Language-Level Support for Co-Creative Programming

Dr. Chris Martens

Director, POEM Lab

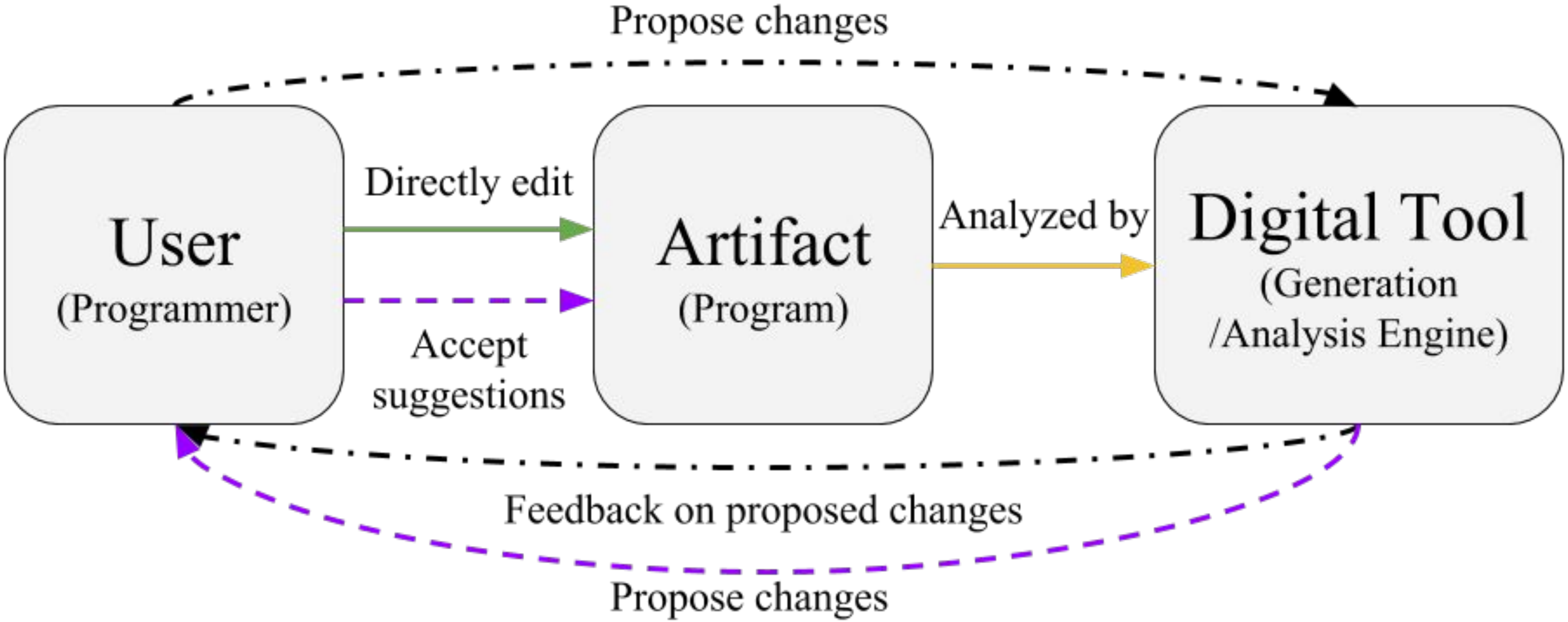
Principles of Expressive Machines

Assistant Professor, Computer Science

NC STATE UNIVERSITY

Or, PCG + FP = 💖

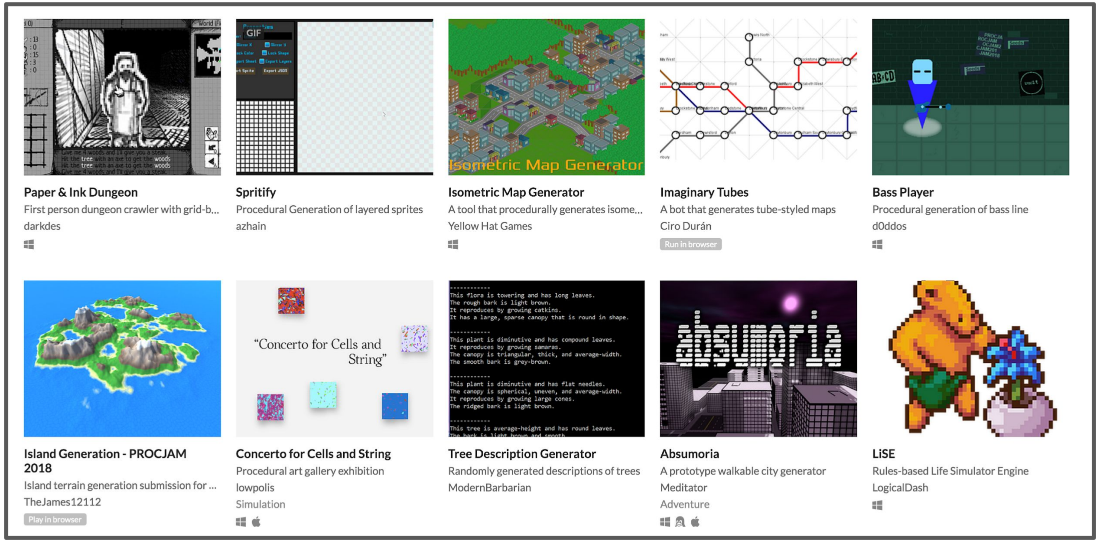
Motivation: Co-Creative Programming



Procedural Content Generation (PCG)

`Generation of art and artifacts through automatic processes`

`Community: ICCC, PCG Workshop, PROCJAM`



Paper & Ink Dungeon
First person dungeon crawler with grid-b...
darkdes

Spritify
Procedural Generation of layered sprites
azhain

Isometric Map Generator
A tool that procedurally generates isome...
Yellow Hat Games

Imaginary Tubes
A bot that generates tube-styled maps
Ciro Durán
Run in browser

Bass Player
Procedural generation of bass line
d0ddos

Island Generation - PROCJAM 2018
Island terrain generation submission for ...
TheJames12112
Play in browser

Concerto for Cells and String
Procedural art gallery exhibition
lowpolis
Simulation

Tree Description Generator
Randomly generated descriptions of trees
ModernBarbarian

Absumoria
A prototype walkable city generator
Meditator
Adventure

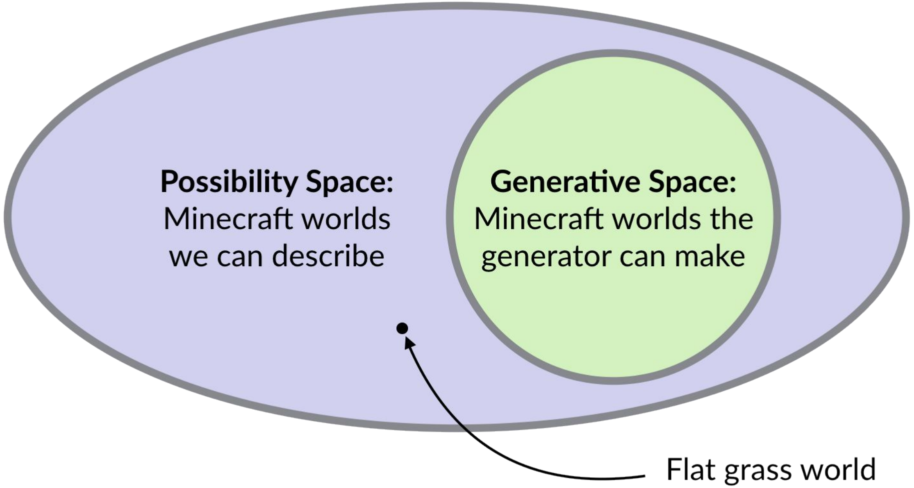
LiSE
Rules-based Life Simulator Engine
LogicalDash

Goals for PCG Language Support

- Make it easier for artists and designers to access PCG techniques.
- Abstractions that support reasoning about generative spaces and their properties
- Provable guarantees about generators: termination, running time, optimality, well-formedness of results, etc.
- Integration with general purpose programming languages

The Generative Space

http://www.possibilityspace.org/tutorial-generative-possibility-space/



Possibility Space:
Minecraft worlds
we can describe

Generative Space:
Minecraft worlds the
generator can make

Flat grass world

This Talk's Key Idea

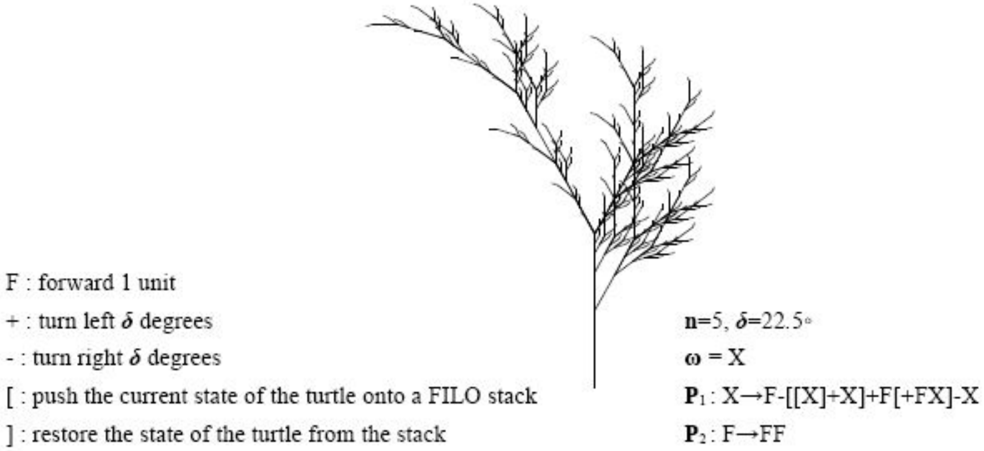
Generative spaces as types
support a unified account of grammar-based PCG, pattern matching, and constraint-based synthesis.

Outline

- Grammar-based PCG
- Deductive grammars (grammars as types)
- In progress: **Sestina** language design
- Future Work

Grammar-Based PCG

L-systems: biologically-inspired plant generation



F : forward 1 unit

+ : turn left δ degrees

- : turn right δ degrees

[: push the current state of the turtle onto a FILO stack

] : restore the state of the turtle from the stack

$\mathbf{n}=5,\ \delta=22.5^{\circ}$

$\omega = X$

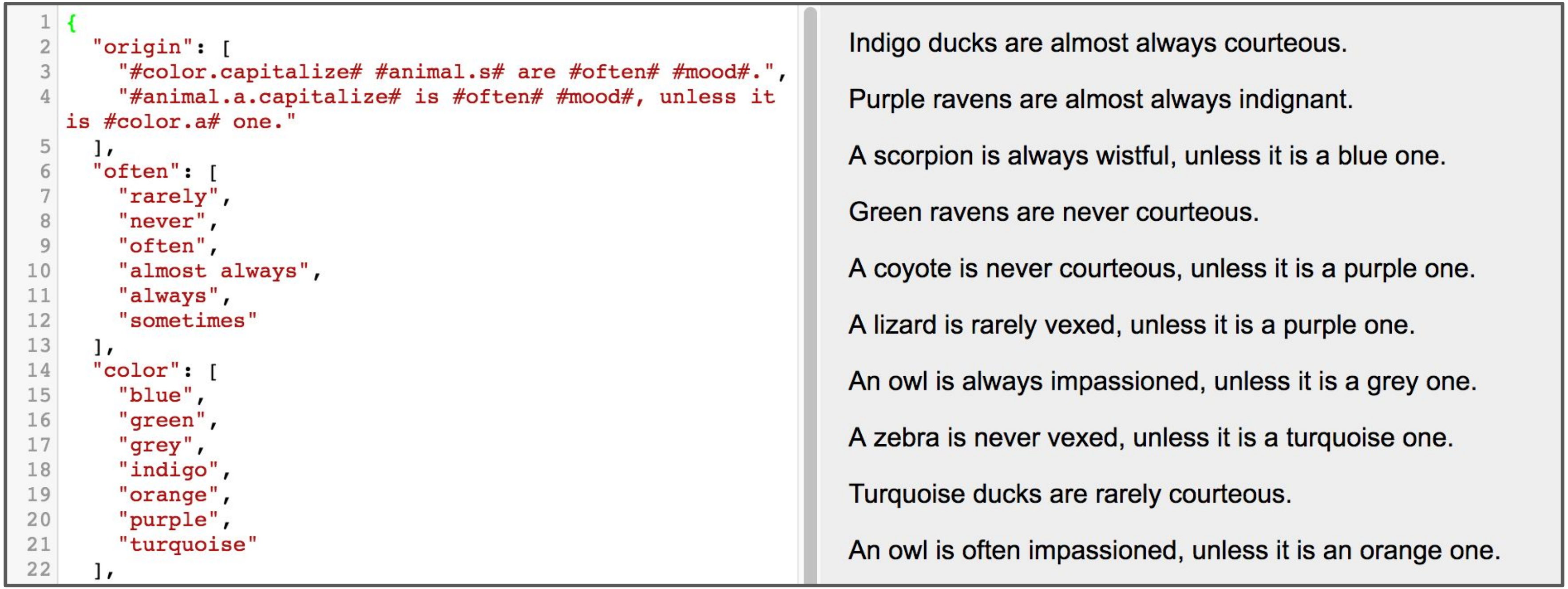
$\mathbf{P}_1 : X \rightarrow F-[[X]+X]+F[+FX]-X$

$\mathbf{P}_2 : F \rightarrow FF$

Tracery (text generation)

Tracery.io

https://beaugunderson.com/tracery-writer



```
{
  "origin": [
    "#color.capitalize# #animal.s# are #often# #mood#.",
    "#animal.a.capitalize# is #often# #mood#, unless it is #color.a# one."
  ],
  "often": [
    "rarely",
    "never",
    "often",
    "almost always",
    "always",
    "sometimes"
  ],
  "color": [
    "blue",
    "green",
    "grey",
    "indigo",
    "orange",
    "purple",
    "turquoise"
  ],
```

```
Indigo ducks are almost always courteous.

Purple ravens are almost always indignant.

A scorpion is always wistful, unless it is a blue one.

Green ravens are never courteous.

A coyote is never courteous, unless it is a purple one.

A lizard is rarely vexed, unless it is a purple one.

An owl is always impassioned, unless it is a grey one.

A zebra is never vexed, unless it is a turquoise one.

Turquoise ducks are rarely courteous.

An owl is often impassioned, unless it is an orange one.
```

Tracery (text generation)

Cheap Bots Done Quick

a strange voyage @str_voyage · 3h
In an instant, day becomes night. We will not sleep tonight.
19 39

a strange voyage @str_voyage · 4h
Our rickety boat requires repairs.
3 10

a strange voyage @str_voyage · 4h
In an instant, day becomes night. We fasten our belongings to the deck.
5 19

a strange voyage @str_voyage · 5h
A vibrant sunset.
10 21

a strange voyage @str_voyage · 5h
The water.
5 18

a strange voyage @str_voyage · 6h
Obsidian cliffs loom over our raft.
3 13

Recursive Story Grammar

"origin": ["Once upon a time, **#story#**"]

"story": ["#hero# the #heroJob# #setSailForAdventure#. **#openBook#**"]

"openBook": ["An old #occupation# told #hero# a story. 'Listen well' she said to #heroThem#, 'to this #strange# #tale#. ' #origin#'","#hero# went home.","#hero# found an ancient book and opened it. As #heroThey# read, the book told #strange.a# #tale#: **#origin#**"]

Environment bindings

```
"origin": ["Once upon a time, #[#setCharacter#]story#"]

"setCharacter": ["[#setPronouns#][hero:#name#][heroJob:#occupation#]"]

"setPronouns": ["[heroThey:they][heroThem:them][heroTheir:their][heroTheirs:theirs]","[heroThey:she][heroThem:her][heroTheir:her][heroTheirs:hers]","[heroThey:he][heroThem:him][heroTheir:his][heroTheirs:his]"]

"setSailForAdventure": ["set sail for adventure","left #heroTheir# home","set out for adventure","went to seek #heroTheir# forture"]
```

Example of a bug

#heroThey# when not defined….

Nondeterministic => only see it sometimes…

Goal: use types to eliminate errors like this

Deductive Grammars

Grammars, formally

$G = \langle N, \Sigma, P, S \rangle$

N	nonterminals, e.g. **Origin**, **Color**
Σ	terminals, e.g. “cat”, “hello”, …
P	production rules, e.g. “**Origin** -> **Color Animal** are **Often Mood**”
S	start symbol, e.g. **Origin**

Grammars, formally

Grammar expressions

α ::= ε | A α | t α (for A in N, t in Sigma)

Strings s ::= ε | t | e^e

Judgment “s matches α”

Deductive Grammars

Nonterminals N correspond to named types

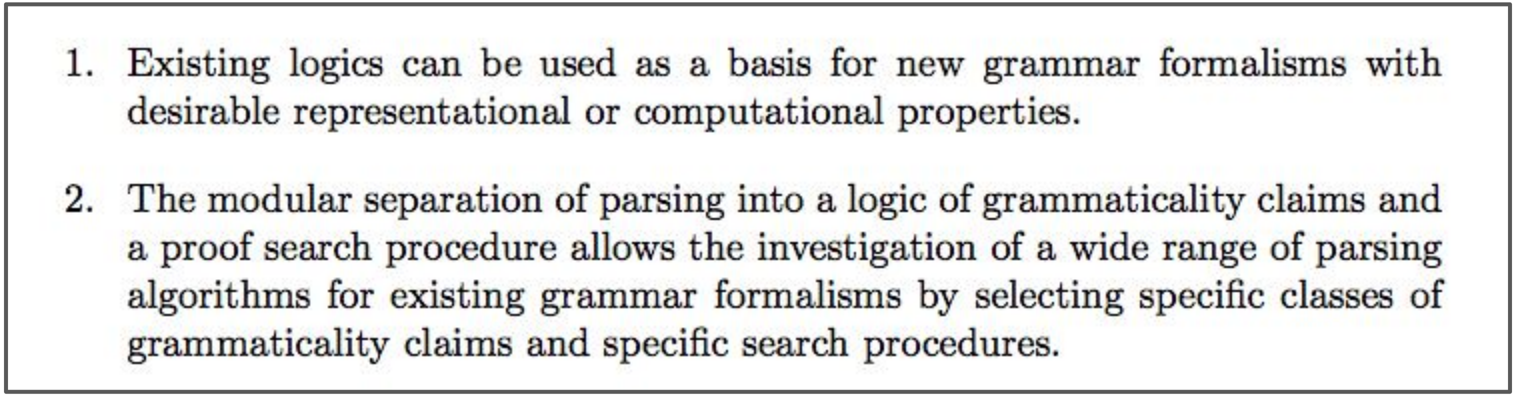
Type checking:

$$\frac{A \text{ -> } \alpha \text{ in } P \qquad e \text{ matches } \alpha}{e : A}$$

Deductive Parsing

Principles and Implementation of Deductive Parsing

Shieber et al., J. Logic Programming 1995



1. Existing logics can be used as a basis for new grammar formalisms with desirable representational or computational properties.
2. The modular separation of parsing into a logic of grammaticality claims and a proof search procedure allows the investigation of a wide range of parsing algorithms for existing grammar formalisms by selecting specific classes of grammaticality claims and specific search procedures.

Deductive Grammars

Beyond strings -- derived rules for sums and products

```
A -> e1 | ... | en

==>

defprop A = OR {TAG1 : |e1| … TAGn : |en|}
```

Expansion alternatives as sums, string concat generalized to products

Deductive Grammars

Beyond strings -- derived rules for sums and products

```
A -> e1 | ... | en
```

==>

```
defprop A = OR {TAG1 : |e1| … TAGn : |en|}
```

Expansion alternatives as sums, string concat generalized to products

Why? So we can **pattern match and project** on generated data with static safety & coverage guarantees

Sestina Language Design

Finite Types

Acknowledgements: Tiannan Chen and Stephen Guy for the example (check out their C++ embedded DSL, GIGL!)

Base types: sets of strings, ranges of numbers

Sums, products, and unit

```
gentype monsterBreed = OR {"direwolf", "dragon", "vampire"}

gentype strongRange = range(5,9) (* = OR {5, 6, 7, 8, 9} *)

gentype weakRange = range(1,4)

gentype monster = AND {tp : monsterBreed,
                       dmg : OR {weakRange,strongRange}}
```

Case Analysis and Projection

```
letgen m : monster in
    case m.tp of
        “dragon” => “The dragon breathes fire at you for ”
                     + m.dmg + “ damage”
        | _ => (*...*)
```

Pronouns example

```
story : string =
  LETGEN
    heroName : name,
    pronouns : pronouns,
    CONCAT
      "Our hero ", heroName, " went into the dungeon to find treasure.",
      pronouns.they, " descended into the final cave, drew ",
      pronouns.their, " sword, and fought the beast who faced ",
      pronouns.them, "."
```

Pronouns example: subtyping/singleton types?

```
gentype pronoun_set = AND {they: string, them: string, their: string}

they_pronouns <: pronoun_set = AND {they: "they", them: "them", their: "their"}
she_pronouns <: pronoun_set = AND {they: "she", them: "her", their: "her"}
he_pronouns <: pronoun_set = AND {they: "he", them: "him", their: "his"}

pronouns <: pronoun_set = OR {they_pronouns, she_pronouns, he_pronouns}
```

Shuffling as a stream

```
(* Primitive for shuffling: turn any gentype into a random
stream
    shuffle : t:gentype => (unit -> t option)  *)

val draw = shuffle card

(* Turn a finite type into a list *)
fun addAll () =
  case draw() of
    SOME c => c::(addAll ())
  | NONE => []
```

Implementation and status

Embedded DSL in Standard ML

https://github.com/chrisamaphone/sestina

Tiny 68-line interpreter (no external syntax yet)

Sums, products, string/range base types, projection, case analysis/pattern matching through SML

Design goals: syntax, recursive types, type signatures, optional labels, base type operations, more base types

Future Work

Probabilities and distributions

(GIGL Syntax)

```
generate DungeonMonster with <* DungeonMonster:
    Monster := weak @ {0.6} | strong @ {0.4},
    Weapon := club @ {0.7} | flail @ {0.3} *>;
```

How likely is a weak monster with a club?

What is the expected value of the monster’s attack damage?

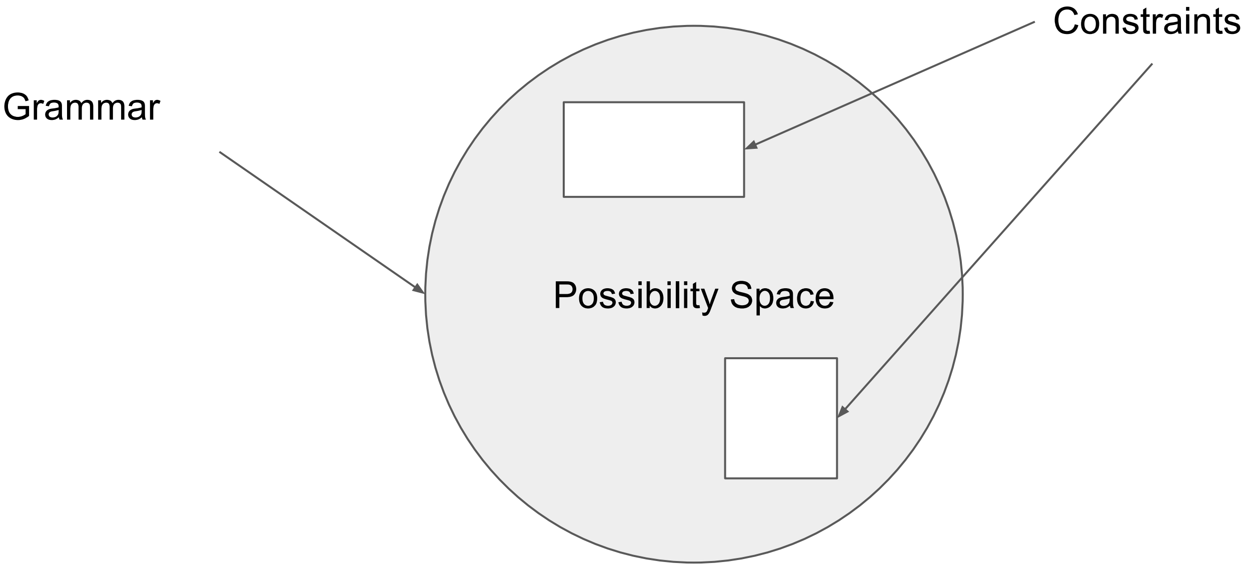
Constructive vs. Subtractive Methods

Grammars: pros: easy to author. Cons: hard to control/refine to only produce the things you want.

A common approach:

- Use a grammar to define a possibility space
- Use a **search-based method** to search through that space for exemplars which meet certain constraints or optimize certain criteria (genetic algorithms, constraint programming, etc.)

Constructive vs. Subtractive Methods



Grammar

Constraints

Possibility Space

Dependent Range Types

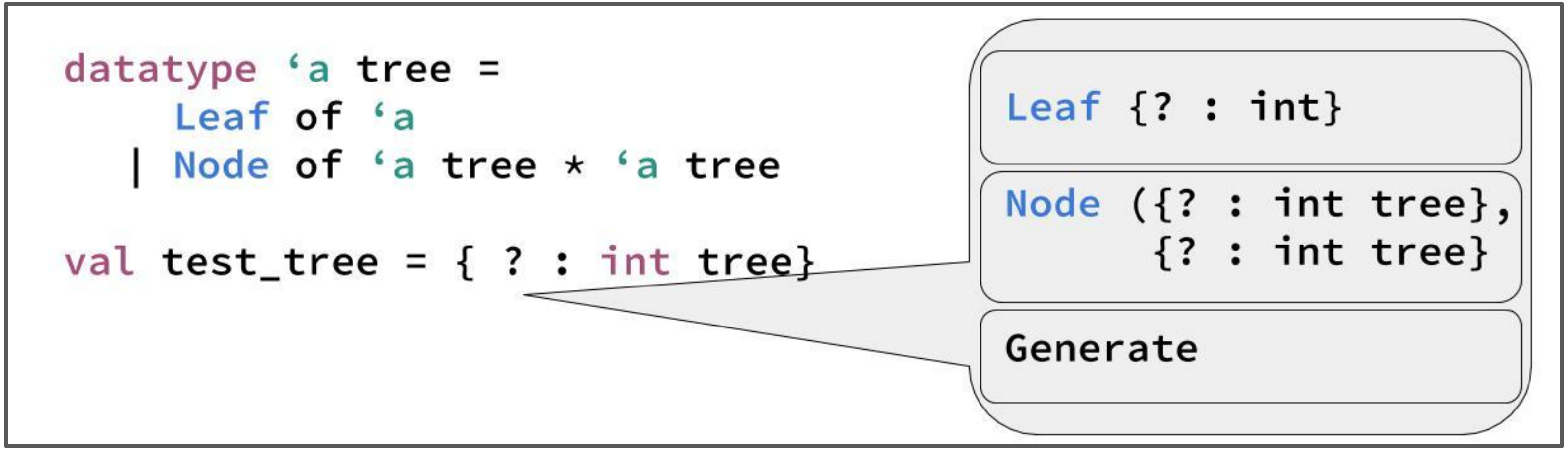
Acknowledgements: Tiannan Chen and Stephen Guy for the example (check out their C++ embedded DSL, GIGL!)

Use case for dependent types

```
gentype range(min,max) = Sigma n:nat. <geq n min, leq n max>

gentype weakRange = range(1,4)

val rageDmg : range(8,11) = letgen d:weakRange in d+7

(* Expanded: *)

Letgen {n=dmg, minproof : geq dmg 1, maxproof : leq dmg 4} : weakRange
in {d+7, geq_succ^7 minproof, leq_succ^7 maxproof}
```

Mixed-initiative program construction

Generation at coding time vs. at running time



```
datatype ‘a tree =
     Leaf of ‘a
   | Node of ‘a tree * ‘a tree

val test_tree = { ? : int tree}
```

Running partial programs

Kate Compton @GalaxyKate · 15 Dec 2018

Yeah, I have a whole dissertation chapter on Casual Creator programming languages, and 90% of it is graceful handling of errors. Either allowing partial specification (ie Tracery's [symbol] notation) or stubbing in defaults or just shrugging and compiling as best you can.

Kate Compton
@GalaxyKate

Replying to @GalaxyKate @chrisamaphone and 3 others

We'd have a much smaller internet if 90s web pages failed to load if you didn't close a <p> tag.

12:54 PM - 15 Dec 2018

3 Retweets 20 Likes

Some other things that are cool and exist

- Agda, Hazel (typed holes and partial program synthesis)
- Type-driven Program Synthesis - Osera, Polikarpova
- PCG Languages: GIGL, Tracery, Marahel

This Talk’s Key Idea

Generative spaces as types
support a unified account of grammar-based PCG, pattern matching, and constraint-based synthesis.

Long-term mission

Enable programming as a co-creative activity through language, editor, and tool design

Thanks!

P O E M

Principles of
Expressive Machines

Chris Martens
@chrisamaphone on Twitter
contextadventure@gmail.com
http://go.ncsu.edu/martens

Sestina: https://github.com/chrisamaphone/sestina

Extra Slides

Functional programming and PCG

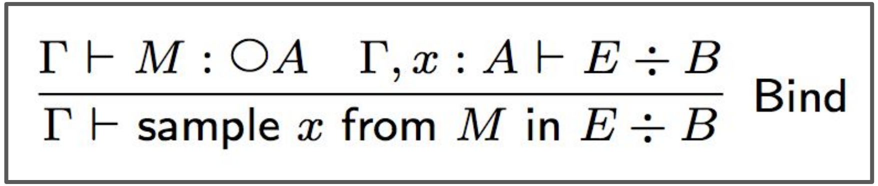
Instead of an effectful, nondeterministic **`unit -> A`**

`params -> T A`

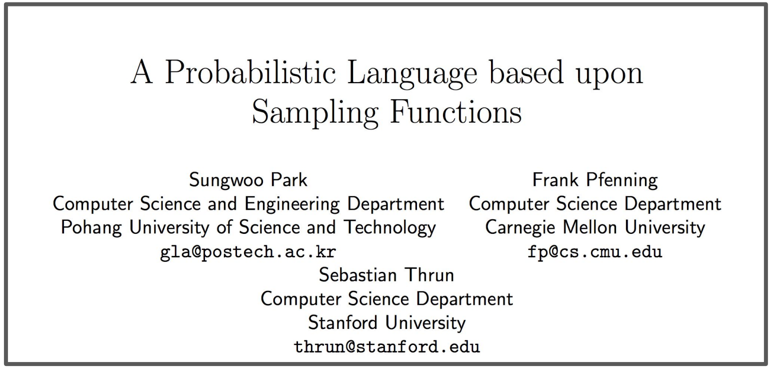
...where **`T A`** is the possibility space

E.g.: probability distribution

Sampling is an effectful (random) operation



$$\frac{\Gamma \vdash M : \bigcirc A \qquad \Gamma, x : A \vdash E \div B}{\Gamma \vdash \mathsf{sample}\ x\ \mathsf{from}\ M\ \mathsf{in}\ E \div B}\ \textsf{Bind}$$



A Probabilistic Language based upon
Sampling Functions

Sungwoo Park
Computer Science and Engineering Department
Pohang University of Science and Technology
gla@postech.ac.kr

Frank Pfenning
Computer Science Department
Carnegie Mellon University
fp@cs.cmu.edu

Sebastian Thrun
Computer Science Department
Stanford University
thrun@stanford.edu