mathit{Num}) \mid \mathit{coll}(\mathit{Shape}, \mathit{Mode}, \mathit{Type}) & \mathit{Shape} &::= \mathit{list} \mid \mathit{bag} \mid \mathit{set} \\
\mid {} & \mathit{prod}(\mathit{FieldId} \twoheadrightarrow \mathit{Type}) \mid \mathit{sum}(\mathit{CaseId} \twoheadrightarrow \mathit{Type}) & \mathit{Mode} &::= \mathit{pos} \mid \mathit{opt}
\end{aligned}
$$

Note that this abstract syntax is designed for extensibility, orthogonality and conceptual simplicity of computation, not for ease of notation in the front-end language. However, this is not a pressing problem, because types feature only implicitly in end-user code.

- The types *any* and *none* denote the upper and lower bound of the type lattice, respectively.
- The constructor *num* denotes numeric types.

$$[\![none]\!]_V = \varnothing \qquad [\![any]\!]_V = Val$$

$$[\![num(bool)]\!]_V = const(\mathbb{B}) \qquad [\![num(nat)]\!]_V = const(\mathbb{N}) \cup \{NaN\}$$
$$[\![num(int)]\!]_V = const(\mathbb{Z}) \cup \{NaN\} \qquad [\![num(rat)]\!]_V = const(\mathbb{Q}) \cup \{NaN\}$$

$$[\![coll(list,opt,t)]\!]_V = thelist\big(\mathbf{L}([\![t]\!]_V)\big) \qquad [\![coll(list,pos,t)]\!]_V = thelist\big(\mathbf{L}([\![t]\!]_V) \setminus \{\varnothing\}\big)$$
$$[\![coll(bag,opt,t)]\!]_V = thebag\big(\mathbf{M}([\![t]\!]_V)\big) \qquad [\![coll(bag,pos,t)]\!]_V = thebag\big(\mathbf{M}([\![t]\!]_V) \setminus \{\varnothing\}\big)$$
$$[\![coll(set,opt,t)]\!]_V = theset\big(\mathbf{P}([\![t]\!]_V)\big) \qquad [\![coll(set,pos,t)]\!]_V = theset\big(\mathbf{P}([\![t]\!]_V) \setminus \{\varnothing\}\big)$$

$$[\![prod(T)]\!]_V = record\big(\{f : FieldId \nrightarrow Val \mid \forall i \in \mathrm{dom}(T).\, f(i) \in [\![T(i)]\!]_V\}\big)$$
$$[\![sum(T)]\!]_V = \bigcup_{i \in \mathrm{dom}(T)} tag\big(\{i\} \times [\![T(i)]\!]_V\big)$$

Figure 1: Extensional Type Semantics

- The constructor *coll* denotes collection types, which come in the shapes *list*, *set* and *bag*, are of the mode *pos* (positive, may not be empty) or *opt* (optional, may be empty) and have a specific element type.
- The constructor *prod* denotes product types, which have a finite set of orthogonal fields specified as a mapping of field identifiers to field types. Field identifiers can be symbolic or positional, such that tuples are merely a special case of products.
- The constructor *sum* denotes sum types, which have a finite set of disjoint cases specified as a mapping of case identifiers to case types. Enumerated types are special cases of sum types, where every case type is the unit type $prod(\varnothing)$.

3.2.1 Semantics of Types

Since here functions, let alone recursive ones, are not values that would need to be typed, a simple set-theoretic semantics can be given that assigns to every type its *extension*. Let there be an untyped value universe specified by the following abstract syntax:

$$\begin{aligned} Val ::= {} & const(\mathbb{Q}) \mid NaN \mid thelist\big(\mathbf{L}(Val)\big) \mid thebag\big(\mathbf{M}(Val)\big) \mid theset\big(\mathbf{P}(Val)\big) \\ & \mid record(FieldId \nrightarrow Val) \mid tag(CaseId \times Val) \end{aligned}$$

For the sake of uniformity, we write the operators $\mathbf{L}$, $\mathbf{M}$ and $\mathbf{P}$ for the finite lists, bags, and subsets of a set, respectively, and $\varnothing$ for the empty list, bag and set. Note that we assume that number spaces are properly embedded: $\mathbb{B} \subset \mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q}$.

The extensional semantics of types is a map $[\![\_]\!]_V : Type \to \mathbf{P}(Val)$, the details are given in Figure 1.

$$\frac{}{none \sqsubseteq t \sqsubseteq any} \qquad \frac{n_1 \sqsubseteq n_2}{num(n_1) \sqsubseteq num(n_2)} \qquad \frac{}{bool \sqsubseteq nat \sqsubseteq int \sqsubseteq rat}$$

$$\frac{s_1 \sqsubseteq s_2 \quad m_1 \sqsubseteq m_2 \quad t_1 \sqsubseteq t_2}{coll(s_1,m_1,t_1) \sqsubseteq coll(s_2,m_2,t_2)} \qquad \frac{}{pos \sqsubseteq opt}$$

$$\frac{\forall i \in dom(T_2).\, T_1(i) \sqsubseteq T_2(i)}{prod(T_1) \sqsubseteq prod(T_2)} \qquad \frac{\forall i \in dom(T_1).\, T_1(i) \sqsubseteq T_2(i)}{sum(T_1) \sqsubseteq sum(T_2)}$$

Figure 2: Subtyping Rules

3.2.2 Empty and Inhabited Types

The following inference rules deduce that a type is empty on a syntactical basis:

$$\frac{}{\text{empty}\, none} \qquad \frac{\forall i \in \text{dom}(T).\, \text{empty}\, T(i)}{\text{empty}\, sum(T)}$$

$$\frac{\text{empty}\, t}{\text{empty}\, coll(s,pos,t)} \qquad \frac{\exists i \in \text{dom}(T).\, \text{empty}\, T(i)}{\text{empty}\, prod(T)}$$

These rules are sound and complete:

$$\text{empty}\, t \iff [\![t]\!]_{\mathrm{V}} = \varnothing$$

3.2.3 Subtyping

Alea types form a lattice. The inference rules of the subtyping calculus are given in Figure 2. The presentation has been geared towards highlighting how independent sublattices are pieced together. As a result, some statements are a little indirect; most notably $s_1 \sqsubseteq s_2$ is merely saying $s_1 = s_2$, since there are no collection subshapes, at least not in the current version of the type system.

The subtyping relation defined by these rules is extensionally sound,

$$t_1 \sqsubseteq t_2 \implies [\![t_1]\!]_{\mathrm{V}} \subseteq [\![t_2]\!]_{\mathrm{V}}$$

but no attempt has been made to make it complete (the converse implication). In particular, there are empty types that are not subtypes of *none*. It is easy to see that the subtyping relation is purely structural; there are no nominal type, let alone subtype declarations.

The subtype relation does indeed give rise to a lattice. The induced join and meet operations likewise have straightforward syntax-directed definitions.

3.2.4 Function Overloading

Getting function overloading right in conjunction with rich type systems, especially with subtyping, is notoriously tricky [2, 8]. Alea makes two simplifying assumptions:

1. Functions have a single implementation that is a partial function defined on a subset of the untyped universe. Hence a function signature of the form $f : t \to u$ means: "If f is applied to an argument value of type t, the result value is defined and has type u". Thus, a function can admit multiple sound type signatures, but the result computed from an argument value must not depend on the type signature used in its checking. Ad-hoc polymorphic functions can still be pieced together from independent pieces with *disjoint* domains.
2. Function polymorphism is *compact*: A polymorphic function may admit infinitely many valid type signatures of the form $f : t \to u$, (such as all instantiations of a type scheme in the Hindley–Milner system,) but for any actual, valid call with an argument of type t', there are only finitely many signatures $f : t_i \to u_i$ with $t' \sqsubseteq t_i$ to consider. The effective actual result type is then the greatest lower bound of the formal result types, $u' = \bigsqcap u_i$, since they all apply as simultaneous guarantees. Then we write $\vdash f : t' \to u'$ in Figure 3 below. The function call is invalid, if the set of matching signatures is empty. How polymorphic type signatures are declared is out of scope here.

The former assumption has consequences for *homomorphic* overloading [7]: For example, rational (exact) division and integer (rounding) division must be considered distinct functions, since they act differently on a pair of integers. The former is described by a single type signature $\mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$, whereas for the latter it is useful to give two type signatures, $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ and $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$.[5]

The latter assumption allows for *parametric* polymorphism in a general and abstract form, without specifying how it is resolved. For example, the addition operation is highly overloaded:

- It acts as the usual arithmetic operation with the signatures $\mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$, $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, and $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$. Note that the latter implies that the sum of Booleans is a natural number.
- It acts as the concatenation operation on lists. This admits an infinite parametric family of valid type signatures, namely all of the form

$$\mathit{coll}(\mathit{list}, m_1, t_1) \times \mathit{coll}(\mathit{list}, m_2, t_2) \to \mathit{coll}(\mathit{list}, m_3, t_3)$$

  with one of $m_1, m_2 \sqsubseteq m_3$ and both $t_1, t_2 \sqsubseteq t_3$. However, for an actual call it suffices to consider the single signature with $m_3 = m_1 \sqcap m_2$ and $t_3 = t_1 \sqcup t_2$.
- It acts analogously as the disjoint union on bags.
- All of the preceding cases form semigroups with some neutral element, and all except list concatenation are commutative. Hence they can be used in *reduce* operations on many collection types, e.g., a bag of numbers, a list of lists, a list of bags, but not a bag of lists.

4 Program Evaluation

In this section we shall outline the semantics of Alea programs. By nature of the language, there is no fundamental difference between evaluation of deterministic expressions, static analysis of outcome distributions, and pseudo-randomized simulation. All three processes share the same basic syntax-directed big-step semantics rules, and are constructive enough to be implemented directly as interpreters. The difference is in the choice of the monad that encapsulates results; namely the identity monad for determinism, the distribution monad for stochastics and a state monad with a pseudo-random generator for simulation, respectively.

[5] For clarity, we use the traditional notation for function signatures here rather than the cumbersome abstract type syntax.

4.1 Abstract Syntax

The internal representation of Alea expressions, on which all evaluation is based, is specified by the following abstract syntax:

$$\begin{aligned} \mathit{Expr} ::= {} & \mathit{var}(\mathit{VarId}) \mid \mathit{const}(\mathit{Val} \times \mathit{Type}) \mid \mathit{app}(\mathit{FunId} \times \mathit{Expr}) \\ & \mid \mathit{choose}(\mathscr{D}(\mathit{Expr})) \mid \mathit{exp}(\mathit{Expr}) \mid \mathit{dist}(\mathit{DistId} \times \mathit{Expr}) \\ & \mid \mathit{let}(\mathit{Expr} \times \mathit{VarId} \times \mathit{Expr}) \\ & \mid \mathit{nswitch}\big(\mathit{Expr} \times (\llbracket \mathit{num}(\mathit{rat}) \rrbracket_{\mathrm{V}} \uplus \{\mathit{default}\} \nrightarrow \mathit{Expr})\big) \\ & \mid \mathit{iter}(\mathit{Expr} \times \mathit{VarId} \times \mathit{Expr}) \\ & \mid \mathit{tuple}(\mathit{FieldId} \nrightarrow \mathit{Expr}) \mid \mathit{select}(\mathit{Expr} \times \mathit{FieldId}) \\ & \mid \mathit{cons}(\mathit{CaseId} \times \mathit{Expr}) \mid \mathit{cswitch}\big(\mathit{Expr} \times (\mathit{CaseId} \nrightarrow \mathit{VarId} \times \mathit{Expr})\big) \end{aligned}$$

All front-end notation described in Section 3.1 can be translated to this form; the technical details are out of scope here.

- The basic term constructs *var*, *const* and *app* denote variable references, constants, and (deterministic) function applications, respectively. Note that the type of a constant must be specified; in practice it is inferred from the form of a literal. All built-in operators are considered functions in this sense.
- The probability-related constructs *choice*, *exp* and *dist* denote fixed distribution, expectation, and drawing from a named distribution, respectively. Note that *dist* takes an *Expr*, thus the distribution parameters can be chosen dynamically.
- The scoping-related constructs *let* and *iter* denote the binding of a single value and the *flatMap* operation on collections of any shape, respectively.
- The branching-related construct *nswitch* is modeled after the `switch` of the C language family.
- The product-related constructs *tuple* and *select* produce and consume record values, respectively.
- The sum-related constructs *cons* and *cswitch* produce and consume tagged values, respectively. *cswitch* is modeled after pattern matching of functional languages, such as `case-of` in Haskell.

4.2 Type Assignment

There are no type specifiers in Alea end-user code. All relevant type information is inferred in a straightforward bottom-up procedure. The rules are depicted in Figure 3. These deduction rules produce unique types, hence they inductively define a partial *type assignment* function:

$$\llbracket e \rrbracket_{\mathrm{T}}(\Gamma) = \begin{cases} t & \text{if } \Gamma \vdash e : t \\ \text{undefined} & \text{if no such } t \end{cases}$$

4.3 Deterministic Evaluation

The rules for deterministic evaluation, which forms the structural basis for other interpretations, are depicted in Figure 4. Note that there are no rules for evaluating the stochastic constructs *choose*, *exp*

$$\frac{\Gamma(x) = t}{\Gamma \vdash \mathit{var}(x) : t} \qquad \frac{v \in [\![t]\!]_{\mathrm{V}}}{\Gamma \vdash \mathit{const}(v,t) : t} \qquad \frac{\Gamma \vdash e : t \qquad \vdash f : t \to u}{\Gamma \vdash \mathit{app}(f,e) : u}$$

$$\frac{\bigwedge_{k=1}^{n} \Gamma \vdash e_k : t_k}{\Gamma \vdash \mathit{choose}\big(\{e_1 \mapsto p_1, \dots, e_n \mapsto p_n\}\big) : \bigsqcup_{k=1}^{n} t_k}$$

$$\frac{\Gamma \vdash e : t \qquad t \sqsubseteq \mathit{num}(\mathit{rat})}{\Gamma \vdash \mathit{exp}(e) : \mathit{num}(\mathit{rat})} \qquad \frac{\Gamma \vdash e : t \qquad \vdash f : t \to u}{\Gamma \vdash \mathit{dist}(f,e) : u}$$

$$\frac{\Gamma \vdash e : t \qquad \Gamma \oplus \{x \mapsto t\} \vdash e' : t'}{\Gamma \vdash \mathit{let}(e,x,e') : t'}$$

$$\frac{\Gamma \vdash e_0 : t \qquad t \sqsubseteq \mathit{num}(\mathit{rat}) \qquad \vdash t \bullet C = \{e_1, \dots, e_k\} \qquad \bigwedge_{k=1}^{n} \Gamma \vdash e_k : u_k}{\Gamma \vdash \mathit{nswitch}(e_0, C) : \bigsqcup_{k=1}^{n} u_k}$$

$$\frac{\Gamma \vdash e : \mathit{coll}(s,m,t) \qquad s \sqsubseteq s' \qquad \Gamma \oplus \{x \mapsto t\} \vdash e' : \mathit{coll}(s',m',t')}{\Gamma \vdash \mathit{iter}(e,x,e') : \mathit{coll}(s', m \sqcup m', t')}$$

$$\frac{\bigwedge_{k=1}^{n} \Gamma \vdash e_k : t_k}{\Gamma \vdash \mathit{tuple}\big(\{i_1 \mapsto e_1, \dots, i_n \mapsto e_n\}\big) : \mathit{prod}\big(\{i_1 \mapsto t_1, \dots, i_n \mapsto t_n\}\big)}$$

$$\frac{\Gamma \vdash e : \mathit{prod}(T) \qquad T(i) = t}{\Gamma \vdash \mathit{select}(e,i) : t} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \mathit{cons}(i,e) : \mathit{sum}\big(\{i \mapsto t\}\big)}$$

$$\frac{\Gamma \vdash e_0 : \mathit{sum}\big(\{i_1 \mapsto t_1, \dots, i_n \mapsto t_n\}\big) \qquad \bigwedge_{k=1}^{n} \Gamma \oplus \{x_k \mapsto t_k\} \vdash e_k : u_k}{\Gamma \vdash \mathit{switch}\big(e_0, C \oplus \{i_1 \mapsto (x_1, e_1), \dots, i_n \mapsto (x_n, e_n)\}\big) : \bigsqcup_{k=1}^{n} u_k}$$

$$\frac{[\![t]\!]_{\mathrm{V}} \subseteq \mathrm{dom}(C)}{\vdash t \bullet C = C\big([\![t]\!]_{\mathrm{V}}\big)} \qquad \frac{[\![t]\!]_{\mathrm{V}} \not\subseteq \mathrm{dom}(C) \qquad \mathit{default} \in \mathrm{dom}(C)}{\vdash t \bullet C = C\big([\![t]\!]_{\mathrm{V}} \cup \{\mathit{default}\}\big)}$$

Figure 3: Type Inference Rules

$$\frac{E(x) = v}{E \vdash var(x) \rightsquigarrow v} \qquad \frac{}{E \vdash const(v,t) \rightsquigarrow v} \qquad \frac{E \vdash e \rightsquigarrow v \qquad [\![f]\!]_{\mathrm{F}}(v) = v'}{E \vdash app(f,e) \rightsquigarrow v'}$$

$$\frac{E \vdash e \rightsquigarrow v \qquad E \oplus \{x \mapsto v\} \vdash e' \rightsquigarrow v'}{E \vdash let(e,x,e') \rightsquigarrow v'}$$

$$\frac{E \vdash e_0 \rightsquigarrow v_0 \qquad E \vdash C'(v_0) \rightsquigarrow v}{E \vdash nswitch(e_0,C) \rightsquigarrow v} \qquad \text{where } C'(v) = \begin{cases} C(v) & \text{if defined} \\ C(default) & \text{otherwise} \end{cases}$$

$$\frac{E \vdash e \rightsquigarrow theS(v_1,\ldots,v_n) \qquad \bigwedge_{k=1}^{n} E \oplus \{x \mapsto v_k\} \vdash e' \rightsquigarrow v'_k}{E \vdash iter(e,x,e') \rightsquigarrow v'_1 \oplus \cdots \oplus v'_n}$$

$$\frac{\bigwedge_{k=1}^{n} E \vdash e_k \rightsquigarrow v_k}{E \vdash tuple\big(\{i_1 \mapsto e_1,\ldots,i_n \mapsto e_n\}\big) \rightsquigarrow \{i_1 \mapsto v_1,\ldots,i_n \mapsto v_n\}}$$

$$\frac{E \vdash e \rightsquigarrow record(V) \qquad V(i) = v}{E \vdash select(e,i) \rightsquigarrow v} \qquad \frac{E \vdash e \rightsquigarrow v}{E \vdash cons(i,e) \rightsquigarrow tag(i,v)}$$

$$\frac{E \vdash e_0 \rightsquigarrow (i,v_0) \qquad C(i) = (x,e) \qquad E \oplus \{x \mapsto v_0\} \vdash e \rightsquigarrow v}{E \vdash cswitch(e_0,C) \rightsquigarrow v}$$

Figure 4: Deterministic Big-Step Evaluation Semantics

and *dist*. The deduction rules produce unique results, hence they inductively define a partial *evaluation* function:

$$[\![e]\!]_{\mathrm{V}}(E) = \begin{cases} v & \text{if } E \vdash e \rightsquigarrow v \\ \text{undefined} & \text{if no such } v \end{cases}$$

Deterministic evaluation respects types and terminates successfully:

$$\left.\begin{array}{r} [\![e]\!]_{\mathrm{T}}\big(\{x_1 \mapsto t_1,\ldots,x_n \mapsto t_n\}\big) = u \\ \neg\,\mathrm{empty}\, u \\ v_1 \in [\![t_1]\!]_{\mathrm{V}};\ldots;v_n \in [\![t_n]\!]_{\mathrm{V}} \end{array}\right\} \implies [\![e]\!]_{\mathrm{D}}\big(\{x_1 \mapsto v_1,\ldots,x_n \mapsto v_n\}\big) \in [\![u]\!]_{\mathrm{V}}$$

4.4 Stochastic Evaluation

The rules for stochastic static evaluation are given partially in Figure 5. The rules for *const* and *let* have been spelled out, in order to demonstrate that they can be derived from the corresponding deterministic rules by sprinkling in the appropriate natural transformations of the distribution monad; δ and μ are used explicitly, whereas functorial $\mathscr{D}$ is implied. In addition, the rules for the three properly stochastic constructs *choose*, *exp* and *dist* are given. The deduction rules produce unique results, hence they define a partial *distribution* function:

$$[\![e]\!]_{\mathrm{D}}(E) = \begin{cases} P & \text{if } E \vdash e \rightharpoonup P \\ \text{undefined} & \text{if no such } P \end{cases}$$

$$\frac{}{E \vdash const(v,t) \rightharpoonup \delta(v)}$$

$$\frac{\bigwedge_{k=1}^{n} E \vdash e_k \rightharpoonup Q_k}{E \vdash choose\big(\{e_1 \mapsto p_1, \ldots, e_n \mapsto p_n\}\big) \rightharpoonup \mu\big(\{Q_1 \mapsto p_1, \ldots, Q_n \mapsto p_n\}\big)}$$

$$\frac{E \vdash e \rightharpoonup \{const(x_1) \mapsto p_1, \ldots, const(x_n) \mapsto p_n\} \qquad m = \sum_{k=1}^{n} p_n \cdot x_n}{E \vdash exp(e) \rightharpoonup \delta(const(m))}$$

$$\frac{E \vdash e \rightharpoonup P = \{v_1 \mapsto p_1, \ldots, v_n \mapsto p_n\} \qquad \bigwedge_{k=1}^{n} [\![f]\!]_{\mathrm{F}}(v_k) = Q_k}{E \vdash dist(f,e) \rightharpoonup \mu\big(\{Q_1 \mapsto p_1, \ldots, Q_n \mapsto p_n\}\big)}$$

$$\frac{E \vdash e \rightharpoonup P = \{v_1 \mapsto p_1, \ldots, v_n \mapsto p_n\} \qquad \bigwedge_{k=1}^{n} E \oplus \{x \mapsto v_k\} \vdash e' \rightharpoonup Q_k}{E \vdash let(e,x,e') \rightharpoonup \mu\big(\{Q_1 \mapsto p_1, \ldots, Q_n \mapsto p_n\}\big)}$$

Figure 5: Probabilistic Big-Step Evaluation Semantics (Excerpt)

Stochastic evaluation respects types and terminates successfully:

$$\left.\begin{array}{r} [\![e]\!]_{\mathrm{T}}\big(\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}\big) = u \\ \neg\,\mathrm{empty}\,u \\ v_1 \in [\![t_1]\!]_{\mathrm{V}}; \ldots; v_n \in [\![t_n]\!]_{\mathrm{V}} \end{array}\right\} \implies [\![e]\!]_{\mathrm{D}}\big(\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}\big) \in \mathscr{D}\big([\![u]\!]_{\mathrm{V}}\big)$$

4.5 Pseudo-Randomized Evaluation

The rules for pseudo-random static evaluation are given partially in Figure 6. We write $e \xrightarrow{s \to s'} v$ to say that the evaluation of e, with the random generator being in state s, results in value v, with the random generator transitioning to state s'. We assume that the generator supports an elementary operation, random, that selects a number in $1, \ldots, n$ according to an n-tuple of probabilities.

Note that, unlike the preceding rule systems, this semantics is no longer unique; for example, the *tuple* rule does not specify the sequential order in which fields are processed. For reasonable random generators, however, the order should not affect the distribution of results over all possible states.

5 Application Examples

5.1 Introduction Revisited

A slightly biased instance of the singular Roman coin toss experiment could be specified like this:

```
~bernoulli(0.503) ? @head : @ship
```

A specification of the more complex dice-rolling example is depicted in Figure 7. For a discussion of the concepts of notation and how they are used, see the next subsection. The success of winning, by any amount, is calculated by the Alea interpreter as exactly 0.90661502737169.

$$\frac{}{E \vdash \mathit{const}(v,t) \xrightarrow{s \to s} v}$$

$$\frac{\bigwedge_{k=1}^{n} E \vdash e_k \xrightarrow{s_{k-1} \to s_k} v_k}{E \vdash \mathit{tuple}\big(\{i_1 \mapsto e_1, \ldots, i_n \mapsto e_n\}\big) \xrightarrow{s_0 \to s_n} \{i_1 \mapsto v_1, \ldots, i_n \mapsto v_n\}}$$

$$\frac{\mathrm{random}(s; p_1, \ldots, p_n) = (s', k) \qquad E \vdash e_k \xrightarrow{s' \to s''} v}{E \vdash \mathit{choose}\big(\{e_1 \mapsto p_1, \ldots, e_n \mapsto p_n\}\big) \xrightarrow{s \to s''} v}$$

Figure 6: Pseudo-Random Big-Step Evaluation Semantics (Excerpt)

```
dice1 := ⟨ ~uniform{1..10} | _ ← ⟨1..7⟩ ⟩;
tens := ⟨ d | d ← dice1; d = 10 ⟩;
dice2 := ⟨ ~uniform{1..10} | _ ← tens ⟩;
dice := dice1 + dice2;
succs := (+)⟨ d > 5 | d ← dice ⟩;
 fails  := (+)⟨ d = 1 | d ← dice ⟩;
 diff  := succs − fails ;
bad := succs = 0 ∧ fails > 0;
 verdict := diff > 0 ? @win(diff) : (bad ? @botch : @lose)
```

Figure 7: Complex Example from Section 1

```
1  dice := ⟨ ~uniform{1 .. 6} | _ ← ⟨1 .. 5⟩ ⟩;
2  (
3      Dice:          dice,
4      Aces:          (+)⟨ d | d ← dice; d = 1 ⟩,
5      Twos:          (+)⟨ d | d ← dice; d = 2 ⟩,
6      Threes:        (+)⟨ d | d ← dice; d = 3 ⟩,
7      Fours:         (+)⟨ d | d ← dice; d = 4 ⟩,
8      Fives:         (+)⟨ d | d ← dice; d = 5 ⟩,
9      Sixes:         (+)⟨ d | d ← dice; d = 6 ⟩,
10     Chance:        (+)(dice),
11     ThreeofaKind: (+)(dice) ∗ (max(mults(dice)) ≥ 3),
12     FourofaKind:  (+)(dice) ∗ (max(mults(dice)) ≥ 4),
13     FullHouse:     25 ∗ (mults(dice) = ⟨2, 3⟩),
14     SmallStraight: 30 ∗ (dice ≥ ⟨1 .. 4⟩ ∨dice ≥ ⟨2 .. 5⟩ ∨dice ≥ ⟨3 .. 6⟩),
15     LargeStraight: 40 ∗ (dice ≥ ⟨1 .. 5⟩ ∨dice ≥ ⟨2 .. 6⟩),
16     Yahtzee:       50 ∗ (mults(dice) = ⟨5⟩)
17 )
```

Figure 8: Yahtzee Scoring

5.2 Bonus Use Case: Yahtzee

As a real-world application example, we specify the scoring system of the dice game Yahtzee [3]. Here we consider only the scoring of a single throw of five dice, ignoring the further game mechanics, namely selective rerolling, the thirteen rounds, and the bonuses.

The Alea code for the basic scheme is depicted in Figure 8. Some comments on the notation and proposed style:

- Line 1 makes five invocations of the uniform distribution on the set of numbers $\{1,\ldots,6\}$, and collects the results in a bag. The concept of collections of independent, identically distributed (iid) random variables is quite ubiquitous; hence it could deserve a specific notation, but we demonstrate here that its semantics are already subsumed by bag comprehension.
- Lines 2–17 compute various scores depending on the rolled dice values, and store them in a record.
- Lines 4–9 sum only the dice values that are equal to one, two, etc., respectively.
- Lines 10–12 sum all dice values conditionally. Lines 13–16 take fixed values conditionally. A condition is imposed by multiplication with a Boolean expression.
- Lines 11–13 and 17 make use of the *mults* function, a useful statistic that maps a bag of arbitrary values to the bag of their multiplicities, e.g., $\langle a,a,a,b,c\rangle$ to $\langle 3,1,1\rangle$.
- Lines 14–15 use the *superbag* relation, written as $\geq$.

Analysis of the outcomes is possible by subjecting the depicted expression to further computations. For example, to find the probability of obtaining at least 17 points in the category 'three of a kind', take the expectation value of the corresponding Boolean expression:

```
E(dice := ⟨...⟩; (...).ThreeofaKind ≥17)
```

The result, computed by the Alea stochastic interpreter, is $17/144$.

6 Current Implementation State

A processing tool for Alea programs has been implemented in Java. It features a type checker and two interpreters, one for stochastic analysis and one for pseudo-random evaluation.

While this tool is good enough for first evaluations, for both a full-fledged stochastics teaching aid and a game support engine many features are still missing. For the former, *visualizations* of the structure expressions and the distributions of outcomes, and dependency graphs for parameterized experiments would greatly enhance the experience. For the latter, a *server* infrastructure, where game rules can be deployed and queries of various kind can be answered, is needed.

Apart from such user interfaces, the prospect of *compiling* Alea code to some other language, while performing optimizations, has not been explored.

In general, all software related to Alea shall eventually be released as open source.

7 Conclusion

We have presented Alea, a simple functional programming language geared towards the declarative specification of simple random experiments. Due to the particular nature of the application domain, Alea is both lacking many common features of more general FP languages, and equipped with specific novel features to suit the application. As a result of the choice of features and the design of the front-end syntax, the language has a distinctive elementary mathematics-like feeling, and should be easy and fun to use by non-experts in programming.

Due to the deliberate design choice to make Alea Turing-incomplete, plausible use cases where expressive power is objectively lacking are bound to arise. Thus, the library of opaquely predefined functions and distributions is expected to grow, and more design work concerning language extensions for syntactic sugar and reusable user-defined subprograms is needed.

Already in fairly simple example code, combinatorial explosion can arise, in particular in conjunction with *distinguishable* random elements that are collected in lists. While this is no practical problem for pseudo-random simulation at all, the naïve way to compute distributions, even simple marginal ones, directly from the semantic rules can get disappointingly slow. Since Alea is not referentially transparent with regard to randomness, optimization techniques such as lazy evaluation or program slicing cannot be applied straightforwardly. It appears that stochastic independence can be exploited to address this issue, but fundamental research into the matter has only just started.

References

[1] Richard Allenunne (2009): *Coin tossing through the ages*. The Telegraph. Available at `https://www.telegraph.co.uk/news/science/science-news/6911921/Coin-tossing-through-the-ages.html`.

[2] Giuseppe Castagna, Giorgio Ghelli & Giuseppe Longo (2000): *A Calculus for Overloaded Functions with Subtyping*. In Malcolm P. Atkinson & Ray Welland, editors: Fully Integrated Data Environments: Persistent Programming Languages, Object Stores, and Programming Environments, Springer, Berlin, Heidelberg, pp. 196–233, doi:10.1007/978-3-642-59623-0_11.

[3] Edwin S. Lowe (1956): *Yahtzee*. © Milton Bradley, Hasbro.

[4] Eugenio Moggi (1991): *Notions of computation and monads*. Information and Computation 93(1), pp. 55–92, doi:10.1016/0890-5401(91)90052-4. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[5] Abraham De Moivre (1718): *The Doctrine of Chances: or, A Method of Calculating the Probability of Events in Play*. W. Pearson, London.

[6] Mark Rein-Hagen (1991): *Vampire: The Masquerade*. White Wolf.

[7] Alex Shafarenko & Sven-Bodo Scholz (2004): *General homomorphic overloading*. In: *Proceedings of the 16th International Conference on Implementation and Application of Functional Languages*, IFL'04, Springer, Berlin, Heidelberg, p. 195–210, doi:10.1007/11431664_12.

[8] P. Wadler & S. Blott (1989): *How to make ad-hoc polymorphism less ad hoc*. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, Association for Computing Machinery, New York, NY, USA, p. 60–76, doi:10.1145/75277.75283.