ALEA IACTA EST

A Declarative Domain-Specific Language for Manually Performable Random Experiments

Baltasar Trancón y Widemann[†‡] Markus Lepper[†]

[†] semantics gGmbH

[‡] Brandenburg University of Applied Sciences

TFPiE

2025-01-13

AGENDA

Introduction

Design
- Basics
- Syntax
- Semantics

Applications
- Introductory Examples Revisited
- Bonus Examples
- Implementation

Conclusion

Chance!

By Classical Numismatic Group, Inc. http://www.cngcoins.com, CC BY-SA 3.0

- Toss a denarius coin. Determine whether it shows a head or a ship.

Chance!

- 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.

Background

- Random experiments have a history of millenia in human culture.
  - toss coins, cast bones, roll dice, draw from a bag / urn / card deck, ...
- **Stochastics** as a discipline has evolved from their study.

Background

- Random experiments have a history of millenia in human culture.
  - toss coins, cast bones, roll dice, draw from a bag / urn / card deck, ...
- **Stochastics** as a discipline has evolved from their study.
- Random experiments feature prominently
  - in the teaching of stochastics,
  - in simulations,
  - in games.

Goals

- Declarative domain-specific language for random experiments
- First-order functional programming + randomness

Goals

- Declarative domain-specific language for random experiments
- First-order functional programming + randomness
- Rich statistical datatypes, simple control flow
- Clean semantics

Goals

- Declarative domain-specific language for random experiments
- First-order functional programming + randomness
- Rich statistical datatypes, simple control flow
- Clean semantics
- Easy and intuitive to use for non-expert programmers
- Amenable to static stochastic analysis — Turing-incomplete
- Pseudo-random simulator & game assistant

AGENDA

Introduction

Design
- Basics
- Syntax
- Semantics

Applications
- Introductory Examples Revisited
- Bonus Examples
- Implementation

Conclusion

AGENDA

Introduction

Design
- Basics
- Syntax
- Semantics

Applications
- Introductory Examples Revisited
- Bonus Examples
- Implementation

Conclusion

Probability Distributions

- Consider only rational, discrete, finite distributions.

Probability Distributions

- Consider only rational, discrete, finite distributions.
- Stochastically effective: Calculate the distribution of any random variable that is a **computable** function.

Probability Distributions

- Consider only rational, discrete, finite distributions.
- Stochastically effective: Calculate the distribution of any random variable that is a **computable** function.
- Strong algebraic structure: Commutative Monad
  - Moggi-style denotational semantics for a term language

Probability Distributions

- Consider only rational, discrete, finite distributions.
- Stochastically effective: Calculate the distribution of any random variable that is a **computable** function.
- Strong algebraic structure: Commutative Monad
  - Moggi-style denotational semantics for a term language
- Closed under finitary computations.
- ***Not*** closed under unbounded iteration / recursion:

  Repeat rolling a die until it shows a six.

Data

- Exact numbers: **rational**, unbounded precision
  - with IEEE 754-style NaN
  - subtypes: integer, natural, **boolean**

Data

- Exact numbers: **rational**, unbounded precision
  - with IEEE 754-style NaN
  - subtypes: integer, natural, **boolean**
- Collections: lists, sets, **bags**
  - bags relieve combinatorial pressure
  - bulk operations *map*, *filter*, *reduce*

Data

- Exact numbers: **rational**, unbounded precision
  - with IEEE 754-style NaN
  - subtypes: integer, natural, **boolean**
- Collections: lists, sets, **bags**
  - bags relieve combinatorial pressure
  - bulk operations *map*, *filter*, *reduce*
- Records
  - field names or $1 \ldots n$ for tuples

Data

- Exact numbers: **rational**, unbounded precision
  - with IEEE 754-style NaN
  - subtypes: integer, natural, **boolean**
- Collections: lists, sets, **bags**
  - bags relieve combinatorial pressure
  - bulk operations *map*, *filter*, *reduce*
- Records
  - field names or $1 \ldots n$ for tuples
- Tagged Data
  - like ADT constructors, but free without type declaration
  - tagged unit tuple = *enum* constant

AGENDA

Introduction

Design

- Basics
- Syntax
- Semantics

Applications

- Introductory Examples Revisited
- Bonus Examples
- Implementation

Conclusion

Syntax

- Overall appearance: elementary mathematics
  - graphical (Unicode) operators; no keywords
  - strong but implicit type system

Syntax

- Overall appearance: elementary mathematics
  - graphical (Unicode) operators; no keywords
  - strong but implicit type system
- Arithmetics

```
a^2 + b^2 ≥ c^2
```

Syntax

- Overall appearance: elementary mathematics
  - graphical (Unicode) operators; no keywords
  - strong but implicit type system
- Arithmetics

```
a^2 + b^2 ≥ c^2
```

- Local variables

```
s := (a + b + c) / 2; (s − a) * (s − b)
```

Syntax

- Overall appearance: elementary mathematics
  - graphical (Unicode) operators; no keywords
  - strong but implicit type system
- Arithmetics

```
a^2 + b^2 ≥ c^2
```

- Local variables

```
s := (a + b + c) / 2; (s − a) * (s − b)
```

- Functions & distributions

```
max(gcd(a, b), 17)    k := ~uniform{1, 2, 3, 4}
```

Syntax

- Overall appearance: elementary mathematics
  - graphical (Unicode) operators; no keywords
  - strong but implicit type system
- Arithmetics

```
a^2 + b^2 ≥ c^2
```

- Local variables

```
s := (a + b + c) / 2; (s − a) * (s − b)
```

- Functions & distributions

```
max(gcd(a, b), 17)    k := ~uniform{1, 2, 3, 4}
```

  - distributions are ***not*** referentially transparent:

```
n := ~uniform{+1, −1}; n − n
```

Collections

- Unified treatment of lists, sets and bags
  - homogeneous elements
  - each with a distinguished shape of brackets

```
[1, 2, 3]    {1, 2, 3}    ⟨1, 2, 3⟩
```

Collections

- Unified treatment of lists, sets and bags
  - homogeneous elements
  - each with a distinguished shape of brackets

`[1, 2, 3]` `{1, 2, 3}` `⟨1, 2, 3⟩`

- Bulk operations *`map`*, *`filter`*, *`reduce`*
  - without functions-as-values

Collections

- Unified treatment of lists, sets and bags
  - homogeneous elements
  - each with a distinguished shape of brackets

```
[1, 2, 3]    {1, 2, 3}    ⟨1, 2, 3⟩
```

- Bulk operations *map*, *filter*, *reduce*
  - without functions-as-values
  - *map* & *filter* by comprehensions (Haskell-style +)

```
[ n^2 | n ← L ]    { x ← S | x ≥ 0 }
```

Collections

- Unified treatment of lists, sets and bags
  - homogeneous elements
  - each with a distinguished shape of brackets

```
[1, 2, 3]    {1, 2, 3}    ⟨1, 2, 3⟩
```

- Bulk operations *map*, *filter*, *reduce*
  - without functions-as-values
  - *map* & *filter* by comprehensions (Haskell-style +)

```
[ n^2 | n ← L ]    { x ← S | x ≥ 0 }
```

  - drawing with or **without** replacement

```
{ x + y | x ← S; y ← S }    { x + y | {x, y} ← S }
```

Collections

- Unified treatment of lists, sets and bags
  - homogeneous elements
  - each with a distinguished shape of brackets

```
[1, 2, 3]    {1, 2, 3}    ⟨1, 2, 3⟩
```

- Bulk operations *map*, *filter*, *reduce*
  - without functions-as-values
  - *map* & *filter* by comprehensions (Haskell-style +)

    ```
    [ n^2 | n ← L ]    { x ← S | x ≥ 0 }
    ```

  - drawing with or **without** replacement

    ```
    { x + y | x ← S; y ← S }    { x + y | {x, y} ← S }
    ```

  - *reduce* by semigroup operations

    ```
    min[a, b, c, d]    (+)⟨ x ≥ 0 | x ← B ⟩
    ```

Records and Tuples

- Ad-hoc heterogeneous aggregation
  - no type declaration required

Records and Tuples

- Ad-hoc heterogeneous aggregation
  - no type declaration required
- Positional and/or symbolic field selectors (ML style tuples)

```
(x: 1, y: 2)    (3, 4, 5)    p.x    t.#1
```

Records and Tuples

- Ad-hoc heterogeneous aggregation
  - no type declaration required
- Positional and/or symbolic field selectors (ML style tuples)

```
(x: 1, y: 2)    (3, 4, 5)    p.x    t.#1
```

- Patterns in assignments

```
(a, b, c) := (3, 4, 5)
```

Tagged Values and Case Distinctions

- Ad-hoc disjoint union
  - no type declaration required

Tagged Values and Case Distinctions

- Ad-hoc disjoint union
  - no type declaration required
- Symbolic tags, payload optional (*enum*)

```
@just(42)    @nothing
```

Tagged Values and Case Distinctions

- Ad-hoc disjoint union
  - no type declaration required
- Symbolic tags, payload optional (*enum*)

```
@just(42)    @nothing
```

- Case distinction with elementary pattern matching

```
x ? { @just(n) → n; @nothing → 0 }
```

Tagged Values and Case Distinctions

- Ad-hoc disjoint union
  - no type declaration required
- Symbolic tags, payload optional (*enum*)

```
@just(42)    @nothing
```

- Case distinction with elementary pattern matching

```
x ? { @just(n) → n; @nothing → 0 }
```

  - also for integers (C-style *switch*)

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

Tagged Values and Case Distinctions

- Ad-hoc disjoint union
  - no type declaration required
- Symbolic tags, payload optional (*enum*)

```
@just(42)    @nothing
```

- Case distinction with elementary pattern matching

```
x ? { @just(n) → n; @nothing → 0 }
```

  - also for integers (C-style *switch*)

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

  - special case: booleans

```
a ? b : c    a ? { 1 → b; 0 → c }
```

AGENDA

Introduction

Design

- Basics
- Syntax
- Semantics

Applications

- Introductory Examples Revisited
- Bonus Examples
- Implementation

Conclusion

Types

- Untyped data universe

Types

- Untyped data universe
- Lattice of types
  - structural subtyping for collections, products, sums
  - simple set-theoretic (extensional) semantics

Types

- Untyped data universe
- Lattice of types
  - structural subtyping for collections, products, sums
  - simple set-theoretic (extensional) semantics
- Flexible overloading for library functions
  - homomorphic overloading for arithmetics
  - parametric overloading for collections
  - semigroup overloading for *reduce*

Types

- Untyped data universe
- Lattice of types
  - structural subtyping for collections, products, sums
  - simple set-theoretic (extensional) semantics
- Flexible overloading for library functions
  - homomorphic overloading for arithmetics
  - parametric overloading for collections
  - semigroup overloading for *reduce*
- Decidable, bottom-up inference of principal types

Types

- Untyped data universe
- Lattice of types
  - structural subtyping for collections, products, sums
  - simple set-theoretic (extensional) semantics
- Flexible overloading for library functions
  - homomorphic overloading for arithmetics
  - parametric overloading for collections
  - semigroup overloading for *reduce*
- Decidable, bottom-up inference of principal types
- Sanity checks
  - function argument types
  - *switch* reachability/completeness
  - *reduce* laws
  - ...

Function Overloading

- Each function has a single *partial* definition on the untyped universe.
  - overloading concerns only *total* restrictions (type signatures)
  - consequence: `/` and `//` (*div*) must be distinct functions

Function Overloading

- Each function has a single *partial* definition on the untyped universe.
  - overloading concerns only *total* restrictions (type signatures)
  - consequence: `/` and `//` (*div*) must be distinct functions
- A function may admit several valid type signatures (polymorphism):

  ad-hoc finitely many
  parametric infinitely many

Function Overloading

- Each function has a single *partial* definition on the untyped universe.
  - overloading concerns only *total* restrictions (type signatures)
  - consequence: `/` and `//` (*div*) must be distinct functions
- A function may admit several valid type signatures (polymorphism):

  ad-hoc finitely many
  parametric infinitely many
- Compactness: for a particular argument of type T', only finitely many signatures $T_i \to U_i$ are relevant
  - cover T' with $\bigsqcap T_i \implies$ result is $U' = \bigsqcap U_i$

Function Overloading

- Each function has a single *partial* definition on the untyped universe.
  - overloading concerns only *total* restrictions (type signatures)
  - consequence: `/` and `//` (*div*) must be distinct functions
- A function may admit several valid type signatures (polymorphism):

  ad-hoc finitely many
  parametric infinitely many
- Compactness: for a particular argument of type T', only finitely many signatures $T_i \to U_i$ are relevant
  - cover T' with $\bigsqcap T_i \implies$ result is $U' = \bigsqcap U_i$
- Example:

$$\begin{aligned} \max &: \mathbb{Z} \times \mathbb{N} \to \mathbb{N} & \max(3,4) &: \mathbb{N} \sqcap \mathbb{N} \sqcap \mathbb{Z} \sqcap \mathbb{Q} \\ \max &: \mathbb{N} \times \mathbb{Z} \to \mathbb{N} & \max(3,-4) &: \mathbb{N} \sqcap \mathbb{Z} \sqcap \mathbb{Q} \\ \max &: \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} & \max(-3,-4) &: \mathbb{Z} \sqcap \mathbb{Q} \\ \max &: \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q} & \max(22/7,-4) &: \mathbb{Q} \end{aligned}$$

Semigroups

- Apply a binary function/operator to a collection (instead of a pair of elements).
- Idea: *reduce*

```
(+)[a, b, c, d]    a + b + c + d
```

Semigroups

- Apply a binary function/operator to a collection (instead of a pair of elements).
- Idea: *reduce*

```
(+)[a, b, c, d]    a + b + c + d
```

- Simplified semantics:
  - no *foldl*/*foldr* distinction
  - no explicit empty case

Semigroups

- Apply a binary function/operator to a collection (instead of a pair of elements).
- Idea: *reduce*

```
(+)[a, b, c, d]    a + b + c + d
```

- Simplified semantics:
  - no *foldl*/*foldr* distinction
  - no explicit empty case
- Rely on algebraic (semigroup) properties:

associative	for all collections
commutative	for sets & bags
monoid	for possibly empty collections

Semigroups

- Apply a binary function/operator to a collection (instead of a pair of elements).
- Idea: *reduce*

```
(+)[a, b, c, d]    a + b + c + d
```

- Simplified semantics:
  - no *foldl*/*foldr* distinction
  - no explicit empty case
- Rely on algebraic (semigroup) properties:

associative	for all collections
commutative	for sets & bags
monoid	for possibly empty collections

- Known magically for library functions; currently no mechanism for user definitions.

Evaluation

- Big-step evaluation semantics

Evaluation

- Big-step evaluation semantics
- Three flavors: deterministic, stochastic, pseudo-random
  - choice of computational monad
  - fun fact: in OO implementation, pseudo-random is almost for free

Evaluation

- Big-step evaluation semantics
- Three flavors: deterministic, stochastic, pseudo-random
  - choice of computational monad
  - fun fact: in OO implementation, pseudo-random is almost for free
- Strong normalization for well-typed programs
  - evaluation succeeds for well-typed expressions of nonempty types
  - results in / distributed-over the type inhabitants

Evaluation

- Big-step evaluation semantics
- Three flavors: deterministic, stochastic, pseudo-random
  - choice of computational monad
  - fun fact: in OO implementation, pseudo-random is almost for free
- Strong normalization for well-typed programs
  - evaluation succeeds for well-typed expressions of nonempty types
  - results in / distributed-over the type inhabitants
  - challenge: pseudo-random evaluation & the law of large numbers

AGENDA

Introduction

Design
- Basics
- Syntax
- Semantics

Applications
- Introductory Examples Revisited
- Bonus Examples
- Implementation

Conclusion

AGENDA

Introduction

Design
- Basics
- Syntax
- Semantics

Applications
- Introductory Examples Revisited
- Bonus Examples
- Implementation

Conclusion

Example #1

```
coin := ~uniform{@head, @ship}
```

Example #2

```
dice1 := ⟨~uniform{1..10} | _ ← ⟨1..7⟩⟩;
tens  := ⟨d ← dice1 | d = 10⟩;
dice2 := ⟨~uniform{1..10} | _ ← tens⟩;
dice  := dice1 + dice2;
nhigh := (+)⟨d > 5 | d ← dice⟩;
nones := (+)⟨d = 1 | d ← dice⟩;
diff  := nhigh − nones;

verdict := diff > 0
  ? @succeed(diff)
  : (nhigh = 0 ∧ nones > 0 ? @botch
                           : @fail)
```

Roll seven ten-sided dice.
For every die that shows a ten, roll another
and add it to the pool.
Count the number > 5.
Compare to the number of dice that show 1.

If the difference is positive, you win by that amount.
If there are no values > 5 *but there is a* 1*, you lose badly.*

Example #2

```
dice1 := ⟨~uniform{1..10} | _ ← ⟨1..7⟩⟩;
tens  := ⟨d ← dice1 | d = 10⟩;
dice2 := ⟨~uniform{1..10} | _ ← tens⟩;
dice  := dice1 + dice2;
nhigh := (+)⟨d > 5 | d ← dice⟩;
nones := (+)⟨d = 1 | d ← dice1⟩;
diff  := nhigh − nones;

verdict := diff > 0
  ? @succeed(diff)
  : (nhigh = 0 ∧ nones > 0 ? @botch
                           : @fail)
```

Roll seven ten-sided dice.
For every die that shows a ten, roll another
and add it to the pool.
Count the number > 5.
Compare to the number of ***original*** *dice that show* 1.

If the difference is positive, you win by that amount.
If there are no values > 5 *but there is a* 1*, you lose badly.*

AGENDA

Introduction

Design
- Basics
- Syntax
- Semantics

Applications
- Introductory Examples Revisited
- Bonus Examples
- Implementation

Conclusion

Yahtzee Scoring

```
dice := ⟨ ~uniform{1 .. 6} | _ ← ⟨1 .. 5⟩ ⟩;
(
  Dice:          dice,
  Aces:          (+)⟨ d ← dice | d = 1 ⟩,
  Twos:          (+)⟨ d ← dice | d = 2 ⟩,
  Threes:        (+)⟨ d ← dice | d = 3 ⟩,
  Fours:         (+)⟨ d ← dice | d = 4 ⟩,
  Fives:         (+)⟨ d ← dice | d = 5 ⟩,
  Sixes:         (+)⟨ d ← dice | d = 6 ⟩,
  Chance:        (+)(dice),
  ThreeofaKind:  (+)(dice) * (max(mults(dice)) ≥ 3),
  FourofaKind:   (+)(dice) * (max(mults(dice)) ≥ 4),
  FullHouse:     25 * (mults(dice) = ⟨2, 3⟩),
  SmallStraight: 30 * (dice ≥ ⟨1 .. 4⟩ ∨ dice ≥ ⟨2 .. 5⟩ ∨ dice ≥ ⟨3 .. 6⟩),
  LargeStraight: 40 * (dice ≥ ⟨1 .. 5⟩ ∨ dice ≥ ⟨2 .. 6⟩),
  Yahtzee:       50 * (mults(dice) = ⟨5⟩)
)
```

math.stackexchange.org #5019285

How can I tackle this probability question on deck of cards with replacement?

math.stackexchange.org #5019285

How can I tackle this probability question on deck of cards with replacement?

- Posted 2025-01-04
- Formalized in ALEA 2025-01-04

math.stackexchange.org #5019285

How can I tackle this probability question on deck of cards with replacement?

- Posted 2025-01-04
- Formalized in ALEA 2025-01-04
- Findings:
  - development time ~ 20 min
  - 17 LoC, no corrections necessary
  - complete formal model no longer than high-level English text
  - analysis result agrees with top answer
  - already indicates that user-defined functions are missing

AGENDA

Introduction

Design
- Basics
- Syntax
- Semantics

Applications
- Introductory Examples Revisited
- Bonus Examples
- Implementation

Conclusion

Implementation

- Java application (~ 5 kLoC)
- Lexer & parser: combinator libraries
- Semantic analysis: visitor pattern

Implementation

- Java application (~ 5 kLoC)
- Lexer & parser: combinator libraries
- Semantic analysis: visitor pattern

- DEMO

AGENDA

Introduction

Design
- Basics
- Syntax
- Semantics

Applications
- Introductory Examples Revisited
- Bonus Examples
- Implementation

Conclusion

Summary

- Elementary functional programming with randomness
  - distributions as “anonymous random variables”
  - statistic datatypes & operations

Summary

- Elementary functional programming with randomness
  - distributions as “anonymous random variables”
  - statistic datatypes & operations
- Declarative, but not referentially transparent
  - compare functional–logic programming
  - strong implicit typing catches errors, without getting in the way

Summary

- Elementary functional programming with randomness
  - distributions as “anonymous random variables”
  - statistic datatypes & operations
- Declarative, but not referentially transparent
  - compare functional–logic programming
  - strong implicit typing catches errors, without getting in the way
- Language design emulates traditional mathematics
  - low threshold for non-programmers
- Random experiment idioms facilitated by special features:
  - numeric booleans, counting
  - bag comprehensions
  - structurally free records & tagged unions

Summary

- Elementary functional programming with randomness
  - distributions as “anonymous random variables”
  - statistic datatypes & operations
- Declarative, but not referentially transparent
  - compare functional–logic programming
  - strong implicit typing catches errors, without getting in the way
- Language design emulates traditional mathematics
  - low threshold for non-programmers
- Random experiment idioms facilitated by special features:
  - numeric booleans, counting
  - bag comprehensions
  - structurally free records & tagged unions
- Implementation in Java (very nearly open source)

Future Work

- Grow the library
  - more pre-defined functions

Future Work

- Grow the library
  - more pre-defined functions
- User-definable subprograms
  - type signatures, determinism, ...

Future Work

- Grow the library
  - more pre-defined functions
- User-definable subprograms
  - type signatures, determinism, ...
- Visualization of distributions
  - histograms, pie charts, ...

Future Work

- Grow the library
  - more pre-defined functions
- User-definable subprograms
  - type signatures, determinism, ...
- Visualization of distributions
  - histograms, pie charts, ...
- Code generation
  - embed in other applications

Future Work

- Grow the library
  - more pre-defined functions
- User-definable subprograms
  - type signatures, determinism, ...
- Visualization of distributions
  - histograms, pie charts, ...
- Code generation
  - embed in other applications
- Optimization
  - tune the runtime environment
  - deforestation of collections
  - stochastic independence vs. referential transparency

Collaborations Welcome!

- Curious to try ALEA in teaching?
- Thinking of some interesting example application?
- Suggestions for syntax, semantics or library?
- Contribute a visualization, UI or tool connection?
- ...

`http://bandm.eu/metatools/doc/usage/getit.html`

AGENDA

Formal Stuff

Types

$$
\begin{aligned}
\mathit{Type} ::= {} & \mathit{any} \mid \mathit{none} \\
\mid {} & \mathit{num}(\mathit{Num}) \mid \mathit{coll}(\mathit{Shape}, \mathit{Mode}, \mathit{Type}) \\
\mid {} & \mathit{prod}(\mathit{FieldId} \nrightarrow \mathit{Type}) \mid \mathit{sum}(\mathit{CaseId} \nrightarrow \mathit{Type})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Num} &::= \mathit{bool} \mid \mathit{nat} \mid \mathit{int} \mid \mathit{rat} \\
\mathit{Shape} &::= \mathit{list} \mid \mathit{bag} \mid \mathit{set} \\
\mathit{Mode} &::= \mathit{pos} \mid \mathit{opt}
\end{aligned}
$$

Untyped Values

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

Extensions

$$[\![\mathsf{none}]\!]_{\mathrm{V}} = \varnothing \qquad [\![\mathsf{any}]\!]_{\mathrm{V}} = \mathsf{Val}$$

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

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

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

Empty or Inhabited?

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

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

Subtyping

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

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

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

Abstract Syntax

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

Type Assignment (1/2)

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

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

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

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

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

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

Type Assignment (2/2)

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

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

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

$$\frac{[\![t]\!]_{\mathrm{V}} \subseteq \mathsf{dom}(C)}{\vdash t \bullet C = C([\![t]\!]_{\mathrm{V}})} \qquad \frac{[\![t]\!]_{\mathrm{V}} \notin \mathsf{dom}(C) \qquad \mathsf{default} \in \mathsf{dom}(C)}{\vdash t \bullet C = C([\![t]\!]_{\mathrm{V}} \cup \{\mathsf{default}\})}$$

Evaluation (Deterministic)

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

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

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

$$\frac{E \vdash e \rightsquigarrow \mathrm{the}\, S(v_1, \ldots, v_n) \qquad \bigwedge_{k=1}^{n} E \oplus \{x \mapsto v_k\} \vdash e' \rightsquigarrow v_k'}{E \vdash \mathrm{iter}(e,x,e') \rightsquigarrow v_1' \oplus \cdots \oplus v_n'}$$

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

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

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

Evaluation (Stochastic)

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

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

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

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

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

Evaluation (Pseudo-Random)

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

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

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