Programming Systems

and their Technical Dimensions

“Methodology of Programming Systems” (MOPS)
tutorial session at <programming> ’22
25 March, Porto, Portugal

Joel Jakubovic
University of Kent
`jdj9@kent.ac.uk`

Jonathan Edwards
`jonathanmedwards@gmail.com`

Tomas Petricek
University of Kent
`T.Petricek@kent.ac.uk`

Lots of theory about *programming languages…*

$\rightarrow \forall$ *Based on $\lambda_\rightarrow$ (9-1)*

Syntax

$t ::=$		*terms:*
	x	*variable*
	$\lambda x{:}T.t$	*abstraction*
	$t\ t$	*application*
	$\lambda X.t$	*type abstraction*
	$t\ [T]$	*type application*
$v ::=$		*values:*
	$\lambda x{:}T.t$	*abstraction value*
	$\lambda X.t$	*type abstraction value*
$T ::=$		*types:*
	X	*type variable*
	$T\rightarrow T$	*type of functions*
	$\forall X.T$	*universal type*
$\Gamma ::=$		*contexts:*
	$\varnothing$	*empty context*
	$\Gamma, x{:}T$	*term variable binding*
	Γ, X	*type variable binding*

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-APP2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-APPABS)}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ [T_2] \longrightarrow t_1'\ [T_2]} \quad \text{(E-TAPP)}$$

$$(\lambda X.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \quad \text{(E-TAPPTABS)}$$

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \rightarrow T_2} \quad \text{(T-ABS)}$$

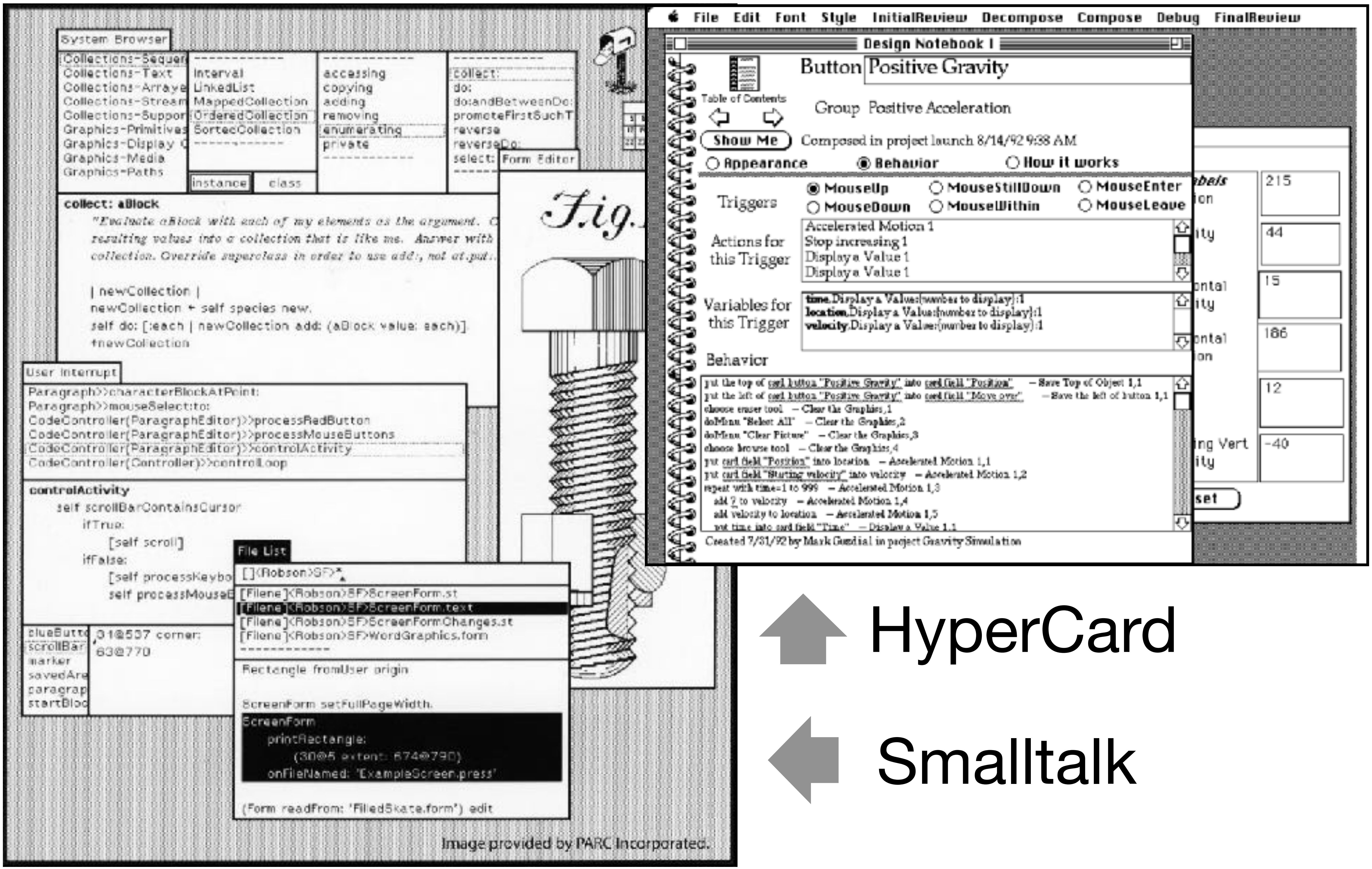
$$\frac{\Gamma \vdash t_1 : T_{11}\rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Gamma \vdash \lambda X.t_2 : \forall X.T_2} \quad \text{(T-TABS)}$$

$$\frac{\Gamma \vdash t_1 : \forall X.T_{12}}{\Gamma \vdash t_1\ [T_2] : [X \mapsto T_2]T_{12}} \quad \text{(T-TAPP)}$$

Figure 23-1: Polymorphic lambda-calculus (System F)

…but how do you theorise stuff like this:



HyperCard

Smalltalk

Informal subset / containment relationship

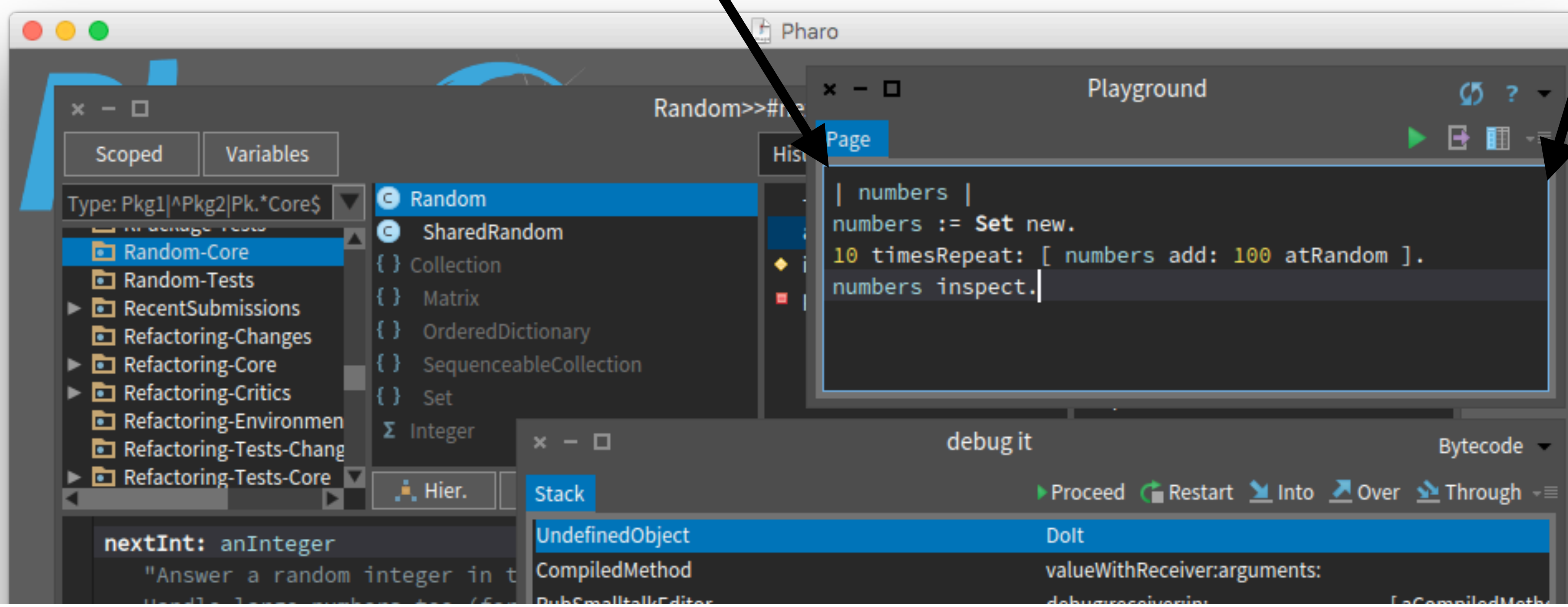
Languages:

```
| numbers |
numbers := Set new.
10 timesRepeat: [ numbers add: 100 atRandom ].
numbers inspect.
```

(but: idealised, formalised)

vs.

Systems:



(specific implementation of the language)

(Good Old?) Systems

(not to scale)

UNIX®

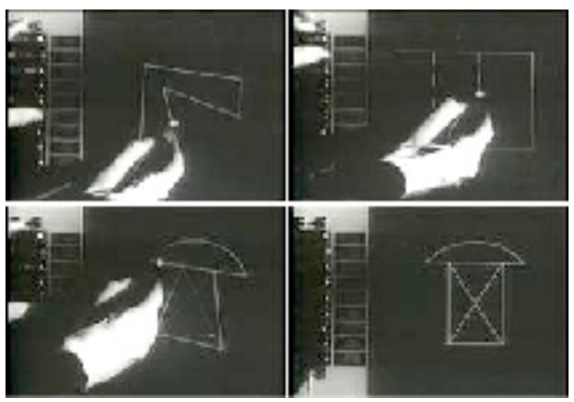
Self

HyperCard

The powerful tool for creating software solutions.

Excel

Wolfram Mathematica

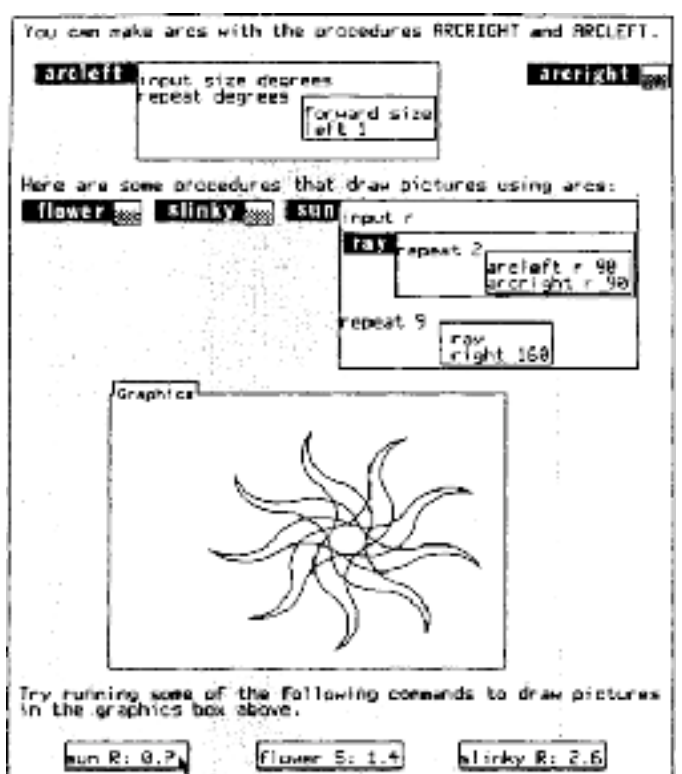


FoxPro

jupyter

Microsoft® Visual Basic®

SCRATCH



LabVIEW™

dBASE™

UNREAL ENGINE

GODOT

Game Maker

unity

What’s currently lacking

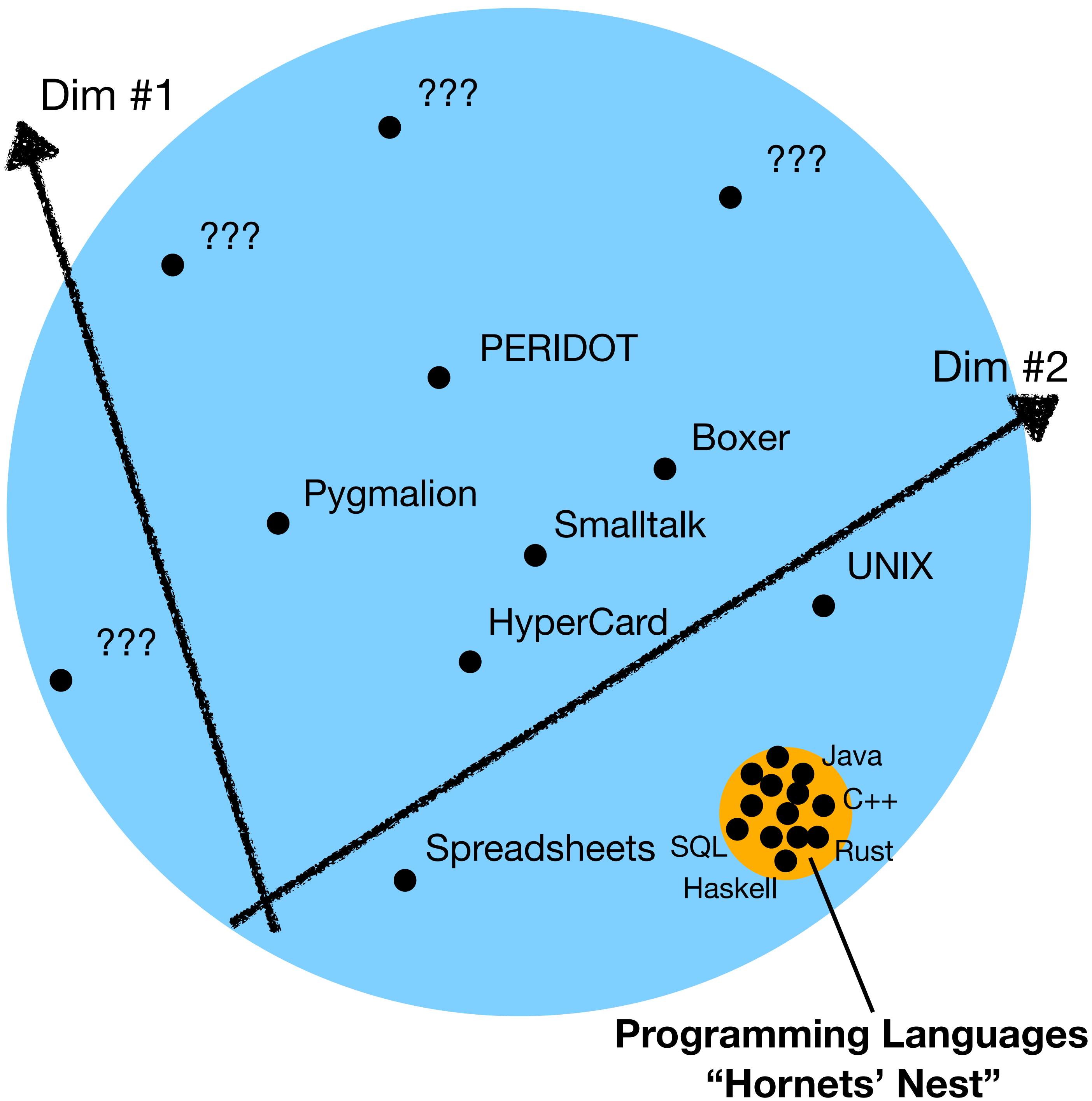
- S*ystems* more general than *languages*
- Stateful environment, GUI, *interacting* with *system,* contra Platonically disembodied code
- Much research building programming systems
- Disconnected, informal, opaque, still *art* not *science*
- How to build on what has been done before…?
- Programming system design “black art” —> collaborative, progressive (scientific?) endeavour?

Introducing “Technical Dimensions” of Programming Systems

For comparing and analysing programming systems. Influences:

- **Cognitive Dimensions of Notation** framework: common vocabulary (we go beyond *notation*)
- **Design Patterns**: common vocabulary with regular format
- Chang’s **Complementary Science:** engage with superseded scientific ideas to better appreciate the present paradigm
- PPIG 2019’s **“Evaluating Programming System Design”**: difficulties with system-focused venues, incorporate multimedia and interactive essays into submission evaluation

Here Be Metaphors…



Desired features of the dimensions:

(and why they're challenging to achieve!)

1. Deeper than mere "notation"

   Systems often emphasise the interface; hard to see beyond it

2. Qualitative yet comparable

   Nontrivial to ensure you can have "more" or "less" of a dimension

3. Not obviously "good" or "bad", tradeoffs welcome

   Dimensions often inspired by standout features of specific systems

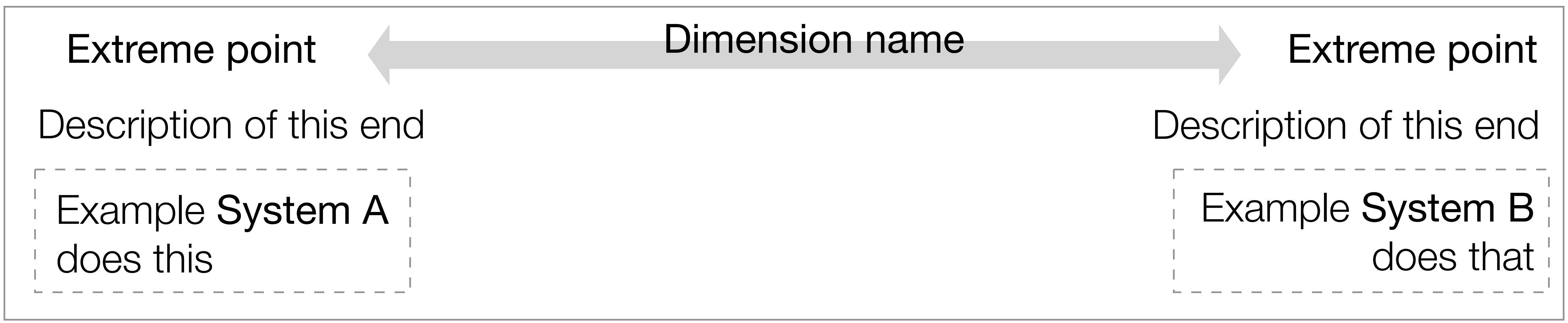
4. Span existing & possible systems, incl. OS-like (Unix, Lisp, Smalltalk) and PLs

   Hard to place every system along every dimension

5. Ideally place PLs in small region of possibility space; reflect similarity as *interactive systems*

   This is true in terms of *interaction,* yet there are still interesting differences between languages e.g. C vs Prolog

Dimensions format



Running example for *most* dimensions: Smalltalk, Spreadsheets

The Dimensions (so far)

Collab doc:
http://tinyurl.com/techdims

Interaction dimensions

Feedback Loops

Modes of Interaction

Abstraction Construction

Customisability dimensions

Staging of Customisation

Externalisability

Additive Authoring

Self-sustainability

Notation dimensions

Multiplicity of Notations

Notational Structure

Notational Uniformity

Expression Geography

"Conceptual Structure" dimensions

Integrity-vs-Openness

Composability

Convenience

Commonality

"Adoptability" dimensions

Learnability

Sociability

"Errors" dimensions

Error Detection

Error Response

Degrees of Automation *(singleton!)*

Interaction Dimensions

How do users manifest their ideas, evaluate the result, and generate new ideas in response?

Gulf of Evaluation

Feedback Loops

Tight ⟷ Wide

Gulf of Execution

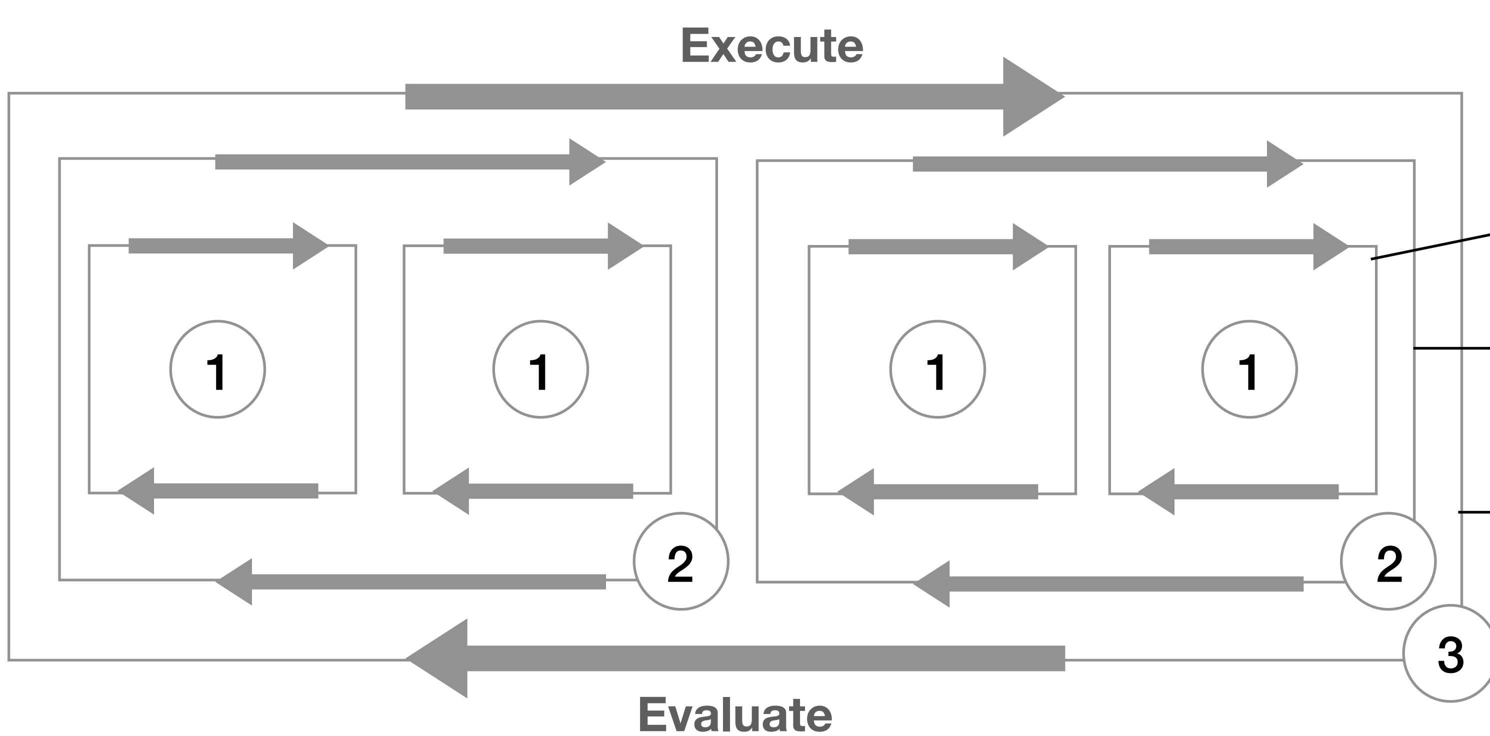
Tight ⟷ Wide

Immediate Feedback
(including Direct Manipulation)

Spreadsheets immediately update all dependent cells when a value is changed; you don't have to manually re-run anything.

Batch Mode

In the **Unix command line,** success is "silent" so you need to manually ask to see part of the state to check if it did what you wanted.



Innermost cycle 1: **Supplementary Medium** (e.g. notebook for working out the code design). Repeat until code is ready to submit.

Intermediate cycle 2: **Static checks.** Repeat until code passes the checks.

Outermost cycle 3: **Runtime observation.** Use the program, notice runtime bugs.

Interaction Dimensions

How do users manifest their ideas, evaluate the result, and generate new ideas in response?

Abstraction Construction

From Concrete ↔ **From Abstract**

You can write example code on example data first, then generalise it later.

You have to start at the abstract level and work your way down.

Pygmalion, a classic "Programming By Example" system, builds programs from concrete example executions.

Spreadsheets let you construct a formula on specific cells, and then drag it over adjacent cells to adapt it to them.

Smalltalk requires you to write classes before instantiating them, and write methods on general symbolic args.

Modes of Interaction

All In One ↔ **Highly Partitioned**

Various feedback loops, from using the running program, editing it and debugging it, are available at any time.

Certain feedback loops only occur together and not with others; they're partitioned into near-disjoint "modes".

Debugging and *running* are not sharply distinguished in **Jupyter notebooks**, which intersperse code blocks with their outputs.

Lisp systems sometimes separate *interpreted* execution (which provides interactive debugging) from *compiled* execution (which doesn't).

“Conceptual Structure” Dimensions

How is meaning constructed? How are internal and external incentives balanced?

Conceptual Integrity ⟷ Conceptual Openness

Smalltalk

“An Operating System is a collection of things that don’t fit into a language. There shouldn’t be one.”—Dan Ingalls [1]

Basic Principle of Recursive Design: give the *parts* (object) the same power as the *whole* (computer).

Everything is an Object, *automatically persisted* through the memory image.

Unix

“Unix succeeds in existing in the postmodern reality of diverse, independently developed, mutually incoherent language- and application-level abstractions, by virtue of its obliviousness to them.”—Stephen Kell [2]

Prescribes basic structure at large/coarse scale (processes, files). At fine scale (variables, functions), Unix says: *do what you want!*

Splits: "application" vs. “device” programming [2]
volatile memory vs. disk storage

Conceptual Integrity	Conceptual Openness
Conscientiously designed	Improvised or evolved
Everything is an X	Integrated mixtures
Rejects constraining norms	Compatible with existing norms
(Maybe) only One Way To Do It	(Probably) Several Ways To Do It
Friction with the outside world	Internal friction / mismatches
“Elegant” structure	Leaky abstractions, edge cases
Appeals to idealism	Appeals to pragmatism

[1] *Design Principles Behind Smalltalk* (1981)

[2] *The operating system: should there be one?* (2013)

“Conceptual Structure” Dimensions

How is meaning constructed? How are internal and external incentives balanced?

Composability

Low ⟷ High

Drop-down list of all available actions

Can *build* a range of unanticipated behaviours

Spreadsheet tool menus and functions available in formulas.

Spreadsheet grid cell references in combination with formulas

Convenience

Low ⟷ High

Small set of things to master

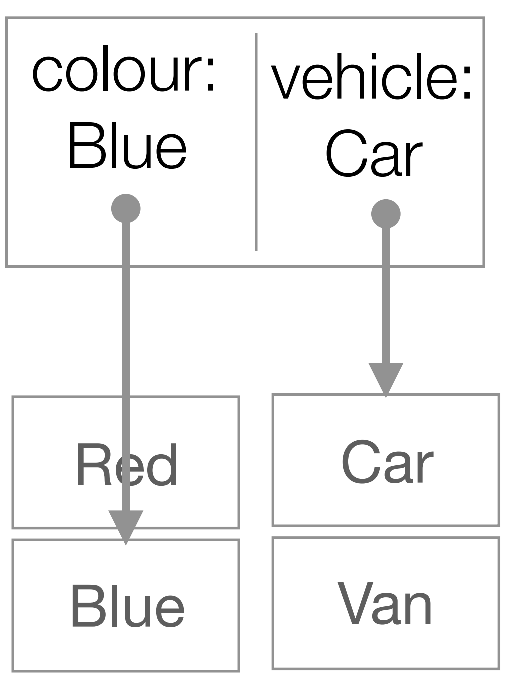
Can focus on *essential complexity*

Scheme provides minimal primitives from which *you* build everything

Smalltalk standard library and built-in data structures

Commonality

Factoring ⟷ Flattening

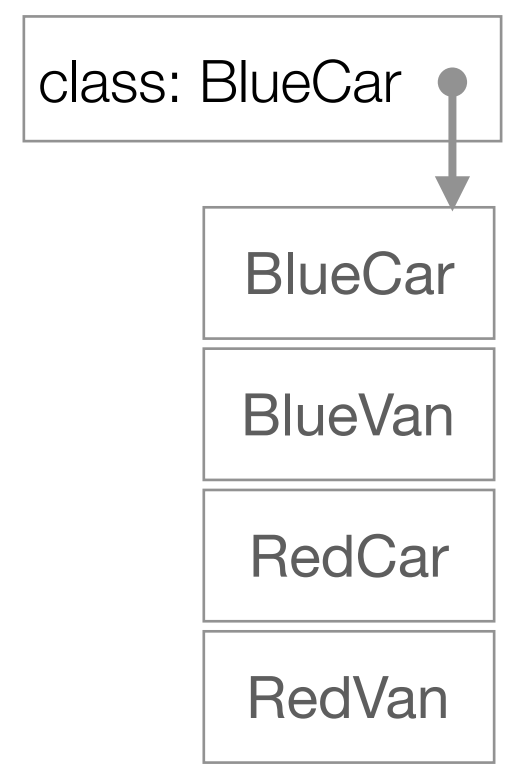


Shared structure is parcelled out and made machine-readable

`Array is Collection`
`Dict is Collection`
`Array.size`
`Dict.size`

Shared structure remains as *copies* on individual instances, which can *diverge*

~~`Array is Collection`~~
~~`Dict is Collection`~~
`Array.`**`length`**
`Dict.size`



“Notation” Dimensions

How are the different textual / visual programming notations related?

Multiplicity

Overlapping ⟷ **Complementing**

Overlapping: Multiple notations represent the same thing. Are any *read-only*? How do changes to one *synchronise* the others to match?

Together/J syncs UML diagrams with Java source code; **Object-Relational Mappers** sync object and DB representations.

Smalltalk’s class browser vs. method editor.

Spreadsheet grid vs. formula vs. Macro notations.

Complementing: Notations used for different aspects of the same thing. Used at the same time? One after the other? Or selected based on difficulty?

Surface notation

Structure: Implicit ⟷ Explicit

Internal notation: Implicit ⟷ Explicit

Internal notation	Surface: Implicit		Surface: Explicit	
Implicit	Sequence editing	**Smalltalk** method code text editing	Sequence rendering (serialization)	**Smalltalk** bytecode generation from AST
Explicit	Structure recovery (parsing)	**Smalltalk** class descriptions	Structure editing	**Spreadsheet** grid interactions

“Notation” Dimensions

How are the different textual / visual programming notations related?

Expression Geography

Rugged ⟷ Smooth

Changing a character results in a valid program which does something very different.

Significant changes in a program’s *behaviour* require significant changes in its *notation*.

Regex and **Perl** have notoriously rugged notation, as well as Unix commands. Exercise care typing `rm -rf ./*`

Direct manipulation of forms in **VB** or cards in **HyperCard** shows continuity in space.

Uniformity

Low ⟷ High

Variety of syntax / local notations. More to learn, more complex to manipulate programmatically, but avoids One-Size-Fits-All restrictions

All notation built out of the same basic pieces. Programmatic simplicity permits e.g. macro systems. Some expressions may feel cumbersome or verbose.

Perl’s syntax contains a wide variety of keywords and symbols, as well as a regex sub-language.

Smalltalk’s source code syntax doesn’t need many keywords; even if/else are expressed as message sends.

Lisp’s notation is highly uniform. No keywords, no infix operators; just nested lists of symbols.

Customisability Dimensions

Once a program exists in the system, how can it be extended and modified?

Staging of Customisation

Transient ⟷ Persistent

Changes made to the running program are “forgotten” if it’s shut down.

Runtime changes are retained through terminations.

Unix distinguishes between volatile storage (processes and their data) and non-volatile (files) throughout the system.

Smalltalk objects just live in the “image”, which is automatically persistent.

Self-sustainability

Low ⟷ High

Sharp distinction between the “implementation” level and the “user” level; different languages, abstractions, etc.

Nothing is “baked in”; any inner workings can be overridden or modified from within the running system.

The compiler or interpreter for a PL (e.g. **C++**) is typically not very changeable from within the code it processes. You’re stuck with it unless you enter the *separate* world of the implementation code, possibly in another language.

Smalltalk goes as far as to let you re-define True as False and break the system!

Customisability Dimensions

Once a program exists in the system, how can it be extended and modified?

Externalizability[1]

Low ⟷ High

State references are highly fragile (e.g. line numbers / memory addresses), or most state can't even be referenced at all (hidden, internal to the runtime.)

A **Smalltalk** VM image is more or less an opaque "blob" only workable via a VM.

You can export+import design elements via "coordinates" which are *stable* to design changes.

Much of the state in a **Web** page can be exported as HTML. Element IDs and CSS classes are stable-*ish* coordinates.

Additive Authoring[2]

Low ⟷ High

Generally, you can only change system behaviour by *overwriting* parts of its specification—you need *write access* to the original source code.

An entire method in **Smalltalk** can be overridden via inheritance, but this does not extend to *finer-grained* data or code behaviour.

Anything can be overridden—back and forth!—by *adding* new instructions for the system to follow, including its behaviour.

Web stylesheets let you override diverse display properties without having to overwrite the CSS code.

[1] *Software and how it lives on: Embedding live programs in the world around them* (2016)

[2] *The Open Authorial Principle: supporting networks of authors in creating externalisable designs* (2018)

"Errors" Dimensions

What does the system consider to be an *error?*
How are they prevented and handled?

Error Detection

Manual ⟷ Auto

Human must watch for errors at runtime.

The semantics of **JavaScript** objects are such that requesting an absent property returns the special value `undefined`. Can't automatically tell whether this was because of a misspelled key vs. intended behaviour!

The system can see that something *will* lead to an error when run, and alert you early.

One of the functions of the **Haskell** type system is to constrain what's allowed so that e.g. misspelled names can be automatically flagged as mistakes.

Error Response

Abort ⟷ Continue

Shut Down Everything! Simplest implementation - just halt, complain, quit.

Unix performs a "core dump" before killing an errant process.

The Show Must Go On! May *ignore*, seek *user assistance* or automatically *recover*

Interlisp's "Do What I Mean" feature attempts to automatically correct misspellings and unbalanced parentheses, deferring to the user if unsuccessful. Similarly, **TeX** pauses and lets you patch in the correct command.

“Adoptability” Dimensions

How does the system facilitate or obstruct adoption by both individuals and communities?

Learnability

General audience ⟷ **Specialist audience**

Targeted at people not already familiar with programming

Targeted at existing programmers or members of a field e.g. physicists, musicians

Boxer was aimed at children’s education but designed to be easy for adults to understand and work with.

Unix was designed explicitly for *programmers* at a time when computers themselves were specialised tools.

Sociability

Social Factors: Code sharing, Q/A sites, documentation, community rules / norms, sense of belonging, Conway’s Law

Programming systems often have a central “guru” or “figurehead” to guide the technical and social evolution (**Smalltalk**+Alan Kay, **Haskell**+SPJ, **Boxer**+diSessa)

Economic Factors: Who contributes? How is development funded? How are money, time, attention and people allocated? How economical is it to *adopt* the system?

Open-source projects are funded by commercial partners or non-profits (e.g. the Blender Foundation)

“Automation” Dimensions

What parts of program logic don’t need to be explicitly specified?

Low-tech ← Degrees of Automation → High-tech

Low-tech		High-tech
“Design-time” memory management (e.g. in bootloader)	Automatic *reservation* of heap blocks (`malloc`/`free`)	Garbage collection
“Programming”	“Good Old-Fashioned AI” (GOFAI)	Machine Learning?

Future work

- Thoroughly apply to example systems (incl. no-code/low-code)
- Add new dims as needed, invite critique and contributions from future collaborators
- Explore previously unexplored combinations

Conclusions

- *Systems* are a broader scope that include *languages*
- No agreement on how to study them
- “Technical Dimensions” are attempt to provide such a methodology
- Open Question: can this start a productive field of research on programming systems?