Programming “systems” deserve a theory too!

A <Programming> 2021 Conversation Starter

Joel Jakubovic
PhD year 2
University of Kent
jdj9@kent.ac.uk

Jonathan Edwards
jonathanmedwards@gmail.com

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

NOTICE: To facilitate high-quality discussion in the session, we ask everyone to **submit 1 position slide** to introduce their contributions.

- Max 2 minutes per person
- Send a PDF by 1 hour before the session (held at Wed 24 March, 5pm UK time) to jdj9@kent.ac.uk
- On the day, we'll quickly review the slides here, and then proceed to participant submissions. We'll then follow with open-ended discussion.

Lots of theory about *programming languages…*

…but how do you theorise *this??*

→ ∀ *Based on λ→ (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*
	$T{\to}T$	*type of functions*
	$\forall X.T$	*universal type*
$\Gamma ::=$		*contexts:*
	$\emptyset$	*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 $\boxed{\Gamma \vdash t : T}$

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

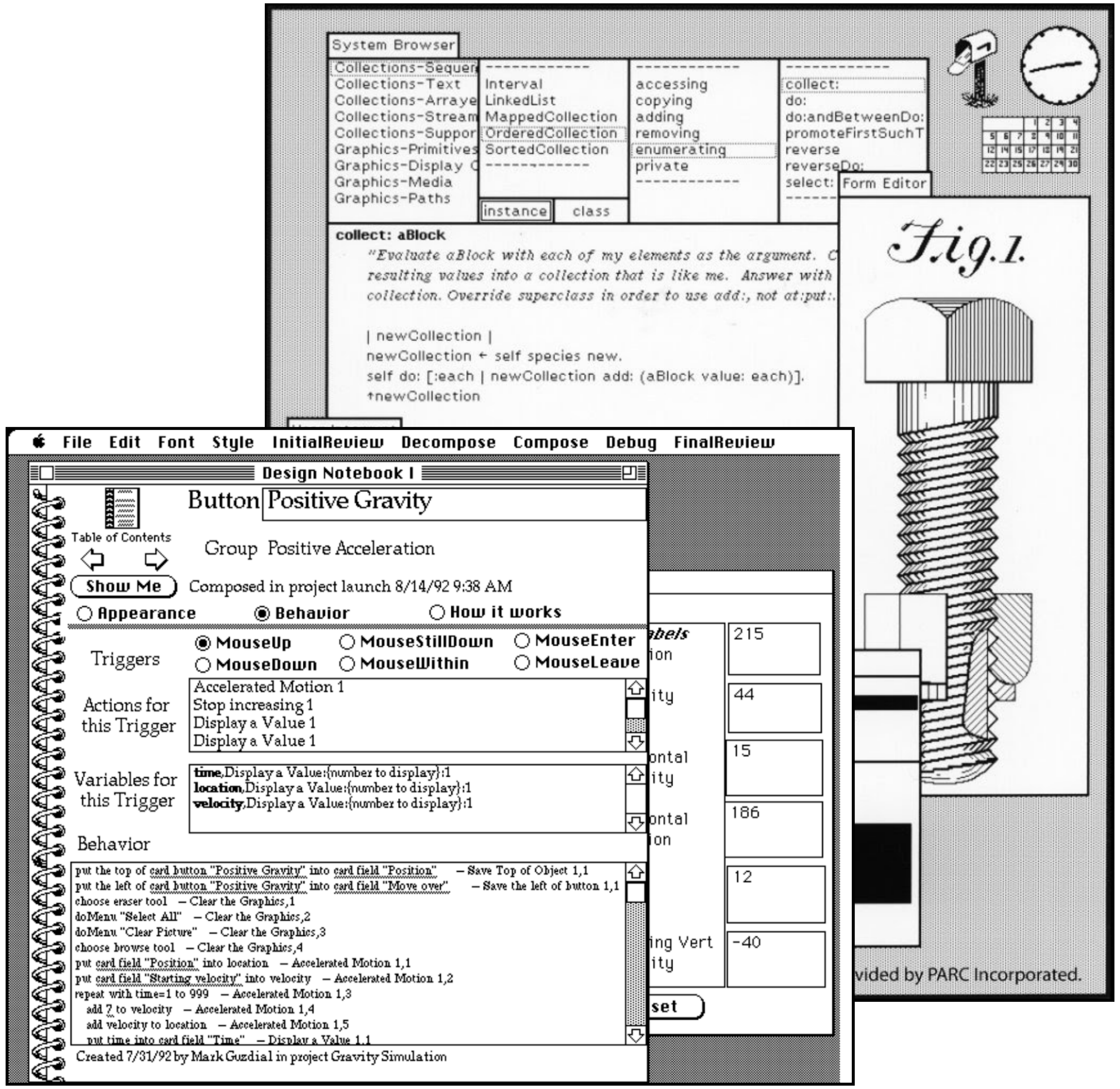
$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1{\to}T_2} \quad \text{(T-Abs)}$$

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

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

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

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



What’s currently lacking

- Programming systems often go beyond programming *languages*.
- Programming is often done in the context of a stateful environment, through a graphical user interface, by *interacting* with the *system* rather than just by writing code.
- Much ongoing research effort focuses on building programming systems that are easier to use, accessible to non-experts, innovative, moldable and/or powerful.
- Such efforts are often **disconnected**. They are informal, guided by the personal vision of the authors and thus are **only evaluable and comparable on the basis of individual experience using them.**
- In other words, they fail to form a coherent body of research. **It isn't clear how to build on what has been done before.**
- Might we turn the “black art” of programming system design into a more easily **collaborative, progressive — even *scientific* — endeavour?**

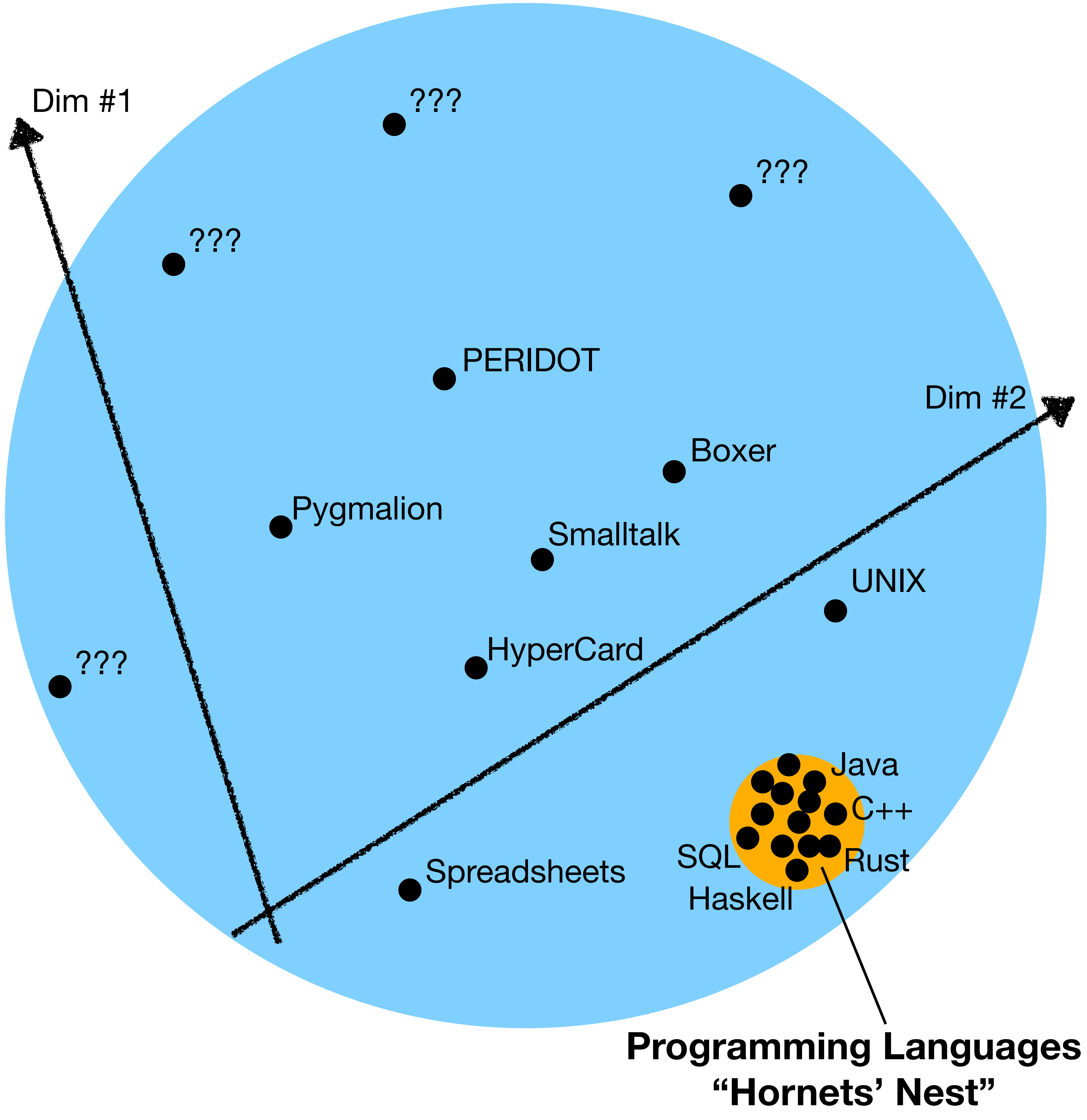
Introducing “Technical Dimensions” of Programming Systems

We’re proposing a set of named **“technical dimensions”** to compare and analyse programming systems. Among our influences are:

- The **Cognitive Dimensions of Notation** framework: a named set of interrelated axes for characterising *notations;* we wish to extend the same approach *beyond* the surface “notation”
- The various **Design Patterns**: a common vocabulary for software engineers, set out in a standard template including summary, details, examples and relations
- Chang’s **Complementary Science:** that it is a valuable activity to revisit the forgotten or superseded science of the past and engage with it in order to better appreciate the present paradigm
- PPIG 2019’s **“Evaluating Programming System Design”:** a survey of the difficulties with system-focused research venues and a look toward incorporating multimedia and interactive essays into the evaluation of submissions

We follow some broad heuristics about what we want them to do:

- Go deeper than mere “notation”
- Not be obviously “good” or “bad”, tradeoffs welcome
- Span a variety of existing and possible systems, including OS-like (Unix, Lisp, Smalltalk) and traditional PLs
- Ideally place PLs in a small region of the space of possibilities to reflect how similar they really are as *interactive programming systems*



Dimension: Feedback Loops

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

Immediate Feedback

- …is where the *evaluation* gulf is imperceptibly small
- …and results are demanded *automatically* without manual polling
- *Direct Manipulation* (DM) is a sub-type, uses proxy of finger or hand

Liveness

- Immediate Feedback is necessary, but not sufficient
- "The thing on the screen is the actual thing" suggests that some measure of DM and bi-directionality may be needed

Examples

- Statically-checked programming languages (see diagram)
- Spreadsheets have the usual DM loop for values and formatting, and a loop for formula editing and use. In this latter loop, execution involves designing and typing formulas; evaluation is often shortened by editor features such as immediate cell previews.

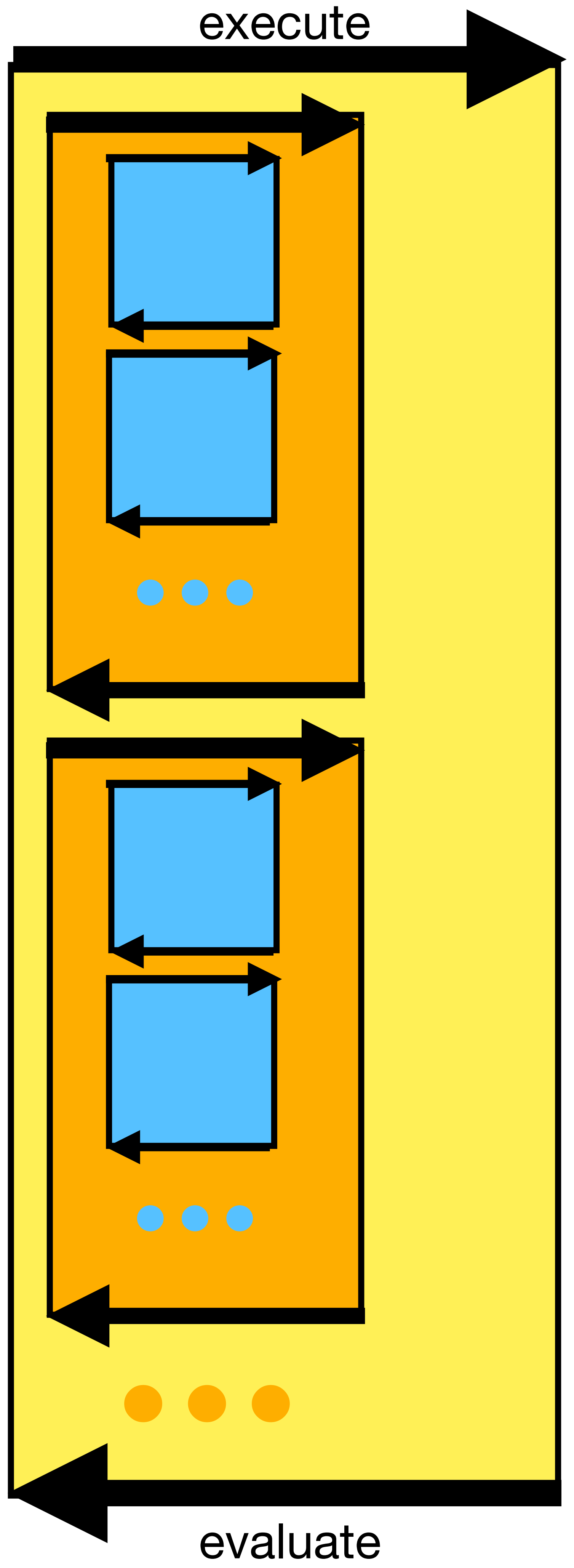
Feedback loop =
Gulf of Execution +
Gulf of Evaluation

Supplementary medium =
e.g. paper notebook for
working out the code design

Cycle 1: Supplementary medium
Repeats until code ready to submit

Cycle 2: Static checks
Repeats until new code is statically valid

Cycle 3: Runtime observation
Repeats until program "works well enough"



Dimension: Notational Structure

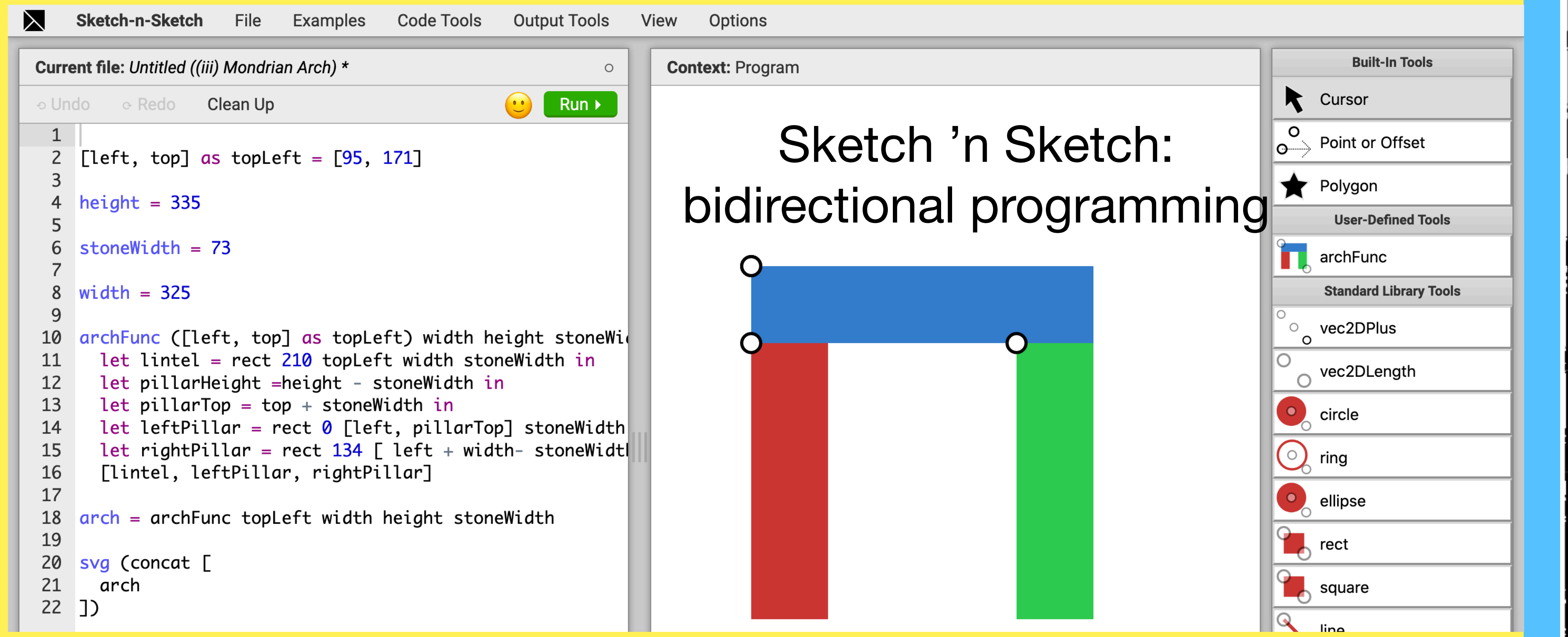
What are the different textual & visual notations through which the system is programmed? How do they overlap or complement each other?

Complementing notations

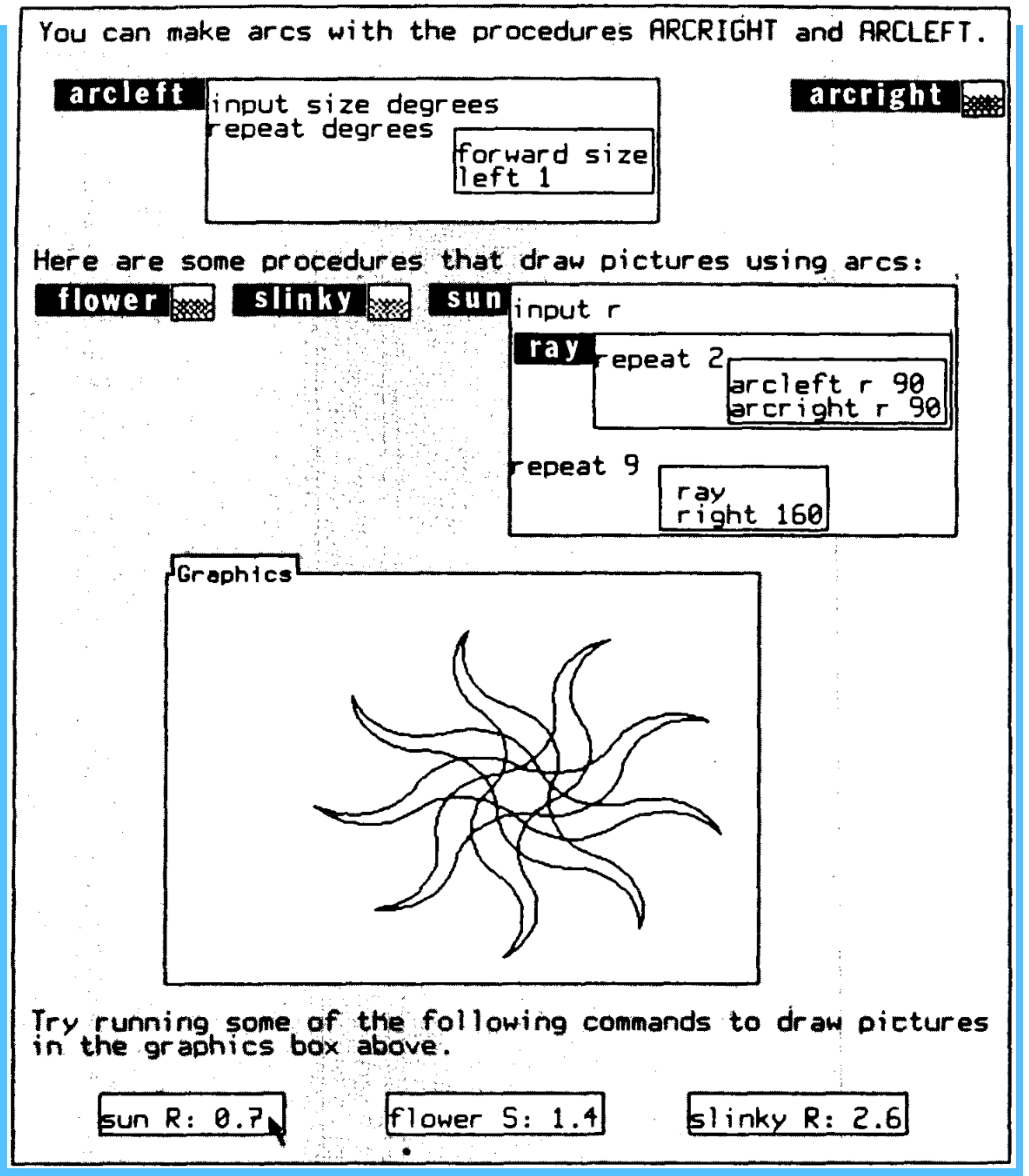
- … represent different parts of the domain e.g. Boxer, HyperCard, spreadsheets
- Excel formulas describe dataflow and arithmetic through the cells, while VBA macros can do more general programming
- How are they connected and how does the user transition between them?
- One can write Excel macros to evaluate formulas, so there is a subset relation here
- Optimised for easy *learnability* at the beginning (formulas), with a sudden jump to learning a conventional programming language (VBA macros)

Overlapping notations

- … fully or partially represent the same thing, e.g. in Sketch ’n Sketch
- Requires synchronization between the different representations e.g. editing with DM causes program code synthesis



Boxer: code within nested box substrate



Dimension: Factoring of Complexity

What are the primitives? How can they be combined? How is *common structure* recognised and utilised?

Composability

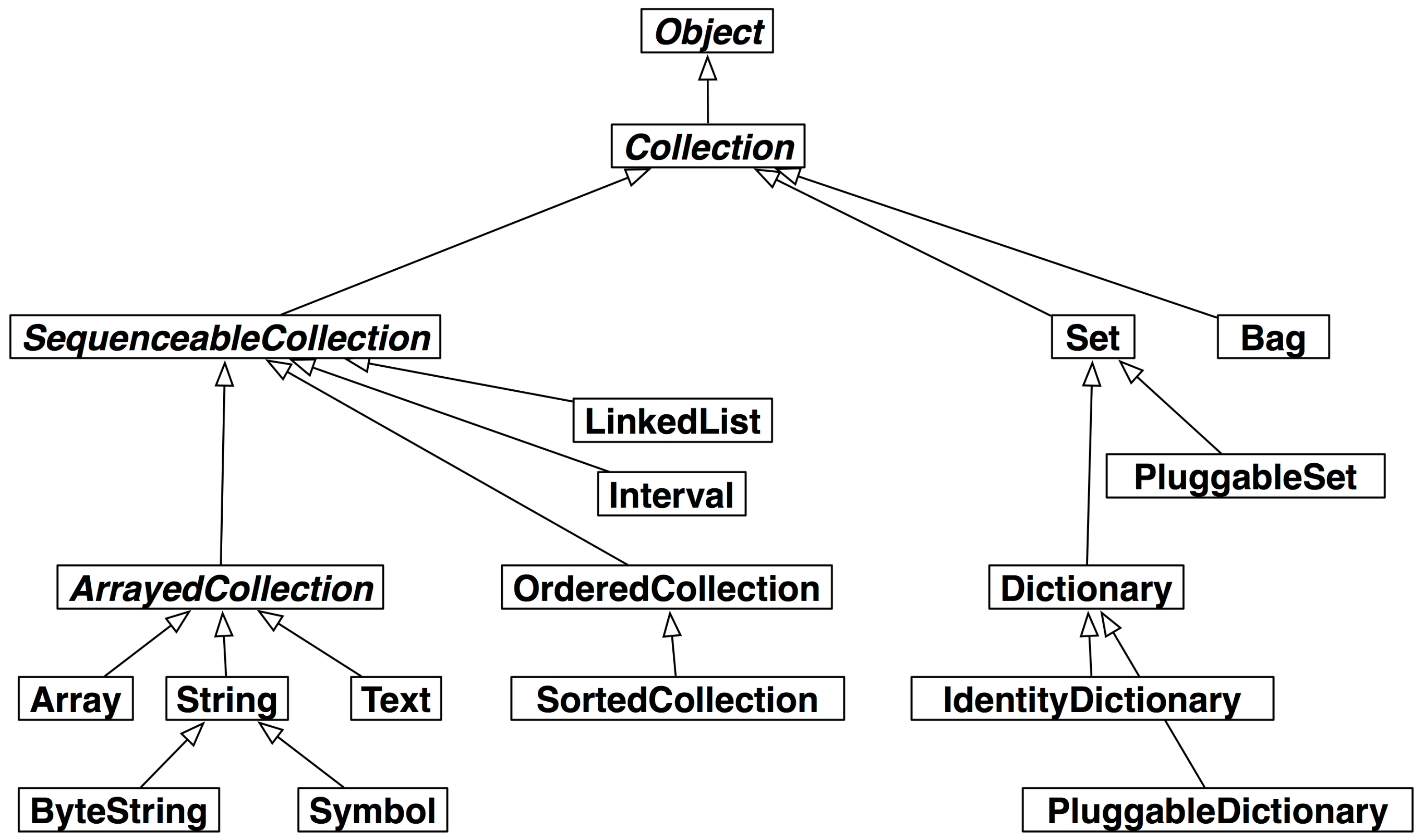
- *"You can get anywhere via a number of smaller steps"*
- There are primitive components with a range of useful combinations, called their *span* by analogy with vectors
- Key to the notion of "programmability" so every *programming* system will have some minimal amount of this

Convenience

- *"You can get to X, Y and Z via one single step"*
- Can take the form of *canonical* solutions and utilities e.g. the expansive Python standard library.
- Specific solution to a specific problem, not necessarily generalisable or composable

Commonality

- *Humans* can see Arrays, Strings, Dicts and Sets all have a "size", but the *computer* needs to be told that they are the "same"
- Commonality can be *factored* out into an explicit structure ("Collection" class); analogous to database *normalization*
- *…*or it can be left implicit; less work, but permits instances to *diverge* (maybe Arrays and Strings have "length", but Dict and Set call it "size") - analogous to *redundancy* in databases.



Smalltalk Collections hierarchy

Flattening & Factoring

- Take a structure we recognise as multidimensional, e.g. "vehicle type" and "colour" are *independently variable*
- System might only permit this as an *exhaustive enumeration*: classes *RedCar, BlueCar, RedVan, BlueVan, RedTrain, BlueTrain*, …
- The computer sees a *flat list* of atoms. The user cannot just "change the colour to Red" programmatically.
- Just as 16 factors to 2*2*2*2, there is implicit structure here that remains un-*factored*.

Other dimensions (no particular order)

Concept of “error”: What does the system consider to be an *error*, and how does it approach their prevention and handling?

Background knowledge: What background knowledge does the system demand in order to be judged on its own merit?

Abstraction mechanisms: What is the relationship between *concrete* and *abstract* in the system? How are abstract entities created from concrete things and vice versa?

Status-quo compatibility: What tradeoffs are made between logical coherence and compatibility with established technologies? Does the system *replace* established technologies or attempt to *adapt* and recombine them?

Information loss: Where is information being destroyed or scrambled in a hard-to-recover manner? (relevant to provenance and bi-directionality)

Self-mutability: to what extent can the system be changed *from within?* What features have been “baked in”, “set in stone”, “hard-coded”?

Target audience: what role does the system encourage for its users? Does it appeal to particular personality types?

Locus of uniformity: What are the central notions or *basic assumptions* defining the worldview imposed by the system, and what is the degree of similarity between different notions?

Four key questions

1. Can we identify a set of technical dimensions which let us *meaningfully compare* different programming systems?

2. How do we distinguish a meaningful, useful technical dimension from just any old ad-hoc observable property of a system? How do we *evaluate* how a proposal fits in the framework?

3. Should we expect to be able to bridge such large conceptual gaps as that between, say, statically-typed *languages* like Haskell and “end-user” environments like HyperCard?

4. Can the dimensions map out a space where existing systems may be plotted, revealing *overlooked combinations* that have not yet been tried?

let’s discuss…

NOTICE: To facilitate high-quality discussion in the session, we ask everyone to **submit 1 position slide** to introduce their contributions.

- Max 2 minutes per person
- Send a PDF by 1 hour before the session (held at Wed 24 March, 5pm UK time) to jdj9@kent.ac.uk
- On the day, we'll quickly review the slides here, and then proceed to participant submissions. We'll then follow with open-ended discussion.