PUC-Rio

Departamento de Informática

PhD Thesis

# Dataflow Semantics for End-User Programmable Applications

*Author:*
Hisham Hashem Muhammad

*Advisor:*
Roberto Ierusalimschy

January 19th, 2016

*This page intentionally contains only this sentence.*

# Contents

# Abstract

End-user programming refers to programming performed by end-users who are not professionals in software development but are specialists in other domains. Scripting is one way to add programmability to applications as an advanced, peripheral feature. Another alternative is to make an end-user language central to the application's UI, as is the case with spreadsheet formulas: programming becomes indistinguishable from using the application proper. Scripting has evolved from ad hoc "little languages" into reusable embeddable languages, which benefit from the advances of programming language research. The state of UI-level end-user programming languages, however, lags behind: those are still developed ad hoc and tied to their domains. We claim that end-user languages should also be developed constructing domain-specific UIs on top of reusable core semantics, much like scripting is now done presenting domain-specific abstractions of top of reusable scripting languages. For this, however, an understanding of the underlying programming model is necessary. In this work, we model the semantics of existing end-user programming languages of different domains, such as spreadsheets and multimedia, aiming to better understand their commonalities and shortcomings. Our focus is on the dataflow paradigm, since this paradigm is representative of a number of applications where end-user programmability takes center stage. Based on this analysis, our goal is to provide a better understanding of dataflow semantics in the context of end-user programming and propose guidelines for the development of dataflow languages for end-user programmable applications.

# Chapter 1

# Introduction

## 1.1 Motivation

*End-user programming* is a term that refers to programming activities performed by end-users, who are not specialists in professional software development but are specialists in other domains, in the context of software developed for their domain of interest [BS14]. Early focus of end-user programming was in *scripting* [Ous98]: embedding a programming language into an application so that users can write programs that drive the application, or at least parts of it. Some examples are Visual Basic for Applications in the Microsoft Office productivity suite and AutoLISP in AutoCAD. However, most end-user programming happens using languages not regarded by their users as programming languages proper, such as spreadsheets and graphical node editors [MSH92]. For this reason, the latter term "end-user development" took over, which avoids having to answer the question of what is and what isn't programming.

Still, those languages are indeed domain-specific languages (DSLs), even if they often have restricted expressivity and if their syntaxes (often a mix of textual elements and graphics) do not make it clear that they are programming languages. In particular, there are applications where the language becomes indistinguishable from the user interface (UI), and using the software means using the language. Examples of scenarios where this happens are spreadsheets such as Excel [Nar93], where using the spreadsheet means using the formula language, and the node graph environment of Pure Data, a multimedia application developed primarily for music [P$^+$15]. Here, we will call these languages *UI-level languages*: programming languages that are apparent to the user as the main UI of the application (not to be confused with languages for constructing GUIs).

There is, therefore, a spectrum of possibilities on how deeply to integrate programmability into end-user applications. It can range from being an optional feature for advanced users, as is often the case with scripting, to being at the core of the application experience, as is the case with UI-level languages.

The programming language community has given much attention to the scripting end of the spectrum of end-user programming. Scripting in applications has evolved through three stages:

- The first stage was the use of so-called "little languages" of the 1970s and 1980s

[Ben86]—examples are small single-purpose languages such as `eqn` and `pic` in Unix [DOK+87], and SCUMM for scripting adventure-style games [1];

- Then, powerful domain-specific languages emerged, such as GraForth, WordBasic in Microsoft Word and AutoLISP in AutoCAD—these languages often adapted the design of existing general-purpose programming languages, producing application-specific dialects;
- Finally, once scripting was identified as a general style of programming [Ous98], we saw the introduction of general-purpose scripting languages such as Tcl, Python and Lua—these languages have embeddable implementations, allowing them to be linked as libraries and reused in many applications. Some of these languages became popular in particular domains in spite of not having been specifically designed for those fields: several graphics programs use Python as a scripting language, and Lua is particularly successful in the game industry.

The UI end of the spectrum, however, lags behind. The state of end-user UI-level languages now is similar to that of scripting languages in the 1980s: most applications develop their own custom "little languages" for end-user programming, tightly coupled to their UIs. At best, ideas are reused from similar efforts, as evidenced by several visual programming languages inspired by Scratch (Stencyl, AppInventor, Snap, Pocket Code), the various node/graph editors in multimedia applications (Nuke, Max/MSP, Blender, Rhino 3D, Antimony, Unreal Engine), and the formula languages for different spreadsheets.

The semantics of these UI-level languages, in particular, are often ill-specified. This has wide-ranging consequences, affecting both users and developers. Users are struck by subtle incompatibilities even when different applications share the same basic metaphors, as is the case with different spreadsheet applications. Developers end up "reinventing the wheel", producing designs that are not informed by PL research, often subject to pitfalls that could have been avoided had the language been designed based on established grounds.

## 1.2 Problem statement

While nowadays scripting is integrated into applications by reusing proven embeddable languages which benefit from the advances of programming language research, end-user UI-level languages are still developed ad hoc, often by developers who are specialists in the domain of the application (e.g. computer graphics, music, statistics, finance), but not in language design and implementation. This distinction manifests itself in many ways, ranging from unclear semantics and little possibility of knowledge reuse from the part of users, down to lack of application interoperability and performance issues. Programming language research is brought to the fold

[1]Originally created by Lucasfilm Games as "Script Creation Utility for *Maniac Mansion*" to be used in a single game; later used in dozens of games, licensed to several companies and even reimplemented as a free-software engine (`http://www.scummvm.org`). This illustrates the "accidental" nature of scripting languages from that era.

afterwards, when trying to find ways to integrate missing functionality or trying to fix issues with the design or implementation [JBB03, FP15].

By focusing on the presentation and direct manipulation of data, many UI-level languages adopt to some extent the dataflow paradigm [JHM04], as it seems a natural fit for programmatic manipulation of data in user interfaces. Thus, the problem domain we focus our attention on is that of dataflow UI-level languages, as these are representative of a number of languages where end-user programmability takes center stage, and the problem we address in particular is that of *ad hoc semantics in dataflow programmable UIs*.

We identify the need for UI-level dataflow languages to go through a similar movement that occurred with scripting languages. Scripting languages evolved from ad hoc languages into standard reusable languages that integrate properly: they can be embedded in an application and can be extended with constructs for dealing with the application's domain.

This research attempts to provide the first steps towards providing designers of end-user programmable applications with a structured knowledge base for reasoning about dataflow semantics for their UI-level languages. Our research question, therefore, can be framed as such: *when designing a dataflow language for an end-user programmable application, which design choices should be taken into consideration with regard to its semantics and what are their effects?*

We begin with a background review of the fields of end-user programming and dataflow languages in Chapter 2. Then, in Chapter 3 we sharpen our focus to define dataflow UI-level languages and present a list of design alternatives for the construction of dataflow languages for end-user programming. With this design space in mind, we analyze existing UI-level dataflow languages and produce semantics for a number of them, understanding their features and shortcomings. These are presented as case studies in Chapters 4, 5 and 6, as well as a review of additional languages in 7. Informed by the design of those existing application-specific languages, we revisit in Chapter 8 our list of design alternatives, and present a discussion of the impacts of various design choices. Finally, we conclude the work in Chapter 9, in which we review our contributions.

# Chapter 2

# Background

## 2.1 End-user programming

*End-user programming* is a term that describes the involvement of users in the addition of functionality to their applications via programming [BS14]. For this to be possible, applications have to be designed with programmability in mind: they should allow new functionality to be built based on the combination of existing ones, and there has to be linguistic support in the application so that the user may express this new functionality.

### 2.1.1 Roles of programming in end-user applications

Depending on the application's design, end-user programming may take a *peripheral* or a *central* role in its use. We define them as such:

- Programming is *peripheral* to an end-user application if users can make effective use of the application with variable degrees of complexity, producing from simple to complex end-results (e.g. documents, queries, etc.), without ever resorting explicitly to the programming capabilities of the application.

- Programming is *central* to an end-user application if users must interact explicitly with the programming capabilities of the application in order to make any non-trivial use of it, even when producing simple end-results. In other words, using the application *is* using the application's programming language.

Examples of peripheral support for end-user programming are macro recorders and embedded scripting languages. Macro recording allows the user to automate sequences of steps, but offers little in the way of abstraction. Embedded scripting languages offer a programmable view to the application, by extending the application with the full power of a Turing-complete programming language. Still, we categorize these features as peripheral to the user of the application, because most users can produce content in the application while ignoring its programmable aspects.

The LibreOffice Writer word processor and the Gimp image editor are examples of scriptable applications, where programming takes a peripheral role. A user

| | | | | | |
|---|---|---|---|---|---|
| *End user* | formulas | macro recorder | node editor | Unix shell | level editor |
| *Domain dev* | macros | textual macros | scripting | shell script | game scripting |
| *Core app* | spreadsheet | word processor | 3D app | C utilities | video game |

Figure 2.1: Nardi's three types of programmers and three-layer architectures in end-user programmable systems

can produce a text document with a very complex layout in LibreOffice Writer or an intricate multi-layered drawing in Gimp without ever touching their scripting abilities. A Gimp user may benefit implicitly from the scripting abilities of the application by using bundled filters and effects available in the application toolset that are implemented as scripts, but from the user's perspective these tools could just as well be built into the application's core. One could conceive an application where all tools are implemented as scripts; still, if the user could use these tools as black boxes without ever touching the programming language, this means the role of programming is not central from an end-user perspective. For this reason, we state that the interaction with the programming facilities needs to be explicit in order to characterize the activity as end-user programming.

A prominent example of an application where end-user programming takes a central role is the spreadsheet. A spreadsheet has an open-ended design in which users can create new solutions for their domains, expressed as calculations in a formula language. Programming is central in the sense that the programmable aspect of the application, the formula language, is unavoidable for any use beyond the trivial case of entering constants into cells: using the formula language is equivalent to using the spreadsheet.

## 2.1.2 The three-layer architecture in end-user programmable applications

When programming support is added in a peripheral role to an application, it is usually to provide advanced flexibility beyond what the base feature set of the application offers: composition of features to automate workflows, iteration to avoid repetitive tasks, interaction with the operating system. Visual Basic for Applications (VBA), embedded in Microsoft Office programs, is an example of this.

This kind of addition also makes sense in applications where this base feature set is provided as a programming language in a central role. This means an application may feature programming in both central and peripheral roles. Modern spreadsheets such as Excel are an example of this. For advanced uses, spreadsheets also offer scripting support: "macros", in spreadsheet parlance, are extension functions implemented as scripts (in Excel, these scripts are implemented in VBA). These extended functions can then be used in the formula language.

A three-layer general architecture like this, with an end-user language on top, a scripting language in the middle, and the core implementation of the application at the bottom, is a common pattern we identify in several successful examples of applications where end-user programming is the central form of interaction.

In "A Small Matter of Programming" [Nar93], Bonnie Nardi reports on studies that identified three types of people who are engaged in programming at different levels: the *end-user*, who may be a sophisticated user in their own domain but who does not particularly care about programming and just wants to use the computer as a tool to solve problems from their domain; the *domain developer* (called a "local developer" in [NM91], "translator" in [Mac90], "tinkerer" in [MCLM90]), who started as an end-user but acquired an appreciation towards programming and is therefore more predisposed to dive into the possibilities offered by a programmable system; and finally, the professional, who had formal training in programming. Figure 2.1 maps these kinds of users to examples of three-language architectures used by end-user-programmable applications.

The existence of different roles among a community of users continues to be observed [DJS11], and the alignment between these three different user profiles and three architectural layers does not seem to be coincidental. We believe that this three-tier architecture is necessary in end-user programmable applications that feature programmability as a central feature in their design. The main user-facing language should be closer to the domain abstractions than a full-fledged general-purpose scripting language: while a language like Python is appropriate for scripting a 3D editor like Blender, a Python command line would never be appropriate as the editor's primary interface. The scripting language exists, thus, as a feature that provides support for when the user's goals outgrow the possibilities of the UI-level language.

### 2.1.3 Scripting languages

Scripting languages complement UI-level languages in end-user programmable applications. They allow advanced users (or even programming professionals) to provide extensions to the UI-level language when needed. Common examples of such extensions are adding custom functions to a spreadsheet's formula language, or designing richer game interactions than those available out-of-the-box in a game level editor.

This is, in a sense, an alternative look at the role of scripting. Ever since Ousterhout's seminal paper [Ous98], scripting languages are primarily regarded in relation to the core applications that sit below them: they are regarded as higher-level alternatives to system programming languages, and as "glue" languages that connect lower-level components. Here, we focus on higher-level programmability of applications. In this context, scripting exists to serve the needs of the end-user language that sits above it, providing unconstrained, Turing-complete extensibility to a UI-level language that remains focused on domain-specific elements. For example, while a scripting language for a 3D editor may deal in terms of typical programmer constructs such as records, lists and associative arrays, its end-user UI-level language should deal in terms of domain constructs such as 3D objects, textures and lighting sources.

The evolution of scripting languages points in the direction we indicate here. When we look at the development history of successful scripting languages such as Python and Lua, we see that these languages started out with clear goals of being easy to program, but over the years their development focus tends to favor adding constructs for advanced programmers. Lua grew from having a single numeric type to having distinct floating-point and integer numbers with bitwise operators, and gained advanced features such as coroutines and lexically-scoped closures. Python gradually shifted its focus away from adding libraries with ready-made components and into improving core language constructs. In contrast, end-user programmable applications like LabVIEW and Max/MSP advertise the number of new library functions added on each new release.

This marks a culture shift in scripting languages. The forerunner of modern scripting languages, Tcl, was seen as inseparable from its UI library to the point that the language was often mistakenly called "Tcl/Tk". Nowadays, the Python community favors adding libraries through the PIP package manager rather than merging them into the core language[1]. We see similar trends in all scripting languages, with their fledgling package managers: Perl and CPAN, Ruby and RubyGems, JavaScript and npm, Lua and LuaRocks.

Scripting languages today are seen less as a deliberately simplified tool for "non-programmers" and more as as a class of languages focused on rapid development, sharing some features such as dynamic type systems, automatic memory management and dynamic code loading. As these languages evolve and these features prove useful for programming in general, they are gaining ground in many fields beyond those originally identified with "scripting". Still, scripting—in the sense of embedding the power of a full programming language to provide advanced control of applications—remains an essential aspect of end-user programming.

## 2.2 Dataflow programming

"Dataflow" in an umbrella term that refers to computation evaluation dictated by the flow of data, the relationship between inputs and outputs in the program, rather than explicit programmer-dictated flow of control. The term "dataflow language" has been used to refer to various families of languages over time. We review some of these languages here to get an outlook of what we mean by the term.

Dataflow programs correspond to directed graphs, where nodes represent operations on data, and arcs represent connections through which data tokens can be sent from node to node. Operations have inputs (incoming edges receiving data) and outputs (outgoing edges sending data). An operation executes when all its inputs receive data tokens. The operation computes its function, and then "fires" the result through one or more outputs, making data available to other nodes. The order of evaluation, thus, depends exclusively on the flow of data, hence the name. In contrast, control-flow[2] oriented languages are those that feature explicit sequencing constructs written by the programmer: all imperative languages fall in this group. In a pure dataflow language, evaluation order is implicit and arbitrary, and there is full referential transparency [Den85]. Purely functional languages certainly fit

[1]Python developer Kenneth Reitz quipped that the "standard library is where modules go to die", as standardizing libraries tends to slow down their evolution for the sake of backward compatibility [Phi13].

[2]We use "dataflow" spelled without a hyphen as this is an established term in the literature to refer to a particular class of languages; "control-flow", on ther other hand, is hyphenated as it refers merely to the notion of a flow of control.

this description. What came to be known as the dataflow paradigm, however, is a particular style of representing these data relationships, in particular the focus on the flow of data, and how to represent it in the face of iteration or recursion. In any case, just like not all functional languages are pure, neither are all dataflow languages.

### 2.2.1 A brief history of dataflow

Early history of dataflow languages is closely tied to that of dataflow computers. In the 1970s, dataflow hardware architectures, with large number of processing elements interconnected to form dataflow graphs, were considered as an alternative to the von Neumann model [WP94]. Given that the dataflow model would be inherently parallel in its design, it was hoped that this would overcome the difficulties of writing concurrent software and the CPU-memory bottleneck in Von Neumann machines. Programming dataflow computers required new languages, as it was particularly hard to map traditional imperative languages to these architectures efficiently.

The motivations of the many dataflow languages created over the years, both textual and graphical, varied widely. In 1973, Kosinski presented DFL [Kos73], a graphical language for operating systems programming, and he motivated the dataflow approach by requirements of paralellism and modularity. Lucid [AW77] is an early example of a textual dataflow language, with a stated goal of making a language amenable to proof reasoning that remained friendly to imperative programmers. Their approach was to make a functional-style language that included iteration constructs. Programs are written in single-assignment style; in iterations, variable updates are written as `next` *`var`* `=` *`exp`*, where occurrences of variable *`var`* in expression *`exp`* represent its value in in the previous iteration.

Since the 1990s, the focus of dataflow languages moved away from performance and paralellism and into software engineering territory, with a particular growth in the field of dataflow visual programming languages [JHM04], of which the most prominent example in the industry is LabVIEW [? ], a commercial application for industrial automation first released in 1986 and marketed to this day, where the graphical language is tightly integrated with the development and execution environment. More recently, research on dataflow shifted its focus once again towards parallelism [LCM+08, GS11, GKM+11, BJ13].

### 2.2.2 Static and dynamic dataflow models

It is often desirable for dataflow programming models to allow the representation of iteration, in which a subgraph executes multiple times. If we consider that the only rule for firing a node is that its all inputs have data ready for consumption, it is conceivable that the early part of a new iteration could begin executing before the previous iteration has finished to run, in a pipeline style of execution. Further, if the iterations of a loop have no data dependencies between them, it should be possible to run all iterations of a loop in parallel. These possibilities, however, complicate both the processing and memory models.

A simple restriction that causes a major simplification to the dataflow model is to add another rule for firing nodes: a node is fired only if all its input ports have data tokens ready for consumption and if its output ports have no data tokens pending for consumption by further nodes. This model is called the *static* dataflow model. In the static dataflow model, memory management is simple, as each arc in the graph represents one storage unit for a data token [Den85]. Synchronous languages such as Lustre [HCRP91] and Lucid [AW77] implement static dataflow models.

For exploiting the full possibilities of dataflow parallelism, *dynamic* models were devised. In common, they all lift the restriction that the output ports need to be empty for a node to execute. A straightforward interpretation of this model is that arcs now represents buffers with multiple data tokens between nodes. Management of memory and processing units becomes more complicated, as it becomes necessary to tag tokens with bookkeping data, for managing concurrent flows of different iterations. Tagged-token models support parallel loops by associating to each input value a tag, indicating which iteration that value is a part of. A node $f$ with input ports $p_1, p_2 ... p_n$ will feature $n$ queues $q_1, q_2 ... q_n$, containing tagged values of the form $(v, i)$, indicating a value $v$ for iteration $i$. The graph will fire $f(v_1, v_2 ... v_n)$ only when it is able to extract from the queues a complete set of tokens $\{(v_x, i) \in q_x | 1 \leq x \leq n\}$ with the same tag $i$.

There is a number of architectural challenges for implementing dynamic dataflow models efficiently: several models were proposed, and this continues to be an area of active research [KSP15]. The choice of dataflow execution model is not only an implementation issue. For one, it affects language design, as typical tagged-token models use explicit looping constructs, which mark points where loop iteration tags in tokens should be incremented or reset [Den85].

### 2.2.3 Data-driven and demand-driven dataflow

Another major design decision when choosing a dataflow model is whether to use data-driven or demand-driven evaluation. These modes of evaluation correspond to what in programming languages is conventionally called, respectively, eager (or strict) and lazy evaluation.

Data-driven evaluation maps to eager evaluation: the availability of input data triggers the evaluation of nodes that are connected to them, producing data for nodes connected further ahead in the graph.

Demand-driven evaluation maps to lazy evaluation: the request of an output causes a node connected to it to be triggered. If that node's input ports have data, the node will execute, producing the output. If inputs are not available, the nodes to which these inputs are connected are then triggered, cascading the triggering backwards until inputs are available. Once inputs are available, nodes are evaluated and their result values propagate forward via their output ports. This way, only the parts of the graph which produce output data execute.

These terms come from the field of computer architecture, when dataflow machines were proposed as alternatives to the von Neumann model. In [TBH82], Treleaven classified architectures as "data-driven" or "demand-driven", but also called data-driven architectures "data-flow computers" and demand-driven architectures "reduction computers". Nowadays, both models are considered styles of dataflow [Hil91, AS94].

### 2.2.4 Uni and bi-directional flow

A field that is closely related to dataflow languages is that of constraint-based systems, but as we will see, these concepts are not equivalent and it is important to establish the distinction here. Constraints allow the specification of relationships between values—for example, one may specify the equation `9*C = 5*(F - 32)` about variables `C` and `F` representing a temperature in Celsius and Fahrenheit, and if either `C` or `F` is updated to a new value, the other one is recomputed so that the equation of the constraint continues to hold. Constraint systems can be classified as one-way or multi-way: one-way constraints are systems where each value is defined by one equation, so that constraints can be solved by propagation as performed by spreadsheet recalculation; multi-way constraints are those where any variable of an equation can trigger an update and where the constraints for a variable can be expressed via a system of equations [SMFBB93].

One-way constraints are a restricted case of multi-way constraints; traditionally, dataflow systems can be seen as an example of one-way constraint systems [DFR14]. For this reason, the term "constraint system" is more often understood to refer to systems based on multi-way constraints: in [ASS96], for example, it is stated that "nondirectionality of computation is the distinguishing feature of constraint-based systems". The power of a multi-way constraint system depends on the power of the solver used to satisfy its systems of equations. For example, Cassowary [BBS01] is a widely used constraint solver supporting linear equations and inequalities. Since we want to focus on core language semantics rather than solver algorithms, our work will restrict itself to traditional dataflow languages, or, in other words, those with one-way constraints only. From a dataflow point of view, a constraint update is a modification of the structure of the dataflow graph.

# Chapter 3

# Design alternatives for dataflow UI-level languages

Research on the dataflow paradigm has seen a decrease in activity since the early 1990s. The revolution of visual languages inspired by GUI systems that was expected by some never came to pass, as object-oriented programming systems remained textual. Massively-parallel dataflow hardware architectures proved difficult to implement efficiently and are now historical artifacts—modern GPUs obtain massive parallelism via vector processing based on imperative machine code.

Practical use of dataflow languages, however, has not gone away, and it is arguably more widespread than ever. Dataflow is the paradigm of choice in a large number of programmable end-user applications. Dataflow-based UIs, especially visual ones based on box-line graph representations, make it easy for the user to inspect intermediate results and therefore understand what is going on in the application [Hil92], making the relationships between data more concrete to the user.

So, dataflow languages continue to be created and used successfully, but mostly away from the sphere of research. A closer look at this class of languages, especially from the perspective of end-user programming, is long overdue. In this chapter, we begin to explore the design space of dataflow languages by presenting a series of design alternatives that come up in their construction.

## 3.1 Hils's classification of design alternatives

In [Hil92], Daniel Hils presented an extensive survey of dataflow visual languages (both domain-specific and general-purpose) and produced a list of design alternatives through which those languages can be classified in various axes. Here, we reuse and expand upon this list while shifting our focus from dataflow visual languages to what we call dataflow UI-level languages. We felt it was necessary to coin this term and to make a distinction from the more common term "dataflow visual languages" for two reasons.

First, we include "UI-level" because we want to focus only on dataflow languages which take *center* stage as the *application interface*; in other words, those languages that are central to the UI, and not an optional component. Being languages that are integrated into an application of a specific domain, this classification excludes

| **Design dimension** | **Design alternatives** |
|---|---|
| Box-line representation | no; yes |
| Iteration | no; limited; yes (cycles); yes (construct) |
| Subprogram abstraction | no; yes |
| Selector/distributor | no; yes |
| Flow of data | uni-directional; bi-directional |
| Sequence construct | no; yes |
| Type checking | no; yes (limited); yes (all types) |
| Higher-order functions | no; yes |
| Execution mode | data-driven; demand-driven |
| Liveness level | 1 (informative); 2 (significant); 3 (responsive); 4 (live) |

Table 3.1: Hils's design dimensions for dataflow visual languages

general-purpose programming languages.

Second, we avoid the term "visual" because we want to cover the entire spectrum of languages that fall within our focus, without having to worry whether a language is visual or not. Notably missing from Hils's work is the most widely used application-specific dataflow language—the spreadsheet formula language. Granted, the relationship between data cells in a spreadsheet is presented symbolically rather than visually, but it is remarkable how, apart from this fact, spreadsheets would fit every other aspect of that study. Our work is concerned with semantics and not syntax; the visual presentation of a language is a syntactic feature.

Apart from this distinction on the criteria for selecting languages, the list of design alternatives can be directly reused in our work. Hils's original list of design alternatives, summarized in Table, 3.1, can be described as follows:

- *box-line representation*: whether the language is presented as a visual graph depicting nodes as boxes and edges as connected by lines or as some other visual approach, such as hierarchical frames (this is the only syntactic aspect discussed, but we retain it in the table for completeness);

- *iteration*: whether the language supports iteration, either through graph cycles or explicit constructs for iterated execution of subgraphs (here we expand on the original yes-no classification to consider the kind of iteration facility and also if some limited form of iteration is available in languages which do not allow iterating arbitrary subgraphs);

- *subprogram abstraction*: whether the language supports abstracting subgraphs as reusable subprograms[1];

- *selector and/or distributor*: whether the language includes classic dataflow constructs for directing the routing of data through a control input. A selector is a function $\sigma(v_1, ..., v_n, k) = v_k$, i.e. the $k$-th input is forwarded to the node's only output; a distributor is a function $\pi(k, v)$, where value $v$ is fired to the node's $k$-th output;

[1]In Hils's original article this item was called "procedural abstraction"; we changed the term to avoid confusion.

- *sequential execution construct*: whether the language breaks away from pure dataflow by providing an explicit construct for specifying the order of evaluation of actions, independently from data dependencies;
- *type checking*: in Hils's classification, this entry discussed whether the arcs in the dataflow graph are typed—in other words, whether the graph is statically typed. Statically-typed graphs provide checking when constructing the graph, disallowing incorrect connections. A dynamically-typed graph may still have typed tokens, resulting in type errors at runtime only. Note that this is different from static and dynamic dataflow models (as discussed in 2.2.2);
- *higher-order functions*: whether the language supports nodes that take functions as arguments. None of the languages studied in this work support this, and in [Hil92] this feature was only present in general-purpose languages and in Hils's own scientific visualization language DataVis [Hil91];
- *execution mode*: whether the language is data-driven or demand-driven, as discussed in Section 2.2.3;
- *liveness level*: a classification developed by Tanimoto [Tan90] with a four-level scale of liveness for visual programming tools. In level 1, "informative", the visual representation is a non-executable auxiliary representation (like flowchart documentation for textual programs). In level 2, "informative and significant", the visual representation is the executable program, but program editing and execution are separate activities. In level 3, "informative, significant and responsive", editing the visual representation triggers updates to the evaluation of the program; there are no separate edit and execution modes. In level 4, "informative, significant, responsive and live", the system updates continuously, updating the visual representation in response to data updates, and not only to user inputs. The liveness level affects the evaluation rules of the language, since levels 3 and 4 allow changing the dataflow graph dynamically.

## 3.2 An extension to Hils's classification

In addition to the axes of comparison proposed by Hils, we identified a number of additional criteria to compare these languages. This new list of design alternatives for dataflow languages both informed and was informed by the study of existing end-user applications and their languages that we will present in the following chapters. We hereby present an extension to the classification proposed by Hils, summarized in Table 3.2, which we believe helps comparing end-user applications of different domains and shines a light on the semantic aspects of their underlying programming languages.

### 3.2.1 Dataflow model

Apart from the classification between data-driven and domain-driven dataflow, perhaps the most important distinction in terms of semantics for a dataflow language is

| Design dimension | Design alternatives |
|---|---|
| Dataflow model | static; dynamic |
| N-to-1 inputs | no; yes (auto-merge); yes (queueing) |
| Time-dependent firing | no; yes |
| Rate-based evaluation | no; synchronous; cyclo-static; quasi-static; dynamic |
| Separate program and UI | no; yes |
| Indirect connections | no; yes (static); yes (runtime-evaluated) |
| Textual sub-language | no; yes (functional); yes (imperative) |

Table 3.2: Additional design dimensions for dataflow end-user languages

whether its dataflow model is static or dynamic, according to definitions presented in Section 2.2.2.

The choice between static and dynamic dataflow models embodies typical computing trade-offs. The static model, with a single data token present per edge, is simpler, and thus easier to implement and also keeps memory use under control. The dynamic model, while more complex, may produce a more efficient execution, maximizing the opportunities for exploiting parallelism.

The question of which one is easier to understand, especially in the context of end-user programming, is not at all obvious. On the one hand, the static model is conceptually simpler and program execution may be easier to introspect. On the other hand, to the user's point of view the restrictions may seem arbitrary: in a number of situations the user may justifiably want to specify cycles (iterative financial calculations, audio-delay feedback, etc.); the dynamic model may present a more free-form experience for the user. Still, misusing cycles is an easy way for the user to make their program go haywire.

### 3.2.2 N-to-1 inputs

A practical question that is closely related to the dataflow model is whether the language allows what we will call "N-to-1 inputs" into a node. A node in a dataflow graph represents a unit of functional execution and may have multiple inputs, akin to a function $f(a, b, c)$ with multiple input arguments. The question here is whether the language allows multiple edges connecting to a *single* input port in a node—in our analogy, that would mean having multiple incoming values for argument $a$ available at the time of executing $f$.

Traditional static dataflow models usually forbid N-to-1 inputs. When two input signals are to be entered to a single input port, an explicit "merge" node needs to be added, which applies some logic to decide which of the inputs is forwarded to the node.

When N-to-1 inputs are allowed, the language needs a policy for dealing with them. Languages employing a dynamic model featuring input queues may naturally allow data from different sources into a single queue. In static dataflow models, N-to-1 inputs may be allowed by making the merge operation implicit.

It is interesting to note that depending on the representation of the dataflow graph, N-to-1 inputs can be not only semantically disallowed, but be made impossible to express syntactically. In a typical textual language, one simply cannot pass two different variables at once to the first argument of a function. That is the case, for example, in spreadsheet formula languages.

### 3.2.3 Time and rate-based evaluation

An aspect of program evaluation that is not much a design decision as much as it is a domain-dependent requirement is the need to take time into account. If a programmable application needs to process events which happen in a time-dependent manner or according to fixed rates, that needs to be reflected in its language. Still, when this necessity arises there is still a design space to be explored on how to handle it.

There are two ways to consider the issue of time. One is that of rate-based execution, which refers to the need of consuming or producing data at a certain frequency. The other is that of time-dependent nodes, that is, nodes that behave differently depending on the time when they were fired [Wip10].

The handling of rate-dependent data often brings performance concerns. Outside of the world of end-user programming, the signal-processing community has been using dataflow models for decades for handling data streams efficiently, while taming the complexity of dynamic dataflow models. *Static dataflow* models have not only bounded memory, but their scheduling can also be determined ahead-of-time, allowing for efficient compilation. However, handling one token at a time is a severe limitation for stream processing. *Synchronous dataflow* [LM87] is an extension of the model in which the number of data samples produced or consumed by each node on each invocation is specified a priori. It still allows compile-time memory and scheduling analysis. *Cyclo-static dataflow* [BELP95] is another extension which extends the expressiveness of the synchronous model while retaining its analysis properties. In it, the consumption and production rates for a node can vary over time, respecting a predetermined cycle. A number of other models have also been proposed to further extend the model of computation: *quasi-static* models such as boolean-controlled dataflow [Buc93] and parameterized dataflow [BB01] attempt to restrict dynamic scheduling to data-dependent nodes while statically scheduling the rest of the graph, with further enhancements continuing to be an area of research [FZHT13, SFG+15].

Time-dependency is closely related to rate-dependency, since predictable rates at a known clock speed results in predictable times, but rate-based evaluation may also be motivated by concerns other than time, such as memory efficiency as buffer sizes can be are minimized with optimized schedules. Time-dependent execution refers to all sorts of dependency on time, such as nodes that return the current time, delay nodes that pause for a given number of microseconds, and so on. Supporting this kind of nodes has its own set of concerns as it can bring a level of indeterminacy to the evaluation, affecting scheduling.

In the context of end-user programming, an application with rate-dependent data processing may combine rate-based and non-rate-based events (e.g. handling audio streams and button clicks). The interaction between these two worlds in the evaluation engine of the language also needs to be taken into account.

### 3.2.4 Separate programming and use views

Another design decision that is worth observing in an end-user programmable application is whether it presents separate interface views for editing the dataflow program and for using the resulting user-written program, or if use of the program happens in the same interface where it is edited. By "separate interface views" we mean here completely distinct presentations: for example, one "edit" view presenting a box-line diagram as the user constructs their program and a "use" view presenting a form when they use it.

Note that this is different from the question having separate "edit" and "use" modes in the first place, which is the distinction between Tanimoto's liveness levels 2 and 3 [Tan90]. These design aspects are orthogonal: an application may present separate "edit" and "use" modes within a single UI paradigm, characterizing liveness level 2; or it could feature distinct UIs for use and editing and yet allow those to be used simultaneously and in a responsive, non-modal, manner matching liveness level 3 or 4.

While strictly speaking the existence of separate views for development and use is a matter of presentation, and hence with immediate effects on syntax, we claim that this design decision is most closely related to a language's pragmatics: having the "source program" always visible or not is a clear indicator of the language's intended mode of use. A language which conflates the use and programming interfaces clearly intends the user and programmer to be one and the same, while a language with separate interfaces for these two scenarios may also steer people towards different roles. Most important to our concerns, here, is whether this design choice for the application affects the overall design of its programming language.

It is also important to note that having separate UIs for creating the user-developed program and using it doesn't mean that the language is at a lower layer according to the architecture discussed in Section 2.1.2. The language continues to be UI-level as long as the main interface for producing documents in the application consists of the environment for interacting with the language. This does not change if once the user is done composing the document (that is, writing the program), they switch to another UI mode for using it.

### 3.2.5 Indirect connections

Dataflow programs are constructed as graphs, and applications using this model need to employ some representation to depict nodes and their connections. The explicitness of relationships between units of computations is arguably a big part of the appeal of dataflow for end-user programming. However, as programs grow in size, graph representations can grow unwieldy. Scalability in visual representations is a concern, embodied in the folk aphorism known as the "Deutsch limit", which states that visual languages cannot have effectively more than 50 visual primitives on screen at the same time [McI98]. A way around this issue is to introduce indirect connections. This presents a trade-off, however: while it reduces visual clutter, it does away with the aforementioned explicitness of relationships.

This is not only a matter of representation, and this becomes clearer when we consider the most popular example of indirect connection in textual languages: a pointer. Indirect connections whose targets are determined at runtime, such as pointers and references, increase the expressive power of the language. This can also have far-reaching impacts to its semantics. For one, node scheduling can no longer be performed statically. In opposition, if the target of an indirect connection is constant, this indirect connection is not unlike a connector in old flowcharts, used to transfer flow from one page to the next.

### 3.2.6 Textual sub-language

The final dimension of design we enumerated is the presence or absence of a textual sub-language included within the UI-level language presented to the user.

The presence of textual sub-languages is a common pattern, even in visual languages. A typical use for them is representing mathematical expressions, since textual syntax based on mathematics such as `a - b + 5` is natural and even some computer conventions such as `*` and `/` for multiplication and division have become broadly understood.[2]

We are also interested in the paradigm of this textual sub-language, as it is indicative of the level of integration with the dataflow language as a whole. A functional sub-language is a natural match to the host dataflow language, representing solely a syntactic shortcut. If the textual sub-language includes imperative features, however, this necessarily means that it goes beyond dataflow and extends the core language's semantics.

## 3.3 Non-dataflow UI-level languages

To make the boundaries of our scope clearer, it is worth dedicating a few words to UI-level languages that fall outside of the dataflow model.

Since end-user applications tend to be in general document-centric, with the user interface dominated by the "current document" and the interaction focusing on manipulating this document, it is natural that data-oriented approaches become appealing choices when one wants to enable end-user programming. However, we do not claim that dataflow languages are the best model of programmable interaction for all kinds of end-user applications.

For example, applications employing a storytelling paradigm [LHML08, MTdS13] benefit from a control-oriented model. In these scenarios, the fundamental building block for the user's programming experience is the *sequencing* construct: "this happens, then that happens". Sequencing is notably absent in the pure dataflow model. However, as we will see in Chapter 6, a dataflow language may include it to simplify operations involving timing.

It is possible to combine data and control flow in graphs. That is the case

[2]One might argue that even in typical programming languages, the sub-grammar of mathematical expressions (including relational and logical operators) is an embedded domain-specific language of its own, as it usually contains a number of syntactic rules that do not apply anywhere else in the language, such as infix operators and precedence rules, and that are often similar even in languages which otherwise vary widely.

of Blueprints[3], the graphical language used for gameplay scripting in the Unreal game engine. There are different kinds of edges in the graph, representing either data connections in dataflow style, or control-flow sequencing. Typical Blueprints programs, however, are of an imperative nature, and the resulting graph resembles a traditional flowchart. This kind of language, therefore, falls beyond our intended scope in the discussion of dataflow semantics.

## 3.4 Case studies

In the following chapters, we will present in-depth case studies analyzing the semantics of three end-user programmable applications:

- Pure Data, a multimedia application focused on audio synthesis widely used by the computer music community;
- the spreadsheet formula language, as used in Excel, LibreOffice and Google Sheets;
- LabVIEW, an engineering application focused on data acquisition.

Those languages were chosen because of their relevancy, their distinct domains, and because they cover different points of the language design space in many aspects. We modelled their semantics by implementing definitional interpreters for them in Haskell, written in the style of structural operational semantics. Each interpreter is written as a set of evaluation functions, one for each language construct. As a whole, this set serves as a specification of the language being interpreted. These functions take as inputs the current state of the execution environment and a language construct to be evaluated, and return the new state of the environment after evaluation of the construct.

The interpreters do not intend to be a complete specification of each language, but aim to capture the notable features of their semantics. We will discuss the languages in terms of the various design dimensions presented in this chapter. We will see to which extent these languages are purely declarative, how time affects the flow graph, the semantics of iteration constructs, their support for abstraction, and so on. In these studies, we identify what are the common patterns that are present, assessing to which extent those languages are "variations on a theme". We also pinpoint aspects where their ad hoc designs show.

The underlying goal is to understand what is the required functionality expected by applications for dataflow-based interfaces and how different languages solve similar problems. Also, having these interpreters side-by-side allows us to reason on the designs of those languages under similar terms.

[3]`https://docs.unrealengine.com/latest/INT/Engine/Blueprints/`

## 3.5 Discussion: On the use of definitional interpreters

Providing the definition of a language through the use of a definitional interpreter written in another language is a classic technique for describing semantics. In his 1972 paper, Reynolds [Rey72] describes the process of writing definitional interpreters, introducing the idea of defunctionalization and an early description of continuations. Both techniques were put to use in our interpreters.

Writing an interpreter for a language, at first glance, seems like a way to unavoidably consider its entire semantics, since all constructs need to be described in the precise language of computer code. However, as observed by Reynolds, the primary concern when writing a definitional interpreter is being aware of which aspects of the language used to implement the interpreter (which Reynolds calls the *defining* language) leak into the semantics of the language to be interpreted (called the *defined* language). Using a construct of the defining language to implement the same construct in the defined language gives us a working implementation, but fails to describe what the semantics of the construct is.

In our interpreters, we made conscious decisions of which semantic aspects to explicitly implement and which ones not. Aspects which are not related to the dataflow evaluation logic are orthogonal to our discussion and not of particular interest, so they were simply reused from the defining language, in our case, Haskell: the addition of numbers, for example, is simply the addition as in Haskell, with the same overflow rules, etc. Aspects which are crucial to the dataflow logic were carefully separated: Haskell's lazy evaluation semantics do not leak into our defined languages (interpreters are defunctionalized, so both eager and lazy evaluation are explicitly implemented). We also avoided the more sophisticated features of Haskell such as monads and type classes, using the language as much as possible as a form of executable $\lambda$-calculus (albeit one with a considerable amount of notational convenience). We hope that this will also prove beneficial for the reader unfamiliar with Haskell. At the same time, we consider a description written in a programming language to be more in line with the interdisciplinary nature of this work, as opposed to one written in the notation of operational semantics used by the programming language community.

It is important to stress here that a definitional interpreter is not written as a "prototype" interpreter. A prototype is typically a concise implementation of a subset of a program, written to give an idea of how the final product would work. Our concern in a definitional interpreter is with the precise specification of the defined language, and while we are implementing subsets of languages, we sacrifice concision whenever precision is important. The end result is quite different from a typical prototype. In particular, no care is given to performance: time, for example, is implemented by simulating the clock and incrementing it manually, resulting in a deterministic and precise description of events.

Finally, using an executable interpreter provides us with an easy way to test our implementation (and hence our definition) with larger, somewhat more practical examples.

# Chapter 4

# Case study: Pure Data

Pure Data [P+15] (also known as Pd) is a graphical language originally designed for music and later expanded for multimedia in general via extension packages. It is widely used in the computer music community [BH15], and is the free software successor of the successful commercial product Max/MSP[1], created by the same author.

## 4.1 Overview of the language

In this section, we give a presentation of the application, which doubles as an overview of its UI-level language. When a user opens Pure Data, they are greeted with a main window which contains the main menu and a blank log panel. The user's first step is to create a new file or open an existing one, which then opens a canvas window where the dataflow graph representing audio synthesis operations is edited and used.

### 4.1.1 Nodes and values

In Pure Data, a program, called a *patch* in Pd jargon[2], is represented as a graph of nodes (operations) and edges (the flow of data between them). The user creates nodes of different kinds in a canvas an

Data flowing between nodes may be either discrete messages (numbers, strings, or lists of these base types) or audio data. Nodes in Pure Data have *creation arguments*, input ports (*inlets*) and output ports (*outlets*). The creation arguments are the initial contents of nodes, given by the programmer via a simple textual language. A node is of one these types:

- *atom box* - represents a value which can be edited by the user while in *run mode*; editing the value (typing a new number or via the mouse scroll wheel) sends the updated value through its output port;

- *message box* - can be clicked by the user while in run mode, producing one or more messages;

[1]`https://cycling74.com/products/max/`

[2]In this section, terms in Pd jargon are presented in italics.

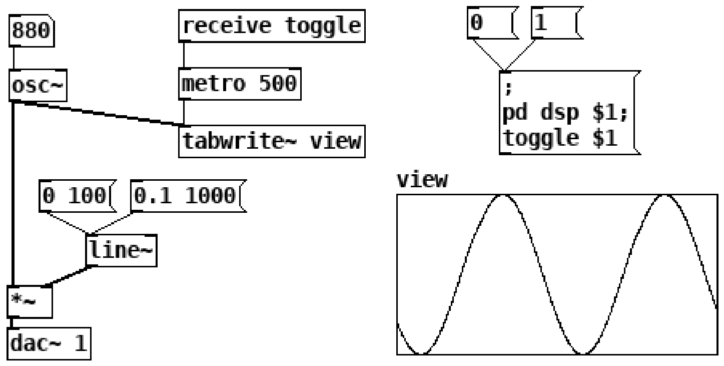


Figure 4.1: A patch demonstrating the oscillator node `osc~`, based on an example from the Pure Data documentation.

- *graphical objects* - receive data and represent it visually, for example, as a plot;
- plain *objects* - represent Pd functions. There are two kinds of Pd functions: those that operate on discrete messages only, and those that operate on audio streams (denoted with a "~" suffix in their names); accordingly, inlets and outlets are also identified as handling messages, audio or both. An object implementing an audio-processing function may have both message and audio inlets and outlets. A non-audio object may only send and receive messages.

Figure 4.1 gives an illustrative example of a Pure Data patch. Thin lines are message connections, thicker lines are audio connections; their types are determined by their outlets. Boxes with a slanted top-right corner are atom boxes, editable in "run mode"; boxes with a recessed right side are clickable message boxes.

This program produces a 880Hz sine wave (via the the [`osc~`] object), which has its amplitude multiplied ([`*~`]) according to a linear function ([`line~`]) prior to be sent to the DSP[3] output ([`dac~`], as in "Digital-to-Analog Converter"). The sine wave is also sampled at a rate of 500Hz ([`metro 500`]) into a visual representation ([`tabwrite~ view`]). The user may interact with the program by setting a new frequency to the oscillator (typing in new values in the atom box that initially contains "`880`"), by clicking the message boxes that reconfigure the linear function ([`0 100`], which will configure [`line~`] to produce a slope from its current value to 0 over the course of 100ms, or "`0.1 1000`", which will cause the value to slide to 0.1 over 1s) or by toggling the update of the graph by sending the value [`0`] or [`1`] to [`metro 500`], done here via an indirect connection for demonstration purposes, sending the value through a receiver object declared as [`receive toggle`]. We will expand on these concepts below.

### 4.1.2 Graph evaluation

Pure Data has two modes of execution, an *edit mode* in which the graph structure can be edited, and a *run mode* in which input values can be produced by interacting with the graph. The DSP engine can be toggled on or off independently of the

[3]Digital Signal Processing

mode of execution. Most importantly, the dataflow program is running at all times: the user can create nodes in the graph, switch to run mode, trigger a loop that produces audio waves, return to edit mode and modify the graph while the loop is still running.

Execution in Pure Data is synchronous. The tree of DSP nodes is processed periodically, updating their buffers and propagating audio data. A complete propagation of audio is called a *DSP tick*, during which only audio nodes are processed. When an input or timing event happens, the entire cascade of messages produced is processed in sequence, before the next DSP tick [P+15]. This means that excessive processing of messages may produce noticeable audio drop-outs, but the audio buffer with a user-specified size mitigates this.

It is possible to wire nodes in the Pd interface so that cycles in the graph are created. If those graphs involve audio wires, the resulting loops in the audio processing chain are detected and propagation is halted. When cycles happen in the message flow, messages may also produce infinite loops, but being dynamic they cannot be detected a priori: those are caught during run mode, reporting a stack overflow.

### 4.1.3 Messages and the textual sub-language

Nodes in Pure Data are similar to spreadsheet cells, in the sense that they are primarily containers for a textual language. Once the user selects "Object" or "Message" in the "Put" menu, a new node with the corresponding shape appears in the canvas and the keyboard focus switches to it immediately. The user then types in the contents of the node, based on which Pure Data determines its number of inlets and outlets, which appear as connectors at the top and bottom of the node's box. For example, in Figure 4.1 node [`metro 500`] has two inlets (with only the left one connected) and one outlet; node [`dac~ 1`] has only one inlet and no outlets.

When editing the graph, apart from connecting outlets to inlets and positioning nodes, everything else is done with the textual language. It is a very basic imperative command language, more similar to the language of primitive interactive shells than that of typical programming languages. When a command is entered in an object box, the first token represents the internal function to be executed (e.g. [`osc~`], [`metro`]) and the remaining tokens are arguments to that function. In message boxes, multiple commands may be entered, separated by a semicolon. The first command is always a message to be sent through the outlet of the message box; in subsequent commands, the first token is the receiver and the remaining tokens are messages. Values received in inlets are available for token substitution using numbered variables such as `$1`.

The system of messaging via named receivers allows the program to send data between nodes that are not explicitly linked via connections. The relationship between a message and a receiver can be thought of as an implicit edge in the dataflow graph. Since the textual language of messages supports variable arguments fed via input ports and the first argument of a command can be set to a variable, receiver destinations can change during execution: this makes the dataflow graph, in effect, mutable at runtime. This, however, is limited to discrete messages, and only the destination of a message can be changed at runtime, and not the identifier a receiver

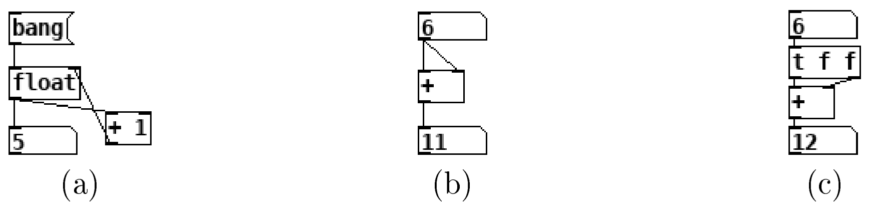


Figure 4.2: Behavior of hot and cold inlets in Pure Data

is "listening" to. The flow of audio data cannot be re-routed while in run mode, but disconnections can be simulated multiplying a signal by 0.

A precise definition of the language semantics will be given in Section 4.2, but to give a feel of the language we will describe two examples from Figure 4.1. The first example is the simplest use of message boxes: message box [`0 100`] simply sends tokens `0` and `100` to [`line~`] when clicked. The second example demonstrates variable substitution and indirect connections. When the user clicks message boxes [`0`] or [`1`], this feeds the value to message box [`;pd dsp $1;toggle $1`], which contains three commands. The first command is empty, indicating there is nothing to be sent to the outlet port. The second command is sent to receiver `pd`, which is Pure Data's internal receiver—here it is used to enable or disable the audio engine. The third command sends the value `0` or `1` to the user-defined receiver `toggle`.

### 4.1.4 Node triggering

To avoid triggering functions with multiple arguments many times, as the input arguments arrive through different paths of the graph, Pure Data has the notion of *hot* and *cold* input ports. Only the hot inlet triggers the node; data arriving to a cold inlet stays buffered there and is only consumed when a hot input port is triggered. This allows, for example, to create a "`float`" node that increments its value every time it is triggered, by sending its output to an increment node ("`+ 1`") and sending the result back via a secondary, cold inlet (see Figure 4.2(a)). This will *not* create an infinite loop, because the result of the increment will only be consumed next time the hot inlet of the node is triggered.

Due to the way Pure Data handles messages, for some node types the order in which connections are performed in its interface can change the semantics of the resulting graph, so that two visually identical graphs can produce different results, depending on the order in which the lines between nodes were drawn in the UI. This behavior is documented as a possible user pitfall in the documentation [P+15]. The example in Figure 4.2(b) was taken directly from the documentation, which says: "Here, I connected the left inlet before connecting the right hand one (although this is not evident in the appearance of the patch)". In fact, disconnecting node [`6`] from [`+`] and connecting the left inlet first makes it work as intended, updating the bottom box with the double of the top box as we edit the value at the top. The documentation suggests as a workaround disambiguating the graph by using the object [`t`] (shorthand for [`trigger`]), which triggers its outlets from right to left, as depicted in Figure 4.2(c).

## 4.2 An interpreter modeling the semantics of Pure Data

We produced an executable model of the core semantics of Pure Data [P+15, Zmö14] written in Haskell, which includes stateful nodes for multiple data types, message and audio objects, identifier-based message passing and the intra-node language for message objects.

Our model implements a simulation of the run mode, in which the graph is fixed and input events are fed into it. It replicates the synchronous behavior of Pure Data, processing cascades of message events and DSP ticks for propagating audio buffers. It does not implement "abstractions" (which is Pure Data jargon for loading subprograms from separate files), and dollar-expansion in object boxes, which is only used when loading abstractions. Still, our model is complete and precise enough so that the numeric values produced in audio buffers can be converted to a sound file format and played back.

This entire section describing the interpreter was written in Literate Haskell [HL15], and is a description of the entire interpreter, including its complete listings. The text was written so that it should be understandable without a detailed reading of the source code, but the sources are nevertheless included for completeness and as supporting material for the interested reader. The source code in `.lhs` format (LaTeX with embedded Haskell) is also available at `https://hisham.hm/thesis/`.

This implementation uses only standard modules included in the Haskell Platform:

```
import Data.Sequence (Seq, fromList, index, update, foldlWithIndex)
import qualified Data.Sequence as Seq (length)
import Data.Foldable (foldl', toList)
import Data.List (sort, intercalate, find)
import Text.Printf
import Data.Fixed
import Data.Binary.Put
import qualified Data.ByteString.Lazy as ByteString
import Control.Monad
import Debug.Trace
```

### 4.2.1 Representation of programs

A Pure Data program (called a "patch") is represented with the *PdPatch* data type in our model, which contains a sequence of nodes, a sequence of connections between nodes, and the pre-computed topological sort of audio connections (stored as a list of integer indices).

```
data PdPatch = PdPatch {
                    pNodes   :: Seq PdNode,
                    pConns   :: Seq (· ▷ ·),
        pDspSort :: [Int]
      }
```

The primitive values in Pure Data are called "atoms": they can be numbers, strings (called "symbols") or opaque pointers. Opaque pointers are used by graphical objects only, so those will be omitted here.

```
data PdAtom = PdFloat   Double
            | PdSymbol String
  deriving (Eq, Ord)
instance Show PdAtom where
  show (PdFloat f)  = show f
  show (PdSymbol s) = s
```

Nodes may be objects, atom boxes or message boxes. In Pure Data, objects are initialized via "creation arguments": a string of arguments, represented here as a list of atoms. We also store in an object its number of inlets and outlets. Atom boxes and message boxes always have one inlet and one outlet each.

```
data PdNode = PdObj       [PdAtom] Int Int
            | PdAtomBox PdAtom
            | PdMsgBox   [PdCmd]
  deriving Show
```

Message boxes contain commands written in the textual sub-language of Pure Data. Here, we represent commands not as a string, but in parsed form, consisting of a receiver and a list of tokens (which may be literal values or numbered references to inlet data (written `$n` in the textual language). Note that a single message box may contain a number of messages.

```
data PdCmd = PdCmd PdReceiver [PdToken]
  deriving Show
data PdReceiver  = PdToOutlet
                 | PdReceiver String
                 | PdRDollar Int
                 | PdReceiverErr
  deriving Show
data PdToken = PdTDollar Int
             | PdTAtom   PdAtom
  deriving Show
```

Finally, we represent the connections of the graph as a sequence of adjacency pairs, where each pair is represented as a $(\cdot \rhd \cdot)$ value, itself composed of two pairs: the node index and outlet index for the source, and the node index and inlet index for the destination. Throughout the interpreter, we will often use the names $(src, outl)$ and $(dst, inl)$ to refer to those indices.

```
data (· ▷ ·) = ((Int, Int) ▷ (Int, Int))
  deriving Show
```

### 4.2.2 Representation of states

The representation of a state in our interpreter is a structure containing the following values: the step count, which will double as our timestamp, since Pure Data has time-based execution; the state for each node; the text contents of the Pd logger window; and future events scheduled by the interpreter.

```
data PdState = PdState {
                  sTs       :: Int,
                  sNStates  :: (Seq PdNodeState),
                  sLog      :: [String],
                  sSched    :: [PdEvent]
               }
  deriving Show
```

The state for each node, on its turn, contains a sequence of atom buffers, one for each inlet, and an internal memory (represented as a list of atoms). Memory consumption during execution is therefore variable, characterizing a dynamic dataflow model.

```
data PdNodeState = PdNodeState (Seq [PdAtom]) [PdAtom]
  deriving Show
```

We represent events with a timestamp, the node index indicating which node was triggered, and a list of atoms representing the event data (such as the number entered by the user in an atom box).

```
data PdEvent = PdEvent {
                  eTs    :: Int,
                  eNidx  :: Int,
                  eArg   :: [PdAtom]
               }
  deriving (Show, Eq, Ord)
```

### 4.2.3 Execution

The execution mode of Pure Data is data-driven. The user triggers events via its interface, and those events cause a cascading series of firings. The user may trigger events by clicking nodes, entering numbers (or using MIDI devices, which are equivalent to entering numbers).

#### 4.2.3.1 Main loop

The execution of the interpreter, therefore, is a loop of evaluation steps. The driver function takes a number of steps, the patch to run, a list of timed events, accumulating a list of states. We are interested in all states, not only the final one, because we want to be able to inspect the results of the execution over time.

Note that the patch itself, $p$, remains unchanged over time. This is typical of a language with liveness level 2: the patch cannot be modified during execution.

```
runSteps :: Int → PdPatch → [PdEvent] → [PdState]
runSteps nSteps p events =
  reverse $ snd $ foldl′ acc (events, [initialState p]) [0 . . (nSteps − 1)]
  where
    absTime :: [PdEvent] → Int → [PdEvent]
    absTime evs ts = map (λe → e {eTs = (eTs e) + ts}) evs
    acc :: ([PdEvent], [PdState]) → Int → ([PdEvent], [PdState])
    acc (events, states@(s : ss)) step =
      (sort (events_next ++ absTime (sSched s′) step), s′ : states)
      where
        (events_curr, events_next) = span (λ(PdEvent ts _ _) → ts ≡ step) events
        s′ = runStep p (s {sSched = [ ]}) events_curr
```

The loop above extracts the sublist of relevant events for the current timestamp, and hands it over to the main evaluation function, *runStep*, which, given a patch, the current state, and a list of events, produces a new state.

Processing a step may produce new future events to be scheduled. These are sorted along with the existing events of the input. Runtime events are produced by the interpreter using relative timestamps (where 0 means “now”), so we adjust them to absolute time using auxiliary function *adjTime*.

The function *runStep* processes events and the DSP tree. Following the specified semantics of Pure Data, this happens in an alternating fashion: all pending messages for a given timestamp are handled, and then the entire DSP tree is processed.

```
runStep :: PdPatch → PdState → [PdEvent] → PdState
runStep p s events =
  let
    s′ = runImmediateEvents p $ foldl′ (runEvent p) s events
    s″ = if (sTs s) ‘mod‘ 2 ≡ 0
      then runDspTree p s′
      else s′
  in
    s″ {sTs = (sTs s) + 1}
```

In our model, the DSP tree is processed at half the rate of the message-based events (hence, *runDspTree* is called at every other run of *runStep*). Assuming that a step in our interpreter is 1 ms, this means the DSP engine runs once every 2 ms (the default configuration of Pd runs the engine every 1.45 ms; with a 64-sample buffer, this amounts to an audio sample rate of 44,100 Hz — with this simplification in our interpreter, we get 36,000 Hz).

The Pure Data documentation specifies that "In the middle of a message cascade you may schedule another one at a delay of zero. This delayed cascade happens after the present cascade has finished, but at the same logical time". So, events scheduled during the current step with a relative timestamp set to zero are immediately executed before running the DSP tree:

```
$runImmediateEvents :: PdPatch \to PdState \to PdState$
$runImmediateEvents\ p\ s =$
  let $z = [ev \mid ev \leftarrow (sSched\ s), eTs\ ev \equiv 0]$
  in if $z \equiv [\,]$
    then $s$
    else $runStep\ p\ s\ z$
```

### 4.2.3.2 Event processing

Two kinds of events can be triggered by the user. Message boxes may be clicked, processing all commands stored inside them, or new numeric values may be entered into atom boxes. We do it producing a synthetic firing of the relevant node.

```
$runEvent :: PdPatch \to PdState \to PdEvent \to PdState$
$runEvent\ p\ s\ event@(PdEvent\ ts\ i_N\ args) =$
  $fire\ p\ (index\ (pNodes\ p)\ i_N)\ args\ (i_N, 0)\ s$
```

The *fire* function invokes the appropriate action for a node, producing a new state.

```
$fire :: PdPatch \to PdNode \to [PdAtom] \to (Int, Int) \to PdState \to PdState$
```

Depending on the type of node, we perform different actions. For message boxes, we feed the incoming atoms into the inlet, and then we fold over its triggering its commands, like when they are clicked by the user. As we will see below in the definition of *runCommand*, this may fire further nodes either directly or indirectly.

```
$fire\ p\ (PdMsgBox\ cmds)\ atoms\ (i_N, inl)\ s =$
  let
    $(PdNodeState\ ins\ mem) = index\ (sNStates\ s)\ i_N$
    $ns' = PdNodeState\ (update\ inl\ atoms\ ins)\ mem$
    $s' = s\ \{sNStates = (update\ i_N\ ns'\ (sNStates\ s))\}$
  in
    $foldl'\ (runCommand\ p\ i_N)\ s'\ cmds$
```

For objects and atom boxes, we hand over the incoming data to the *sendMsg* handler function, which implements the various behaviors supported by different Pure Data objects. The function *sendMsg* returns a tuple with the updated node state, log outputs produced (if any), data to be sent via outlets and new events to be scheduled. We update the state with this data, adjusting the node index of the returned events to point them to that of the current node ($i_N$): a node can only schedule events for itself. Finally, we propagate the data through the outlets, processing them from right to left, as mandated by the Pure Data specification.

```
$fire\ p\ node\ atoms\ (i_N, inl)\ s =$
  let
    $ns = index\ (sNStates\ s)\ i_N$
    $(ns', logw', outlets, evs) = sendMsg\ node\ atoms\ inl\ ns$
    s′ = s {
      sNStates = update i_N ns′ (sNStates s),
      sLog     = (sLog s)   ++ logw′,
      sSched   = (sSched s) ++ (map (λe → e {eNidx = i_N}) evs)
    }
    propagate :: PdState → ([PdAtom], Int) → PdState
    propagate s (atoms, outl) =
      if atoms ≡ [ ]
      then s
      else forEachInOutlet p (i_N, outl) atoms s
  in
    foldl′ propagate s′ (zip (reverse outlets) [length outlets − 1 . . 0])
```

When propagating data, we send it to every connected outlet of a node. A node may have multiple outlets and multiple nodes can be connected to a single outlet. This function takes the patch, a $(node, outlet)$ pair of indices indicating the source of the data, the data itself (a list of atoms), and the current state. It folds over the list of connections of the patch, firing the data to the appropriate inlets of all matching connections.

```
forEachInOutlet :: PdPatch → (Int, Int) → [PdAtom] → PdState → PdState
forEachInOutlet p srcPair atoms s =
  foldl′ handle s (pConns p)
  where
    handle :: PdState → (· ▷ ·) → PdState
    handle s (from ▷ (to@(dst, inl)))
      | srcPair ≡ from = fire p (index (pNodes p) dst) atoms to s
      | otherwise      = s
```

Pure Data commands are written in its textual language. Commands may include references to data obtained via inlets of the node using the $\$n$ notation. For example, sending `10 20` to a message box containing `pitch $2 velocity $1` connected to an object box `print` will print to the log window the string `pitch 20 velocity 10`.

In function *runCommand* below, we run a given command *cmd* on a node (with index $i_N$) by first obtaining the inlet data currently stored in the node state. Then we perform \$-expansion on the command's tokens. Then, based on the receiver of the message, we route it through the graph (forwarding it to every outlet, in a classic dataflow fashion) or symbolically, sending it to all objects configured as a receivers for the given name.

```
runCommand :: PdPatch → Int → PdState → PdCmd → PdState
runCommand p i_N (PdState ts nss logw evs) cmd =
  let
    (PdNodeState ins mem) = index nss i_N
    inletData = index ins 0
    (recv, atoms) = dollarExpansion cmd inletData
    ns' = PdNodeState (update 0 [] ins) mem
    nss' = update i_N ns' nss
    s' = PdState ts nss' logw evs
  in
    case recv of
    PdToOutlet →
      forEachInOutlet p (i_N, 0) atoms s'
    PdReceiver r →
      forEachReceiver p r atoms s'
    PdReceiverErr →
      printOut [PdSymbol "$1: symbol needed as message destination"] s'
```

The process of $-expansion is a simple substitution, where the receiver must be a string. Invalid indices are converted to zero. (In Pure Data, they also produce an error message to the log window, but here we omit this for brevity.) We also handle here a few syntactic shortcuts: a messages with a sole number like `1.0` expands to `float 1.0`; lists starting with a number get the prefix `list`.

```
ffor a f = fmap f a
dollarExpansion :: PdCmd → [PdAtom] → (PdReceiver, [PdAtom])
dollarExpansion (PdCmd recv tokens) inlData =
  (recv', atoms')
  where
    inlAt n = if n < length inlData then inlData !! n else PdFloat 0
    recv' =
      case recv of
      PdRDollar n →
        case inlAt n of
          PdSymbol s → PdReceiver s
          _          → PdReceiverErr
      _ → recv
    atoms' =
      normalize $ ffor tokens (λtoken →
        case token of
        PdTDollar n    → inlAt n
        PdTAtom atom → atom
      )
    normalize atoms@[PdFloat f]      = (PdSymbol "float" : atoms)
    normalize atoms@(PdFloat f : xs) = (PdSymbol "list" : atoms)
    normalize atoms                  = atoms
```

Indirect connections are handled similarly to outlet connections, but instead of folding over the list of connections, we fold over the list of nodes, looking for objects declared as `receive` *name*. Note that the search happens over the statically-declared list of nodes of the patch. While it is possible construct a message at runtime and determine the receiver dynamically, it is not possible to change the identifier of a `receive` node at runtime.

```
forEachReceiver :: PdPatch → String
                           → [PdAtom]
                           → PdState → PdState
forEachReceiver p name atoms s =
  foldlWithIndex handle s (pNodes p)
  where
    handle :: PdState → Int → PdNode → PdState
    handle s dst (PdObj (PdSymbol "receive" : (PdSymbol rname : _)) _ _)
      | name ≡ rname = forEachInOutlet p (dst, 0) atoms s
    handle s _ _ = s
```

### 4.2.3.3 Audio processing

The processing of audio nodes is very different from that of message nodes. Before execution, the audio nodes are topologically sorted, producing an order according to which they are evaluated on each DSP update. For simplicity, we do not compute this order at the beginning of execution, and merely assume it is given as an input (in the *dspSort* field of $p$).

As the list of nodes is traversed, each object is triggered (applying the *performDsp* function) and then the new computed value of its audio buffer is propagated to the inlets of the nodes to which they are connected.

```
runDspTree :: PdPatch → PdState → PdState
runDspTree p s =
  s {sNStates = nss'}
  where
    nss' = foldl' handle (zeroDspInlets (sNStates s) (pDspSort p)) (pDspSort p)
    handle :: (Seq PdNodeState) → Int → (Seq PdNodeState)
    handle nss iN =
      foldl' (propagate outputs) nss'' (pConns p)
      where
        obj = index (pNodes p) iN
        ns@(PdNodeState ins mem) = index nss iN
        (outputs, mem') = performDsp obj ns
        nss'' = update iN (PdNodeState ins mem') nss
        propagate :: [[PdAtom]] → (Seq PdNodeState) → (· ▷ ·) → (Seq PdNodeState)
        propagate outputs nss ((src, outl) ▷ (dst, inl))
          | src ≡ iN    = addToInlet (dst, inl) (outputs !! outl) nss
          | otherwise = nss
```

Each audio node has a 64-sample buffer that needs to be cleared before each traversal. Note that this is different from handling inlets in message objects: for message objects, the inlets become empty once consumed. Here, we need the inlet buffers to be filled with zeros.

```
zeroDspInlets :: (Seq PdNodeState) → [Int] → (Seq PdNodeState)
zeroDspInlets nss dspSort =
  fromList $ clearNodes 0 (toList nss) (sort dspSort)
    where
      zeroInlets :: Int → (Seq [PdAtom])
      zeroInlets n = fromList $ replicate n (replicate 64 (PdFloat 0.0))
      zeroState :: PdNodeState → PdNodeState
      zeroState (PdNodeState ins mem) =
        PdNodeState (zeroInlets (Seq.length ins)) mem
      clearNodes :: Int → [PdNodeState] → [Int] → [PdNodeState]
      clearNodes i_N (st : sts) indices@(i : is)
        | i_N ≡ i      = zeroState st : clearNodes (i_N + 1) sts is
        | otherwise = st           : clearNodes (i_N + 1) sts indices
      clearNodes i_N nss [] = nss
      clearNodes i_N []   _ = []
```

The reason why we fill the inlets with zeros is because when multiple nodes connect to the same inlet in a DSP object, additive synthesis is performed: the values of the incoming buffer are added to the current contents of the inlet buffer, subject to saturation (audio values are internally floats between -1.0 and 1.0).

```
addToInlet :: (Int, Int) → [PdAtom] → (Seq PdNodeState) → (Seq PdNodeState)
addToInlet (dst, inl) atoms nss = update dst ns' nss
  where
    saturate (PdFloat f) = PdFloat (max (−1.0) (min 1.0 f))
    satSum (PdFloat a, PdFloat b) = saturate $ PdFloat (a + b)
    ns@(PdNodeState ins mem) = index nss dst
    atoms_old = index ins inl
    atoms_new = fmap satSum (zip atoms_old atoms)
    ns' = PdNodeState (update inl atoms_new ins) mem
```

In Section 4.2.4.8 we will present *performDsp*, which implements the various DSP objects supported by this interpreter.

#### 4.2.3.4 Initial state

Finally, for completeness of the execution model, we present here the functions that create the initial state.

```
emptyInlets :: Int → Seq [PdAtom]
emptyInlets n = fromList (replicate n [])
initialState :: PdPatch → PdState
initialState (PdPatch nodes _ _) = PdState 0 (fmap emptyNode nodes) [] []
  where
  emptyNode node =
    case node of
      PdAtomBox atom   → PdNodeState (emptyInlets 1)   [atom]
      PdObj        _ inl _ → PdNodeState (emptyInlets inl) []
      PdMsgBox   _        → PdNodeState (emptyInlets 1)   []
```

### 4.2.4 Operations

The graphical language of Pure Data is graph-based and contains only nodes and edges. The contents of nodes (object boxes, message boxes and atom boxes) are textual. Like there are two kinds of edges (message and audio), there are also two kinds of objects. Audio-handling objects are identified by a ~ suffix in their names (the Pure Data documentation calls them “tilde objects”. In our interpreter, plain objects are implemented in the *sendMsg* function (Section **??**) and tilde objects are implemented in the *performDsp* function (Section 4.2.4.8).

For printing to the log, we present a simple auxiliary function that adds to the output log of the state value.

```
printOut :: [PdAtom] → PdState → PdState
printOut atoms s =
  s {sLog = (sLog s) ++ [intercalate " " $ map show atoms]}
```

The implementation of all non-audio nodes is done in the *sendMsg* function, which pattern-matches on the structure of the node (which includes the parsed representation of its textual definition).

```
sendMsg :: PdNode → [PdAtom] → Int → PdNodeState
           → (PdNodeState, [String], [[PdAtom]], [PdEvent])
```

Unlike the *runCommand* function used in the firing of message boxes, which causes global effects to the graph evaluation (via indirect connections) and therefore needs access to the whole state, *sendMsg* accesses only the node’s private state, producing a triple containing the new private node state, any text produced for the output log, a list of messages to be sent via the node’s outlets and any new events to be scheduled.

Similarly to *sendMsg*, we define a single function that performs the operations for all audio-processing objects:

```
performDsp :: PdNode → PdNodeState → ([[PdAtom]], [PdAtom])
```

The *performDsp* function takes the object, its node state and outputs the audio buffer to be sent at the node’s outlets, and the updated internal data for the node.

We did not implement the full range of objects supported by Pure Data since our goal was not to produce a full-fledged computer music application, but we included a few representative objects that allow us to demonstrate the interpreter and the various behaviors of objects.

#### 4.2.4.1 Atom boxes

When given a float, atom boxes update their internal memory and propagate the value. When given a `bang`, they just propagate the value.

```
sendMsg (PdAtomBox _) (PdSymbol "float" : fl) 0 _ =
  (PdNodeState (fromList [ ]) fl, [ ], [PdSymbol "float" : fl], [ ])
sendMsg (PdAtomBox _) [PdSymbol "bang"] 0 ns@(PdNodeState _ mem) =
  (ns, [ ], [PdSymbol "float" : mem], [ ])
```

#### 4.2.4.2 An object with side-effects: `print`

The `print` object accepts data through its inlet and prints it to the log console. It demonstrates the use of the log console as a global side-effect.

```
sendMsg (PdObj (PdSymbol "print" : xs) _ _) (PdSymbol "float" : fs) 0 ns =
  (ns, ["print: " ++ (intercalate " " $ map show (xs ++ fs))], [], [])
sendMsg (PdObj (PdSymbol "print" : xs) _ _) (PdSymbol "list" : ls) 0 ns =
  (ns, ["print: " ++ (intercalate " " $ map show (xs ++ ls))], [], [])
sendMsg (PdObj (PdSymbol "print" : xs) _ _) atoms 0 ns =
  (ns, ["print: " ++ (intercalate " " $ map show atoms)], [], [])
```

#### 4.2.4.3 An object with hot and cold inlets: `+`

In Pure Data, the first inlet of a node is the “hot” inlet; when data is received through it, the action of the node is performed. When data arrives in “cold” inlets, it stays queued until the “hot” inlet causes the object to be evaluated.

The `+` object demonstrates the behavior of hot and cold inlets. When a number arrives in the hot inlet, it sums the values in inlets 0 and 1 and sends it through its outlet. When a `bang` arrives in the hot outlet, the most recently received values in the inlet buffers are used for the sum instead.

```
sendMsg (PdObj [PdSymbol "+", n] _ _) [PdSymbol "float", fl] 0
            (PdNodeState ins mem) =
  let
    (PdFloat val_0) = fl
    inlet_1 = index ins 1
    (PdFloat val_1) = if inlet_1 ≡ [] then n else head inlet_1
    mem' = [PdFloat (val_0 + val_1)]
    ns' = PdNodeState (update 0 [fl] ins) mem'
  in
    (ns', [], [PdSymbol "float" : mem'], [])
sendMsg (PdObj [PdSymbol "+", n] _ _) [PdSymbol "bang"] 0
            (PdNodeState ins mem) =
  let
    inlet_0 = index ins 0
    (PdFloat val_0) = if inlet_0 ≡ [] then (PdFloat 0) else head inlet_0
    inlet_1 = index ins 1
    (PdFloat val_1) = if inlet_1 ≡ [] then n else head inlet_1
    mem' = [PdFloat (val_0 + val_1)]
    ns' = PdNodeState ins mem'
  in
    (ns', [], [PdSymbol "float" : mem'], [])
```

#### 4.2.4.4 Objects producing timed events: `delay` and `metro`

The `delay` object demonstrates how objects generate future events. We handle four cases: receiving a `bang` message schedules a `tick` event. When received, it outputs a `bang` to the node's outlets.

```
sendMsg (PdObj [PdSymbol "delay", PdFloat time] inl _) [PdSymbol "bang"] 0 ns =
  (ns, [], [], [PdEvent (floor time) 0 [PdSymbol "tick"]])
sendMsg (PdObj (PdSymbol "delay" : t) inl _) [PdSymbol "tick"] 0 ns =
  (ns, [], [[PdSymbol "bang"]], [])
```

The `metro` node, on its turn, expands on the `delay` functionality, implementing a metronome: it sends a series of `bang` messages at regular time intervals. It also has a second inlet which allows updating the interval.

We handle four cases: receiving a `bang` message to start the metronome, receiving a `stop` message to stop it, and receiving the internally-scheduled `tick` when the metronome is either on or off.

```
sendMsg (PdObj (PdSymbol "metro" : xs) inl _) [PdSymbol "bang"] 0
            (PdNodeState ins mem) =
  let
    inlet₁ = index ins 1
    (PdFloat time) = head (inlet₁ ++ mem ++ xs ++ [PdFloat 1000])
    ns' = PdNodeState (emptyInlets inl) [PdFloat time, PdSymbol "on"]
  in
    (ns', [], [[PdSymbol "bang"]], [PdEvent (floor time) 0 [PdSymbol "tick"]])
sendMsg (PdObj (PdSymbol "metro" : xs) inl _) [PdSymbol "stop"] 0
            (PdNodeState ins [PdFloat time, PdSymbol "on"]) =
  (PdNodeState ins [PdFloat time, PdSymbol "off"], [], [], [])
sendMsg (PdObj (PdSymbol "metro" : xs) inl _) [PdSymbol "tick"] 0
            ns@(PdNodeState ins [PdFloat time, PdSymbol "on"]) =
  (ns, [], [[PdSymbol "bang"]], [PdEvent (floor time) 0 [PdSymbol "tick"]])
sendMsg (PdObj (PdSymbol "metro" : xs) inl _) [PdSymbol "tick"] 0
            ns@(PdNodeState ins [_, PdSymbol "off"]) =
  (ns, [], [], [])
```

#### 4.2.4.5 Message handlers for audio objects: `osc~` and `line~`

Some audio objects in Pure Data also accept messages. The `osc~` object implements a sinewave oscillator. Sending a float to it, we configure its frequency, which is stored in the node's internal memory. Note that the actual oscillator is not implemented here, but in the DSP handler for this object type in function *performDsp*, in Section 4.2.4.8.

```
sendMsg (PdObj (PdSymbol "osc~" : _) _ _) [PdSymbol "float", PdFloat freq] 0
            (PdNodeState ins [_, position]) =
  (PdNodeState ins [PdFloat ((2 * pi) / (32000 / freq)), position], [], [], [])
```

The `line~` object implements a linear function over time. It can be used, for example, to implement gradual changes of frequency or amplitude. Its internal memory stores values *current*, *target* and *delta*. It accepts a message with two floats, indicating the new target value and the time interval to take ramping from the current value to the new target.

```
sendMsg (PdObj [PdSymbol "line~"] _ _)
            [PdSymbol "list", PdFloat amp, PdFloat time] 0
            (PdNodeState ins mem) =
  let
    [PdFloat current, PdFloat target, PdFloat delta] =
        if mem ≢ [] then mem else [PdFloat 0, PdFloat 0, PdFloat 0]
    mem′ =
        [PdFloat current, PdFloat amp, PdFloat ((amp − current) / (time ∗ 32))]
  in
    (PdNodeState ins mem′, [], [], [])
```

### 4.2.4.6 Cold inlets

Since cold inlets are passive and only store the incoming data in the inlet buffer without executing any node-specific operation, the implementation for cold inlets can be shared by all types of node.

```
sendMsg node (PdSymbol "float" : fs) inl (PdNodeState ins mem) | inl > 0 =
  (PdNodeState (update inl fs ins) mem, [], [], [])
sendMsg node atoms inl (PdNodeState ins mem) | inl > 0 =
  (PdNodeState (update inl atoms ins) mem, [], [], [])
```

### 4.2.4.7 Data objects: `float` and `list`

The `float` and `list` objects store and forward data of their respective types. They have two inlets for accepting new data. When given data through its first inlet, the object stores it in its internal memory and outputs the value through the outlet. When given data through its second inlet, it only stores the value. When given a unit event (called `bang` in Pure Data), it outputs the most recently received value (or the one given in its creation argument, or zero as a fallback).

```
sendMsg cmd@(PdObj (PdSymbol "float" : xs) inl _) atoms 0 ns =
  dataObject cmd atoms ns
sendMsg cmd@(PdObj (PdSymbol "list" : xs) inl _) atoms 0 ns =
  dataObject cmd atoms ns
dataObject (PdObj (PdSymbol a : xs) inl _) [PdSymbol "bang"]
  (PdNodeState ins mem) =
  let
    inlet₁ = index ins 1
    Just mem′ = find (≢ []) [inlet₁, mem, xs, [PdFloat 0]]
  in
    (PdNodeState (emptyInlets inl) mem′, [ ], [PdSymbol a : mem′], [ ])
dataObject (PdObj (PdSymbol a : xs) inl _) (PdSymbol b : fl) _ | a ≡ b =
  (PdNodeState (emptyInlets inl) fl, [ ], [PdSymbol a : fl], [ ])
```

#### 4.2.4.8 Audio handling operations: `osc~`, `line~` and `*~`

Audio handling is performed by function *performDsp*, which implements cases for each type of audio object.

Object `osc~` is the sinewave oscillator. It holds two values in its internal memory, *delta* and *position*, through which a wave describing a sine function is incrementally computed.

We handle two cases here: when the internal memory is empty, the parameters are initialized according to the *freq* creation argument; when the memory is initialized, we produce the new buffer calculating 64 new values, determine the next position to start the wave in the next iteration, store this value in the internal memory, and output the buffer through the node's outlet.

```
performDsp obj@(PdObj [PdSymbol "osc~", PdFloat freq] _ _) (PdNodeState ins [ ]) =
  performDsp obj (PdNodeState ins [PdFloat ((2 ∗ pi) / (32000 / freq)), PdFloat 0])
performDsp (PdObj [PdSymbol "osc~", _] _ _)
           (PdNodeState ins [PdFloat delta, PdFloat position]) =
  let
    osc :: Double → Double → Double → Double
    osc position delta idx = (position + (delta ∗ idx)) ‘mod′‘ (2 ∗ pi)
    output = map (PdFloat ∘ sin ∘ osc position delta) [0 . . 63]
    nextPosition = osc position delta 64
    mem′ = [PdFloat delta, PdFloat nextPosition]
  in
    ([output], mem′)
```

As described in Section 4.2.4.5, the `line~` object implements a linear ramp over time. As in `osc~` we handle two cases: when the internal memory of the object is empty, in which case we initialize it; and when it is initialized with *current*, *target* and *delta* values. The function varies linearly over time from *current* to *target*, after which, it stays constant at *target*.

```
performDsp obj@(PdObj [PdSymbol "line~"] _ _) (PdNodeState ins [ ]) =
  performDsp obj (PdNodeState ins [PdFloat 0, PdFloat 0, PdFloat 0])
performDsp (PdObj [PdSymbol "line~"] _ _)
           (PdNodeState ins [PdFloat current, PdFloat target, PdFloat delta]) =
  let
    limiter = if delta > 0 then min else max
    output = map PdFloat $ tail $ take 65
               $ iterate (λv → limiter target (v + delta)) current
    mem′ = [last output, PdFloat target, PdFloat delta]
  in
    ([output], mem')
```

The `*~` object multiplies the data from inlets 0 and 1. It is used, for example, to modify the amplitude of an audio wave.

```
performDsp (PdObj [PdSymbol "*~"] _ _) (PdNodeState ins []) =
  let
    mult (PdFloat a) (PdFloat b) = PdFloat (a * b)
    output = zipWith mult (index ins 0) (index ins 1)
  in
    ([output], [])
```

Finally, this is a default handler for *performDsp* that merely produces a silent audio buffer.

```
performDsp obj ns =
  ([toList $ replicate 64 $ PdFloat 0.0], [])
```

### 4.2.5 Demonstration

In Appendix A, we present a practical demonstration of the interpreter in use. We run the patch depicted in Figure 4.3. In includes atom boxes, objects and message boxes, and features message and audio processing, variable expansion, indirect messages and delayed execution. Running the interpreter emulates Pure Data's "use" mode: the graph cannot be modified, but atom boxes can receive new values and message boxes can be clicked, producing events. The interpreter simulates this interactive experience by receiving as input a list of interaction events with timestamps.

Appendix A also includes a main wrapper function that launches the interpreter and converts its output to .WAV format. The resulting audio file produced by the execution of the above graph by the interpreter when given a specific series of inputs can be played at `https://hisham.hm/thesis/`.

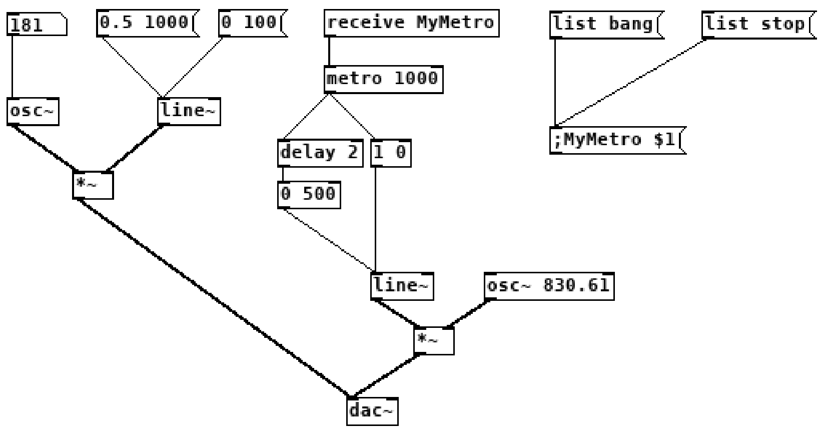
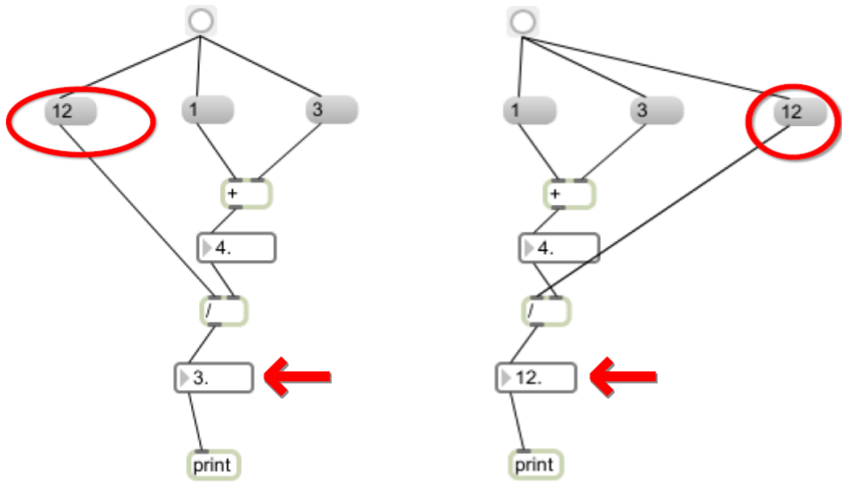


Figure 4.3: A Pure Data patch equivalent to the example code in Appendix A



(a) (b)

Figure 4.4: Impact of semantically significant layout in Max/MSP: Two graphs with identical sets of nodes and edges producing different results. Image adapted from [GKHB09].

## 4.3 Discussion: Syntax and semantics in visual languages

The design of end-user programmable applications is a field that spans both the worlds of end-user application design and of programming language design. These two areas are often distant from each other. The programming language design community is most often concerned with professionals, and most software written by end-user application developers nowadays is not programmable. These different groups naturally tend to be biased towards different aspects of design.

This separation happens to the detriment of end-user programming language design. Research in visual languages, in particular, is almost by definition focused primarily on syntax, given the area itself is defined by a style of representation. But while the programming language community may be guilty of sometimes dismissing matters of syntax and perpetuating arguably poor syntax in the name of familiarity [Has96, Tra05], the neglect of semantics in the design of the syntax of end-user applications has much graver consequences.

As we saw in Section 4.1.4, Pure Data specifies the right-to-left order in which outlets are processed, but the order in which messages are fired from various connections of a single outlet depends on the order the connections were made, making it possible to produce two visually identical graphs with different behavior. This is a major flaw in the language's syntax, and one that could be easily fixed by exposing in the syntax the ordered nature of the outgoing connections. Two possible solutions would be to draw the connector lines next to each other (and not starting from the same point) making the outlet wider as needed, or to make connection lines visually distinct (e.g. attaching a number to the lines, or simply painting them with different colors according to their sort order).

In Max/MSP, the commercial variant of Pure Data, it is not possible to produce two identical graphs with two behavior, but the solution chosen by its developers may be even worse. Max/MSP establishes that the order of message propagation follows that of the visual layout of a patch, that is, the coordinates of nodes on the canvas [GKHB11]. This means that moving nodes around, without changing their connections, can alter the behavior of the graph, leading to situations like the one depicted in Figure 4.4: two graphs with identical sets of nodes and edges producing different results. It is fair to assume that this counters any intuition a user may have about the interpretation of a diagram.

If syntax evolves at a slow pace in the world of textual languages, it may well be because it has reached a "local maximum" in the design space, where current syntaxes are "good enough", solutions to typical design requirements are well-known and it would take a major leap to move into something different. In the world of visual languages, it seems clear that we have not yet reached this point.

# Chapter 5

# Case study: spreadsheets

The spreadsheet is arguably the most successful end-user programmable application [SSM05]. Microsoft Excel is the most successful product of this kind, as part of Microsoft Office, a productivity suite with over 1.2 billion users, with an estimated 750 million users of Excel [Mic14]. The spreadsheet formula language is therefore the most popular programming language in the world.

The importance of this class of applications as well as concerns with the reliability of spreadsheets produced by users[1] have inspired academic work in areas such as debugging [BGB14], testing [CFR06] and expressivity [JBB03] of spreadsheets. Often, these works involve modeling the spreadsheet application in order to reason about it. Formal models of spreadsheets applied to research work usually simplify considerably their semantics of execution, assuming a model of computation with no specifications for error conditions and without advanced features such as indirect references [AE06, CSV09].

Real-world spreadsheets, however, are nothing but simple. Their design has evolved over the years, but to this day, spreadsheets follow the design paradigm of VisiCalc, created in 1979 for the Apple II. The user interface of a spreadsheet is dominated by a grid view, inspired by physical paper spreadsheets. Each cell of this grid presents a value, which may be calculated by a formula, which may reference values calculated by other cells. In the 1980s and 1990s several applications competed for this market, of which VisiCalc failed to maintain dominance. Lotus 1-2-3, Quattro Pro, Multiplan and Excel all introduced new features, such as instant recalculation, formula auto-fill (where the user can produce new formulas by dragging the cursor from one cell, producing formulas in new cells following a predictable pattern), multiple worksheets, and so on. As they adopted each other's features, the design of spreadsheet applications coalesced to that of Excel today, and by the 2000s the competition among proprietary spreadsheets was essentially over. The only popular alternatives to Excel that emerged since then did not gain adoption due to their feature set, but due to non-functional characteristics: LibreOffice Calc[2] became the main free software spreadsheet; Google Sheets is the most popular web-

[1]Losses caused by spreadsheet errors are calculated in the scale of millions of dollars [SR12].

[2]Originally StarCalc, a proprietary spreadsheet that was part of StarOffice, a productivity suite developed in the 1990s by German company StarDivision. This company was bought by Sun Microsystems, which open-sourced StarOffice as OpenOffice. LibreOffice emerged as a fork of OpenOffice after Oracle's acquisition of Sun.

based spreadsheet. For both products (and their users), compatibility with Excel is a major concern. Reproducing the semantics of Excel, therefore, ought to be considered a major goal for these projects.

For a long time, there was no specification whatsoever of the semantics of Excel, or even of its file format. It was only in 2006, 21 years after the initial release of Excel, that a specification was published detailing its file format [ISO12], due to political push towards open formats. Its semantics, however, remain vague. Unsurprisingly, as we will see below, these major competitors fail to implement full compatibility with Excel's formula language. Interestingly, even Excel Online[3], also produced by Microsoft as a response to Google Sheets, fails to implement the semantics of the formula language correctly.

## 5.1 The formula language

We studied the formula language as implemented by five spreadsheet applications:

- Microsoft Excel 2010, the spreadsheet market leader, matching the standardized document for the .xlsx format [ISO12], which is still current at the time of this writing;
- LibreOffice Calc 5, the leading free software spreadsheet, whose behavior also matches the latest specification documents for its file format [OAS11];
- Google Sheets, a prominent web-based spreadsheet[4];
- Microsoft Excel Online, Microsoft's own web-based version of Excel[5];
- Microsoft Excel for Android version 16.0.7127.1010, Microsoft's mobile version of Excel[6].

All five implementations have incompatibilities to various levels, but they are similar enough so that they can be understood as dialects of the same language. The exposition below presents this formula language as implemented by these spreadsheets, discussing it from a programming language design perspective. We focus primarily on Excel, since the other applications mostly follow its design, but we highlight their variations whenever they appear. In particular, Microsoft Excel for Android presents very close behavior to that of Excel Online. Whenever mobile Excel is not explicitly mentioned, the reader can assume that its behavior matches that of Excel Online.

Given both Google Sheets and Excel Online are server-based applications, their behavior may change at any time, so all observations about them are current at the time of writing.

[3]Available at `https://office.live.com/start/Excel.aspx`
[4]`https://sheets.google.com`
[5]`https://office.live.com/start/Excel.aspx`
[6]`https://play.google.com/store/apps/details?id=com.microsoft.office.excel`

| | English | Portuguese |
|---|---|---|
| Numbers | `6.2831` | `6,2831` |
| Function names | `SQRT(9)` | `RAIZ(9)` |
| Argument separators | `SUM(6,0.2831)` | `SOMA(6;0,2831)` |
| Literal matrices | `{1,2;3,4}` | `{1;2\3;4}` |
| Function arguments | `CELL("type",A1)` | `CÉL("tipo",A1)` |

Table 5.1: Syntactic changes in localized versions of Excel: all but the last one can be automatically converted by the application.

| | Microsoft Excel | LO Calc | Google Sheets | Excel Online |
|---|---|---|---|---|
| Refer to a sheet | `Sheet2!B1` | `Sheet2.B1` | `Sheet2!B1` | `Sheet2!B1` |
| Array formulas | `{=`*fn*`}` | `{=`*fn*`}` | `=ARRAYFORMULA(`*fn*`)` | N/A |
| Nested arrays | Error | Error | Flattened | Error |

Table 5.2: Some syntactic incompatibilities between spreadsheets

### 5.1.1 Syntax

Nowadays, all spreadsheet products use roughly the same formula language: the user of any spreadsheet user will be familiar with expressions such as `=A5+SUM(B10:B20)`. From a programming point of view, spreadheets have been described as a "first-order functional languages" [AE06]. The formula language is a language of expressions, with a number of predefined operators, such as `+` (addition), `&` (string concatenation), as well as a large number of built-in functions, such as `SQRT` and `SUM`. At first glance, it is not unlike the sub-language of expressions with infix operators and function calls contained in a number of programming languages.

The syntax of the language changes across translated versions of Excel. The names of functions are localized: for example, `SUM()` becomes `SOMA()` in the Portuguese version. Also, in many languages the comma is used as a decimal separator for numbers, forcing other uses of commas to be replaced by semicolons, and semicolons to be replaced by backslashes. Table 5.1 lists those differences. The application stores function names internally in English, so these and other operator changes are automatically translated when a file by opened in another version of Excel. This automatic conversion, unfortunately, is not complete. Some functions use a set of predefined strings as a kind of enumeration, such as `CELL("type", A1)`. These arguments were translated in localized versions of the functions, and these break when loaded in a different language: a function written in a Portuguese version of Excel as `CÉL("tipo", A1)` becomes `CELL("tipo", A1)` when loaded in an English version of Excel, which produces an error.

Even at a syntactic level, we found that while the studied applications have similar formula languages, they were all incompatible to each other. The surface similarity is certainly meant to lower the learning curve for users who are moving from one application to the other, but beyond the basics, incompatibilities show. Table 5.2 lists some of these incompatibilities. It is notable that even though only a mere three features are listed, no two columns in the table are alike.

| | Microsoft Excel | LO Calc | Google Sheets | Excel Online |
|---|---|---|---|---|
| `=TYPE({1})` | 64 | 64 | 1 | 64 |
| `=TYPE({1,1})` | 64 | 64 | 64 | 64 |
| `=TYPE(1+{1,2})` | 64 | 64 | 1 | 64 |
| `=TYPE({1,2}/0)` | 64 | 64 | 16 | 64 |
| `=TYPE(Z99)` *(Empty)* | 1 | `Err:502` | 1 | 1 |
| `=TYPE(A1)` *(Self)* | 0 *(Warning)* | `Err:502` | `#REF!` | 0 |

Table 5.3: Behavior of the `TYPE` function in spreadsheets

### 5.1.2 Values and types

The formula language is typed and it features scalar and matrix values. Scalar values may be of type boolean, error, string or number. Matrices are bidimensional, with unidimensional arrays as a particular case, and may contain scalar values of heterogenous types, but matrices cannot contain other matrices. Google Sheets accepts nested matrix syntax, but matrices are in effect flattened: `{1,{2,3},4}` is syntactically valid but it is equivalent to `{1,2,3,4}`. In all other three spreadsheets, nested matrix literals produce a syntax error.

The matrix notation with curly brackets can be only used to represent literals. It is not a general matrix constructor, and can only contain scalar literals, and not subexpressions: while `{1,2,3}` is valid, both `{1+1}` and `{A1}` are not.

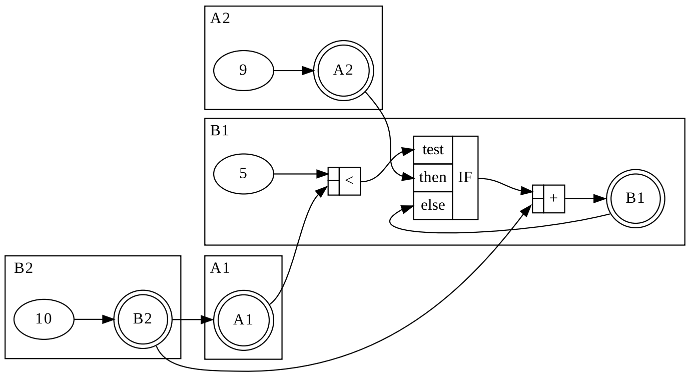
Most contexts feature automatic coercions, but not all. The expression `="1"+"2"` returns the number `3`, as does `=SQRT("9")`. But functions taking arrays of numbers, such as `SUM`, skip all string values. Therefore, given cells A1 to A3 containing `1`, `2` and `"100"`, we have that `=A1+A2+A3` returns `103`, and `=SUM(A1:A3)` returns `3`. Oddly, it does coerce boolean values, so, replacing A1 with `TRUE` in the previous example still yields the same results for both formulas.

Formula expressions may contain and manipulate scalars and matrices, including matrix literals, but cells can only represent scalar values. Each cell in the grid may be either empty or have contents entered by the user. *Cell contents* may be either nothing, a formula or a scalar literal value entered (attempting to enter a matrix literal causes it to be simply interpreted as a string). A cell also has a result value, based on its contents. *Cell values* may be either nothing (coerced to either `0` or `""` as necessary) or a scalar value produced from the calculation of the formula. Cells may also have formatting metadata that are only relevant to the UI presentation of values, but are not distinct types: for example, percentages and dates are still values of type number; colors and fonts are also formatting metadata of a cell.

Whenever a scalar value is expected and a matrix value is given, the value at position $(1, 1)$ of the matrix is returned. The UI of a spreadsheet displays cell values by default. Some functions, however, operate on cell contents—that is, there are functions $f$ so that $f$`(10)` and $f$`(A1)` with `A1=10` produce different results. An example is the function `TYPE`, which returns the data type of the cell contents as a number. Given a cell as an argument, `TYPE` returns `1` if the cell contains a number literal, `2` for strings and `4` for booleans. If the cell contains a formula, it returns `8` regardless of the data type of the formula's result, unless the formula results in

| | A | B |
|---|---|---|
| 1 | `=B2` | `=IF(5<A1;A2;B1)+B2` |
| 2 | `9` | `10` |

(a) A simple spreadsheet. A1 evaluates to 10, B1 evaluates to 19



(b) The same spreadsheet represented as a dataflow graph

Figure 5.1: The usual representation of a spreadsheet with a grid layout and a textual formula language, and its conceptual dataflow graph displaying data dependencies.

an error, in which case it returns `16`, or if it contains a matrix literal, in which case it returns `64` whether the expression results in an error or not. When given an expression as an argument, `TYPE` returns the type of the value of the evaluated expression. This complicated behavior, which is not implemented consistently among spreadsheets (Table 5.3 shows some incompatibilities), illustrates how poorly data types are presented to users in spreadsheets.

## 5.2 Evaluation model

The collection of cells and formulas in a spreadsheet forms a dataflow graph, and evaluation of each cell follows a top-down evaluation of the abstract syntax tree of its formula. Figure 5.1(a) depicts the typical visual representation of a spreadsheet (here, with all formulas exposed for clarity—normally only one formula is visible at a time, and cells display their computed values). Figure 5.1(b) depicts the same data as a dataflow graph[7]. This top-down evaluation corresponds to a typical demand-driven dataflow model.

The evaluation rules of individual nodes, however, are far from simple. Built-in functions `IF()` and `CHOOSE()` are evaluated lazily (`IF(1<2, 10, 0/0)` returns `10` and `0/0` is never evaluated), but functions `AND()` and `OR()` are not. These two functions do not perform short-circuiting: `OR(TRUE, 10)` returns `TRUE` but `OR(TRUE, 0/0)` returns `#DIV/0!`, an error value. To check that the evaluation disciplines of `IF` and `AND`/`OR` are indeed different, we escaped the purely functional formula language

[7]A quick remark on syntax: while the representation in Figure 5.1(a) is definitely more concise, 5.1(b) makes it a lot more evident that there is a cycle.

by writing a BASIC macro that produces a side-effect, popping up a dialog box. In both Excel and LibreOffice, the spreadsheets supporting BASIC macros, `OR(TRUE, PopDialogBox())` pops a dialog box but `IF(1<2, 10, PopDialogBox())` does not. Google Sheets supports JavaScript macros, and while its API explicitly blocks IO side-effects such as dialog boxes in formula macros, we were able to reproduce this test by writing a recursive function that causes a stack overflow, with similar results. Excel Online does not support running macros of any kind.

Errors are propagated as the formula is evaluated from left to right. This behavior is relevant to language compatibility since errors can be detected by functions such as `ISNA()`, which returns `TRUE` for `#N/A` and `FALSE` for any other error or non-error values. Given cells `A1` containing `=1/0` (evaluates to `#DIV/0`) and `A2` containing `#N/A`, `ISNA(A1+A2)` evaluates to `FALSE` and `ISNA(A2+A1)` evaluates to `TRUE`.

### 5.2.1 Array formulas

An *array formula* is a formula that is marked to be evaluated in a special array-oriented evaluation model. In Excel and LibreOffice a formula is marked as an array formula by confirming its entry pressing Ctrl+Shift+Enter, and the UI displays the formula enclosed in brackets, as in `{=A1:B5+10}`; Google Sheets uses a function-style annotation, as in `=ARRAYFORMULA(A1:B5+10)`. In the array formula evaluation mode, when ranges are given as arguments to scalar operations (such as + in the above example), the range is decomposed and the operation is performed for each element. The results of an array formula, therefore, may extend to several cells. Once the user enters an array formula, the required number of cells is filled with the results, with the initial cell being the top-left entry of the result matrix.

For the array formula above, `{=A1:B5+10}`, the result is a matrix with two columns and five rows, in which each cell is filled as if the range was substituted by a scalar corresponding to the given offset in the range, such that, for each cell *(x, y)* of the resulting matrix, its value is equivalent to that of `=INDEX(A1:B5,` *x*`,` *y*`)+10` (where `INDEX` is the function that takes element in row x, column y of the given range). This example belies the complexity in the evaluation of array formulas, for a simple substitution of ranges for their elements is not sufficient. When a function such as `SUM()` expects an array argument, the full range is given to the function.

This showcases a behavior that is very different from that of most programming languages: evaluation of sub-expressions is context-sensitive. The way a sub-expression is evaluated may vary according to an enclosing function call, perhaps several levels up in the syntax tree of the expression. The expected types of arguments for built-in functions define whether expressions given to them will have scalar or matrix evaluation. Within an array formula, for scalar arguments, arrays are destructured so that a scalar element is given.

The UI presentation of the array formula also influences the resulting values of cells: an array formula is always entered over a rectangular group of one or more cells. This group has a user-defined size that may or may not match the size of the matrix value of the array formula. The default size of the cell group matches that of the matrix value, but the cell group may be resized by the user by dragging the selection corner. Growing the group beyond the size of the result matrix may result

| | | Microsoft Excel | LO Calc | Google Sheets | Excel Online |
|---|---|---|---|---|---|
| `=SUM(SQRT({10,20}))` | F | 7.6344 | 3.1622 | 3.1622 | 7.6344 |
| | AF | 7.6344 | 7.6344 | 7.6344 | |
| `=SUM(SQRT(A1:A2))` | F | #VALUE! | #VALUE! | #VALUE! | #VALUE! |
| | AF | 7.6344 | 7.6344 | 7.6344 | |
| `=SUM(SQRT(INDIRECT({"A1","A2"}))` | F | #VALUE! | 3.1622 | 3.1622 | #VALUE! |
| | AF | #VALUE! | 7.6344 | 3.1622 | |
| `=SUM(INDIRECT({"A1","A2"})` | F | 10 | 10 | 10 | 10 |
| | AF | 10 | 30 | 10 | |
| `=SUM(MINVERSE(A1:B2))` | F | 4.163E-17 | 27756E-17 | 0 | #VALUE! |

Table 5.4: Formula evaluation incompatibilities between spreadsheets

in cells filled with `#N/A`; conversely, shrinking the cell group may hide parts of the result matrix.

We say it "may" result in `#N/A` because the precise semantics are a bit more complicated: if any of the dimensions of the result matrix is 1, increasing the size of the cell group in this dimension will produce repeated values in that dimension. For example, if the array formula `={10,20}` which produces a single-row matrix is inserted in a $3 \times 2$-group `A1:B3`, then the row will be duplicated, and cells `A1:A3` and `B1:B3` will present the same contents, namely: `10`, `20` and `#N/A`. If the matrix representation of an array formula value is $1 \times 1$ (which is also the case if the result value is not a matrix), all values in its cell group will be identical.

Array formulas are a niche feature: from the Enron corpus of 15935 industrial spreadsheets used in [AHH15], we independently assessed that 185 of them use array formulas (1.16%). However, arrays are pervasive in Excel: ranges such as `A1:A5` reference arrays of cells, and common functions such as `SUM()` take arrays as arguments. Functions that expect arrays as arguments evaluate these arguments in an *array context*, producing different values than arguments evaluated in a *scalar context*. In Excel, an array context of evaluation can produce iterated calculation of scalar functions, like in the context of array formulas. This is not implemented in LibreOffice or Google Sheets. In these two applications, iterated execution happens only in array formulas. Interestingly, Excel Online, which does not support array formulas, does implement iterated execution in array contexts of plain formulas. Table 5.4 illustrates these incompatibilities (in the second column of this table, F denotes plain formula mode, AF denotes array formula mode). In the first two examples, the enclosing `SUM` function imposes an array context over scalar function `SQRT`, triggering its iterated calculation in Excel and Excel Online plain formulas. In the last two examples, assuming `A1` and `A2` contain `10` and `20` and `B1` and `B2` contain `30` and `40`, `INDIRECT({"A1","A2"})` produces an array `{10,20}` which is coerced to scalar `10` in all modes except LibreOffice's array formula mode; in the third example, Excel and Excel Online fail to propagate the array context in a doubly-nested function. In the last row, we see that the spreadsheets also have inconsistencies in their evaluation order leading to observable differences in their results due to floating point calculations.

Array formulas are a powerful feature: they implement a separate evaluation model for formulas, and have been used to demonstrate that spreadsheets can model relational algebra [Tys10]. Still, they are usually disregarded when discussing the semantics of spreadsheets, and do not feature on any of the works cited in this chapter. The Excel documentation is vague when explaining their evaluation logic, resorting to examples [Mic16]. In fact, in the standardization process of spreadsheet file formats, the complete specification of formula evaluation was a contention issue: the draft specification of the OASIS OpenFormula did not specify formula evaluation, which led a Microsoft Office team member to raise issues about it [Jon05]. However, Microsoft's own specification did not fully specify formula evaluation at the time either , and even the following draft of OpenFormula did not specify array formulas [OAS06]. Eventually, specification of array formulas were included in both OASIS ODF 1.2 [OAS11] and Microsoft Office Open XML [ISO12], but even then the specification was informal and mostly driven by examples, in both documents.

## 5.3 An interpreter modeling spreadsheet semantics

In the previous sections we gave a general overview of the spreadsheet language, taking into account the familiarity most readers probably have with this kind of application and focusing only on its more peculiar aspects. Now, we proceed with a more formal and complete presentation. In this section, we present definitional interpreter designed to model the core semantics of spreadsheets, with a focus on the dataflow language at its core. Our intention here is to illustrate the various design decisions that go into specifying precise semantics for a spreadsheet containing a realistic set of features, showcasing how complexity arises from what is usually seen as a conceptually simple language. We believe that this helps to explain the number of incompatibilities between different implementations that we found and described in our work.

We chose to model most closely the semantics of LibreOffice, which is the spreadsheet for which the most detailed specification documents are available.

As in the interpreter for Pure Data, this section was also written in Literate Haskell, including the complete listings of the intepreter, and its source code in `.lhs` format is also available at `https://hisham.hm/thesis/`.

This implementation uses only standard modules included in the Haskell Platform:

```
module XlInterpreter where
import Data.Char (ord, chr, toUpper)
import Data.Fixed
import Data.List (foldl')
import Data.Map.Strict as Map (Map, foldlWithKey, member, empty, lookup)
import Data.Set as Set (Set, member, singleton)
import Data.Map.Strict (insert)
import qualified Data.Set as Set (insert)
```

### 5.3.1 Representation of programs

A spreadsheet program (called a "worksheet") is represented with the *XlWorksheet* data type in our model, which contains a map from row-column coordinates to cells.

In modern spreadsheet applications, a complete document is a set of worksheets (called a workbook). For simplicity, we did not implement support for multiple worksheets since this does not affect evaluation significantly.

```
data XlWorksheet = XlWorksheet XlCells
  deriving Show
type XlCells = Map.Map ⦇·,·⦈ XlCell
```

We represent row-column pairs with the notation $⦇\cdot,\cdot⦈$. It contains a pair of addresses, representing row and column, and each of which may be absolute (represented as $\langle n \rangle$) or relative (represented as $\langle n \rangle_R$).

```
data ⦇·,·⦈ = ⦇XlAddr, XlAddr⦈
  deriving (Eq, Ord)
data XlAddr = ⟨Int⟩     -- (absolute address)
            | ⟨Int⟩_R   -- (relative address)
  deriving (Eq, Ord)
```

Cells contain formulas. As explained in Section 5.2.1, formulas can be evaluated in a special mode called "array formula". The indication if the formula will be evaluated as an array formula is a property of the cell, not the formula.

In the spreadsheet interface, a single array formula is presented as covering a range of cells. In our interpreter, we replicate the formula in each cell of the range, and annotate it with an $(x, y)$ coordinate indicating which element of the range matrix they represent, with the top-left cell being entry $(0, 0)$. We will call this pair the offset of a cell in an array formula.

```
data XlCell = XlCell    XlFormula
            | XlAFCell XlFormula (Int, Int)
  deriving Show
```

A formula, on its turn, may be a literal value, a reference to another cell, a reference to a range of cells, or a function, which has a name and a list of arguments. Our interpreter, thus, manipulates expressions as trees of *XlFun* nodes, assuming that the textual formula language has already been parsed into this format.

```
data XlFormula = XlLit  XlValue
               | XlRef  ⦇·,·⦈
               | XlRng ⦇·,·⦈ ⦇·,·⦈
               | XlFun String [XlFormula]
  deriving Show
```

Finally, values are numbers, strings, booleans, errors and matrices of literals. We represent all matrices as 2-dimensional, stored as a list of lists, which each inner list representing a row (a unidimensional array is a 2-dimensional matrix with a single row). We also have a special value for an empty cell, due to its special coercion rules (implemented in Section 5.3.5.7).

```
data XlValue = XlNumber Double
             | XlString String
             | XlBool   Bool
             | XlError  String
             | XlMatrix [[XlValue]]
             | XlEmpty
  deriving Eq
```

For convenience we define a few instances of the *Show* type class that will prove useful later when running the interpreter. In particular, for display purposes we convert absolute row-column coordinates to the familiar “A1” notation.[8]

```
instance Show ⦇·, ·⦈ where
  show (⦇r, ·⦈@⟨rn⟩ c@⟨cn⟩) =
    "<" ++ [chr (cn + 65)] ++ (show (rn + 1)) ++ ">"
  show ⦇r, c⦈ =
    "R" ++ show r ++ "C" ++ show c
instance Show XlValue where
  show (XlNumber d) = num2str d
  show (XlString s) = show s
  show (XlBool b)   = show b
  show (XlError e)  = show e
  show (XlMatrix m) = show m
  show XlEmpty      = ""
instance Show XlAddr where
  show ⟨n⟩ = show n
  show ⟨n⟩_R = "[" ++ show n ++ "]"
```

## 5.3.2 Representation of states

The state of a spreadsheet consists of the map of cells, which stores the cells and their contents (that is, the formulas), and the map of values, which stores the computed value for each cell. Both are indexed by row-column coordinates.

```
data XlState = XlState XlCells XlValues
type XlValues = Map.Map ⦇·, ·⦈ XlValue
```

From a programming language perspective, interaction with a spreadsheet consists exclusively of replacing formulas in cells. We represent these as events that

[8]We made a simplification here by presenting absolute coordinates using strings such as `B5`. In spreadsheets, such an identifier actually represents a relative coordinate, with `$B$5` being the absolute equivalent. The `A1` notation hides the fact that coordinates in spreadsheets are relative by default (which explains their behavior when copying and pasting cells).

contain the absolute coordinates and the formula to be entered to a cell. In the case of array formulas, a rectangular range (denoted by the top-left and bottom-right cells) covering one or more cells must be given. A single formula will then apply to that range as a group.

```
data XlEvent = XlSetFormula      (|·,·|) XlFormula
             | XlSetArrayFormula (|·,·|) (|·,·|) XlFormula
  deriving Show
```

### 5.3.3 Execution

The execution of a spreadsheet is demand-driven. The user triggers the evaluation by editing a cell, which causes its value to be recomputed. When computing the value of a cell, other cells may be referenced, so they are computed as well, and the process continues recursively. Conversely, other cells may reference the newly-edited cell, so their values need to be recomputed as well.

#### 5.3.3.1 Main loop

Since we are interested in the dynamic semantics (that is, what happens with the program state over time as it runs), we model our interpreter as a loop of evaluation steps. The function *runEvents* implements this loop, taking as inputs a worksheet (a spreadsheet document containing the initial contents of cell formulas) and a list of events. For each event, it calls the main evaluation function, *runEvent*, until it produces the final state, containing the resulting cells and their values.

Unlike the interpreter modelling Pure Data in Chapter 4, we return only the final state, since inspecting the final result of the spreadsheet is usually sufficient for understanding its behavior. Tracing the intermediate results is an easy modification if desired.

```
runEvents :: XlWorksheet → [XlEvent] → XlState
runEvents sheet@(XlWorksheet cells) events =
  foldl′ runEvent (XlState cells Map.empty) events
```

When we process an event in *runEvent*, we need to update the cells that were entered and then perform the necessary recalculations. Since we are not concerned with performance and formulas are in principle purely functional (which is not true in real-world spreadsheets due to functions such as `TODAY` which reads the system clock, but is true in our interpreter), we simply discard the previous map of values and recompute all cells in the worksheet. One way to avoid this computational expense would be to maintain data structures that keep track of reverse dependencies for each cell, but we avoid this optimization here for simplicity. Real-world spreadsheets further restrict the recalculation by limiting it to cells which are currently visible in their user interface.[9]

[9]We were able to empirically verify this when we produced a spreadsheet with a formula that crashed LibreOffice. The application only crashed when the offending cell was scrolled into view.

Our interpreter does avoid recalculating a cell if it was already calculated in the current pass as a dependency of a previous cell. Also, it keeps track of which cells are currently being visited, for detecting circular references.

```
runEvent :: XlState → XlEvent → XlState
runEvent env@(XlState cells _) event =
  let
     cells′ = updateCells cells event
     acc :: XlValues → $(\!|\cdot,\cdot|\!)$ → XlCell → XlValues
     acc vs rc cell =
       if Map.member rc vs
       then vs
       else
         let (v′, vs′) = calcCell (Set.singleton rc) cells′ vs rc cell
         in insert rc v′ vs′
  in
     XlState cells′ (Map.foldlWithKey acc Map.empty cells′)
```

An event may update a single cell in case of a regular formula, or many cells at a time in case of an array formula applied over a range. Function *updateCells* covers both cases:

```
updateCells cells event@(XlSetFormula rc fml) =
  insert rc (XlCell fml) cells
updateCells cells event@(XlSetArrayFormula $rc_{from}$ $rc_{to}$ fml) =
  fst $ foldRange $rc_{from}$ $rc_{from}$ $rc_{to}$ (cells, (0, 0)) id $op_{cell}$ $op_{row}$
     where
       $op_{cell}$ (cells, (x, y)) rc   = (insert rc (XlAFCell fml (x, y)) cells, (x + 1, y))
       $op_{row}$ _ r (cells, (x, y)) = (cells, (0, y + 1))
```

To iterate over ranges, we define a folding function *foldRange*, which loops over the 2-dimensional range applying two accumulator functions: one which runs on each cell, and one that runs as each row is completed.

```
foldRange :: $(\!|\cdot,\cdot|\!)$ → $(\!|\cdot,\cdot|\!)$ → $(\!|\cdot,\cdot|\!)$       -- cell position and addresses for the range
               → r                    -- a zero-value for the fold as a whole
               → (r → c)              -- an initializer function for each row
               → (c → $(\!|\cdot,\cdot|\!)$ → c)        -- function to run on each cell
               → (r → Int → c → r)    -- function to run on each complete row
               → r
foldRange pos $rc_{from}$ $rc_{to}$ zero $zero_{row}$ $op_{cell}$ $op_{row}$ =
  let
     ($r_{min}$, $c_{min}$, $r_{max}$, $c_{max}$) = toAbsRange pos $rc_{from}$ $rc_{to}$
     handleRow $acc_{row}$ r = $op_{row}$ $acc_{row}$ r $v_{row}$
       where
         handleCell $acc_{cell}$ c = $op_{cell}$ $acc_{cell}$ $(\!|\langle r\rangle, \langle c\rangle|\!)$
         $v_{row}$ = foldl′ handleCell ($zero_{row}$ $acc_{row}$) [$c_{min}$ .. $c_{max}$]
  in
    $foldl'\ handleRow\ zero\ [r_{min} \mathinner{..} r_{max}]$
```

It is important to note that, when handling array formulas, *updateCells* expands a single array formula spanning a range of cells into a number of individual *XlAFCell* entries in the *cells* map, each of them containing the $(x, y)$ offset to indicate their relative position in the rectangular range to which the array formula was applied.

This makes two important assumptions. First, that it is possible to compute each position of an array formula individually. This assumption is not critical. At worst, it would wasteful in cases such as matrix multiplication, where each cell would cause the whole matrix to be calculated and then converted down to the scalar corresponding to the cell's position.

The second assumption is that the computation of a given cell from an array formula's range is independent of the total size of the range as specified by the user when the array formula was created. In general, this assumption holds in spreadsheet applications, but we were able to identify corner cases in Excel where an array formula returns different results when entered in a single cell versus being entered in a range. For example, assuming `A1` contains the string `C1`, `B1` contains the string `D1`, `C1` contains `9` and `D1` contains `16`, entering `=SQRT(INDIRECT(A1:B1))` in cell `E2` results in the value `3`; but entering the same formula with the range `E2:F2` selected causes the value of both cells to be `#VALUE!`. In LibreOffice (and in our interpreter), they evaluate to `3` and `4`. By behaving differently according to the range size selected during initial entry, Excel adds a dependency to the calculation of cells that is invisible in its UI. This interpreter avoids this problem by using calculation strategies similar to those in LibreOffice and Google Sheets.

### 5.3.3.2 Resolving addresses

Relative coordinates are used extensively in spreadsheets, but whenever they are used they need to be resolved into absolute addresses. Also, whenever the interpreter uses ranges, it needs to ensure that they are normalized as absolute coordinates with the top-left cell first and the bottom-right cell second.

When relative addresses are given, they are resolved relative to the coordinate of the cell being evaluated, which we will refer throughout as the cell's position.

Functions *toAbs* and *toAbsRange* normalize coordinates and ranges, respectively:

```
$toAbs :: (\!|\cdot, \cdot|\!) \rightarrow (\!|\cdot, \cdot|\!) \rightarrow (\!|\cdot, \cdot|\!)$
$toAbs\ pos@(\!|r_p, c_p|\!)\ cell@(\!|r, c|\!) = (\!|(absAddr\ r_p\ r), (absAddr\ c_p\ c)|\!)$
  where
    $absAddr :: XlAddr \rightarrow XlAddr \rightarrow XlAddr$
    $absAddr\ \_$            $c@\langle\_\rangle = c$
    $absAddr\ \langle b\rangle$         $\langle c\rangle_R$   $= \langle(b + c)\rangle$
    $absAddr\ b@\langle\_\rangle_R\ \_$        $= \bot$
$toAbsRange :: (\!|\cdot, \cdot|\!) \rightarrow (\!|\cdot, \cdot|\!) \rightarrow (\!|\cdot, \cdot|\!) \rightarrow (Int, Int, Int, Int)$
$toAbsRange\ pos\ rc_{from}\ rc_{to} =$
  let
    $(\!|\langle r_{min}\rangle, \langle c_{min}\rangle|\!) = toAbs\ pos\ rc_{from}$
    $(\!|\langle r_{max}\rangle, \langle c_{max}\rangle|\!) = toAbs\ pos\ rc_{to}$
    $r_{min} = min\ r_{min}\ r_{max}$
    $r_{max} = max\ r_{min}\ r_{max}$
    $c_{min} = min\ c_{min}\ c_{max}$
    $c_{max} = max\ c_{min}\ c_{max}$
  in
    $(r_{min}, c_{min}, r_{max}, c_{max})$
```

## 5.3.4 Calculating cell values

To determine the value of a cell, the interpreter evaluates the cell's formula, potentially recursing to evaluate other cells referenced by that formula. The *calcCell* function takes as arguments a set of cell addresses currently being recursively visited (to detect cycles), the table of cell formulas, the current table of values, the cell position and the actual cell to compute. The function produces the calculated value of the cell along with the map of all values, since other calls may have been computed along the way.

$calcCell :: Set\ (\!|\cdot,\cdot|\!) \rightarrow XlCells \rightarrow XlValues \rightarrow (\!|\cdot,\cdot|\!) \rightarrow XlCell \rightarrow (XlValue, XlValues)$

A major complication in the semantics of a spreadsheet application is the fact that there are two distinct modes of evaluation: one for regular formulas, and one for array formulas. Further, different kinds of functions in formulas evaluate their arguments in different ways: borrowing from the terminology of the language Perl, some functions evaluate their arguments in a scalar context (that is, they expect their arguments to produce a scalar value), and some evaluate arguments in an array context. This gives us four evaluation rules in total.

This is the core of the incompatibility between spreadsheet formula languages. As our examples in Section 5.2.1 demonstrate, each application uses a different set of rules as to when to switch to array evaluation, and to what to do in each evaluation mode.

Note that the presence of different evaluation rules affects not only array formulas. As illustrated in Figure 5.4, Excel performs array-style evaluation in subformulas for certain functions even when not in array formula mode.

In our implementation, we modularized these decisions into a number of functions implementing different ways of evaluating a formula, in array and scalar contexts.

Then, to represent an evaluation mode, the interpreter features a data type *XlEvaluator* which, besides carrying a few context values for convenience, includes a coercion function *eToScalar* to obtain a scalar function according to the context of a cell (as we will see in more detail below), and two evaluation functions, one for each of the possible evaluation contexts: *eArray* and *eScalar*.

```
data $XlEvaluator = XlEvaluator$ {
  $ePos$       $:: (\!|\cdot,\cdot|\!)$,
  $eOffset$    $:: (Int, Int)$,
  $eVisiting$  $:: Set\ (\!|\cdot,\cdot|\!)$,
  $eCells$     $:: XlCells$,
  eToScalar :: (|·, ·|) → (Int, Int) → XlFormula → XlFormula,
  eArray    :: XlEvaluator → XlValues → XlFormula → (XlValue, XlValues),
  eScalar   :: XlEvaluator → XlValues → XlFormula → (XlValue, XlValues)
}
```

We opted to implement evaluation functions that follow the OpenDocument specification. With this, we achieved a good (but deliberately not full) degree of compatibility with LibreOffice in the subset of spreadsheet features implemented in this interpreter.

For calculating the value of a regular cell, the interpreter employs an evaluator that uses functions *intersectScalar* to convert non-scalar to scalars, *evalScalarFormula* for evaluating scalar arguments, and *evalFormula* for evaluating non-scalar arguments. We will see the definition of these functions in Section 5.3.4.1. Once the evaluator is defined, *calcCell* triggers the scalar evaluation function on the formula.

```
calcCell visiting cells vs pos@(|⟨r⟩, ⟨c⟩|) (XlCell formula) =
  evalScalarFormula ev vs formula
  where
    ev = XlEvaluator {
      ePos       = pos,
      eOffset    = (0, 0),
      eCells     = cells,
      eVisiting  = visiting,
      eToScalar  = intersectScalar,
      eScalar    = evalScalarFormula,
      eArray     = evalFormula
    }
```

For calculating cells marked as array formulas, the interpreter uses a different evaluator. For coercing non-scalars into scalars, it uses a different function, *matrixToScalar*. For scalar evaluation of arguments, it uses the same function *evalScalarFunction* as above, but for non-scalar evaluation, it uses *iterateFormula*. Both *matrixToScalar* and *iterateFormula* will be defined in Section 5.3.4.2.

The implementation of *calcCell* for array formulas also triggers the calculation by applying this mode's scalar evaluator, but here the result value is further filtered through a coercion function (*scalarize*), to ensure that a scalar value is ultimately displayed in the cell.

```
calcCell visiting cells vs pos (XlAFCell formula (x, y)) =
  scalarize ev $ (eScalar ev) ev vs formula
  where
    ev = XlEvaluator {
      ePos       = pos,
      eOffset    = (x, y),
      eCells     = cells,
      eVisiting  = visiting,
      eToScalar  = matrixToScalar,
      eScalar    = evalScalarFormula,
        eArray     = iterateFormula
    }
scalarize :: XlEvaluator → (XlValue, XlValues) → (XlValue, XlValues)
scalarize ev (v, vs) = (v′, vs)
  where
    (XlLit v′) = matrixToScalar (ePos ev) (eOffset ev) (XlLit v)
```

#### 5.3.4.1 Regular cell evaluation

When the interpreter evaluates a formula in a scalar context, it runs the evaluator's scalar conversion function on the formula prior to evaluating it proper. If the formula is an array or a range, it will be converted to a scalar. If it is a scalar or a function, it will be evaluated as-is.

```
evalScalarFormula ev vs formula =
  evalFormula ev vs formula′
  where
    formula′ = (eToScalar ev) (ePos ev) (eOffset ev) formula
```

The conversion function for regular cells, *intersectScalar*, is defined as follows.

For array literals, element $(0, 0)$ is returned. Empty arrays have inconsistent behavior across spreadsheets. When given an empty array, Excel rejects the formula, pops a message box alerting the user and does not accept the entry. Excel Online does not display a message, but marks the cell with a red dashed border. LibreOffice exhibits a very inconsistent behavior: `={}` displays as an empty cell; `=10/{}` evaluates to `#VALUE!` but both `=ABS({})` and `=ABS(10/{})` evaluate to `0`; however, `=ABS(A1)` where `A1` is `{}` evaluates to `#VALUE!`. In our interpreter, we simply return the `#REF!` error for all uses of `{}`, replicating the behavior of Google Sheets.

For ranges, the resulting value depends on the shape of the range and the position in the spreadsheet grid where the formula was entered. If the range is a vertical $(n \times 1)$ or horizontal $(1 \times n)$ array, the evaluation follows an "intersection" rule: the value returned is that of the element of the range that is perpendicularly aligned with the position of the formula. For example, for a formula in cell `G5` that references `A1` in a scalar context, the value in `A5` will be returned. Likewise, if that same cell `G5` references `E1:K1`, the value obtained will be that in cell `G1`. If there is no intersection or if the range has any other shape, `#VALUE!` is returned.

```
intersectScalar :: ⦇·, ·⦈ → (Int, Int) → XlFormula → XlFormula
intersectScalar pos@⦇⟨r⟩, ⟨c⟩⦈ _ formula =
  case formula of
  XlLit (XlMatrix [])   → XlLit (XlError "#REF!")
  XlLit (XlMatrix [[]]) → XlLit (XlError "#REF!")
  XlLit (XlMatrix m)    → XlLit (head (head m))
  XlRng $rc_{from}$ $rc_{to}$       →
    case toAbsRange pos $rc_{from}$ $rc_{to}$ of
    ($r_{min}$, $c_{min}$, $r_{max}$, $c_{max}$)
        | ($c_{min}$ ≡ $c_{max}$) ∧ (r ⩾ $r_{min}$) ∧ (r ⩽ $r_{max}$) → XlRef ⦇⟨r⟩, ⟨$c_{min}$⟩⦈
    | $(r_{min} \equiv r_{max}) \wedge (c \geqslant c_{min}) \wedge (c \leqslant c_{max}) \rightarrow XlRef\ (\!|\langle r_{min}\rangle, \langle c\rangle|\!)$
  $\_ \rightarrow XlLit\ (XlError$ "#VALUE!")
$f \rightarrow f$
```

### 5.3.4.2 Cell evaluation for array formulas

When a cell is marked as an array formula, it follows a different evaluation process. As we saw in the definition of the array formula evaluator in function *calcCell* (Section 5.3.4), for scalar contexts we use the same evaluation function as in regular cells, *evalScalarFormula*. However, in array formulas this function uses a different conversion function: *eToScalar* is defined as *matrixToScalar*.

Function *matrixToScalar* extracts a scalar value from a non-scalar based on the offset $(x, y)$ relative to the range for which the array formula was defined. This way, as *runEvent* calculates cell values for each position of an array formula, the evaluation of each cell will extract a different value from non-scalars produced during the calculation of the formula. For example, if we enter `=A1:B2` as an array formula in range `D10:E11`, cell `D11` has offset $(1, 0)$ and will obtain the value of cell `B1`.

The area designated by the user for an array formula does not necessarily have the same dimensions as the non-scalar being displayed in it. The OpenDocument specification lists a series of rules for filling the exceeding cells, which the *displayRule* function below implements. Excel and LibreOffice also implement these rules; Google Sheets does not.

```
$matrixToScalar :: (\!|\cdot,\cdot|\!) \rightarrow (Int, Int) \rightarrow XlFormula \rightarrow XlFormula$
$matrixToScalar\ pos\ (x, y)\ f =$
  case $f$ of
    $XlLit\ (XlMatrix\ m) \rightarrow$
      $displayRule\ x\ y\ (foldl'\ max\ 0\ (map\ length\ m))\ (length\ m)$
        $(\lambda x\ y \rightarrow XlLit\ \$\ m\ !!\ y\ !!\ x)$
    $XlRng\ rc_{from}\ rc_{to} \rightarrow$
      $displayRule\ x\ y\ (1 + c_{max} - c_{min})\ (1 + r_{max} - r_{min})$
        $(\lambda x\ y \rightarrow XlRef\ (\!|\langle(r_{min} + y)\rangle, \langle(c_{min} + x)\rangle|\!))$
        where
          $(r_{min}, c_{min}, r_{max}, c_{max}) = toAbsRange\ pos\ rc_{from}\ rc_{to}$
    $f \rightarrow f$
  where
    $displayRule :: Int \rightarrow Int \rightarrow Int \rightarrow Int \rightarrow (Int \rightarrow Int \rightarrow XlFormula)$
                $\rightarrow XlFormula$
    $displayRule\ x\ y\ x_{size}\ y_{size}\ getXY$
      | $x_{size} > x \wedge y_{size} > y = getXY\ x\ y$
      | $x_{size} \equiv 1 \wedge y_{size} \equiv 1 = getXY\ 0\ 0$
      | $x_{size} \equiv 1 \wedge x > 0$       $= getXY\ 0\ y$
      | $y_{size} \equiv 1 \wedge y > 0$       $= getXY\ x\ 0$
      | $otherwise$                $= XlLit\ \$\ XlError$ "#N/A"
```

Function *iterateFormula* implements the special evaluation mode for array formulas. When given a function where any argument is a range or a matrix, it produces a matrix with results. It does this by first checking each argument and determining the maximum dimensions used by an argument ($x_{max}$ and $y_{max}$). Then, it iterates from $(0, 0)$ to $(x_{max}, y_{max})$, evaluating the function in scalar context once for each entry. In each evaluation, it uses a modified version of the list of arguments, in which each non-scalar argument is converted to a scalar based on the current $(x, y)$ offset.

If the given function has no non-scalar arguments, it is evaluated normally by *evalFormula*.

```
iterateFormula :: XlEvaluator → XlValues → XlFormula → (XlValue, XlValues)
iterateFormula ev vs (XlFun name args) =
  if x_max > 1 ∨ y_max > 1
  then (λ(m, vs') → (XlMatrix m, vs')) $ foldl' doRow ([], vs) [0 .. y_max − 1]
  else evalFormula ev vs (XlFun name args)
  where
    y_max = foldl' getY 1 args
      where
        getY a (XlLit (XlMatrix m)) = max a (length m)
        getY a (XlRng rc_from rc_to)    = max a (1 + r_max − r_min)
          where
            (r_min, _, r_max, _) = toAbsRange (ePos ev) rc_from rc_to
        getY a _ = a
    x_max = foldl' getX 1 args
      where
        getX a (XlLit (XlMatrix m)) = max a (maxRowLength m)
          where
            maxRowLength :: [[XlValue]] → Int
            maxRowLength m = foldl' (λa' row → max a' (length row)) 1 m
        getX a (XlRng rc_from rc_to) = max a (1 + c_max − c_min)
          where
            (_, c_min, _, c_max) = toAbsRange (ePos ev) rc_from rc_to
        getX a _ = a
    doRow :: ([[XlValue]], XlValues) → Int → ([[XlValue]], XlValues)
    doRow (m, vs) y = appendTo m $ foldl' doCell ([], vs) [0 .. x_max − 1]
      where
        doCell :: ([XlValue], XlValues) → Int → ([XlValue], XlValues)
        doCell (row, vs) x = appendTo row $ evalFormula ev vs f'
          where
            f' = XlFun name (map ((eToScalar ev) (ePos ev) (x, y)) args)
        appendTo xs (v, vs) = (xs ++ [v], vs)
iterateFormula ev vs f = evalFormula ev vs f
```

## 5.3.5 Operations

The last part of the interpreter is function *evalFormula*, which implements the evaluation of the various operations available in the textual formula language. Given an evaluator, the current map of values, and a formula, it produces the calculated value of the formula and a new map of values (since other cells may be calculated as part of the evaluation of this formula).

```
evalFormula :: XlEvaluator → XlValues → XlFormula → (XlValue, XlValues)
```

The function *evalFormula* implements the various language constructs as follows.

#### 5.3.5.1 Literals, references and ranges

When a formula is just a literal, its value is returned and the map of cell values remains unchanged.

```
evalFormula ev vs (XlLit v) = (v, vs)
```

When a formula is a reference to another cell, *evalFormula* first converts the reference address to its absolute value relative to the cell's position. Then, it detects circular references by checking the *eVisiting* set of the evaluator. If the reference is valid, it checks in the map of values if the value was already calculated. If the cell is unset, we return the special value *XlEmpty*. Finally, if the cell contains a formula which needs to be calculated, we calculate it with *calcCell* and store the resulting value in an updated map of values.

```
evalFormula ev vs (XlRef ref′) =
  let
    ref = toAbs (ePos ev) ref′
  in
    if ref ∈ (eVisiting ev)
    then (XlError "#LOOP!", vs)
    else
      case Map.lookup ref vs of
      Just v    → (v, vs)
      Nothing →
        case Map.lookup ref (eCells ev) of
        Nothing  → (XlEmpty, vs)
        Just cell →
          (v′, vs″)
          where
            (v′, vs′) = calcCell (Set.insert ref (eVisiting ev)) (eCells ev) vs ref cell
            vs″ = insert ref v′ vs′
```

For evaluating ranges, *evalFormula* uses *foldRange* to iterate over the range, invoking the scalar evaluation function (*eScalar*) for each element, producing a matrix of values.

```
evalFormula ev vs (XlRng from to) =
  let
    (m, vs′) = foldRange (ePos ev) from to ([], vs) $zero_{row}$ $op_{cell}$ $op_{row}$
    where
      $zero_{row} :: ([[XlValue]], XlValues) \rightarrow ([XlValue], XlValues)$
      $zero_{row}\ (\_, vs) = ([\,], vs)$
      $op_{cell} :: ([XlValue], XlValues) \rightarrow (\!|\cdot, \cdot|\!) \rightarrow ([XlValue], XlValues)$
      $op_{cell}\ (row, vs)\ rc =$
        $addToRow\ \$\ (eScalar\ ev)\ ev\ vs\ (XlRef\ rc)$
          where $addToRow\ (v, vs') = (row +\!\!+ [v], vs')$
      $op_{row} :: ([[XlValue]], XlValues) \rightarrow Int \rightarrow ([XlValue], XlValues)$
          $\rightarrow ([[XlValue]], XlValues)$
      $op_{row}\ (m, \_)\ r\ (row, vs) = (m +\!\!+ [row], vs)$
  in
    $(XlMatrix\ m, vs')$
```

#### 5.3.5.2 IF, AND, and OR

The IF function takes three arguments. It tests the first argument, and if evaluates to *XlBool True* it evaluates the second argument and returns it; otherwise, it evaluates and returns the third argument. Note that arguments are evaluated lazily, as is typical in constructs of this type in programming languages.

```
$evalFormula\ ev\ vs\ (XlFun$ "IF" $[i, t, e]) =$
  let
    $(v_i, vs_i) = toBool\ \$\ (eScalar\ ev)\ ev\ vs\ i$
    $(v_r, vs_r) =$
      case $v_i$ of
      $(XlError\ \_)$       $\rightarrow (v_i, vs_i)$
      $(XlBool\ True)$  $\rightarrow (eScalar\ ev)\ ev\ vs_i\ e$
      $(XlBool\ False)$ $\rightarrow (eScalar\ ev)\ ev\ vs_i\ t$
      $\_$                   $\rightarrow ((XlError$ "#VALUE!"$), vs_i)$
  in
    $(v_r, vs_r)$
```

The AND and OR functions in spreadsheets, however, are evaluated strictly, not performing the usual short-circuit expected of them in programming languages. They always evaluate both arguments, and return and error if either argument fails.

```
$evalFormula\ ev\ vs\ (XlFun$ "AND" $[a, b]) =$
  let
    $(v_a, vs') = toBool\ \$\ (eScalar\ ev)\ ev\ vs\ a$
    $(v_b, vs'') = toBool\ \$\ (eScalar\ ev)\ ev\ vs'\ b$
    $v_r =$ case $(v_a, v_b)$ of
      $(XlError\ \_, \_)$                  $\rightarrow v_a$
      $(\_, XlError\ \_)$                  $\rightarrow v_b$
      $(XlBool\ True, XlBool\ True)$ $\rightarrow v_a$
      $\_$                                   $\rightarrow XlBool\ False$
  in
```

```
    (v_r, vs'')
evalFormula ev vs (XlFun "OR" [a, b]) =
  let
    (v_a, vs') = toBool $ (eScalar ev) ev vs a
    (v_b, vs'') = toBool $ (eScalar ev) ev vs' b
    v_r = case (v_a, v_b) of
      (XlError _, _)     → v_a
      (_, XlError _)     → v_b
      (XlBool True, _)   → v_a
      (_, XlBool True)   → v_b
      _                  → XlBool False
  in
    (v_r, vs'')
```

#### 5.3.5.3 SUM

The SUM function illustrates the use of array evaluation. Each argument is evaluated using the *eArray* function of the evaluator *ev*, and their results are added producing the final result $v_r$. Thus, when used in an array formula, the evaluation of its arguments is done using *iterateFormula* (Section 5.3.4.2), producing a *XlMatrix* of results that is then iterated to perform the sum. This allows, for example, to use `=SUM(SQRT(A1:A10))` to obtain a sum of squares, even though function SQRT is a scalar function that does not support ranges on its own.

It is worth noting that the coercion rules used by SUM are different from those used by + (Section 5.3.5.6). While SUM skips string values (which may appear, for example, as part of a range), the + function attempts to coerce them into numbers.

```
evalFormula ev vs (XlFun "SUM" args) =
  let
    doSum s@(XlString _) v               = v
    doSum v               s@(XlString _) = v
    doSum (XlBool b)      (XlNumber n)   = XlNumber (bool2num b + n)
    doSum (XlNumber n)    (XlBool b)     = XlNumber (bool2num b + n)
    doSum (XlNumber a)    (XlNumber b)   = XlNumber (a + b)
    (v_r, vs_r) = foldl' handle (XlNumber 0, vs) args
      where
        handle (acc, vs_acc) arg =
          let
            (v_a, vs_b) = (eArray ev) ev vs_acc arg
            v_sum =
              case v_a of
                XlError _    → v_a
                XlMatrix m   → foldl' (foldl' (checkErr doSum)) acc m
                XlBool b     → checkErr doSum acc v_a
                XlNumber n   → checkErr doSum acc v_a
                _            → XlError "#VALUE!"
          in
            $(v_{sum}, vs_b)$
  in
    $(v_r, vs_r)$
```

#### 5.3.5.4 `INDIRECT`

The `INDIRECT` function converts a string describing a reference or range written in "A1" notation to the actual reference or range. This feature adds support for runtime-evaluated indirect connections to the dataflow graph of a spreadsheet. A cell can effectively act as a pointer to another cell.

Different spreadsheets vary in their semantics when supporting non-scalar indirect references. Here, we opted for implementing it in a straightforward way: we evaluate the argument in a scalar context, coercing it to string, and then evaluate the indirect reference in a scalar context as well. When used in an array formula, `INDIRECT` can handle non-scalar arguments due to the scalar conversion performed by *matrixToScalar* (Section 5.3.4.2).

Auxiliary function *toRC* converts addresses in "A1" alphanumeric format to the internal row-column numeric format. For simplicity, this interpreter only support columns A to Z, and we assume the string is well-formed and do not perform error checking.

```
$evalFormula\ ev\ vs\ (XlFun$ "INDIRECT" $[addr]) =$
  let
    $toRC :: String \to (\!|\cdot,\cdot|\!)$
    $toRC\ (l : num) = (\!|\langle((read\ num) - 1)\rangle, \langle((ord\ l) - 65)\rangle|\!)$
    $convert\ s =$
      case $break\ (\equiv$ ':' $)\ s$ of
      $(a1,$ ':' $: b2) \to (XlRng\ (toRC\ a1)\ (toRC\ b2))$
      $\_$              $\to (XlRef\ (toRC\ s))$
    $(v_a, vs_b) = toString\ \$\ (eScalar\ ev)\ ev\ vs\ addr$
    $(v_r, vs_r) =$
      case $v_a$ of
      $XlError\ e\ \to (v_a, vs_b)$
      $XlString\ s \to (eScalar\ ev)\ ev\ vs_b\ (convert\ s)$
      $\_$          $\to ((XlError$ "#VALUE!"$), vs_b)$
  in
    $(v_r, vs_r)$
```

#### 5.3.5.5 String operations

For illustrative purposes, we define a function that operates on strings: the substring function `MID`, and the concatenation operator `&`. These are useful for demonstrating the coercion rules in examples. In particular, it is interesting to observe how the empty cell coerces to different values: with `A1` being empty, `="Hello"&A1` results in `"Hello"`, and `=1/A1` results in `#DIV/0!`.

```
$evalFormula\ ev\ vs\ (XlFun$ "MID" $[v_{str}, v_{sum}, v_{len}]) =$
  let
    $(v'_{str},\ vs')\ = toString\ \ \$\ (eScalar\ ev)\ ev\ vs\ \ v_{str}$
    $(v'_{sum}, vs'')\ = toNumber\ \$\ (eScalar\ ev)\ ev\ vs'\ v_{sum}$
    $(v'_{len},\ vs''')= toNumber\ \$\ (eScalar\ ev)\ ev\ vs''\ v_{len}$
    $doMid\ (XlString\ str)\ (XlNumber\ start)\ (XlNumber\ len) =$
      $XlString\ \$\ take\ (floor\ len)\ \$\ drop\ (floor\ start - 1)\ str$
    $doMid\ \_\ \_\ \_ = XlError$ "#VALUE!"
    $v = doMid\ v'_{str}\ v'_{sum}\ v'_{len}$
  in
    $(v, vs''')$

$evalFormula\ ev\ vs\ (XlFun$ "&" $[a, b]) =$
  let
    $(v_a, vs')\ = toString\ \$\ (eScalar\ ev)\ ev\ vs\ \ a$
    $(v_b, vs'') = toString\ \$\ (eScalar\ ev)\ ev\ vs'\ b$
    $doConcat\ (XlString\ sa)\ (XlString\ sb) = XlString\ (sa +\!\!+\ sb)$
    $doConcat\ \_\ \qquad\qquad\quad \_ \qquad\qquad\quad = XlError$ "#VALUE!"
    $v = checkErr\ doConcat\ v_a\ v_b$
  in
    $(v, vs'')$
```

#### 5.3.5.6 Mathematical operations and equality

A few unary and binary mathematical operations are defined here. They all follow the same pattern, encapsulated as functions *unOp* and *binOp* defined further below. The division operator additionally checks for division-by-zero, returning `#DIV/0!`.

```
$evalFormula\ ev\ vs\ (XlFun$ "SQRT" $[v]) = unOp\ \ sqrt\ ev\ vs\ v$
$evalFormula\ ev\ vs\ (XlFun$ "ABS"  $[v]) = unOp\ \ abs\ \ ev\ vs\ v$
$evalFormula\ ev\ vs\ (XlFun$ "+" $[a, b])\ \ = binOp\ (+)\ \ ev\ vs\ a\ b$
$evalFormula\ ev\ vs\ (XlFun$ "-" $[a, b])\ \ = binOp\ (-)\ \ ev\ vs\ a\ b$
$evalFormula\ ev\ vs\ (XlFun$ "*" $[a, b])\ \ = binOp\ (*)\ \ ev\ vs\ a\ b$
$evalFormula\ ev\ vs\ (XlFun$ "/" $[a, b]) =$
  let
    $(v_a, vs')\ = toNumber\ \$\ (eScalar\ ev)\ ev\ vs\ \ a$
    $(v_b, vs'') = toNumber\ \$\ (eScalar\ ev)\ ev\ vs'\ b$
    $doDiv\ (XlNumber\ n_a)\ (XlNumber\ 0)\ \ = XlError$ "#DIV/0!"
    $doDiv\ (XlNumber\ n_a)\ (XlNumber\ n_b) = XlNumber\ (n_a\ /\ n_b)$
    $doDiv\ \_\ \qquad\qquad\quad \_ \qquad\qquad\quad = XlError$ "#VALUE!"
    $v = checkErr\ doDiv\ v_a\ v_b$
  in
    $(v, vs'')$
```

The equality operator is notable in which is does not perform number and string coercions as the other functions (that is, `=2="2"` returns `FALSE`). However, it does coerce booleans to numbers, probably as a compatibility leftover from when spreadsheets did not have a separate boolean type. The OpenDocument specification [OAS11] states that a conforming implementation may represent booleans as a subtype of numbers.

```
evalFormula ev vs (XlFun "=" [a, b]) =
  let
    (v_a,  vs') = (eScalar ev) ev vs a
    (v_b,  vs'') = (eScalar ev) ev vs' b
    doEq (XlNumber n_a) (XlNumber n_b) = XlBool (n_a ≡ n_b)
    doEq (XlString sa)  (XlString sb)  = XlBool (sa ≡ sb)
    doEq (XlBool ba)    (XlBool bb)    = XlBool (ba ≡ bb)
    doEq (XlNumber n_a) (XlBool bb)    = XlBool (n_a ≡ bool2num bb)
    doEq (XlBool ba)    (XlNumber n_b) = XlBool (bool2num ba ≡ n_b)
    doEq _              _              = XlBool False
    v = checkErr doEq v_a v_b
  in
    (v, vs'')
evalFormula ev vs (XlFun _ _) = (XlError "#NAME?", vs)
```

Functions *unOp* and *binOp* are convenience functions that encapsulate the pattern for common unary and binary numeric functions. They evaluate their arguments in a scalar context, check if any of the arguments evaluated to an error, and perform the operation *op*.

```
unOp :: (Double → Double)
       → XlEvaluator → XlValues → XlFormula → (XlValue, XlValues)
unOp op ev vs v =
  let
    (v', vs') = toNumber $ (eScalar ev) ev vs v
    v'' = case v' of
      e@(XlError _) → e
      (XlNumber n)  → XlNumber $ op n
      _             → XlError "#VALUE!"
  in
    (v'', vs')
binOp :: (Double → Double → Double) → XlEvaluator → XlValues
        → XlFormula → XlFormula → (XlValue, XlValues)
binOp op ev vs a b =
  let
    (v_a, vs')  = toNumber $ (eScalar ev) ev vs a
    (v_b, vs'') = toNumber $ (eScalar ev) ev vs' b
    doOp (XlNumber n_a) (XlNumber n_b) = XlNumber (op n_a n_b)
    doOp _              _              = XlError "#VALUE!"
    v = checkErr doOp v_a v_b
  in
    (v, vs'')
```

#### 5.3.5.7 Type conversions

We conclude the presentation of the interpreter with the remaining utility functions that perform various type conversions and checks.

Function *num2str* is a converter that presents rational and integer values in their preferred notation (that is, with and without a decimal point, respectively). Function *bool2num* converts booleans to 0 or 1.

```
num2str :: Double → String
num2str n = if fromIntegral (floor n) ≢ n then show n else show (floor n)
bool2num :: Bool → Double
bool2num b = if b ≡ True then 1 else 0
```

Functions *toNumber*, *toString* and *toBool* attempt to convert a value to the specified type, producing a *XlError* value if the input is not convertible.

```
toNumber :: (XlValue, XlValues) → (XlValue, XlValues)
toNumber (v, vs) = (coerce v, vs)
  where
    coerce (XlString s)   = case reads s :: [(Double, String)] of
                              []       → XlError "#VALUE!"
                              [(n, _)] → XlNumber n
    coerce (XlBool b)     = XlNumber (bool2num b)
    coerce (XlEmpty)      = XlNumber 0
    coerce (XlMatrix _)   = XlError "#VALUE!"
    coerce v              = v
toString :: (XlValue, XlValues) → (XlValue, XlValues)
toString (v, vs) = (coerce v, vs)
  where
    coerce (XlNumber n) = XlString (num2str n)
    coerce (XlBool b)   = XlString (if b ≡ True then "1" else "0")
    coerce (XlEmpty)    = XlString ""
    coerce (XlMatrix _) = XlError "#VALUE!"
    coerce v            = v
toBool :: (XlValue, XlValues) → (XlValue, XlValues)
toBool (v, vs) = (coerce v, vs)
  where
    coerce (XlNumber 0) = XlBool False
    coerce (XlNumber _) = XlBool True
    coerce (XlString s) = case map toUpper s of
                            "TRUE"  → XlBool True
                            "FALSE" → XlBool False
                            _       → XlError "#VALUE!"
    coerce (XlEmpty)    = XlBool False
    coerce (XlMatrix _) = XlError "#VALUE!"
    coerce v            = v
```

| | Microsoft Excel | LO Calc | Google Sheets | Excel Online | Excel mobile |
|---|---|---|---|---|---|
| A1 | | `Err:522` | `#REF!` | `100` | `100` |
| B1 | | `Err:522` | `#REF!` | `0` | `0` |
| error dialog | | no | no | no | yes |

Table 5.5: Behavior upon circular references, after the following sequence: B1 to 100, A1 to =B1, B1 to =A1

Function *checkErr* checks input values for errors before performing a binary operation. The order errors are evaluated is relevant: if the first argument contains an error, it takes precedence.

```
checkErr :: (XlValue → XlValue → XlValue) → XlValue → XlValue → XlValue
checkErr op e@(XlError _) _             = e
checkErr op _             e@(XlError _) = e
checkErr op a             b             = op a b
```

## 5.3.6 Demonstration

In Appendix B we present a demonstration of use of this interpreter, showcasing its features. We also produced a series of tests that correspond to sections of the OpenDocument specification for the `.ods` format [OAS11] and the ISO Open Office XML specification for the `.xlsx` format [ISO12], as well as our own additional tests that cover some unspecified behavior. All examples and tests are available in `https://hisham.hm/thesis/`.

## 5.4 Discussion: Language specification and compatibility issues

One might argue that the various language issues present in formula languages discussed in this chapter are due to specification blunders early in the history of spreadsheets, forever preserved in the name of backwards compatibility. But the insufficient concern with precise semantics of spreadsheets is not only historical, as it manifests itself in compatibility issues between modern variants, even by the same vendor. Further, when evaluating the compatibility of various spreadsheet implementations it is necessary to define what exactly is meant by compatibility.

When looking at spreadsheets as *documents*, one tends to think about "file format compatibility" as such: an application should be able to load a file and render it correctly. This definition of compatibility is insufficient, as it does not account the dynamic semantics of the language, that is, how the state of the program changes over time as the program executes. When one looks at a spreadsheet as an *interactive program*, then the newly-loaded document defines only the initial state of the program and further edits to cells are inputs that cause state updates.

Compatible languages should have equivalent dynamic semantics. Under this definition, Excel, Excel Online and mobile Excel, all three by Microsoft, are not compatible: there are identical sequences of formula edits that one can perform over the same spreadsheet which lead to different results in each program. In other words, the dynamic semantics of their formula languages differ. Case in point, all three variants of Excel have different behavior in face of circular references, as illustrated in Figure 5.5. In desktop and mobile Excel, the application pops a dialog warning the user about the circular references; in the web-based version no such warning is present. More important, however, is the difference in produced values: when a user produces a loop between two cells in desktop Excel, both cells instantly produce error values; in Excel Online and mobile Excel, the most recently updated cell produces the value zero and the other one retains its previous value.

That such a striking difference in behavior has made it to production seems to show that the UI was not treated as a well-defined language whose behavior was meant to be duplicated. While care has certainly been taken to ensure that Excel Online and mobile have good compatibility with Excel, apparently this was taken to mean only that they should load Excel files and produces identical *initial* results. The behavior of Excel Online is especially worrying, as circular references produce invalid values silently.

# Chapter 6

# Case study: LabVIEW

LabVIEW[1] is a tool for data acquisition, instrument control and industrial automation, developed by National Instruments. It is a proprietary commercial application, with extensive support for typical engineering domains such as digital signal processing. The application consists of a graphical programming environment, including a large amount of bundled functionality for data acquisition, as well as support for hardware also produced by its vendor. LabVIEW programs can be compiled and deployed stand-alone, depending only on a runtime package. LabVIEW is noted as a major success story of a visual programming language in the industry [JHM04]. In it, program code is represented as a set of graphical diagrams.

The programming language of the LabVIEW environment is called G. However, since there are no other implementations of G or any specification other than the implementation of LabVIEW itself, it is customary to refer to the language as LabVIEW or use both names interchangeably [KKR09, MS98]; for simplicity, we will call the both the application and its language LabVIEW throughout the text.

## 6.1 Overview of the language

In LabVIEW, programs are called *virtual instruments* (VIs), as their interfaces mimic laboratory instruments, with buttons, scopes and gauges. This is a clear nod to its application domain, since the developer of the tool is also a vendor of physical hardware instruments.

As depicted in Figure 6.1, each VI has always two parts:

- the *front panel*, which is the user program's interface. It contains elements that provide inputs and outputs to the program, presented as graphical widgets for interaction or visualization.
- the *block diagram*, which is the dataflow graph. It contains all elements that are present in the front panel (this time in iconic mode) as well as any additional nodes which represent functions to be applied to data, effectively constructing the program.

[1] `http://ni.com/labview/`

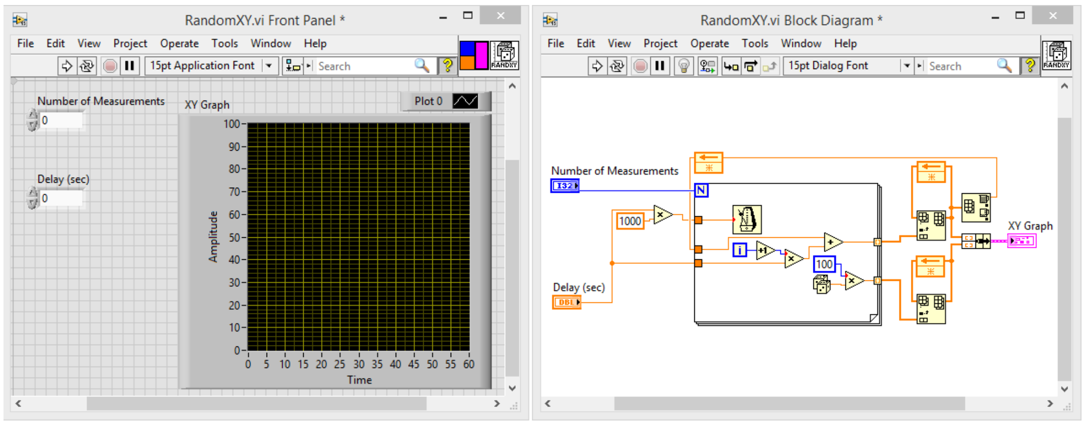


Figure 6.1: The main interface of LabVIEW. Each virtual instrument appears in two windows: the front panel (left) and the block diagram (right).

All widgets included in the front panel are either input elements, called *controls* in LabVIEW, or output elements, called *indicators*. This dichotomy leads to a much simpler model for UI programming, as opposed to typical GUI frameworks where one needs to implement handlers to various events that read-write widgets may produce. Indicators appear as write-only nodes in the block diagram and read-only widgets in the front panel; contrariwise, controls appear as data-entry widgets in the front panel that are read-only nodes in the block diagram.

When running a program inside the LabVIEW environment, the block diagram is still visible when running, but it is read-only. At runtime, the interaction with the block diagram is limited. It can only be decorated with temporary probes for debugging purposes, attached to wires. When a LabVIEW program is compiled and deployed, only the front panel is visible.

It is possible to update control values from the block diagram using more advanced features of the language that are less evident in its UI, but the environment is conducive to a simpler input/output discipline which presents data flowing from controls, to the block diagram, and finally to indicators.

### 6.1.1 Execution modes

There are two modes of execution, which can be launched, respectively, by the first two icons in the toolbars depicted in Figure 6.1. In the LabVIEW UI they are simply called “Run” and “Run continuously”. We will therefore call these modes *single-shot* and *continuous*.

In single-shot mode, all nodes and controls structures at the top-level graph are fired at most once; control structures may loop, causing inner nodes to fire multiple times. To begin the execution of this mode, LabVIEW fires all controls and nodes that do not depend on other objects, and execution continues until there are no more values to be propagated. When all values propagate to the appropriate indicators, their values are updated in the UI and the execution halts, bringing LabVIEW back to edit mode. However, if a diagram contains an infinite loop, for example, the program will run forever. Since controls are only fired at the beginning of execution, further inputs entered by the user while a single-shot execution runs have no effect on that execution. Controls and indicators retain their last values.

Continuous mode is equivalent to enclosing the entire program in an infinite loop and firing controls on each iteration. Each iteration of this continuous run is equivalent to one single-shot execution: all controls are fired, and the graph evaluation essentially restarts on each step, except that two kinds of objects, shift registers and feedback nodes, also retain their values from iteration to iteration. These two objects, which we will describe in detail in the next section, are the only nodes that can represent cyclic connections in the graph. In effect, single-shot execution is acyclic, and cycles can only propagate values from one iteration of a graph (or subgraph) to the next.

Two restrictions ensure that at most one value arrives at an input port during an iteration: first, that cycles only happen across iterations; second, that only one wire can be connected to an input port. This characterizes a static dataflow model, for which no buffers are necessary in input ports. This greatly simplifies scheduling and memory management: it is not possible to produce a stack overflow through an execution cycle, or a buffer overflow in nodes (for there are no buffers). As we will see below, however, the presence of aggregate data types brings back concerns about memory management.

Each control is fired only once in single-shot mode and only at the beginning of each iteration of the main graph in continuous mode. This means that having a long-running main graph in continuous mode leads to an unresponsive VI.

### 6.1.2 Data types and wires

The language features primitive and structured data types. It supports a large number of primitive basic types: 8, 16, 32 and 64 bit integers; fixed-point, floating-point and complex numbers of various sizes. For structured data, LabVIEW includes single and multi-dimensional arrays, as well as record types called *clusters*.

Controls, nodes and indicators are connected through *wires*, which is how edges in the dataflow graph are called. Apart from a special “dynamic” wire which sees limited use in LabVIEW as it demands special conversions, wires are in general statically typed: each wire has a data type assigned to it. In the UI, the color, thickness and stripe pattern of the wire indicates its type: the color represents the base type (integer, floating point, array, cluster), the thickness represents data dimensions (scalar, single or multidimensional array) and additional styling such as stripe patterns are used for particular types, such as waveforms and errors, which are just predefinitions for particular cluster types. For example, a waveform is a cluster containing on array of data points, a numeric timestamp and a numeric interval value between data samples. Error data flows as a cluster of three values: a boolean indicator an error condition, a 32-bit integer with the error code, and a string containing source information.

Some types support automatic coercions. For example, it is possible to connect an integer output port to a floating-point input port. The resulting wire has integer type: coercion happens at the input edge of the wire.

Not all types can be visually distinguished in the interface. Apart from the

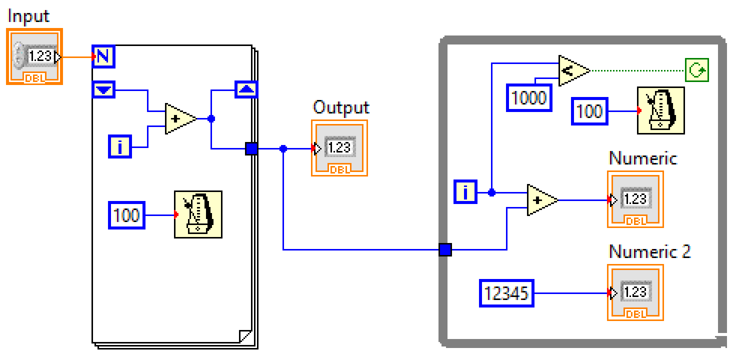


Figure 6.2: Looping constructs in LabVIEW, a "for" and a "while" loop.

especially predefined clusters like waveform and error, all user-defined cluster types look alike. For these wires, the contextual help window serves as a useful aid, describing the complete type of the wire under the mouse cursor.

Representing data structures is a well known difficulty in the area of dataflow [JHM04]. LabVIEW takes a simple approach: structured data such as arrays and clusters still flow as a single dataflow packet. The transfer of a whole array between two nodes happens as a single firing through an array-typed wire. To compensate for the low granularity of arrays in the flow of data, LabVIEW offers a number of nodes with complex array operations. New functionality to make it easier to manipulate arrays continues to be added. All three of the new plug-ins introduced in LabVIEW 2015 that were suggested by users of the vendor's discussion forums deal with array types: "Change to Array or Element", "Size Array Constants to Contents", "Transpose 2D Array". This indicates that users need high-level array operations.

### 6.1.3 Looping and cycles

From the end-user's perspective, LabVIEW programs are graphs that can contain cycles, but these are controlled via the use of explicit *feedback nodes* and structured looping constructs.

The LabVIEW UI enforces that the only connections producing explicit cycles in a graph are those connecting feedback nodes: wiring any two objects in the graph producing a cycle automatically inserts a feedback node between them. This feedback node exists solely to store the value in between executions of the graph. In Figure 6.1, feedback nodes appear as light orange nodes containing a ← sign.

Structured looping constructs are available, mirroring those available in traditional textual languages. The "for-loop" and "while-loop" constructs act as their familiar equivalents. In both cases, the looping construct appears in the UI as a frame inside the graph: they are both depicted in Figure 6.2. The "while" structure, presented at the right, always runs at least once and the condition can be negated by clicking on the green ↻ symbol, so it is more like either a "do-while" or a "repeat-until" construct. The frame of the loop encloses a subgraph and controls its iteration. The subgraph may produce one or more values that are sent out when the iteration completes.

Values may also be sent from one iteration of the loop to the next through the use of *shift registers*. A shift register appears as a pair of small nodes at the edges of the loop frame: one incoming connector at the right and one outgoing connector at the left. The "for" loop at the left in Figure 6.2 showcases a shift register. Shift registers are conceptually similar to how textual dataflow languages like Lucid allow iteration, with `x` and `next x` holding distinct values. The presence of a shift register denotes an implicit cycle in the graph.

In short, feedback nodes are constructs for sending data through iterations of the main graph when in continuous mode, and shift registers are constructs for sending data through iterations of a loop subgraph. It is notable that while functionally very similar, they have very different representations in the UI.

### 6.1.4 Timing

LabVIEW offers two wait functions: "Wait Until Next ms Multiple" and "Wait". The former monitors a millisecond counter and waits until it reaches a multiple of a given number, controlling the loop execution rate by holding back the loop step. This is designed for the synchronization of loops, typically when performing device reads and outputs to indicators in the interface. Note that the multiple may happen right after the loop starts, so the delay for the first iteration of the loop is indeterminate. The latter function, "Wait", always waits the specified amount of time, effectively adding a pause of constant size between steps of the loop.

The firing of a wait node inside a loop construct holds back the next step of the loop, effectively controlling the execution rate, assuming the rest of the code in the loop takes less time to execute than the configured delay. When multiple parallel loops exist in the graph, using "Wait Until Next ms Multiple" allows one to synchronize them to millisecond precision, which is often good enough for the domain of data acquisition hardware. It is amusing to note that the icon for "Wait Until Next ms Multiple" (visible in Figure 6.2) is a metronome, a device for counting tempo in music; in Pure Data, the function for generating periodic messages is called `metro`, referencing the same device. The image of the metronome reinforces the idea of "orchestration" between parallel agents.

### 6.1.5 Tunnels

Tunnels are nodes that send or receive data into or out of a structure. By connecting a wire from a node inside a structure to another node outside it or vice versa, a tunnel is automatically created at the edge of the structure frame.

When using tunnels to send values into or out of loop structures, the values are transferred only at the beginning or at the end of the execution of the loop. Figure 6.3 illustrates the behavior of input and output tunnels in loops. We have three while-loops in which the termination condition is connected to a boolean button, the iteration counter is connected to a numeric indicator, and the loop timing is controlled to execute one iteration per 1000 ms. In Loop A, both the inputs and outputs are connected through tunnels. The end result is that the termination value

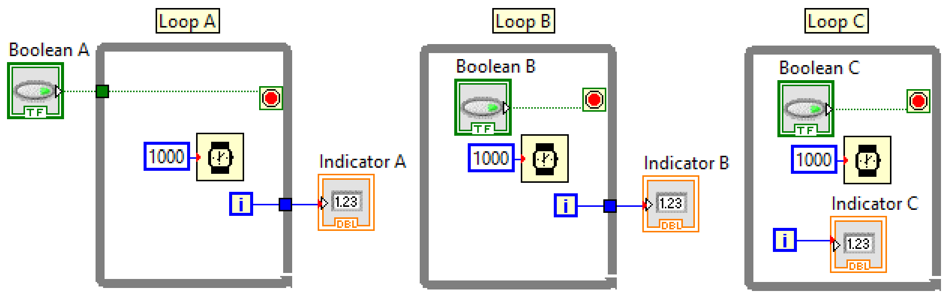


Figure 6.3: Interaction of loops and tunnels in LabVIEW. Loop A never updates its output; Loop B updates only at the end; Loop C updates every second.

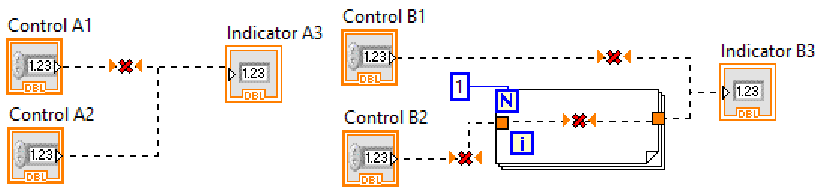


Figure 6.4: Connection errors in LabVIEW. Connecting two data sources (A1 and A2) directly to an input (A3) produces a helpful error message. If one of the connections goes through a tunnel, however (B2), this produces a tunnel direction inference error.

becomes fixed in the loop as the value of Boolean A goes through the tunnel. If the value of Boolean A at the beginning of execution is true, the loop runs for only one iteration and 0 is sent to Indicator A. If Boolean A is false at the beginning of execution, the loop never stops, and clicking the button has no effect. In Loop B, Boolean B is inside the loop, so its value is sent to the conditional terminal on each iteration. The loop stops once Boolean B is set to true, and only then the iteration value is sent via the output tunnel to Indicator B. In Loop C, the interface displays Indicator C being updated once a second, as long as Boolean C is false.

Because the direction of tunnels is inferred, incorrect connections involving tunnels produce less useful messages than similar connections not going through a tunnel. Figure 6.4 illustrates how tunnel inference affects error messages. When one connects two controls (A1 and A2) to an indicator (A3), this produces one error message that says “a wire can be connected to only one data source”. When one attempts a similar connection (B1 and B2 to B3), but one of these data sources (B2) goes through a tunnel, this produces three identical error messages that say “wire connected to an undirected tunnel: a tunnel on this wire cannot determine whether it is an input or an output”. This is typical of error messages involving inference: the inference engine of the language detects that a unification was not possible, but cannot tell which one of the two mismatching elements is the incorrect one.

### 6.1.6 Other control structures

LabVIEW also supports other control structures, two of which will be briefly discussed here: “case” and “sequence”. Both structures hold an arbitrary number of subgraphs. The “case” structure is presented as a frame that has a number of pages,

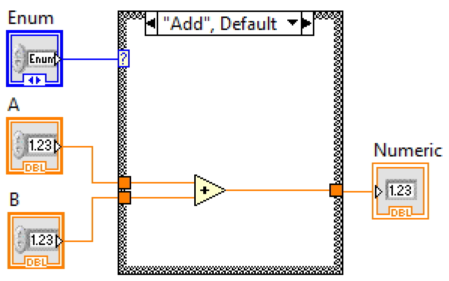


Figure 6.5: ”Case” structure in LabVIEW

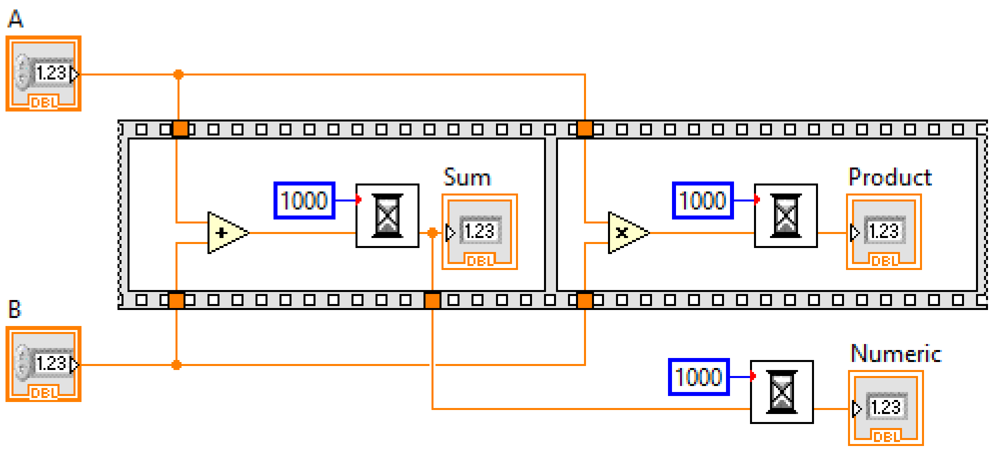


Figure 6.6: ”Sequence” structure in LabVIEW

each of them holding a subgraph for each case. It accepts an enumeration value as input to select the active page. Figure 6.5 illustrates a “case” structure. The enumeration selects which operation is to be applied to the two inputs A and B.

The “sequence” structure is presented in the UI as film roll with a series of frames, each holding a subgraph to be executed one after the other. This is an escape from the pure dataflow model, and provides a way to force a particular control flow structure regardless of data dependencies. Figure 6.5 illustrates a “sequence” structure. Note that inputs A and B arrive to the input tunnels of both frames immediately, but the second frame will only execute after the first frame finishes. The output tunnel for the first frame will only fire after one second, so the “Numeric” indicator and the “Product” indicator will be updated at the same time.

## 6.2 An interpreter modeling the semantics of LabVIEW

A difficulty in discussing the semantics of LabVIEW is that is has no published specification. Its documentation often resorts to examples to explain concepts, and does not serve an an exhaustive specification of the language. Previous attempts on the formalization of LabVIEW have been restricted to subsets of the language, and based themselves on its user manual and experimenting with the tool itself. In [MS98], Mok and Stuart map a subset of the language to RTL (real-time logic), a first-order logic used for describing real-time and embedded systems; they note that design decisions had to be made in points where the precise behavior was not clear. In [KKR09], Kaufmann et al. map a purely functional subset of LabVIEW into a dialect of Common Lisp used by the ACL2 theorem prover.

Like previous work in the literature, we designed the model based on LabVIEW's documentation and experimentation with the tool itself. We limited ourselves to the core logic of graph evaluation, the main control structures, and a few nodes that would allow us to run examples and model time-based execution. Additional features of LabVIEW that were not implemented include support for multiple non-reentrant VIs, global variables (which are not really global variables in the traditional sense but actually references to external VIs); object-oriented features, advanced event handling for controls and indicators, and object references. Still, we believe this work to be a more detailed model than the ones previously available in the literature. For instance, it models sequences and nodes with side-effects.

This implementation uses only standard modules included in the Haskell Platform:

```
module LvInterpreter where
import Data.Sequence (Seq, fromList, index, update, elemIndexL)
import qualified Data.Sequence as Seq (length, take)
import Data.Char
import Data.List
import Data.Maybe
import Data.Bits
import Data.Foldable (toList)
import Data.Generics.Aliases (orElse)
```

### 6.2.1 Representation of programs

A program in LabVIEW (called a VI, or Virtual Instrument) is a graph connecting different kinds of objects. In LabVIEW terminology, these objects are called *controls*, which are input-only, *indicators*, which are output-only, and *nodes*, which are all other operations. Throughout the implementation, we will use this nomenclature; in particular the name "node" will be used only for graph objects which are not controls or indicators. Graph objects are connected through wires. To avoid confusion with objects in the interpreter implementation, we will refer to graph objects (controls, indicators and nodes) as *elements*.

We represent a VI as a record containing a series of lists, enumerating controls, indicators, nodes and wires. Controls, indicators and nodes are paired with their names for display purposes only. The list of wires constitutes an adjacency list for the graph connections.

```
data LvVI = LvVI {
                vCtrls  :: [(String, LvControl)],
                vIndics :: [(String, LvIndicator)],
                vNodes :: [(String, LvNode)],
                vWires :: [LvWire]
              }
    deriving Show
```

A control in LabVIEW is an input widget in the VI's front panel, which also gets a representation as an object in the block diagram. However, since LabVIEW includes structured graphs composed of subgraphs representing structures such as for- and while-loops, we build these graphs in the interpreter recursively, declaring subgraphs as *LvVI* objects. For this reason, we use controls and indicators not only to represent GUI objects of the front panel, but also inputs and outputs of subgraphs. To do this, we declare a number of types of controls: a plain control that corresponds to a GUI object; an "auto" control that represents an automatically-generated input value, such as the increment count in a for-loop; a "tunnel" control, which is an input that connects data from the enclosing graph to the subgraph; and a "shift-register" control, which is the input terminator for shift registers (a construct to send data across iterations of a loop).

```
data LvControl = LvControl LvValue
               | LvAutoControl
               | LvTunControl
               | LvSRControl LvValue
  deriving Show
```

An indicator in LabVIEW is an output widget in the VI's front panel. Like controls, indicators are represented both in the front panel (as a GUI widget) and in the block diagram (as a connectable object). For the same reasons as explained above for controls, we have different kinds of indicators: the plain indicator, which represents a GUI indicator proper; the "shift-register" indicator, which sends data to its respective shift-register control (represented by the numeric index of the control in its constructor) for the next execution of a loop; and the "tunnel" indicator, which sends data out of the subgraph back to the enclosing graph.

Tunnel indicators can be of different types: "last value", which sends out the value produced by the last iteration of the subgraph; "auto-indexing", which produces an array accumulating all values received by the tunnel across all iterations of the subgraph; and "concatenating", which concatenates all values received. Here, we implement the "last value" and "auto-indexing" modes, since the "concatenating" mode is a mere convenience that could be achieved by concatenating the values of the array returned in the "auto-indexing" mode.

The LabVIEW interface enables auto-indexing by default when sending data out of for-loops, but this can be overridden by the user in the UI.

```
data LvIndicator = LvIndicator LvValue
                 | LvSRIndicator Int
                 | LvTunIndicator LvTunnelMode
  deriving Show
```

```
data LvTunnelMode = LvAutoIndexing
                  | LvLastValue
  deriving Show
```

There are several kinds of nodes in LabVIEW. The vast majority are functions, but there are also control structures, constants and feedback nodes.

Functions are identified in our implementation by their their names. They can have zero or more input ports, and zero or more output ports.

There are various kinds of control structures. Due to the fact that many of them share code in our implementation, we grouped them in the *LvStructure* type constructor: those are while-loops, for-loops, sequences, and sub-VIs. The case-structure controls a list of sub-VIs, and for this reason is handled separately with the *LvCase* constructor.

A constant is a node that holds a value. It has a single output port and immediately fires its value.

A feedback node holds the value it receives through its input port and fires it the next time the program is executed, when running in continuous mode as explained in Section 6.1.1.

```
data LvNode = LvFunction String
            | LvStructure LvStrucType LvVI
            | LvCase [LvVI]
            | LvConstant LvValue
            | LvFeedbackNode LvValue
  deriving Show
data LvStrucType = LvWhile
                 | LvFor
                 | LvSequence
                 | LvSubVI
  deriving Show
```

LabVIEW supports a large number of primitive numeric types: single, double and extended-precision floating-point numbers; fixed-point numbers; signed and unsigned integers of 8, 16, 32 and 64 bits; single, double and extended-precision complex numbers. We chose to implement only one floating-point and one integer type.

Besides these, the interpreter also supports the following types: strings; booleans; the *clusters*, which are a heterogeneous tuple of values (working like a record or "struct"); and homogeneous arrays.

Unlike LabVIEW, our implementation allows arbitrarily recursive types (e.g. we support a cluster of arrays of arrays of clusters).

Though LabVIEW supports arrays of clusters, and clusters of arrays, it does not support arrays of arrays. The recommended alternative is to use an "array of cluster of array": an array where elements are single-element clusters containing an array. This limitation is an explicit design decision, harking back to the development of LabVIEW 2.0 in 1988[2].

[2]`https://forums.ni.com/t5/LabVIEW-Idea-Exchange/Add-Support-for-Array-of-Array/idi-p/1875123`

Since we assume that input programs are properly type-checked, implementing the same restrictions that LabVIEW enforces to aggregate data types could be easily done in the type-checking step.

```
data LvValue = LvDBL Double
             | LvI32 Int
             | LvSTR String
             | LvBool Bool
             | LvCluster [LvValue]
             | LvArr [LvValue]
  deriving (Show, Eq, Ord)
```

A wire is a connection between two objects, represented as a source-destination pair of port addresses. Each port address, denoted ⦇$t, e, p$⦈, is a triple containing the element type (control, indicator or node), the element index and the port index within the element. For the source tuple, the port index denotes the element's output port; for the destination tuple, it denotes the input port.

```
data LvWire = LvWire {
                  wSrc :: ⦇·, ·, ·⦈,
                  wDst :: ⦇·, ·, ·⦈
                }
  deriving Show
data ⦇·, ·, ·⦈ = ⦇LvElemType, Int, Int⦈
  deriving Eq
instance Show ⦇·, ·, ·⦈ where
  show ⦇typ, eidx, pidx⦈ =
    "{" ++ show typ ++ " " ++ show eidx ++ ", " ++ show pidx ++ "}"
data LvElemType = LvC
                | LvI
                | LvN
  deriving (Show, Eq)
```

### 6.2.2 Representation of state

The representation of a state in our interpreter is a record containing the following values: the timestamp, a scheduler queue listing the next elements that need to be processed, and three sequences that store the internal states of nodes, controls and indicators. For controls and indicators, the sequences store their values. A VI always initializes controls and indicators with default values. Elements in the scheduler queue are denoted as ⦇$t, e$⦈, where $t$ is the type of the element (control, indicator or node) and $e$ is the numeric index of the element in its appropriate list in the *LvVI* object.

```
data LvState = LvState {
                   sTs          :: Int,
                   sPrng      :: Int,
                   sSched     :: [⦇·, ·⦈],
                   sNStates   :: Seq LvNodeState,
                   sCtrlVals  :: Seq LvValue,
                   sIndicVals :: Seq LvValue
                 }
     deriving Show
data ⦇·, ·⦈ = ⦇LvElemType, Int⦈
     deriving Eq
instance Show ⦇·, ·⦈ where
     show ⦇typ, eidx⦈ =
       "{" ++ show typ ++ " " ++ show eidx ++ "}"
```

For node states, the interpreter stores the contents of the input ports and an optional continuation. Each input port may be either empty or contain a single value, in accordance with the static dataflow model.

```
data LvNodeState = LvNodeState {
                     nsInputs :: Seq (Maybe LvValue),
                     nsCont   :: Maybe LvCont
                   }
  deriving Show
```

For functions, we use continuations to model computations that run over time. An operation that needs to continue running beyond the current timestamp implements the rest of the computation as a separate function, which will be scheduled to run at the next time tick. In the *LvKFunction* constructor we store the continuation function itself (*kFn*) and the values that will be passed to it (*kArgs*). These values act as the operation's internal memory. A continuation function returns either *LvReturn*, which contains the result values to be sent through the function's output ports, or *LvContinue*, which encapsulates the next continuation to be executed as the operation resumes running.

For subgraph structures, such as loops, the continuation of its execution is the state of the sub-VI. Note that, this way, the interpreter models a hierarchical tree of scheduler queues, as each structure node keeps an *LvState* with its own *sSched* queue. This way, multiple subgraphs can run concurrently.

```
data LvCont = LvKFunction {
                kFn   :: LvWorld → [LvValue] → (LvWorld, LvReturn),
                kArgs :: [LvValue]
              }
            | LvKState LvState
instance Show LvCont where
  show (LvKFunction _ args) = "KFunction(" ++ show args ++ ")"
  show (LvKState s)         = "KState[" ++ show s ++ "]"
data LvReturn = LvReturn [LvValue]
              | LvContinue LvCont
```

In all functions implementing LabVIEW nodes, we include an additional argument and an additional result representing access to side-effects that affect the state of the external world.

These extra values allow us to model impure functions whose effects depend not only on the inputs received through wires in the dataflow graph. In particular, this allows us to model the relationship between graph evaluation and time.

In our model, a simplified view of this "external world" is implemented as the *LvWorld* type. It consists of a read-only timestamp, which we will use as a model of a "system clock" for timer-based functions, and the read-write pseudo-random number generator (PRNG) state, which can be consumed and updated.

```
data LvWorld = LvWorld {
                    wTs :: Int,
                    wPrng :: Int
                  }
```

Note that *LvWorld* is a subset of our *LvState* object, which represents the memory of the VI being executed. In this sense, this is the part of the outside world that is visible to the function.

### 6.2.3 Execution

The execution mode of LabVIEW is data-driven. The user enters data via controls, which propagate their values through other nodes, eventually reaching indicators, which provide feedback to the user via their representations in the front panel.

This interpreter models a single-shot execution (as discussed in Section 6.1.1). Continuous execution is semantically equivalent as enclosing the entire VI in a while-loop.

#### 6.2.3.1 Main loop

The execution of the interpreter is a loop of evaluation steps, which starts from an initial state defined for the VI and runs producing new states until a final state with an empty scheduler queue is produced.

```
runVI :: LvVI → IO ()
runVI vi =
  loop (initialState 0 42 vi)
  where
    loop s = do
      print s
      case sSched s of
        [] → return ()
        _ → loop (run s vi)
```

#### 6.2.3.2 Initial state

The initial state consists of the input values entered for controls, the initial values of indicators, and empty states for each node, containing the appropriate number of empty slots corresponding to their input ports. It also contains the initial schedule, which is the initial list of graph elements to be executed.

```
initialState :: Int → Int → LvVI → LvState
initialState ts prng vi =
  LvState {
    sTs         = ts + 1,
    sPrng       = prng,
    sCtrlVals   = fromList $ map (makeCtrlVal ∘ snd)  (vCtrls vi),
    sIndicVals  = fromList $ map (makeIndicVal ∘ snd) (vIndics vi),
    sNStates    = fromList $ mapIdx makeNState        (vNodes vi),
    sSched      = initialSchedule vi
  }
  where
    makeNState :: (Int, (String, LvNode)) → LvNodeState
    makeNState (i, (name, node)) = LvNodeState {
                                     nsInputs = emptyInputs $ nrInputs i node,
                                     nsCont   = Nothing
                                   }
    nrInputs :: Int → LvNode → Int
    nrInputs i (LvFunction _)         = nrConnectedInputs i vi
    nrInputs _ (LvConstant _)         = 0
    nrInputs _ (LvStructure _ subvi)  = length $ vCtrls subvi
    nrInputs _ (LvCase subvis)        = length $ vCtrls (head subvis)
    nrInputs _ (LvFeedbackNode _)     = 1
    makeCtrlVal :: LvControl → LvValue
    makeCtrlVal (LvControl   v) = v
    makeCtrlVal (LvSRControl v) = v
    makeCtrlVal _               = LvI32 0
    makeIndicVal :: LvIndicator → LvValue
    makeIndicVal (LvIndicator v)                = v
    makeIndicVal (LvTunIndicator LvAutoIndexing) = LvArr [ ]
    makeIndicVal _                              = LvI32 0
    mapIdx :: ((Int, a) → b) → [a] → [b]
    mapIdx fn l = zipWith (curry fn) (indices l) l
emptyInputs :: Int → Seq (Maybe LvValue)
emptyInputs n = fromList (replicate n Nothing)
```

The initial schedule is defined as follows. All controls, constants and feedback nodes are queued. Then, all function and structure nodes which do not depend on other inputs are queued as well. Here, we make a simplification and assume that VIs do not have any functions with mandatory inputs missing. This could be verified in a type-checking step prior to execution.

Note also that the code below implies the initial schedule follows the order of nodes given in the description of the *LvVI* record, leading to a deterministic execution of our intpreter. LabVIEW does not specify a particular order.

```
initialSchedule :: LvVI → [(|·, ·|)]
initialSchedule vi =
  map (|LvC, ·|) (indices $ vCtrls vi)
  ++ map (|LvN, ·|) (filter (λi → isBootNode i (vNodes vi !! i)) (indices $ vNodes vi))
  where
    isBootNode _ (_, LvConstant _) = True
    isBootNode _ (_, LvFeedbackNode _) = True
    isBootNode i (_, LvFunction _)                 | nrConnectedInputs i vi ≡ 0 = True
    isBootNode i (_, LvStructure LvWhile _)        | nrConnectedInputs i vi ≡ 0 = True
    isBootNode i (_, LvStructure LvSubVI _)        | nrConnectedInputs i vi ≡ 0 = True
    isBootNode i (_, LvStructure LvSequence _)     | nrConnectedInputs i vi ≡ 0 = True
    isBootNode _ _ = False
```

A node can only be fired when all its connected inputs have incoming data. We specifically check for connected inputs because some LabVIEW nodes have optional inputs. We assume here for simplicity that the type-checking step prior to execution verified that the correct set of mandatory inputs has been connected. Here, we derive the number of connections of a node from the list of wires.

```
nrConnectedInputs :: Int → LvVI → Int
nrConnectedInputs idx vi =
  1 + foldl' maxInput (−1) (vWires vi)
  where
    maxInput :: Int → LvWire → Int
    maxInput mx (LvWire _ (|LvN, i, n|)) | i ≡ idx = max mx n
    maxInput mx _ = mx
```

#### 6.2.3.3 Event processing

The main operation of the interpreter consists of taking one entry off the scheduler queue, incrementing the timestamp, and triggering the event corresponding to that entry. Every time we produce a new state, we increment the timestamp. The timestamp, therefore, is not a count of the number of evaluation steps, but is a simulation of a system clock, to be used by timer operations.

```
run :: LvState → LvVI → LvState
run s vi
 | null (sSched s) = s
 | otherwise =
  case sSched s of
  (q : qs) → let s₀ = s {sTs = (sTs s) + 1, sSched = qs}
             in  runEvent q s₀ vi
```

An event in the queue indicates the graph element to be executed next. Function *runEvent* takes a $(\!|\cdot,\cdot|\!)$ that identifies the element, a state and a VI, and produces a new state, with the results of triggering that element:

$runEvent :: (\!|\cdot,\cdot|\!) \rightarrow LvState \rightarrow LvVI \rightarrow LvState$

When triggering a control, its effect is to fire its value through its sole output port.

```
$runEvent\ (\!|LvC, idx|\!)\ s_0\ vi =$
  $fire\ vi\ cv\ (\!|LvC, idx, 0|\!)\ s_0$
  **where**
    $cv = index\ (sCtrlVals\ s_0)\ idx$
```

When triggering a node for execution, the event may be triggering either an initial execution from data fired through its input ports, or a continuation of a previous execution that has not finished running. In the former case, the interpreter fetches the data from the node's input ports and clears it from the node state, ensuring incoming values are consumed only once. In the latter case, the inputs come from the data previously stored in the continuation object and the node state is kept as is. Once the inputs and state are determined, *runEvent* calls *runNode*, which produces a new state and may produce data to be fired through the node's output ports.

```
$runEvent\ (\!|LvN, idx|\!)\ s_0\ vi =$
  $foldl'\ (\lambda s\ (p, v) \rightarrow fire\ vi\ v\ (\!|LvN, idx, p|\!)\ s)\ s_2\ pvs$
  **where**
    $ns = index\ (sNStates\ s_0)\ idx$
    $(s_1, inputs) =$
      **case** $nsCont\ ns$ **of**
      $Nothing \rightarrow startNode$
      $Just\ k \quad \rightarrow continueNode\ k$
    $(s_2, pvs) = runNode\ (snd\ \$\ vNodes\ vi\ !!\ idx)\ s_1\ inputs\ idx$
    $startNode = (s_1, inputs)$
      **where**
        $s_1$          $= updateNode\ idx\ s_0\ clearState\ [\,]$
        $inputs$       $= toList\ (nsInputs\ ns)$
        $clearState$ $= ns\ \{nsInputs = clear\}$
        $clear$        $= emptyInputs\ (Seq.length\ (nsInputs\ ns))$
    $continueNode\ k = (s_1, inputs)$
      **where**
        $s_1$      $= s_0$
        $inputs = $ **case** $k$ **of**
                   $LvKFunction\ \_\ kargs \rightarrow map\ Just\ kargs$
                   $LvKState\ \_$               $\rightarrow \bot$
```

When updating the internal state of a node, we use the auxiliary function *updateNode*, which increments the timestamp, optionally appends events to the scheduler queue, and replaces the node state for the node at the given index.

```
updateNode :: Int → LvState → LvNodeState → [⦇·, ·⦈] → LvState
updateNode idx s ns sched =
  s {
    sTs       = sTs s + 1,
    sSched    = sSched s ++ sched,
    sNStates = update idx ns (sNStates s)
  }
```

### 6.2.3.4 Firing data to objects

As shown in the previous section, when objects are triggered for execution, they may produce new values which are fired through their output ports. The function *fire* iterates through the adjacency list of wires, identifying all outward connections of an object and propagating the value to their destination nodes.

```
fire :: LvVI → LvValue → ⦇·, ·, ·⦈ → LvState → LvState
fire vi value addr s =
  foldl′ checkWire s (vWires vi)
    where
    checkWire s (LvWire src dst) =
      if addr ≡ src
      then propagate value vi dst s
      else s
```

When a value is propagated to an indicator, its value is stored in the state, with the appropriate handling for different kinds of tunnel indicators.

```
propagate :: LvValue → LvVI → ⦇·, ·, ·⦈ → LvState → LvState
propagate value vi ⦇LvI, dnode, _⦈ s =
  let
    (_, indicator) = vIndics vi !! dnode
    newValue       =
      case indicator of
      LvIndicator _                       → value
      LvSRIndicator _                     → value
      LvTunIndicator LvLastValue          → value
      LvTunIndicator LvAutoIndexing → let arr = index (sIndicVals s) dnode
                                      in insertIntoArray arr value []
  in
    s {
      sTs          = sTs s + 1,
      sIndicVals = update dnode newValue (sIndicVals s)
    }
```

When a value is propagated to a node, the interpreter stores the value in the *nsInputs* sequence of the node state. Then, it needs to decide whether the node needs to be scheduled for execution.

```
propagate value vi (|LvN, dnode, dport|) s =
  s {
    sTs      = sTs s + 1,
    sSched   = sched',
    sNStates = nss'
  }
  where
    nss     = sNStates s
    ns      = index nss dnode
    inputs' = update dport (Just value) (nsInputs ns)
    nss'    = update dnode (ns {nsInputs = inputs'}) nss
    sched'  =
      let
        sched = sSched s
        entry = (|LvN, dnode|)
      in
        if shouldSchedule (snd $ vNodes vi !! dnode) inputs' ∧ entry ∉ sched
        then sched ++ [entry]
        else sched
```

To determine if a node needs to be scheduled, the interpreter checks if all its required inputs contain values. For function nodes, this means that all mandatory arguments must have incoming values. For structures, it means that all tunnels going into the structure must have values available for consumption.

This interpreter implements a single node accepting optional inputs, `InsertIntoArray` (Section 6.2.5.2); for all other nodes, all inputs are mandatary.

Feedback nodes are never triggered by another node: when they receive a value through its input port, this value remains stored for the next single-shot execution of the whole graph. Constants do not have input ports, so they cannot receive values.

```
shouldSchedule :: LvNode → Seq (Maybe LvValue) → Bool
shouldSchedule node inputs =
  case node of
    LvFunction name   → shouldScheduleNode name
    LvStructure _ vi  → shouldScheduleSubVI vi inputs
    LvCase vis        → shouldScheduleSubVI (head vis) inputs
    LvFeedbackNode _  → False
    LvConstant _      → ⊥
  where
    shouldScheduleNode name =
      isNothing $ elemIndexL Nothing mandatoryInputs
      where
        mandatoryInputs =
          case nrMandatoryInputs name of
            Nothing  → inputs
            Just n   → Seq.take n inputs
    shouldScheduleSubVI :: LvVI → Seq (Maybe LvValue) → Bool
  shouldScheduleSubVI vi inputs =
    isNothing $ find unfilledTunnel (indices $ vCtrls vi)
      where
        unfilledTunnel cidx =
          case vCtrls vi !! cidx of
            (_, LvTunControl) → isNothing (index inputs cidx)
            _                 → False
nrMandatoryInputs :: String → Maybe Int
nrMandatoryInputs "InsertIntoArray" = Just 2
nrMandatoryInputs _ = Nothing
indices :: [a] → [Int]
indices l = [0 .. (length l − 1)]
```

### 6.2.4 Nodes and structures

The function *runNode* takes care of implementing the general logic for each kind of node. For functions, it handles the management of continuations; for structures, it triggers their subgraphs according to each structure's rules of iteration and conditions of termination.

The function *runNode* takes a node, an input state, a list of input values, the integer index that identifies the node in the VI, and produces a new state and a list of index-value pairs, listing values to be sent through output ports.

```
runNode :: LvNode → LvState → [Maybe LvValue] → Int
             → (LvState, [(Int, LvValue)])
```

#### 6.2.4.1 Constant nodes

When executed, a constant node simply sends out its value through its single output port.

```
runNode (LvConstant value) s₁ _ _ =
  (s₁, [(0, value)])
```

#### 6.2.4.2 Feedback nodes

A feedback node behaves like a constant node: it sends out the value it stores through its output port. In spite of having an input port, a feedback node is only triggered at the beginning of the execution of the graph, as determined by the initial state (Section 6.2.3.2) and firing rules (Section 6.2.3.4).

In our model, an *LvFeedbackNode* always takes an initialization value. In the LabVIEW UI, this value can be left out, in which case a default value for the appropriate data type, such as zero or an empty string, is implied.

```
runNode (LvFeedbackNode initVal) s₁ inputs _ =
  (s₁, [(0, fromMaybe initVal (head inputs))])
```

#### 6.2.4.3 Function nodes

When running a function node, the interpreter first checks if it has an existing continuation pending for the node. If there is one, it resumes the continuation, applying the function stored in the continuation object $k$. Otherwise, it triggers the function (identified by its name) using *applyFunction*.

The function may return either a *LvReturn* value, which contains the list of result values be propagated through its output ports, or a *LvContinue* value, which contains the next continuation $k'$ to be executed. When a continuation is returned, the node itself (identified by its address *idx*) is also scheduled back in the queue, and no values are produced to be sent to the node's output ports.

```
$runNode\ (LvFunction\ name)\ s_1\ inputs\ idx =$
  $\textbf{let}$
    $nss \quad\ = sNStates\ s_1$
    $ns \quad\ \ = index\ nss\ idx$
    $world\ s = LvWorld\ \{wTs = sTs\ s, wPrng = sPrng\ s\}$
    $ret \quad\ =$
      $\textbf{case}\ nsCont\ ns\ \textbf{of}$
      $Nothing \rightarrow applyFunction\ name\ (world\ s_1)\ inputs$
      $Just\ k \quad \rightarrow kFn\ k\ (world\ s_1)\ (catMaybes\ inputs)$
    $(w, mk, q, pvs) =$
      $\textbf{case}\ ret\ \textbf{of}$
      $(w, LvReturn\ outVals) \rightarrow (w, Nothing, [\,], zip\ (indices\ outVals)\ outVals)$
      $(w, LvContinue\ k') \quad\ \rightarrow (w, Just\ k', \ [(\!|LvN, idx|\!)], [\,])$
    $updateWorld\ w\ s = s\ \{sPrng = wPrng\ w\}$
  $\textbf{in}$
    $(updateWorld\ w\ \$\ updateNode\ idx\ s_1\ ns\ \{nsCont = mk\}\ q, pvs)$
```

#### 6.2.4.4 Control structures

The interpreter supports five kinds of control structures: for-loop, while-loop, sequence, case and sub-VI. They are all implemented similarly, by running a subgraph (itself represented as an instance of *LvVI*, like the main graph), and storing a state object for this subgraph as a continuation object of the node state for the enclosing graph (represented as *LvState*, like the main state). Running this subgraph may take several evaluation steps, so the enclosing graph will continuously queue it for execution until it decides it should finish running. Each time the scheduler of the enclosing graph triggers the structure node, it will run the subgraph consuming one event of the internal state's own scheduler queue. This will, in effect, produce a round-robin of all structures that may be running concurrently.

This common behavior is implemented in the *runStructure* function that will be presented below. The implementations of *runNode* for all structures use *runStructure*, differing by the way they control triggering and termination of subgraphs.

The for-loop provides *runStructure* with a termination function *shouldStop* which determines if the loop should stop comparing the value of the counter control (at index 0) with the limit control (at index 1). Also, it uses the helper function *initCounter* to force the initial value of control 0 when the structure is triggered for the first time (that is, when it is not resuming a continuation).

```
runNode (LvStructure LvFor subvi) s₁ inputs idx =
  runStructure subvi shouldStop s₁ idx (initCounter s₁ idx inputs)
  where
    shouldStop s =
      (i + 1 ⩾ n)
      where
        LvI32 i  = index (sCtrlVals s) 0
        LvI32 n = coerceToInt $ index (sCtrlVals s) 1
    coerceToInt v@(LvI32 _) = v
    coerceToInt (LvDBL d)    = LvI32 (floor d)
```

The while-loop structure in LabVIEW always provides an iteration counter, implemented in the interpreter as a counter control at index 0. As in the for-loop, it is initialized using the helper function *initCounter*. The termination function for the while-loop checks for the boolean value at the indicator at index 0.

```
runNode (LvStructure LvWhile subvi) s₁ inputs idx =
  runStructure subvi shouldStop s₁ idx (initCounter s₁ idx inputs)
  where
    shouldStop s =
      ¬ test
      where
        LvBool test = index (sIndicVals s) 0
```

Sequence nodes in LabVIEW are a way to enforce order of execution irrespective of data dependencies. In the LabVIEW UI, sequences are presented as a series of frames presented like a film-strip. In our interpreter, we implement each frame of the film-strip as a separate *LvStructure* object containing a boolean control at input port 0 and a boolean indicator at output port 0. Frames of a sequence are connected through a wire connecting the frame's indicator 0 to the next frame's control 0. This way, we force a data dependency between frames, and the implementation of *runNode* for sequences pushes a boolean value to output port 0 to trigger the execution of the next frame in the sequence. This connection is explicit in our model, but it could be easily hidden in the application's UI.

```
runNode (LvStructure LvSequence subvi) s₁ inputs idx =
  let
    (s₂, pvs) = runStructure subvi (const True) s₁ idx inputs
    ns₂       = index (sNStates s₂) idx
    nextq = [(0, LvBool True) | isNothing (nsCont ns₂)]
  in
    (s₂, pvs ++ nextq)
```

Case structures are different from the other ones because they contain a list of subgraphs. All subgraphs representing cases are assumed to have the same set of controls and indicators, and they all have a numeric control at index 0 which determines which case is active. LabVIEW denotes cases using enumeration types, but in the interpreter we simply use an integer.

When a case node is triggered, *runNode* needs to choose which VI to use with *runStructure*. In its first execution, it reads from the input data sent to control 0; in subsequent executions, when those inputs are no longer available, it reads directly from the control value, which is stored in the node state. Note that since case VIs have the same set of controls and indicators, they are structurally equivalent, and the initialization routine in Section 6.2.3.2 simply uses the first case when constructing the initial empty state.

A case subgraph does not iterate: it may take several schedule events to run through a full single-shot execution, but once the subgraph scheduler queue is empty, it should not run again. For this reason, the termination function is simply *const True*.

```
runNode (LvCase subvis) s1 inputs idx =
  let
    ns1 = index (sNStates s1) idx
    n = case nsCont ns1 of
      Nothing → case inputs of
                  Just (LvI32 i) : _ → i
                  _                  → 0
      Just _ →    (λ(LvI32 i) → i) $
                  fromMaybe (error "no input 0") $ index (nsInputs ns1) 0
    (s2, pvs) = runStructure (subvis !! n) (const True) s1 idx inputs
    s3 =
      case nsCont ns1 of
        Nothing → let
                    ns2 = index (sNStates s2) idx
                    inputs = update 0 (Just (LvI32 n)) (nsInputs ns2)
                    ns3 = ns2 {nsInputs = inputs}
                  in
                    updateNode idx s2 ns3 []
        Just _ →  s2
  in
    (s3, pvs)
```

Finally, a sub-VI structure has a simple implementation, where we launch the subgraph with *runStructure*, directing it to run once and performing no additional operations to its state.

```
runNode (LvStructure LvSubVI subvi) s1 inputs idx =
  runStructure subvi (const True) s1 idx inputs
```

The core to the execution of all structure nodes is the *runStructure* function, which we present here. This function takes as arguments the subgraph to execute, the termination function to apply, the enclosing graph's state, and the index of the structure in the enclosing VI; it returns a pair with the new state and a list of port-value pairs to fire through output ports.

```
runStructure :: LvVI
             → (LvState → Bool)
             → LvState → Int → [Maybe LvValue]
             → (LvState, [(Int, LvValue)])
```

Its execution works as follows. First, it determines $sk_1$, which is the state to use when running the subgraph. If there is no continuation, a new state is constructed using *initialState* (Section 6.2.3.2), with the input values received as arguments entered as values for the structure's controls. If there is a continuation, it means it is resuming execution of an existing state, so it reuses the state stored in the *LvKState* object, merely updating its timestamp.

Then, it calls the main function *run* (Section 6.2.3.3) on the subgraph *subvi* and state $sk_1$. This produces a new state, $sk_2$. If the scheduler queue in this state is not empty, this means that the single-shot execution of the graph did not finish. In this case, the interpreter stores this new state in a continuation object *nextk* and enqueues the structure in the main state so it runs again.

If the scheduler queue is empty, *runStructure* runs the termination check *shouldStop* to determine if it should schedule a new iteration of the subgraph. If a new iteration is required, a new state is produced with *nextStep*, which increments the iterator and processes shift registers.

At last, if the execution does not produce a continuation, this means the structure terminated its single-shot execution: the values of the indicators are sent out to the structure's output ports.

```
runStructure subvi shouldStop s₁ idx inputs =
  let
    nss  = sNStates s₁
    ns   = index nss idx
    ts'  = sTs s₁ + 1
    prng = sPrng s₁
    sk₁ =
      case nsCont ns of
      Nothing              → setCtrlVals inputs (initialState ts' prng subvi)
      Just (LvKState st) → st {sTs = ts'}
    setCtrlVals inputs s =
      s {
        sTs       = sTs s + 1,
        sCtrlVals = fromList (zipWith fromMaybe (toList $ sCtrlVals s) inputs)
      }
    sk₂ = run sk₁ subvi
    nextk
      | ¬ (null (sSched sk₂)) = Just (LvKState sk₂)
      | shouldStop sk₂         = Nothing
      | otherwise          = let LvI32 i = index (sCtrlVals sk₂) 0
                             in Just (LvKState (nextStep subvi sk₂ (i + 1)))
    qMyself = [(|LvN, idx|) | isJust nextk]
    s₂ = s₁ {
      sTs      = sTs sk₂ + 1,
      sPrng    = sPrng sk₂,
      sSched   = sSched s₁ ++ qMyself,
      sNStates = update idx (ns {nsCont = nextk}) nss
    }
    pvs = zip (indices $ vIndics subvi) (toList $ sIndicVals sk₂)
  in
    (s₂, if isJust nextk then [ ] else pvs)
```

Structure nodes use the following auxiliary functions, already mentioned above. Function *initCounter* checks whether the node state has a continuation, and initializes the iteration counter if it doesn't. Function *nextStep* resets the scheduler for the state of the subgraph, and implements the shift register logic, copying values from indicators marked as *LvSRIndicator* to their corresponding controls in the new state.

```
initCounter :: LvState → Int → [Maybe LvValue] → [Maybe LvValue]
initCounter s idx inputs =
  case nsCont (index (sNStates s) idx) of
  Nothing → Just (LvI32 0) : tail inputs
  _       → inputs
nextStep :: LvVI → LvState → Int → LvState
nextStep vi s i′ =
  s {
    sTs       = sTs s + 1,
    sSched    = initialSchedule vi,
    sCtrlVals = cvs″
  }
  where
    cvs′ = update 0 (LvI32 i′) (sCtrlVals s)
    cvs″ = foldl′ shiftRegister cvs′ $ zip (vIndics vi) (toList (sIndicVals s))
    shiftRegister :: Seq LvValue → ((String, LvIndicator), LvValue) → Seq LvValue
    shiftRegister cvs ((_, LvSRIndicator cidx), ival) =
      update cidx ival cvs
    shiftRegister cvs _ = cvs
```

### 6.2.5 Operations

The final section of the interpreter is the implementation of the various operations available in the language as function nodes. These operations are implemented as cases for function *applyFunction*, which takes a string with the name of the function, an instance of the outside world, the list of input values, and produces a return, which may be a list of results or a continuation, along with the updated state of the world.

```
applyFunction :: String → LvWorld → [Maybe LvValue] → (LvWorld, LvReturn)
```

However, in the spirit of dataflow, most function nodes implement pure functions (that is, they do not read or affect the outside world). We represent them as such, removing the occurrences of *LvWorld* from the signature:

```
applyPureFunction :: String → [Maybe LvValue] → LvReturn
```

To fit the interpreter's execution model, these pure functions can then be converted to match the expected signature using the following combinator, which is able to convert the signature of *applyPureFunction* into that of *applyFunction*, by simply forwarding the *LvWorld* object unchanged:

```
withWorld :: (a → r) → (w → a → (w, r))
withWorld f = λw args → (w, f args)
```

Our goal in this interpreter is not to reproduce the functionality of LabVIEW with respect to its domain in engineering, but to describe in detail the semantics of the dataflow language at its core. For this reason, we include below only a small selection of functions, which should be enough to illustrate the behavior of the interpreter through examples.

The following pure functions are implemented: arithmetic and relational operators (Section 6.2.5.1), array functions `Array Max & Min` and `Insert Into Array` (Section 6.2.5.2), and `Bundle` (a simple function which packs values into a cluster).

To demonstrate impure functions, the interpreter includes the timer function `Wait Until Next Ms` (Section 6.2.5.4) and the PRNG function `Random Number` (Section 6.2.5.3).

```
applyPureFunction name =
  case name of
  "+"                → numOp (+) (+)
  "-"                → numOp (−) (−)
  "*"                → numOp (∗) (∗)
  "/"                → numOp (/) div
  "<"                → boolOp (<) (<)
  ">"                → boolOp (>) (>)
  "ArrayMax&Min"     → returnArrayMaxMin
  "InsertIntoArray"  → returnInsertIntoArray
  "Bundle"           → returnBundle
  otherwise          → error ("No rule to apply " ++ name)
  where
    returnArrayMaxMin [Just (LvArr a)] =
      LvReturn (arrayMaxMin a)
    returnInsertIntoArray (Just arr : Just vs : idxs) =
      LvReturn [insertIntoArray arr vs (map toNumber idxs)]
      where toNumber i = if isNothing i
                         then (−1)
                         else (λ(Just (LvI32 n)) → n) i
    returnBundle args =
      LvReturn [LvCluster (catMaybes args)]
```

### 6.2.5.1 Numeric and relational operators

LobVIEW nodes automatically perform coercions between integers and doubles. Since ports in our implementation do not carry type information (it assumes the input VI has been type-checked prior to execution), we pragmatically include the coercion logic directly in the implementation for numeric and relational operator nodes, codified in the *binOp* function, to which the *numOp* and *boolOp* functions below delegate.

It is worth noting that the LabVIEW UI gives visual feedback when a coercion takes place, by adding a small circle attached to the input port. This could be considered an automatically inserted coercion node, not unlike the automatic insertion of feedback nodes. However, since these are not separate nodes in LabVIEW (for instance, they cannot be probed as separate objects by the LabVIEW debugging facilities, unlike feedback nodes), we chose to not implement them as separate nodes, so keep node structure in input programs for this interpreter more alike to that of actual LabVIEW programs.

```
numOp :: (Double → Double → Double)
  → (Int → Int → Int) → [Maybe LvValue] → LvReturn
numOp op_d op_i = LvReturn ∘ return ∘ binOp op_d LvDBL op_i LvI32
boolOp :: (Double → Double → Bool)
  → (Int → Int → Bool) → [Maybe LvValue] → LvReturn
boolOp op_d op_i = LvReturn ∘ return ∘ binOp op_d LvBool op_i LvBool
binOp :: (Double → Double → t) → (t → LvValue)
  → (Int → Int → t1) → (t1 → LvValue)
  → [Maybe LvValue] → LvValue
binOp op_d t_d _ _ [Just (LvDBL a), Just (LvDBL b)] = t_d (op_d a b)
binOp op_d t_d _ _ [Just (LvI32 a),  Just (LvDBL b)] = t_d (op_d (fromIntegral a) b)
binOp op_d t_d _ _ [Just (LvDBL a), Just (LvI32 b)]  = t_d (op_d a (fromIntegral b))
binOp _ _ op_i t_i [Just (LvI32 a),  Just (LvI32 b)]  = t_i (op_i a b)
binOp _ _ _ _ _ = ⊥
```

### 6.2.5.2 Array functions

Representing aggregate data structures and processing them efficiently is a recognized issue in dataflow languages [JHM04]. LabVIEW includes support for arrays and clusters, and provides a large library of functions to support these data types. We illustrate two such functions in the interpreter.

`Array Max & Min` is a function that takes an array and produces four output values: the maximum value of the array, the index of this maximum value, the minimum value of the array, and the index of this minimum value. The design of this node reflects one concern which appears often in the LabVIEW documentation and among their users: avoiding excessive array copying. While languages providing similar functionality typically provide separate functions for `min` and `max`, here the language provides all four values at once, to dissuade the user from processing the array multiple times in case more than one value is needed. LabVIEW also provides a control structure called `In Place Element Structure`, not implemented in this interpreter, where an array and one or more indices are entered as inputs, producing input and output tunnels for each index, so that values can be replaced in an aggregate data structure without producing copies. More recent versions of LabVIEW avoid array copying through optimization, reducing the usefulness of this control structure.

```
arrayMaxMin a =
  if null a
  then [LvDBL 0, LvI32 0,        LvDBL 0, LvI32 0]
  else [maxVal,    LvI32 maxIdx, minVal,    LvI32 minIdx]
    where
      (maxVal, maxIdx) = foldPair (>) a
      (minVal, minIdx) = foldPair (<) a
      foldPair op l = foldl1 (λ(x, i) (y, j) → if op x y
        then (x, i)
        else (y, j))
                                    (zip l (indices l))
```

An example of a surprisingly large amount of functionality condensed into one function node is LabVIEW's `Insert Into Array` operation. To insert into an array $x$ a value $y$, this nodes features as input ports the target array ($x$), the data to be inserted ($y$, which may also be an array) and one indexing input port for each dimension of $x$. However, only one indexing port can be connected; the other ones must remain disconnected, and this indicates on which dimension the insertion should take place.

The behavior of the function changes depending on which of the inputs are connected and what are the number of dimensions of array $x$ and data $y$.

Given a $n$-dimensional array $x$, value $y$ must be either an $n$ or $(n-1)$-dimensional array (or in the case of inserting into a 1-D array, it must be either a 1-D array or an element of the array's base type).

For example, if $x$ is a 2D array $p \times q$ and $y$ is a 1D array of size $n$, if the first indexing input is connected, it inserts a new row into the matrix, producing an array $p+1 \times q$; if the second index is connected, it inserts a new column, and the resulting array size is $p \times q + 1$. This also works in higher dimensions: for example, one can insert a 2D matrix into a 3D array along one of its three axes.

When the dimensions are the same, the results are different: inserting an array of size $m \times n$ into an array of size $p \times q$ may produce an array of size $p + m \times q$ or $p \times q + n$. For all operations, the dimensions of $y$ are cropped or expanded with null values (such as zero or the empty string) to match the dimensions of $x$.

```
insertIntoArray :: LvValue → LvValue → [Int] → LvValue
insertIntoArray vx vy idxs =
  case (vx, vy, idxs) of
  (LvArr lx, _, [ ]) → insertIntoArray vx vy [length lx]
  (LvArr lx@(LvArr x: _), LvArr ly, −1 : is) → recurseTo is lx (next x lx ly)
  (LvArr lx@(LvArr x: _), LvArr ly, i : _)   → insertAt  i  lx (curr x lx ly)
  (LvArr lx,                      _,         i : _)   → insertAt  i  lx (base vy)
  where
    (next, curr, base) =
      if ndims vx ≡ ndims vy
      then (λ_ lx ly      → resizeCurr id lx ly,
            λ_ lx ly      → resizeLower lx ly,
            λ(LvArr ly) → ly)
      else ( λx _ ly      → resizeCurr id x ly,
             λx _ ly      → [LvArr (resizeAll x ly)],
             λ_           → [vy])
    insertAt i lx ly = LvArr $ take i lx ++ ly ++ drop i lx
    recurseTo is lx ly = LvArr $ zipWith (λa b → insertIntoArray a b is) lx ly
    resizeCurr childOp xs@(x: _) ys =
      map childOp $ take (length xs) $ ys ++ (repeat ∘ zero) x
      where
        zero (LvArr l@(x: _)) = LvArr (replicate (length l) (zero x))
        zero (LvDBL _)         = LvDBL 0.0
        zero (LvI32 _)         = LvI32 0
        zero (LvSTR _)         = LvSTR ""
        zero (LvBool _)        = LvBool False
        zero (LvCluster c)     = LvCluster (map zero c)
        zero (LvArr [ ])       = LvArr [ ]
    resizeLower (x: _) ys = map (childResizer x) ys
    resizeAll xs@(x: _) ys = resizeCurr (childResizer x) xs ys
    childResizer (LvArr x) = λ(LvArr a) → LvArr (resizeAll x a)
    childResizer _ = id
    ndims (LvArr (v: _))  = 1 + ndims v
    ndims (LvArr [ ])     = 1
    ndims _               = 0
```

#### 6.2.5.3 Random Number

`Random Number` is an example of an impure function which produces a side-effect beyond the value sent through its output port. In our definition of the "outside world", which is part of the ongoing state computed in our model, we have the state of the pseudo-random number generator, which needs to be updated each time this node produces a value.

In this interpreter, we implement the PRNG using the 32-bit variant of the Xorshift algorithm [Mar03].

```
applyFunction "RandomNumber" w [] =
  let
    mask = foldl1 (λv b → v .|. bit b) (0 : [0 .. 31])
    n0   = wPrng w
    n1   = (n0 `xor` (n0 `shiftL` 13)) .&. mask
    n2   = (n1 `xor` (n1 `shiftR` 17)) .&. mask
    n3   = (n2 `xor` (n2 `shiftL` 25)) .&. mask
    f    = abs $ (fromIntegral n3) / 2 ↑ 32
  in (w {wPrng = n3}, LvReturn [LvDBL f])
```

#### 6.2.5.4 `Wait Until Next Ms`

Node `Wait Until Next Ms` demonstrates both the use a value coming from the outside world (the timestamp) and the use of a continuation. Its goal is to wait until the timestamp matches or exceeds the next multiple of the given argument. Using this object in loop structures that are running concurrently causes them to iterate in lockstep, if the inserted delay is long enough. This is a simple way to produce an acceptable level of synchronization for the typical domain of instrument data acquisition which LabVIEW specializes on.

When the function is applied, it immediately returns a continuation, containing the function *waitUntil* and the target timestamp *nextMs* as its argument. As we saw in Section 6.2.4.3, this will cause the function to be rescheduled. The implementation of *waitUntil* checks the current time received in the *LvWorld* argument: if it has not reached the target time, the function returns another continuation rescheduling itself; otherwise, it returns producing no value, since the function node for this operation has no output ports. This node relies on the fact that a (sub)graph as a whole keeps running as long as some node is scheduled.

```
applyFunction "WaitUntilNextMs" w [Just (LvI32 ms)] =
  (w, LvContinue $ LvKFunction waitUntil [LvI32 nextMs])
  where
    ts = wTs w
    nextMs = ts − (ts `mod` ms) + ms
    waitUntil w@(LvWorld now _) arg@[LvI32 stop]
      | now ⩾ stop = (w, LvReturn [])
      | otherwise   = (w, LvContinue $ LvKFunction waitUntil arg)
applyFunction "WaitUntilNextMs" vst [Just (LvDBL msd)] =
  applyFunction "WaitUntilNextMs" vst [Just (LvI32 (floor msd))]
```

Finally, we finish the definition of *applyFunction* by delegating the remaining functions to *applyPureFunction*.

```
applyFunction n w a = (withWorld ∘ applyPureFunction) n w a
```

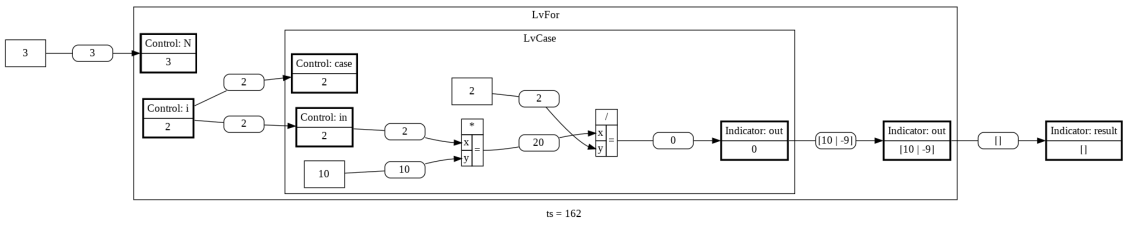


Figure 6.7: An animation frame produced by converting the output of the interpreter.

### 6.2.6 Demonstration

In Appendix C, we demonstrate the execution of our interpreter through a few example programs that showcase LabVIEW's various control structures. We developed a rendering pipeline to produce a visualization of the execution. Running the interpreter prints in its output the *LvVI* data structure and a series of *LvState* data structures, one for each execution step of the main graph. We then parse this output using a Lua script to generate a series of `.dot` files, one for each step, describing the graph alongside the state values. We then convert each `.dot` file to an image file containing a diagram using GraphViz[3] and finally combine all frames into a video file using FFMPEG[4]. The resulting video contains an animation of the evaluation of the graph over time, with values moving across nodes. Figure 6.7 shows a sample frame from one of the animations produced. All materials are available at [5].

## 6.3 Discussion: Is LabVIEW end-user programming?

When we discuss end-user programming, what defines the concept is not a particular programming paradigm, set of constructs or UI style. End-user programming is about the fact that the person doing the programming is the one who will use the resulting program, and, as a secundary point, that they are not programmers by profession. That is indeed not always the case in LabVIEW, which is used by professional programmers who build, compile and deploy programs for other end-users.

However, two aspects warrant its presence in the discussion on end-user programming languages taking place in this work. First, although it is used as a traditional programming language by software professionals, LabVIEW is also heavily used by end-users in engineering and physics fields, and the design of the language is heavily informed by this fact. One might even argue that the language is better suited to small-scale rapid end-user programming than to large-scale software development. Second, it is a particularly interesting subject in the design space of end-user programming because it is developed as a programming environment for writing data acquisition programs rather than a programmable data acquisition application. This

[3]`http://graphviz.org`
[4]`http://ffmpeg.org`
[5]`http://hisham.hm/thesis/`

has effects in the design of the resulting language. A good way to observe these effects is by contrasting LabVIEW and Pure Data.

### 6.3.1 LabVIEW and Pure Data compared

While both LabVIEW and Pure Data are visual dataflow languages that present programs as graph diagrams connecting node objects, the LabVIEW environment is a lot more similar to that of a typical programming language IDE. Beyond the visual presentation, a fundamental difference is that in LabVIEW the block diagram with the graph and front panel with the UI widgets are the "source code" of the program, which can then be executed, presenting the UI windows which are the "program" to be used. In Pure Data, there is no such distinction: the graph is the "document", which is edited in "edit mode" as the programming takes place, and which is later manipulated in "run mode" as the music is produced. This means both applications have two modes of operation—in LabVIEW, execution is toggled with the "play" and "stop" buttons of the UI. However, in Pure Data there is no distinction between what is being edited during creation and the end result; not only the interface is the same: most importantly, the DSP engine remains running while in "edit mode", so a musician can transition between these two modes during a performance. Recall that even while in "edit mode" the dataflow program is still running.

In Pure Data's "run mode", interaction happens via manipulation of values directly in the graph nodes or by clicking nodes to trigger messages: the program structure is transparent. While it is possible to hide the graph structure in Pure Data through the use of subprograms and indirect messages, the environment does not lead the user in this direction; it is more natural to present the graph, and it helps understanding the effect of editing values. This visibility is common practice in the field: some hardware synthesizers even include in their chassis drawings of their high-level audio flow diagrams, to help the musician make sense of how the various buttons relate to each other in the overall synthesis.

When running a LabVIEW program, there is no way to affect the program itself by interacting with the graph during execution. The only form of interaction allowed is via the front panel or other attached inputs (such as hardware instruments). When a LabVIEW program is compiled and deployed, the graph is not even visible, let alone editable by the user. A deployed LabVIEW program, therefore, is not itself end-user programmable.

In Pure Data, the user of the program and the developer are often the same: a computer musician programming audio synthesis and sequencing. Even when that is not the case—and there are communities where Pure Data programs are shared—the explicit nature of the graph structure invites users to tweak the patches to their liking, producing their own sounds.

This would lead us to conclude that Pure Data has a stronger focus on end-users, and by catering also to a professional audience, LabVIEW would be more difficult for newcomers. However, by including features common to typical professional programming environments, such as a clearer distinction between types and error messages targeting at the specific points of failure, LabVIEW makes it actually easier to understand problems in the dataflow graph than Pure Data.

Language features such as (auto-generated) feedback nodes make it easier to understand and debug cycle constructs; data coercion nodes (also auto-generated) make explicit any precision loss—a problem that also affects musicians using Pure Data, where it is perceived as degraded audio quality.

Both Pure Data and LabVIEW feature multiple types, including numbers, strings and table objects holding aggregated data. LabVIEW has a richer set of types, and is statically typed; Pure Data has a simpler set and is dynamically typed. Both of them make a visual distinction among edge types in the graph: Pure Data displays audio connections as thicker lines and message connections as thinner lines; LabVIEW uses colors, thickness and stripe patterns to indicate the various data types it supports. It is easy to make an invalid connection in Pure Data, for example connecting a string outlet to a float inlet, which will cause a runtime error being logged. In LabVIEW, the mismatch is caught as the user tries to make the connection; if the data is coercible, a conversion node is automatically inserted.

Through a combination of language and environment features, LabVIEW happens to be an easier language to program for, even though engineers typically have more formal training in programming-related fields than musicians. Still, a large community of musicians thrives using software such as Pure Data (and its proprietary relative Max/MSP) even without the facilities that professional programmers have grown used to. This shows us that the abilities of end-users should not be underestimated, and invites us to consider how much those end-users could benefit if the languages they work on incorporated more from established programming language design practices.

# Chapter 7

# Some other languages

With our extensive discussion of Pure Data (Chapter 4), spreadsheets (Chapter 5) and LabVIEW (Chapter 6), we covered a wide range of the space of design alternatives discussed in Chapter 3. In this chapter, we extend our panorama of dataflow languages through overviews of three more applications, which can now be presented through the frame of reference of the languages presented earlier:

- Reaktor [Nat15] - a music application for constructing modular synthesizers;
- VEE [Agi11] - an engineering application for data acquisition and test-and-measurement;
- Blender [Ble17] - a 3D computer graphics software.

## 7.1 Reaktor

Reaktor (depicted in Figure 7.1) is a commercial application by Native Instruments for constructing synthesizers, which can be used either stand-alone or as a plugin for different music applications. Its interface has distinct Edit and Play modes. It features separate Panel Layouts with the input and output widgets for interaction and a Structure View with the dataflow graph for editing. Reaktor supports abstracting sub-graphs into units. A program is presented as an “instrument”, which acts as a synthesizer or an effects unit, often with a front-end interface that mimics equivalent hardware.

Reaktor has two dataflow environments with distinct languages, called Primary and Core. Instruments are created using high-level operators called “modules” which are combined into “macros” using a dataflow language in Primary mode. The modules themselves are also written as graphs composed of “core cells”, in a distinct environment called Core mode.

Like Pure Data, Reaktor’s Primary mode has two types of wires: audio and event wires. Further, modules may be monophonic or polyphonic. Connecting a polyphonic module to a monophonic module flags a type error, marking the wire in red. A “voice combiner” module can be used to mix down a polyphonic signal into a monophonic one. Only one wire can be connected to a port, but some modules such as the Multiply operation are variadic, allowing more input ports to be created

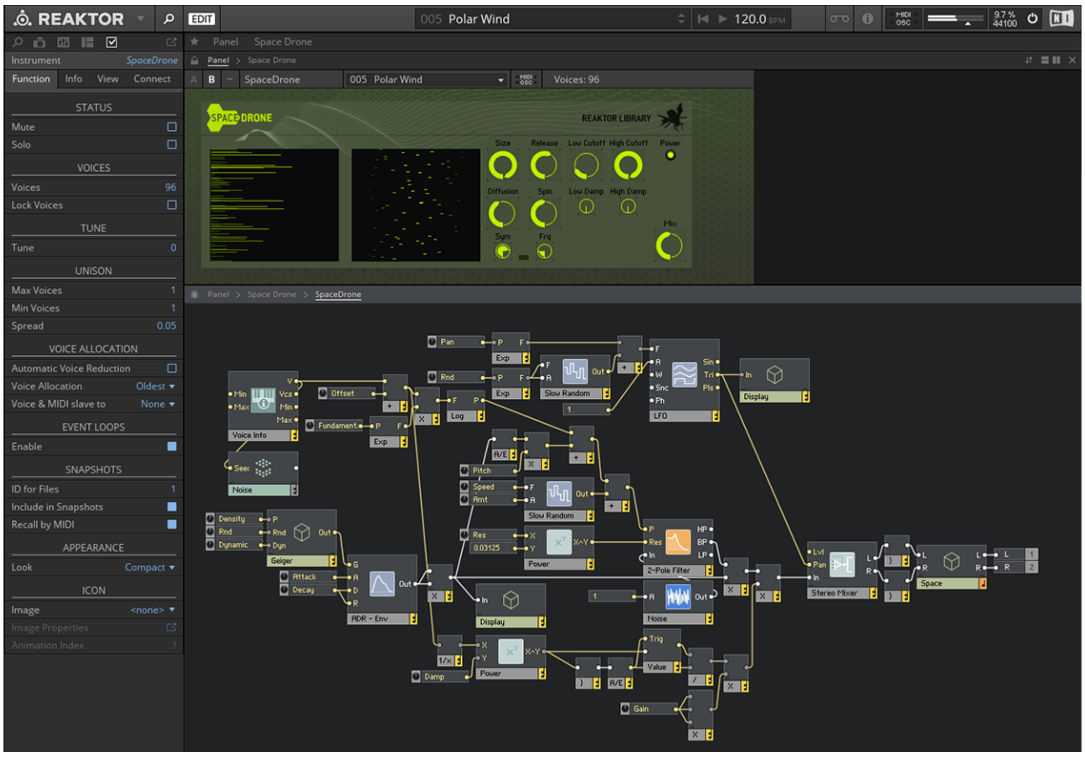


Figure 7.1: A screenshot of Reaktor. Source: `https://www.native-instruments.com`

as needed. The order through which audio is processed is deterministic, and the UI has a "Module Sorting Debug Tool" option for displaying the ordering explicitly, deemed in the manual as "crucial" in case of structures with feedback [Nat15].

When advanced users peek into the implementation of Primary modules, they enter Core mode. While also a visual dataflow language, Core mode uses lower-level data types, such as scalars of types `float` and `int` (including low-level concerns such as undefined behavior in type conversions and denormals in IEEE 754 floating-point), arrays and custom structured types called "bundles" (akin to LabVIEW's "clusters"). Memory storage in the style of shift registers are represented as special wire types. While at first glance Core seems an extension of Primary, they are fundamentally distinct languages. For instance, their semantics for event propagation are different. In Primary, an event propagated for more than one output is forwarded to each destination in turn. In Core, replicated events arrive logically simultaneously, so if one output is plugged into two inputs of the same cell, a single firing event is produced (whereas in Primary, this would produce two events).

Reaktor notably lacks a textual scripting mode. This absence is noted even by the music practitioner community: a major music industry magazine, Sound On Sound, states in its review of the latest version of Reaktor that "integration of a scripting language would help serious developers who sometimes feel the restrictions of a purely visual-based approach"[1]. Still, it is interesting to note that the application also uses a three-tier architecture, with a higher-level language in a central role and a

[1]`http://www.soundonsound.com/reviews/native-instruments-reaktor-6`

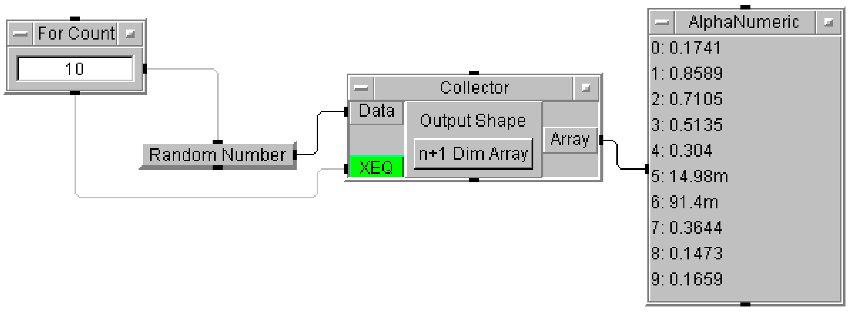


Figure 7.2: Mix of data and control flows in VEE.

lower-level language in a peripheral role, on top of the built-in application facilities.

## 7.2 VEE

VEE (Visual Engineering Environment) is a tool for development of engineering test-and-measurement software, based around a visual programming language. It was originally released by Hewlett-Packard in 1991, and is currently marketed by Keysight Technologies [Agi11].

VEE supports integration with other languages for scripting. MATLAB integration support is included. Additionally, VBA scripting and an Excel integration library are also supported on Windows.

There are several data types, including integer, real, complex, waveform, enum, text, record and multidimensional arrays. Connections, however, are not typed. Most nodes perform automatic type conversions. VEE also supports named variables, which can be used via Set Variable and Get Variable nodes, allowing for indirect connections in the graph.

There are five kinds of ports for connecting wires (called *pins* in VEE): data pins, which carry data; sequence pins, designed only for affecting the firing sequence; execute pins, which force an immediate execution when fired; control pins, which affect the internal state of a node but do not cause data propagation; and error pins, for error handling. Sequence pins are prominent in VEE and break the pure dataflow model. In a comparison between LabVIEW and VEE in an engineering magazine, it is said that "LabVIEW follows the data-flow paradigm of programming rigorously. HP VEE is loosely based on data-flow diagrams with a large measure of flow-charting sprinkled with a little decision table." [BHHW99]. Connected subgraphs of a disjoint graph can be launched separately, by connecting separate "Start" buttons to the sequence pins of input nodes of each subgraph. Data pins can connect to sequence pins; for example, an `If-Then-Else` node can be used either in the dataflow way, outputting a value according to a condition, or in the flowchart way, triggering a subgraph. The `For Count` iteration node is often used in conjunction with sequence and execute pins. Figure 7.2 illustrates the mix of data and control flows in VEE. Ports at the top and bottom of nodes are sequence pins; the green port is an execute pin; other pins at the left and right are data pins. This graph has two dataflow connections (black wires) and two control flow connections (gray wires). The `For`

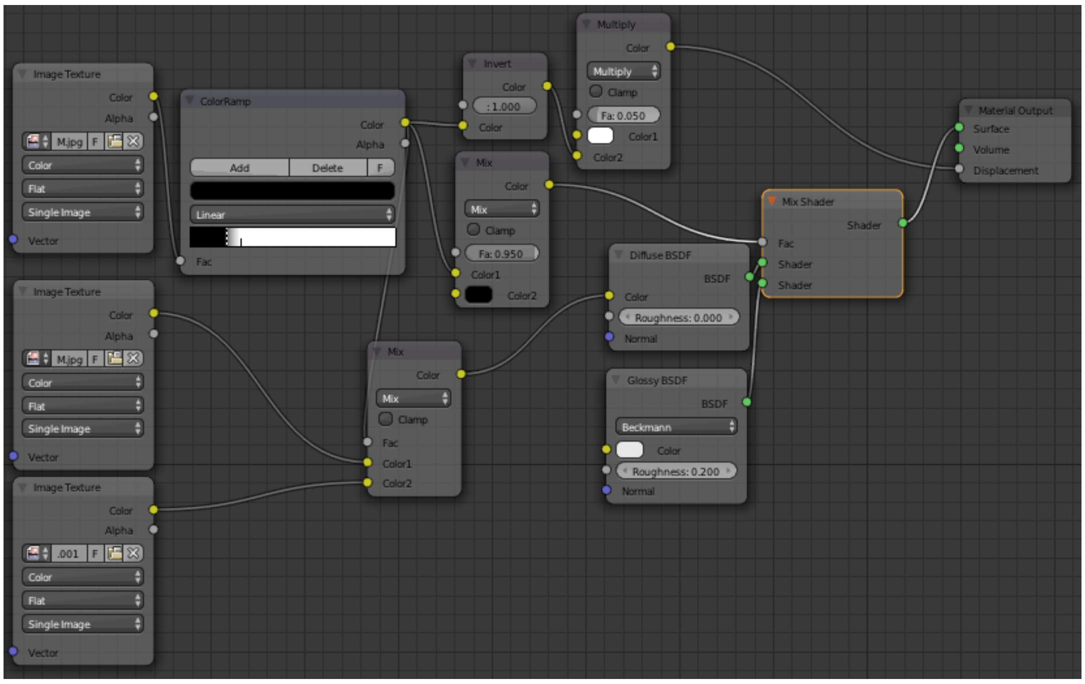


Figure 7.3: The node editor for creating materials in Blender. Source: `https://www.blender.org`

`Count` output data pin triggers the sequence input pin of `RandomNumber` ten times; at the end of the count, the sequence output pin of `For Count` triggers the execute pin of `Collector`, which then outputs the array to `AlphaNumeric`. The sequence output pin of a node fires after the evaluation of all subgraphs connected to that node's data outputs is complete, which may include triggering sequence pins in nested subgraphs. A description of the intricate evaluation semantics of sequence pins in VEE is given in [GJ98].

Through the use of various kinds of pins, VEE gives a fine grained control of (and responsibility for) the execution flow to the user. The user needs to be aware that different connectors of a node have different effects in the evaluation logic. To an extent, this is similar to the situation in Pure Data, where hot and cold inlets also have different triggering rules.

There are two kinds of subprogram abstractions: UserObjects and UserFunctions. A UserObject is merely a collapsable subgraph. To reuse it, the object needs to be cloned, producing a new subgraph instance which can be edited separately from the original copy. A UserFunction is a reusable subgraph, for which all uses are references to the same instance.

## 7.3 Blender

Blender is an open source 3D graphics software that is used for professional animation, video games, effects, modeling and art. It has a large number of features, several of which are programmable. Blender includes a dataflow language, called in its documentation simply "node language", that is used in different parts of the application. The documentation also mixes the presentation of the language, explaining concepts such as sockets and properties, with the presentation of the UI, explaining menus and keyboard shortcuts.

The node editor is used as the UI for creating materials in the render engine, for compositing and for texture editing. Figure 7.3 shows a screenshot of the materials editor. Each of these activities has its own set of nodes available to it. A restricted form of the node language called the "Logic Editor" is also available for integrating 3D objects with game scripting code. The remainder of this section will focus on the more general Node Editors available for the other tasks.

Programs in the Blender node language are directed acyclic graphs. It has four data types: color, number, vector and shader. Ports (called *sockets*) have different colors according to their data types. Nodes have *properties*, which are additional input arguments that are "inlined" in the node (that is, it is not necessary to create numeric nodes with constants and connect them to a node to parameterize these arguments). For example, in Figure 7.3 node `Mix` has two color inputs (indicated by yellow sockets); `Color1` is connected to the previous node `ColorRamp`, and `Color2` is set to black internally via a property.

Node Groups are collection of nodes that can be reused within a file and throughout different files, and the documentation describes them as "similar to functions in programming". Recursion is not allowed. Node groups can have particular inputs and outputs from its internal nodes that are made available to the outside, in a similar manner to how tunnels work in LabVIEW. These tunnels are accessible when the node group is collapsed both as sockets and as properties. Oddly, when Node Groups made accessible to a different file (through the `File ▷ Append` menu, which appends the content of one file to another), node groups are called "Node Trees".

Blender is scriptable in Python, and the node editor is also available to the Python scripting layer, so plug-ins can create additional node editors for other activities. An example of the power of this combination is the popular Animation Nodes add-on [2], which adds a new graph-based system for controlling animations, allowing users to specify visually tasks that previously required Python code. The scripting interface, however, is not without its limitations. For instance, it allows manipulating a node tree in the material editor, but not creating new node types. The Cycles Render Engine supports creating new node types written in OSL (Open Shading Language), for implementing texture shaders. A user then used this shader language to create the equivalent of a Boolean if-then-else node, that was missing in the materials editor. Another user in the same forum thread where this node was announced mentioned this node was equivalent to the "Condition" node in competing product Maya.[3]

[2]`https://github.com/JacquesLucke/animation_nodes`

[3]`https://blenderartists.org/forum/showthread.php?354949-Cycles-Boolean-Node-(Not-Shader)`

## 7.4 Discussion: Dataflow end-user programming, then and now

Dataflow end-user programming has come a long way since the early days of the paradigm. Twenty-five years ago, most user-centric languages based on dataflow were developed in a research context [Hil92]. Now, we were able to concentrate our research exclusively in production languages that are in widespread use, and we were able to select representative examples from a wider pool of options. When looking at the world of contemporary dataflow end-user applications, it is clear that the paradigm has established itself in certain fields:

- Spreadsheets: Excel, LibreOffice, Google Sheets
- Audio synthesis: Max/MSP, Pure Data, Reaktor, vvvv[4], AudioMulch[5]
- Video compositing: Nuke[6], Natron[7]
- 3D modeling: Maya[8], Blender, Grasshopper[9]
- Engineering: LabVIEW, VEE, Expecco[10], DASYLab[11]

One might speculate the reasons for the model's success in particular areas. In some cases, it seems that a trailblazing application paved the way for the emergence of similar applications (VisiCalc for spreadsheets, LabVIEW for engineering). In other cases, visual languages seem to be a natural fit for professionals of certain areas, as is the case with multimedia (video, 3D, music). It is also worth pointing out that, looking at historical dataflow applications, their areas also tend to be similar: music, image processing, graphics.

It is worth considering whether the paradigm could have a wider range of application. Given that spreadsheets are used by professionals of all sorts of fields, it seems that the computational model is accessible to a wide range of users. In a certain sense, a spreadsheet is a general-purpose programming language for numeric applications. Domain-specific applications could explore the familiarity of this model to provide functionality tailored to different areas in a programmable environment.

[4]`https://vvvv.org/`
[5]`http://www.audiomulch.com/`
[6]`https://www.thefoundry.co.uk/products/nuke/`
[7]`http://natron.fr/`
[8]`http://www.autodesk.com/products/maya/overview`
[9]`http://www.grasshopper3d.com/`
[10]`https://www.exept.de/en/products/expecco`
[11]`http://www.mccdaq.com/DASYLab-index.aspx`

# Chapter 8

# Design alternatives critiqued

In this chapter, we revisit the classifications of design alternatives for dataflow end-user programmable languages presented in Chapter 3, illuminated by the study of contemporary languages of this kind presented throughout this work, and in particular the in-depth case studies of Chapters 4 to 6. We present here a critique of these various design choices and the impact of their inherent trade-offs. As we consider each of the design dimensions proposed in the survey by Hils [Hil92] and the additional ones proposed in this work, our aim is to discuss which choices have shown to be especially appropriate or inappropriate for different scenarios. We consider especially appropriate those choices that contribute to the usefulness of the application within its domain. We consider inappropriate design choices those that are related to the various pitfalls and shortcomings that we identified in the design of the various languages studied and presented in the previous chapters.

We begin by presenting in Table 8.1 an update to the table presented in [Hil92], applying the list of design alternatives from that work (discussed in Section 3.1) to a different set of contemporary dataflow languages, while also extending it with the additional design dimensions as presented in Section 3.2. The only language from that survey that is also in our list is LabVIEW [Nat01]. It is worth noting that in Hils's larger set of 15 visual dataflow programming languages the majority of them were academic projects, many of them short-lived. In our shorter list, we restricted ourselves to successful languages (both proprietary and open-source) with a proven track in terms of longevity and user base.

The following sections discuss these design dimensions organized logically into four groups. Being the focus of this work, the first three sections discuss semantic aspects, namely graph evaluation, language features and type checking; the fourth section groups the remaining aspects. Unless otherwise stated, all remarks about Excel below apply to all spreadsheets discussed in Chapter 5.

## 8.1 Graph evaluation

In this section, we discuss design aspects that refer to the evaluation of the dataflow graph as a whole. In other words, we will discuss aspects which affect the design of the graph evaluator's main loop and not that of specific nodes. This way, we present the discussion of semantic aspects in a top-down fashion, the same way we

| *General information* | **Pure Data** | **Excel** | **LabVIEW** | **Reaktor** | **VEE** | **Blender** |
|---|---|---|---|---|---|---|
| **Main reference** | [P$^+$15] | | [Nat01] | [Nat15] | [Agi11] | [Ble17] |
| **Licensing** | 3-clause BSD | Proprietary | Proprietary | Proprietary | Proprietary | GNU GPL v2+ |
| **Initial release** | 1996 | 1985 | 1986 | 1999 | 1991 | 1995 |
| **Latest release** | 2016 | 2016 | 2016 | 2015 | 2013 | 2017 |
| **Application domain** | Music | Office | Engineering | Music | Engineering | 3D graphics |
| *Design alternatives* [Hil92] | **Pure Data** | **Excel** | **LabVIEW** | **Reaktor** | **VEE** | **Blender** |
| **Box-line representation** | Yes | No | Yes | Yes | Yes | Yes |
| **Iteration** | Yes (cycles) | Limited | Yes (construct) | Limited | Yes | No |
| **Subprogram abstraction** | Yes | No | Yes | Yes | Yes | Yes |
| **Selector/distributor** | Yes | Yes | Yes | Yes | Yes | Yes |
| **Flow of data** | Uni | Uni | Uni | Uni | Uni | Uni |
| **Sequence construct** | No | No | Yes | No | Yes | No |
| **Type checking** | Limited | No | Yes | Yes | No | Yes |
| **Higher-order functions** | No | No | No | No | No | No |
| **Execution mode** | Data-driven | Demand-driven | Data-driven | Demand-driven | Data-driven | Data-driven |
| **Liveness level** [Tan90] | 2 | 3 | 2 | 2 | 2 | 3 |
| *Additional design alternatives* | **Pure Data** | **Excel** | **LabVIEW** | **Reaktor** | **VEE** | **Blender** |
| **Dataflow model** | Dynamic | Static | Static | Static | Static | Static |
| **N-to-1 inputs** | Yes | No | No | No | No | No |
| **Separate edit/use views** | No | No | Yes | Yes | Yes | No |
| **Time-dependent firing** | Yes | No | Yes | Yes | Yes | No |
| **Rate-based evaluation** | Synchronous | No | No | Synchronous | No | No |
| **Indirect connections** | Yes | Yes | Yes | Yes | Yes | No |
| **Dynamic connections** | Yes | Yes | Yes | No | No | No |
| **Textual sub-language** | Imperative | Functional | Imperative | No | Imperative | No |
| **Scripting** | Python, Lua | VBA | MATLAB | Reaktor Core | MATLAB | OSL, Python |

Table 8.1: A comparison of contemporary dataflow UI-level languages

presented the interpreters in previous chapters.

### 8.1.1 Static and dynamic dataflow models

When creating a dataflow-based system, the first design decision to take with regard to the language is which dataflow model to use; in other words, which criteria will be used for firing nodes. Given that the essence of dataflow is purely functional, any order of evaluation (and thus any sequence of firings) should produce identical results. However, since real-world programs are usually not purely functional and include visible effects, the underlying dataflow model can become apparent to the end-user: for example, in a static model, a fast operation can be held back by a slow operation further ahead in the pipeline, due to the lack of queueing. Dynamic dataflow models avoid these bottlenecks, but their more complicated models present different trade-offs [Den85, KSP15], some of which become apparent in the resulting language. Of the languages compared in this work, only Pure Data employs a dynamic model. A side-effect of this model is that ordering issues arise. As discussed in Section 4.3, we believe these issues were not addressed appropriately.

Another side of this trade-off which favors static and synchronous dataflow models is that understanding and debugging a dataflow graph is easier when there is a single token per arc (or a fixed number of values, as in the case of synchronous dataflow), and only one iteration of a loop is running at a time. For end-user programming, understandability is more important than parallel efficiency, so it is our view that end-user dataflow languages should present a static dataflow view of program execution. When the language contains explicit looping constructs, it should be possible to achieve a dynamic flow of execution in certain loops as a user-transparent optimization, if the contents of a loop are known to be purely functional, for example. Such an optimization could then be automatically disabled when the user is probing the flow of tokens for debugging purposes, restoring a one-token-per-wire view.

### 8.1.2 Data-driven and demand-driven dataflow

In the classic literature on the dataflow paradigm, the data-driven and demand-driven models are presented as equally proeminent, complementary approaches [TBH82]. Yet in the context of end-user programming, we identify a tendency towards data-driven execution, considering not only the six languages selected here, but also the ones that were preliminarily studied to perform the selection.

We believe this is understandable because the data-driven approach more directly maps to the mental model one has about the evaluation of a graph, with the order of execution matching the way the data flows from input nodes toward output nodes.

Only two languages in this study employ demand-driven evaluation: Excel and Reaktor. Demand-driven evaluation is natural in a spreadsheet because cells are written as textual expressions, which translate to an expression tree that is evaluated top-down, that is, starting from the output node. The fact that Excel is demand-driven is mostly transparent, because of its static dataflow model, its lack of time-dependent firing and purely functional nature (with no imperative textual sub-language). Beyond the basic intuition about expression trees, one way to verify that a spreadsheet is indeed demand-driven is by forcing side-effects via the scripting layer of the application, and confirming that these only happens when cells are scrolled into view. We confirmed this successfully in all spreadsheets analyzed[1].

The case of Reaktor is a good illustration of how demand-driven execution can be a poor choice for end-user applications. Although not stated explicitly in its documentation and indistinguishable in most cases, the evaluation of Reaktor's Primary mode is demand-driven. This can be inferred from the fact that graphs that do not connect to an audio output need to have their terminal nodes marked as "always active" in order to trigger continuous evaluation. In one tutorial from the vendor, the documentation instructs to add a dummy "lamp" output marked as always-on just to achieve this same effect [2].

### 8.1.3 Uni and bi-directional dataflow

All languages studied in this work employ the traditional uni-directional style of dataflow. In Hils's work, only one language featured bi-directional flow of data: Fabrik [IWC+88], a language for designing user interfaces.

Bi-directional constraint systems have been present in research for end-user development systems since the early days of Sketchpad [Sut63] and the paradigm continues to be researched for this day [Sch17], but what we observe in industry practice is that the simpler uni-directional model has become established as the norm in dataflow. GUI construction systems (notably the same domain as Fabrik) have taken up bi-directional constraint systems [Jam14], with Cassowary [BBS01] being integrated into Apple's standard GUI libraries, but those are used as an internal component and not as end-user programming languages of their own.

### 8.1.4 N-to-1 inputs

Pure Data is the only language in our study that supports N-to-1 inputs, that is, multiple wires connecting to a single input port in a node. The way it handles these inputs is deeply linked to its dataflow evaluation model. For audio wires, which use a static and synchronous model, the incoming data is merged using additive synthesis (that is, the input argument for the port is a buffer where each sample value is the sum of the sample values at the corresponding positions from the incoming wire buffers). For message wires, which use a dynamic model, it queues inputs, causing multiple incoming inputs arriving to a single port through different wires to fire the node multiple times. Queueing also leads to concerns with ordering, as discussed in Section 4.3.

It seems clear that disallowing N-to-1 inputs leads to a simpler conceptual model and less suprising behavior. Not all domains have an obvious choice on what to do when merging inputs (and even in Pure Data's domain of audio processing, some

[1] The JavaScript-based scripting layer of Google Sheets tries to prevent side-effects (we could not make it pop up a message box as we did in the other applications), but we still verified the demand-driven execution by writing a computation-intensive script that caused a noticeable delay.

[2] `https://support.native-instruments.com/hc/en-us/articles/209588249-How-to-Use-an-Event-Table-as-Copy-Buffer-in-REAKTOR-5`

synthesizers use subtractive synthesis, for example) so having an explicit merge node is a clearer why of presenting what is happening with the data. The convenience that N-to-1 inputs bring could be obtained by automatically inserting those nodes when multiple wires are plugged to an input port, similarly to how LabVIEW auto-inserts feedback nodes.

### 8.1.5 Timing considerations

When considering issues of timing, let us look at both time-dependent firing and rate-based evaluation at once. We see in Table 8.1 three distinct patterns. We have two languages that support time-dependent firing and use a synchronous dataflow model for rate-based evaluation: Pure Data and Reaktor; two languages that support time-dependent firing but do not feature rate-based evaluation (that is, using a purely static model): LabVIEW and VEE; and two languages that do not have either: Excel and Blender.

The approach of these languages with regard to timing is linked to their domains and to the kind of data and activity they perform. Pure Data and Reaktor, both of them music applications, operate on audio streams. Processing digital audio in real-time requires rate-based evaluation, and music creation demands the ability to specify transformations of data based on time. LabVIEW and VEE, engineering applications, require supporting activities such as periodic reading and writing of data, but since they don't have a single domain-specific target for these processing rates which can be made fully implicit in the evaluation loop, as it happens, for example, with music applications do, processing buffers at standard rates such as "48000 Hz 24-bit stereo" ($48000 \times 3 \times 2 = 288000$ bytes per second). Since sampling rates of various data acquisition instruments supported by those engineering tools vary, the user needs to essentially construct the rate-processing loop by hand, using arrays and delay objects. Both LabVIEW and VEE define "waveform" types as abstractions to help in this task, but those are no more than a "typedef" of a record type, storing an array and a timestamp.

Languages whose domains do not deal with time avoid time-based evaluation features entirely: Excel and Blender have no support them (except for the occasional Excel function like `TODAY()`, but that is clearly not integrated with the language's evaluation model—for one, the cell value does not update automatically as time passes).

Another issue related to timing that is often a concern in language design is synchronization. In end-user applications, timing constraints depend on the domain. In the field of music, for example, timing precision matters up to the scale of human-perceptible audio latency, which is in the order of a few hundredths of seconds. This means that synchronization can be often satisfactorily approximated via real-time clock events. As we saw in the case of LabVIEW, timing primitives with millisecond precision are used to cause iterations of parallel loops to proceed in tandem. When a language combines two evaluation models, as is the case of Pure Data, which uses dynamic dataflow for messages and synchronous dataflow for audio, it is also important to avoid synchronization issues. Pure Data solves this adequately by alternating message and audio evaluation while using an audio buffer to give the evaluation of the message cascades enough time to run. This can still lead to audio drop-outs if message processing is excessive, but musicians nowadays are used to the notion that heavy computations can make audio stutter, known in the community jargon as "audio dropouts", and adapt accordingly.

### 8.1.6 Indirect and dynamic connections

As discussed in Section 3.2.5, while the presence of indirect connections is a syntactic feature, the occurrence of connections determined at runtime has semantic consequences. Dynamic connections make it impossible to determine a static schedule for node evaluation in advance (as is done for audio nodes in Pure Data, for example), to optimize in-place replacement of buffers to avoid array copying (since any intermediate node may be fired at any time) and to reliably detect loops in advance.

Pure Data, Excel and LabVIEW support dynamic connections. As should be no surprise by now, Pure Data supports dynamic connections only for message data, not audio. An indirect connection, dynamic or not, consists of two nodes, a sender and a receiver. In Pure Data, only the sender node can have its target dynamically defined; the identifier of a receiver node cannot be changed at runtime.

Of the three languages that support dynamic connections, Pure Data presents them as a basic feature of the language; Excel and LabVIEW treat them as advanced features, in the form of Excel's `INDIRECT` function and LabVIEW's object reference system. This may be related to the fact that, given that Pure Data already employs a dynamic dataflow model, these connections behave like any other ones. In static dataflow systems like Excel and LabVIEW, the use of dynamic connections can cause issues. It is easy to find on the internet examples of user problems with dynamic connections in both Excel[3] and LabVIEW[4].

## 8.2 Language features

We now move to the second part of the discussion on semantic aspects. Here, we discuss the design of specific nodes and features that may or may not be present in a dataflow language.

[3]"No #REF! error, the cell just doesnt update with the new value (just stays exactly the same), even though the reference is correct and the referenced cell is obviously updated" "I actually tried rebooting, didnt help." `https://www.wallstreetoasis.com/forums/excel-help-cells-do-not-update-when-they-reference-another-excel-file`

[4]Typing "*labview object reference*" in Google auto-suggests "*labview object reference is invalid*". In one of the results, the LabVIEW knowledge base reports "This is a documented known issue that occurs in LabVIEW Real-Time versions 2014 and 2015. After making a modification to the VI, Error 1055 is thrown from any property node attempting to access the dynamic refnum. In order to resolve this error, close and re-open the VI." (`http://digital.ni.com/public.nsf/allkb/2E848F065A18570986257F3800708328`).

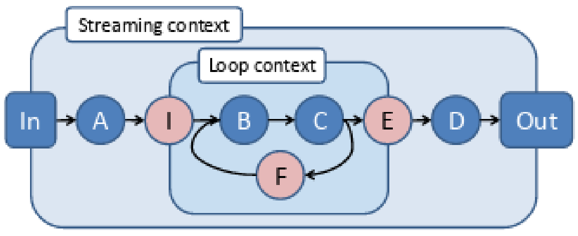


Figure 8.1: Loop contexts in Naiad [MMI+13], featuring a very similar structure to that of Show and Tell and LabVIEW

### 8.2.1 Selector/distributor

Selector and distributor nodes are the most basic features in a dataflow language. Unsurprisingly, all languages studied here implement them and there is not much design variation among them. For more complex data types, such as images in Blender and audio in Pure Data and Reaktor, a selector $\sigma(k, v_1, v_2)$ can work like a "mixer" node, in which $k$ is a blend value between 0 and 1 instead of a boolean.

### 8.2.2 Iteration

In stark contrast with selectors and distributors, iteration constructs are the ones that show the greatest amount of variation in design among dataflow languages, leading even to a survey specifically about it [MP00]. And indeed, no two languages among those presented in this work implement iteration the same way. We consider here iteration in a broad sense of the word, defining it as any language feature that allows evaluating a subgraph a number of times.

Pure Data, being a dynamic dataflow language, allows for cycles, and this is a simple way of producing iteration. Evidently, cycles are only allowed in the dynamic part of the language, that is, between message-handling nodes. Cycles between audio nodes are detected and rejected as soon as the DSP engine is activated.

In Excel, a array formulas may be considered a limited form of iteration, since they allow evaluating multiple times for a given range of values a single expression tree. Some array formula patterns combining common functions were turned into predefined functions which always evaluate their arguments in an array context, like `SUMPRODUCT` and `SUMIF`[5].

LabVIEW features structured constructs for looping, based on the idea of a frame around a subgraph, with values flowing between iterations through shift registers. The design of this construct is essentially the same as that of Hierarchical Data Flow introduced in Show and Tell [Kim85, KCM86]. More recently, this model of structured loops in static dataflow graphs has been reinvented in Naiad [MMI+13], a modern distributed system for Big Data processing, which advertises static dataflow with structured loops as one of its main features. Figure 8.1, from [MMI+13],

[5]Excel users invented an idiom to produce the equivalent of the missing function `SUMPRODUCTIF` without resorting to array formulas: the expression `SUMPRODUCT(--(C1:C10="flag"),A1:A10,B1:B10))` uses double negation to coerce boolean values into 0 or 1, annulling elements of the product when the condition does not match.

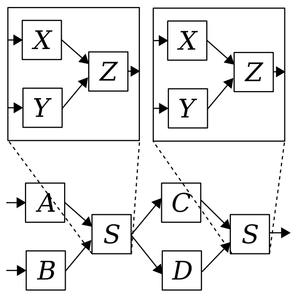


Figure 8.2: A graph containing two occurrences of a subgraph $S$

illustrates the similarities: nodes $I$ and $E$ work like LabVIEW tunnels, and node $F$ works like a shift register, with the difference that Naiad employs a tagged-token system to allow for parallel iterations.

Reaktor has no iteration structure per se, but it features a node called "Iteration" which acts as a counter, producing a series of values that can be used for firing other subgraphs and indexing values. VEE has a similar iteration node to Reaktor, but because of its support for sequence pins that dictate control flow, it is a more powerful construct for triggering arbitrary subgraphs. VEE also performs an implicit *map* operation when passing an array to a function that expects a scalar, similarly to Excel's array context. Finally, Blender notably lacks an iteration construct.

### 8.2.3 Sequence construct

A sequence construct is a way to specify that one subgraph should execute after another, without having a data dependency between them (akin to sequencing two statements $s_1; s_2$ in textual programming languages). As such, it is a fundamentally imperative construct.

The only languages from our survey to include explicit sequencing constructs are LabVIEW and VEE. Given that they are both engineering applications and are the only ones to feature constructs clearly named after their textual-language counterparts such as “for” and “while”, we speculate that explicit sequencing was added to ease the transition from users who had some previous programming experience.

It is interesting to note that all the other applications do not have sequencing constructs. This seems to indicate that for domain specialists without preconceived notions about programming, imperative constructs are not a necessity and declarative programming can be used successfully.

### 8.2.4 Subprogram abstractions

The ability to abstract away subprograms is a commonplace feature in modern programming languages. In the dataflow model, a subprogram abstraction means replacing a subgraph with a node that represents it. In graphical languages, this feature becomes especially necessary to tame the visual clutter of the graph representation.

In end-user programming languages, and especially visual ones, abstractions present semantic complications that are usually missing in languages for professional programmers. A typical motivation for abstracting a subgraph is to reuse it. In the example of Figure 8.2, subgraph $S$ appears twice in the main diagram. Expanding both occurences of $S$, one would find the same subgraph with nodes $X$, $Y$ and $Z$. When copying a node representing an abstracted subgraph for reuse, end-users have different intuitions whether these two nodes are references to the same subgraph or if they are two separate copies that can be modified without affecting the other one. Translating to the world of textual languages, this is the question whether an abstraction behaves as a newly-declarated function used in two places, or if it is merely a visual (syntactic) abbreviation, akin to those achieved by code-folding text editors.

Moreover, when end-users perceive an abstraction as a single subgraph referenced in two different places, then there is the question if they perceive the subgraph as reentrant: in other words, whether they see the two invocations of the subgraph as fully independent executions (that is, like usual function calls where each invocation has its own activation record), or if they see the shared subgraph as a single entity in memory. In case of stateful nodes, this is especially relevant, because that determines if multiple invocations of the abstraction in the main graph affect each other or not (which would be equivalent, for example, to declaring all local variables in a C function as being `static` or not).

Note that these two issues of copying and reentrancy are related: copies naturally have no reentrancy problems. We have therefore three possible behaviors:

1. copying a subgraph produces a new, unrelated subgraph with identical contents;

2. copying a subgraph produces a new reference to the same subgraph, but each reference produces a new instance in memory at runtime;

3. copying a subgraph produces a new reference to the same subgraph, which has a single instance in memory at runtime.

Figure 8.3 illustrates the effect of editing the graph from Figure 8.2 under these different behaviors. Consider that the user expands the right-hand occurrence of $S$ and changes $Z$ to $W$. In the first scenario, editing the second occurrence of $S$ does not affect the first (Figure 8.3(a1)), and their executions will also be independent (Figure 8.3(a2)). In the second scenario, all references point to the same subgraph (Figure 8.3(b1)), but the execution of each instance is independent (Figure 8.3(b2)). In the third scenario, there is a single copy of $S$, both in the diagram (note that Figure 8.3(c1) is identical to Figure 8.3(b1)) and in memory (Figure 8.3(b1)), meaning that the execution is not re-entrant.

Seasoned programmers used to textual languages will expect the behavior of scenario 2, with a single representation of a function in the program and separate instances in memory as it executes, which is the best one in terms of code reuse and safe execution. Note, however, that of the three scenarios depicted in Figure 8.3, this is the only one where the visual presentation when editing the program does not represent the behavior in memory.

Different applications approach these issues in different ways. Pure Data has two ways of representing subprograms: "subpatches", which behave according to scenario

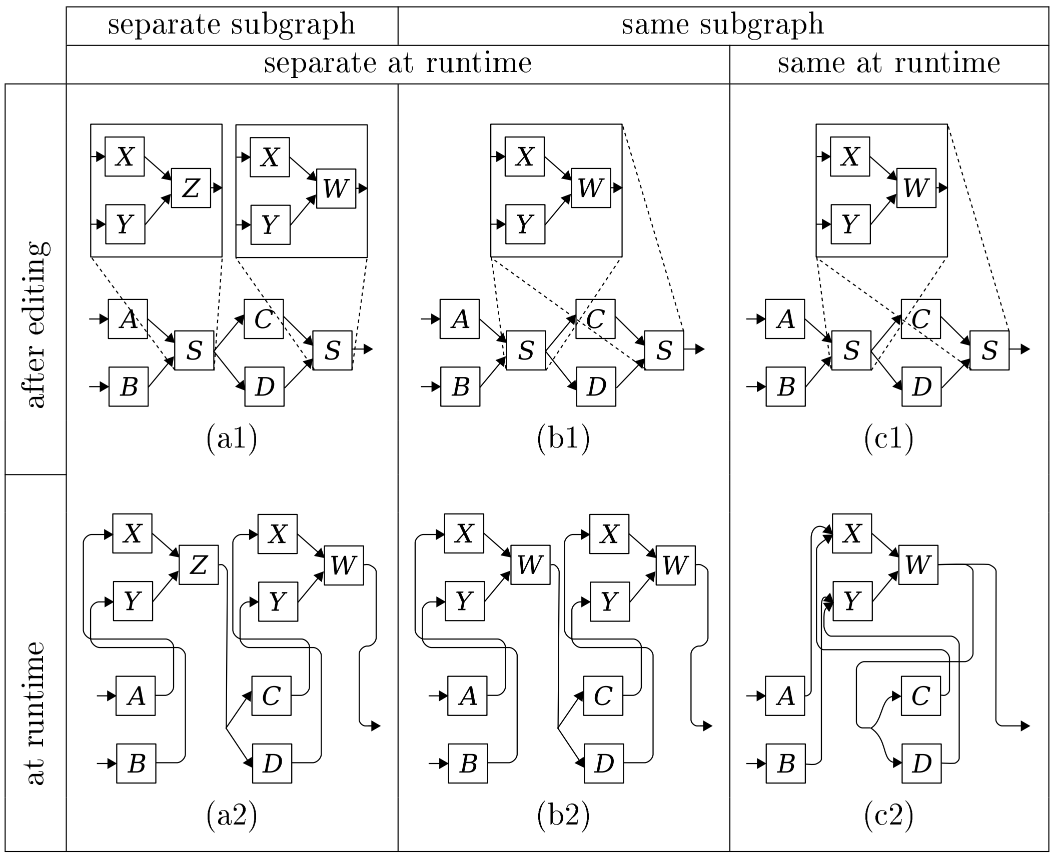


Figure 8.3: Different alternatives for the behavior of subprograms

1, and are stored as part of the same file as the main graph, and "abstractions", which are stored as separate graph files, and behave according to scenario 2. A problem arises, though, when saving abstractions. Pure Data persists the internal state of graphs when saving them, so when a patch contains multiple instances of an abstraction, the internal state of only one of them is saved. Users are advised to make the initialization of their abstractions stateless through the use of creation arguments, but they still look for workarounds to save their state.[6]

In LabVIEW, sub-VIs are not reentrant by default [Nat13]. There is a single instance in memory, as in scenario 3 above. Note that this leads effectively to a situation with N-to-1 wires leading to input ports, even though LabVIEW does not allow it otherwise. This breaks LabVIEW's static dataflow model and introduces queueing. The user can enable reentrancy, producing one instance in memory per reference, changing its behavior to scenario 2. Sub-VIs cannot be recursive. Each instance remains in memory even when not running, to save the state of shift registers and feedback nodes. There is also a third setting, in which LabVIEW creates a pool of instances as a way to reduce memory usage, but in this case sub-VIs become stateless.

A Node Group in Blender shows in its collapsed box a number that represents the number of "linked" instances. When a Node Group is copied, this number increments in all instances to denote that the same Node Group is being used in multiple places.

[6]`http://forum.pdpatchrepo.info/topic/8803/state-saving-abstractions/14`

The user can turn an instance into an independent copy by clicking this number.

### 8.2.5 Higher-order functions

None of the languages presented here include support for defining higher-order functions. In fact, the only languages with support for user-defined higher-order functions in Hils's original survey are all either general-purpose programming languages (outside of the scope of this work) or Hils's own DataVis [Hil91], a research language for scientific visualization. In [FPK93], higher-order functions for dataflow visual languages are again discussed only in the context of general-purpose languages.

It is unsurprising that successful end-user programmable applications lack higher-order functions: those applications feature languages tailored for their specific domains, and a language is most effective when it is designed to work in terms of objects of its intended domain [Wes15]. Domain specialists think in terms of objects of their domain: numbers in a spreadsheet represent monetary values, a matrix in a graphics editor represents an image. Functions as first-class objects are a reification of programs. First-order functions represent programs that are operations on objects of the domain: a function $transpose : Score \times Key \to Score$ represents the work of transposing a musical score into another key (e.g. from $C\sharp$ to $B\flat$), and as such it is an activity within the specialist's domain. Higher-order functions are one step removed from the domain in terms of abstraction: they are programs that are operations on other programs. The addition of higher-order functions to the set of first-class values, thus, makes the universe of discourse[7] strictly larger, and the added objects are familiar (and of interest) to computing professionals, not to domain specialists. An argument of the same nature could be made in the opposite direction of the abstraction ladder, to explain the absence of low-level access to bits and bytes.

## 8.3 Type checking

This work has not focused much on type checking issues, since the type systems of all languages studied are very simple. In at least one case, the type system of a language was deliberately simplified by the language designers, with the restriction on recursive data types in LabVIEW. Still, there are interesting observations to be made about type checking in dataflow end-user applications.

We define type checking in the context of dataflow as a verification provided by the environment, prior to execution, that the types of values expected by an input port matches the types of values produced by an output port connected to it. In the case of an environment with liveness level 2, this means that the system reports a type mismatch before entering the "use" mode. In the case of a responsive environment (i.e. one with liveness level 3), we expect the system to flag a type incompatibility immediately as the user attempts to add an invalid connection, disallowing the creation of the wire in the first place.

[7]in Boole's original sense [Boo54]

LabVIEW performs type checking, and the various data types supported are visually identified through colors in nodes and wires. When a user draws incorrect wires those remain in the diagram marked as such and running the program is not possible. As explained in Chapter 6, LabVIEW's inference system for tunnel directions can cause previously correct wires to become flagged as incorrect, sometimes with wide-ranging and confusing results, and unhelpful error messages. The well-intentioned convenience provided by the inference system has proven to be, in our opinion, inappropriate for an end-user programming system.

Pure Data offers limited type checking, in the sense that the interface provides a clear separation between message and audio data, differentiating both nodes and wires of these two kinds and disallowing mismatching connections, but type mismatches between various message data types are not checked. In various senses, Pure Data works almost like two languages in one, with its single-typed synchronous model for audio flow on one side, and a dynamically-typed dynamic dataflow model for message passing on the other.

Both Blender and Reaktor provide visual hints about the types of their input and output ports, and only allow connecting wires between ports of compatible types. Type errors are impossible.

Excel and VEE are dynamically typed: type errors are reported only at runtime. Since Excel is a responsive application and connections between cells are given by the user textually, there is no way for the language to prevent type errors in the manner of Blender and Reaktor. However, one could conceive of a statically-typed spreadsheet that, in the event of cell errors, instead of merely producing an error value for the whole formula[8], produced an error message indicating which term of the expression caused the type error, as modern compilers do.

There are other interesting issues related to types which we did not focus in this work. Efficient handling of data structures and strategies to avoid excessive copying of data between nodes are matters of concern [FP15, FBH16], especially in less restricted evaluation models.

## 8.4 Other aspects

Finally, we discuss other aspects with primarily syntactic or pragmatic impact. Since these affect the design of the language as a whole, choices made one way or another in these aspects may influence other choices in the semantic aspects discussed above.

### 8.4.1 Liveness, representation and modes

All languages from this study score either 2 or 3 in Tanimoto's liveness scale [Tan90]. Recalling the meaning of each level, we have that in level 2 the visual representation is the executable program, and that in level 3 the visual representation is responsive: editing the visual representation triggers updates to the evaluation of the program.

Level 2 is therefore a syntactic feature: all visual languages fit this criterion. All languages presented in this work apart from the spreadsheets use box-line graph

[8]Not to mention that such errors in dynamically-typed spreadsheets are subject to often-arbitrary error propagation semantics as we have seen in Chapter 5.

representations, but the (semantically significant!) spatial layout of a spreadsheet is also a visual representation of the program.

Level 3 refers to the lack of separate "edit" and "use" modes. Depending on the the language, a responsive loop may have a semantic impact or not: if a spreadsheet was made non-responsive, with explicit "edit" and "use" modes, that would only mean that recalculations would have to be triggered explicitly by a "Run" button (which would briefly send the program into "use" mode and back). Effectively, our interpreter from Chapter 5 would be unchanged. For a program with long-running loops, however, adding a responsive interface would bring new questions about the language's behavior: what would happen in a responsive version of LabVIEW if graphs could gain or lose nodes and wires as the program runs? Our interpreter from Chapter 6 would be very different.

None of the programs we analyzed implements liveness at Tanimoto's level 4, in which the program is "responsive and live" in the sense that results update continually as the program is edited beyond merely reacting to the user's edits. Pure Data does continue to produce audio if the user switches from "use" back to "edit" mode, but the sets of actions allowed in each mode are disjoint. So, in that sense, it does not qualify for level 3 in terms of inputs but it reaches level 4 in terms of outputs. That may in fact indicate a shortcoming in Tanimoto's classification method.

A more useful observation may be that the applications that attain level 3 of liveness, Excel and Blender, share some important traits in their semantics that make their responsiveness possible: they combine static dataflow with the absence of time-dependent firings and no side-effecting nodes. This allows both applications to re-evaluate subgraphs as needed and present the user an instant update each time they make an edit.

### 8.4.2 Textual sub-language and scripting

There are two distinct aspects with regard to integration with textual languages. The first aspect is what we call a textual sub-language, which is a textual part of the UI-level language. As such, it is a uniquely syntactic distinction: the interpreters in Chapters 4 and 5 implement their textual parts as AST nodes that are intertwined with the program representation as a whole. The second aspect is the integration of a scripting language in the application, reflecting the architecture described in Chapter 2.

The textual sub-languages of Excel and Pure Data are at the forefront of their respective applications, and they are central to their dataflow languages as the dataflow languages are central to the UI as a whole. In line with the three-layer architecture, both applications allow for more advanced scripting as well. Excel, as part of the Microsoft Office family, integrates with Visual Basic for Applications. The vanilla package of Pure Data does not ship with a scripting engine by default, but it is extensible through plug-ins and there are extensions available that add Python and Lua scripting, the latter being available from the main Pure Data community site and included by default in some distributions of the application. Max/MSP is also extensible with a number of options of scripting language plugins, including Python, Lua, Ruby and JavaScript.

Reaktor and Blender are the only two languages to lack a textual sub-language. Blender integrates with textual languages at the scripting language level, but Reaktor presents its scripting layer as a second graphical language. It is notable how, in spite of avoiding textual languages, the design of Reaktor evolved to the same three-layer structure described in 2.1.2: Reaktor Core was introduced in 2005, adding more powerful, lower-level programming capabilities to the application. In many aspects, the semantics of the higher-level Reaktor Primary language resemble those of Pure Data, and the semantics of the lower-level Reaktor Core resemble those of LabVIEW (further fueling the discussion introduced in Section 6.3.1).

LabVIEW has a somewhat blurred boundary between its textual sub-language and its scripting capabilities: it offers a gradient of options, starting from a functional "expression node" in which a single-variable mathematical expression can be entered textually; a "formula node" in which small imperative programs can be written in a safe subset of C, accessing data only via the node's input and output ports; a "script node" which accepts MathScript, a subset of MATLAB; and a "MATLAB node", which connects to an external MATLAB instance for execution. The latter two nodes allow for side-effects.

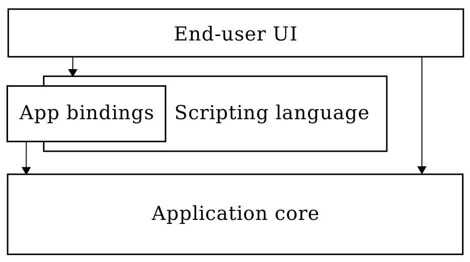
## 8.5 Discussion: An architectural pattern for end-user programmable applications

The architecture of today's end-user programmable applications is typically an extension of that of scriptable applications (Figure 8.4(a)), adding an ad hoc end-user language accessible via the application's interface (Figure 8.4(b)). To move past ad hoc end-user languages and get us closer to the situation we have in the scripting world, it is necessary to take into account the fact that these languages need to be fully customized to their domain.

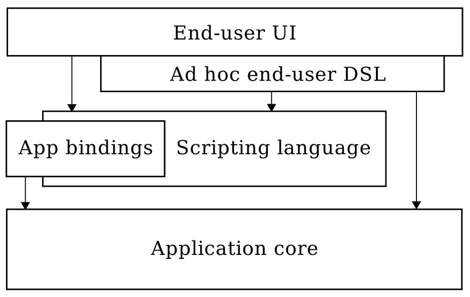
An approach for this is to produce a DSL on top of a reusable component (Figure 8.4(c)). For this, the implementation of dataflow languages needs to become reusable. One possibility in this direction is the development of a dataflow engine, exposing to the application developer building blocks based on well-understood design alternatives, such as those discussed in Chapter 8. The development of the UI-level layer would become then an integration process, similar to what currently happens with scripting languages.

One concern when exposing the functionality of an application as two different languages is a possible discontinuity in the abstractions provided (termed the semiotic continuum principle in [dSBdS01]), so that the scripting layer contains functionality that is unrepresentable in the UI, or vice-versa. Note that this kind of discontinuity can happen between any two layers of abstraction. As a practical example, our previous work developing a translator of Lua 5.0 to C based on the Lua/C API[9] uncovered some shortcomings in said API. What that work did was to attempt to perform a projection of the language into its API, producing a definition of the semantics of Lua 5.0 programs in terms of its Lua/C API, using

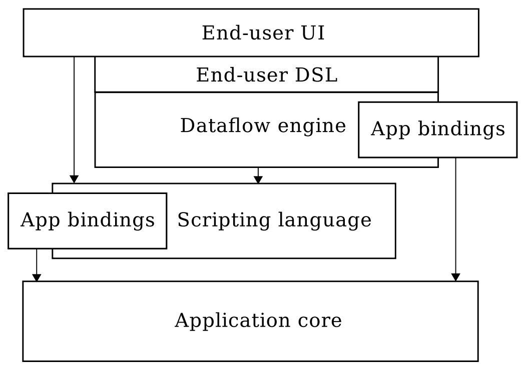
[9]`https://github.com/hishamhm/luatoc`



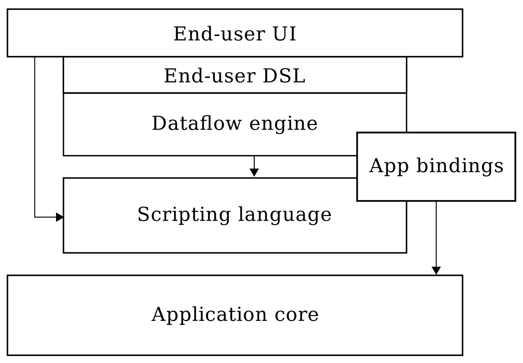
(a) Scriptable application



(b) Scriptable end-user programmable application



(c) UI-level DSL provided by a dataflow engine



(d) UI-level and scripting languages with shared bindings

Figure 8.4: Architectural patterns of programmable applications

pure C exclusively for representing control flow. The fact that the resulting translator had to produce strings of Lua code for some operations meant that not all aspects of the language were readily interoperable through the API. The Lua/C API was subsequently amended in Lua 5.1, allowing for a full projection without string evaluation.

Making sure that the different layers of a programmable application project correctly onto each other can be challenging, especially when the languages at each level and their binding APIs evolve in parallel. The possibility for API discontinuities are greater when there are different paths towards the application core (as in Figures 8.4(b) and 8.4(c)). A way to ensure this consistency between the end-user UI-level language and the scripting language would be to share the application bindings, thus providing a single path to the application core (Figure 8.4(d)). There are interesting possibilities of how to achieve this, such as compatible lower-level APIs or using the scripting language to implement the dataflow engine.

# Chapter 9

# Conclusion

End-user languages for UI interaction are today in the state where scripting languages were in the 1980s: they are custom ad hoc languages, with their implementation often mixed with that of the application itself, and their design reflecting an organic evolution. In the world of scripting, this has since been replaced by out-of-the-box implementations of widely used languages, reused among many applications, with their design reflecting an evolution towards suitability for multiple domains. Most importantly, this notion of a "scriptable application" composed by a low-level core and a scripting language extended with bindings to this core has become a common architectural pattern [Ous98].

We aimed to bring a similar evolution to end-user UI-level languages one step closer to reality. In earlier drafts of this work, our initial goal was to map the design space of dataflow end-user languages, identify the various design options, and from there construct a reusable language in which these various options were provided as building blocks, so that an application developer could construct the UI-level language for their application by combining these blocks at will. However, providing application writers with a toolkit of language building blocks could save them considerable development effort, but the resulting languages would still be ad hoc. As we developed our research, we realized that we needed to step back and perform a deeper analysis of the design space instead. Not only it is important to know what the choices are, but it is fundamental to understand the effects of these choices.

Our work, thus, made the following primary contributions, each one leading logically to the next:

**Mapping the design space of dataflow end-user language semantics.** The apparent simplicity of visual diagrams embedded in application interfaces is deceiving. A number of design decisions go into building a dataflow UI-level language. Much of the earlier research work on this class of languages went into studying its visual aspects. Here, we focused on the semantic aspects of those languages, lifting the veil on their underlying complexity.

**A critique of design alternatives for dataflow end-user languages.** This map of the design space proved to be appropriate as a conceptual framework to compare languages of this kind efficiently. We applied this classification into a group of successful end-user applications, allowing us to discuss the effects of each of the mapped design dimensions based in actual practice. This allowed us to verify which design choices worked best and which ones caused problems.

**Identifying interdependencies in dataflow design choices.** Our study concluded that many of the dimensions identified in the design space for this class of languages are dependent on each other. The evaluation provided in Chapter 8 showed that many languages were only able to pick one choice over another in certain design aspects because of their choices in other aspects. This stresses the importance of understanding these design aspects as a whole, at the risk of having one choice bringing unexpected consequences later in the design. It also confirms that a sound design cannot be achieved by merely combining building blocks at will.

In the course of this work, we also made the following secondary contributions:

**A specification of realistic spreadsheet semantics.** The existing literature on spreadsheets to this day has always restricted itself to simplified models of their semantics [Tys13, AE06, AAGK03, AHH15], ignoring the variations across different implementations. The major omissions in official specification documents [ISO12, OAS11] and the incompatibilities between implementations from the same vendor suggest that these languages are not understood in detail at all. In this work we provide what we believe to be the most comprehensive formal specification of a realistic spreadsheet semantics so far.

**Executable models of Pure Data and LabVIEW.** In a similar vein, both Pure Data and LabVIEW are relevant languages that have been studied in academia, not only within their respective domains but also by the programming language community [BJ13, BH15, MS98, KKR09]. None of these languages had any kind of specification, and this work provides the first realistic models of their core semantics.

**Insights on multi-language application architecture.** Throughout this work, we made some observations on language and application architecture that we believe to be novel in the literature: the concept of roles of end-user programming languages, with the distiction between central and peripheral end-user languages (Section 2.1.1); identifying Nardi's three profiles of users with the presence of a three-layer architecture in successful end-user programmable applications (Section 2.1.2); the effects of API bindings and architectural alternatives to ensure proper language projections in multi-language designs (Section 8.5).

All in all, this work provided a better understanding of the state of dataflow languages in the context of end-user programming, with a view towards the advancement of this field. We believe the the evolution of scripting languages hints at a possible path for the evolution of UI-level languages. Scripting languages evolved from initially ad hoc shell and configuration languages, as these started to make use of the lessons learned by earlier high-level programming languages. We believe

UI-level languages will follow the same path, adopting lessons from decades of research in dataflow. In our view, reusable UI-level dataflow languages will eventually become a reality, provided that the application architecture is considered as a whole and the design constraints are well-understood. This in itself is an exciting avenue for future work.

# Bibliography

[AAGK03] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. *A type system for statically detecting spreadsheet errors*, pages 174–183. 10 2003.

[AE06] Robin Abraham and Martin Erwig. Type inference for spreadsheets. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, pages 73–84, New York, NY, USA, 2006. ACM.

[Agi11] Agilent Technologies. *VEE 9.3 User's Guide*, 2011.

[AHH15] E. Aivaloglou, D. Hoepelman, and F. Hermans. A grammar for spreadsheet formulas evaluated on two large datasets. In *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pages 121–130, Sept 2015.

[AS94] Arnon Avron and Nada Sasson. Stability, sequentiality and demand driven evaluation in dataflow. *Formal Aspects of Computing*, 6(6):620–642, 1994.

[ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[AW77] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Commun. ACM*, 20(7):519–526, jul 1977.

[BB01] B. Bhattacharya and S.S. Bhattacharyya. Parameterized dataflow modeling for dsp systems. *Trans. Sig. Proc.*, 49(10):2408–2421, October 2001.

[BBS01] Greg J. Badros, Alan Borning, and Peter J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, dec 2001.

[BELP95] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static data flow. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 3255–3258 vol.5, May 1995.

[Ben86] Jon Bentley. Programming pearls: Little languages. *Commun. ACM*, 29(8):711–721, aug 1986.

[BGB14] Daniel W. Barowy, Dimitar Gochev, and Emery D. Berger. Checkcell: Data debugging for spreadsheets. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 507–523, New York, NY, USA, 2014. ACM.

[BH15] Gregory Burlet and Abram Hindle. An empirical study of end-user programmers in the computer music community. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 292–302, Piscataway, NJ, USA, 2015. IEEE Press.

[BHHW99] Ed Baroth, Chris Hartsough, Amy Holst, and George Wells. Evaluation of LabVIEW 5.0 and HP VEE 5.0 - Part 2. *EE, Evaluation Engineering*, 38(5):5, may 1999.

[BJ13] Karim Barkati and Pierre Jouvelot. Synchronous programming in audio processing: A lookup table oscillator case study. *ACM Comput. Surv.*, 46(2):24:1–24:35, dec 2013.

[Ble17] Blender Foundation. Blender, 2017.

[Boo54] George Boole. *An investigation of the laws of thought, on which are founded the mathematical theories of logic and probabilities*, chapter 3, page 30. Project Gutenberg EBook #15114 (2005), 1854.

[BS14] Margaret M. Burnett and Christopher Scaffidi. *The Encyclopedia of Human-Computer Interaction, 2nd Ed.*, chapter End-User Development. The Interaction Design Foundation, Aarhus, Denmark, 2014.

[Buc93] Joseph Tobin Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model.* PhD thesis, 1993. AAI9431898.

[CFR06] Jeffrey Carver, Marc Fisher, II, and Gregg Rothermel. An empirical evaluation of a testing and debugging methodology for excel. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 278–287, New York, NY, USA, 2006. ACM.

[CSV09] Jácome Cunha, João Saraiva, and Joost Visser. From spreadsheets to relational databases and back. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '09, pages 179–188, New York, NY, USA, 2009. ACM.

[Den85] Jack B. Dennis. Models of data flow computation. In Manfred Broy, editor, *Control Flow and Data Flow - Concepts of Distributed Programming*, Berlin Heidelberg, 1985. Springer.

[DFR14] Camil Demetrescu, Irene Finocchi, and Andrea Ribichini. Reactive imperative programming with dataflow constraints. *ACM Trans. Program. Lang. Syst.*, 37(1):3:1–3:53, nov 2014.

[DJS11] Sebastian Draxler, Adrian Jung, and Gunnar Stevens. Managing software portfolios: a comparative study. In *End-User Development*, pages 337–342. Springer, 2011.

[DOK+87] D. Dougherty, T. O'Reilly, S.G. Kochan, P.H. Wood, and O'Reilly & Associates. *UNIX Text Processing*. Hayden Books UNIX library system. Hayden Books, 1987.

[dSBdS01] C.S de Souza, S.D.J Barbosa, and S.R.P da Silva. Semiotic engineering principles for evaluating end-user programming environments. *Interacting with Computers*, 13(4):467 – 495, 2001.

[FBH16] Vincent Foley-Bourgon and Laurie Hendren. Efficiently implementing the copy semantics of matlab's arrays in javascript. In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 72–83, New York, NY, USA, 2016. ACM.

[FP15] Bruno Ferreira and Fernando Quintão Pereira. The Dinamica virtual machine for geosciences. In *Brazilian Symposium on Programming Languages - SBLP*, 2015.

[FPK93] Alex Fukunaga, Wolfgang Pree, and Takayuki Dan Kimura. Functions as objects in a data flow based visual language. In *Proceedings of the 1993 ACM Conference on Computer Science*, CSC '93, pages 215–220, New York, NY, USA, 1993. ACM.

[FZHT13] Joachim Falk, Christian Zebelein, Christian Haubelt, and Jürgen Teich. A rule-based quasi-static scheduling approach for static islands in dynamic dataflow graphs. *ACM Trans. Embed. Comput. Syst.*, 12(3):74:1–74:31, April 2013.

[GJ98] Steven Greenbaum and Stanley Jefferson. A compiler for HP VEE. *Hewlett-Packard Journal*, 49(2):98–122, may 1998.

[GKHB09] Nicolas Gold, Jens Krinke, Mark Harman, and David Binkley. Clone detection for max/msp patch libraries (poster abstract). In *Digital Music Research Network Workshop*, 2009.

[GKHB11] Nicolas Gold, Jens Krinke, Mark Harman, and David Binkley. Cloning in Max/MSP patches. In *Proceedings of International Computer Music Conference 2011*, pages 159–162, Huddersfield, UK, July 2011. International Computer Music Association.

[GKM+11] L. Gantel, A. Khiar, B. Miramond, A. Benkhelifa, F. Lemonnier, and L. Kessal. Dataflow programming model for reconfigurable computing. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, pages 1–8, June 2011.

[GS11] Gagan Gupta and Gurindar S. Sohi. Dataflow execution of sequential imperative programs on multicore architectures. In *Proceedings of the*

*44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 59–70, New York, NY, USA, 2011. ACM.

[Has96] Haskell Wiki. Wadler's Law. `https://wiki.haskell.org/Wadler's_Law`, 1996.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, Sep 1991.

[Hil91] Daniel D. Hils. DataVis: A visual programming language for scientific visualization. In *Proceedings of the 19th Annual Conference on Computer Science*, CSC '91, pages 439–448, New York, NY, USA, 1991. ACM.

[Hil92] Daniel D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, 3:69–101, 1992.

[HL15] Ralf Hinze and Andres Löh. Guide to lhs2 (for version 1.19). `https://hackage.haskell.org/package/lhs2tex-1.19/src/doc/Guide2.pdf`, apr 2015.

[ISO12] ISO. ISO/IEC 29500-1:2012 – Office Open XML File Formats, 2012.

[IWC+88] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. Fabrik: A visual programming environment. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '88, pages 176–190, New York, NY, USA, 1988. ACM.

[Jam14] Noreen Jamil. Constraint solvers for user interface layout. arXiv preprint arXiv:1401.1031, jan 2014.

[JBB03] Simon Peyton Jones, Alan Blackwell, and Margaret Burnett. A user-centred approach to functions in Excel. In *In ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 165–176. ACM Press, 2003.

[JHM04] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, mar 2004.

[Jon05] Brian Jones. Comments from Tim Bray on OpenDocument. `http://blogs.msdn.com/b/brian_jones/archive/2005/10/04/477127.aspx`, oct 2005.

[KCM86] Takayuki Dan Kimura, Julie W. Choi, and Jane M. Mack. A visual language for keyboardless programming. Technical Report WUCS-86-06, Washington University, St. Louis, jun 1986.

[Kim85] Takayuki Dan Kimura. Hierarchical dataflow model: A computation model for small children. Technical Report WUCS-85-05, Washington University, St. Louis, may 1985.

[KKR09] Matt Kaufmann, Jacob Kornerup, and Mark Reitblatt. Formal verification of LabVIEW programs using the ACL2 theorem prover. In *8th International Workshop on the ACL2 Theorem Prover and Its Applications*, ACL2 '09, pages 82–89, New York, NY, USA, 2009. ACM.

[Kos73] Paul R. Kosinski. A data flow language for operating systems programming. *SIGPLAN Not.*, 8(9):89–94, jan 1973.

[KSP15] Krishna Kavi, Charles Shelor, and Domenico Pace. Concurrency, synchronization, and speculation–the dataflow way. *Advances in Computers*, 96:47–104, 2015.

[LCM$^+$08] Yuan Lin, Yoonseo Choi, S. Mahlke, T. Mudge, and C. Chakrabarti. A parameterized dataflow language extension for embedded streaming systems. In *Embedded Computer Systems: Architectures, Modeling, and Simulation, 2008. SAMOS 2008. International Conference on*, pages 10–17, July 2008.

[LHML08] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. Coscripter: Automating & sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1719–1728, New York, NY, USA, 2008. ACM.

[LM87] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept 1987.

[Mac90] Wendy E. Mackay. Patterns of sharing customizable software. In *Proceedings of the 1990 ACM Conference on Computer-supported Cooperative Work*, CSCW '90, pages 209–221, New York, NY, USA, 1990. ACM.

[Mar03] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, 8(1):1–6, 2003.

[McI98] David McIntirye. Comp.lang.visual - frequently-asked questions list, mar 1998.

[MCLM90] Allan MacLean, Kathleen Carter, Lennart Lövstrand, and Thomas Moran. User-tailorable systems: Pressing the issues with buttons. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 175–182, New York, NY, USA, 1990. ACM.

[Mic14] Microsoft. Microsoft by the numbers. `https://news.microsoft.com/bythenumbers/ms_numbers.pdf`, 2014.

[Mic16] Microsoft. Guidelines and examples of array formulas. `https://support.office.com/en-us/article/Guidelines-and-examples-of-array-formulas-3BE0C791-3F89-4644-A062-8E6E9ECEE523`, 2016.

[MMI+13] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[MP00] M. Mosconi and M. Porta. Iteration constructs in data-flow visual programming languages. *Comput. Lang.*, 26(2-4):67–104, July 2000.

[MS98] A.K. Mok and D. Stuart. An rtl semantics for labview. In *Aerospace Conference, 1998 IEEE*, volume 4, pages 61–71 vol.4, Mar 1998.

[MSH92] Brad A Myers, David Canfield Smith, and Bruce Horn. Report of the "End-User Programming" working group. In Brad A. Myers, editor, *Languages for developing user interfaces*, chapter 19, pages 343–366. AK Peters, Ltd., 1992.

[MTdS13] Ingrid Teixeira Monteiro, Eduardo Tiomno Tolmasquim, and Clarisse Sieckenius de Souza. Going back and forth in metacommunication threads. In *12th Brazilian Symposium on Human Factors in Computing Systems*, IHC '13, pages 102–111, Porto Alegre, Brazil, Brazil, 2013. SBC.

[Nar93] Bonnie A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing.* MIT press, 1993.

[Nat01] National Instruments. *Getting Started With LabVIEW.* National Instruments, Austin, Texas, 321527e-01 edition, nov 2001.

[Nat13] National Instruments. Reentrancy: Allowing simultaneous calls to the same SubVI. LabVIEW 2013 Help, jun 2013.

[Nat15] Native Instruments. *REAKTOR 6 - Building in Primary.* Berlin, Germany, 6.0.1 edition, nov 2015.

[NM91] B. A. Nardi and J. R. Miller. Computer-supported cooperative work and groupware. chapter Twinkling Lights and Nested Loops: Distributed Problem Solving and Spreadsheet Development, pages 29–54. Academic Press Ltd., London, UK, UK, 1991.

[OAS06] OASIS. OpenFormula Format for Office Applications (OpenFormula) - Rough Draft. `https://www.oasis-open.org/committees/download.php/16826/openformula-spec-20060221.html`, 02 2006.

[OAS11] OASIS. Open Document Format for Office Applications (OpenDocument) Version 1.2 - Part 2: Recalculated Formula (OpenFormula) Format. `http://docs.oasis-open.org/office/v1.2/os/OpenDocument-v1.2-os-part2.html`, 9 2011.

[Ous98] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, mar 1998.

[$P^+15$] Miller Puckette et al. Pd documentation. `http://msp.ucsd.edu/Pd_documentation/index.html`, 2015.

[Phi13] Dusty Phillips. Dead batteries included. O'Reilly Radar, October 2013.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 717–740, New York, NY, USA, 1972. ACM.

[Sch17] Toby Schachman. Apparatus: a hybrid graphics editor and programming environment for creating interactive diagrams, 2017.

[$SFG^+15$] Tobias Schwarzer, Joachim Falk, Michael Glaß, Jürgen Teich, Christian Zebelein, and Christian Haubelt. Throughput-optimizing compilation of dataflow applications for multi-cores using quasi-static scheduling. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, pages 68–75, New York, NY, USA, 2015. ACM.

[SMFBB93] Michael Sannella, John Maloney, Bjorn Freeman-Benson, and Alan Borning. Multi-way versus one-way constraints in user interfaces: Experience with the deltablue algorithm. *Softw. Pract. Exper.*, 23(5):529–566, may 1993.

[SR12] Marton Sakal and Lazar Rakovic. Errors in building and using electronic tables: Financial consequences and minimisation techniques. *International Journal on Strategic Management and Decision Support Systems in Strategic Management*, 17(3):29–35, 2012.

[SSM05] Christopher Scaffidi, Mary Shaw, and Brad Myers. Estimating the numbers of end users and end user programmers. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '05, pages 207–214, Washington, DC, USA, 2005. IEEE Computer Society.

[Sut63] Ivan Edward Sutherland. *Sketchpad, a man-machine graphical communication system.* PhD thesis, Massachusetts Institute of Technology, jan 1963.

[Tan90] S. L. Tanimoto. VIVA: a visual language for image processing. *Journal of Visual Languages and Computing*, 1:127–139, 1990.

[TBH82] Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins. Data-driven and demand-driven computer architecture. *ACM Computing Surveys*, 14(1):93–143, March 1982.

[Tra05] Laurence Tratt. The importance of syntax. `http://tratt.net/laurie/blog/entries/the_importance_of_syntax.html`, may 2005.

[Tys10] Jerzy Tyszkiewicz. Spreadsheet as a relational database engine. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 195–206, New York, NY, USA, 2010. ACM.

[Tys13] J. Tyszkiewicz. The Power of Spreadsheet Computations. *ArXiv e-prints*, jul 2013.

[Wes15] David M. West. The cuban software revolution: 2016-2025. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 267–281, New York, NY, USA, 2015. ACM.

[Wip10] Matthieu Wipliez. *Compilation infrastructure for dataflow programs.* PhD thesis, INSA de Rennes, 2010.

[WP94] Paul G. Whiting and Robert S. V. Pascoe. A history of data-flow languages. *IEEE Ann. Hist. Comput.*, 16(4):38–59, dec 1994.

[Zmö14] Iohannes M. Zmölnig. How to write an external for Pure Data. `http://iem.at/pd/externals-HOWTO/`, March 2014.

# Appendix A

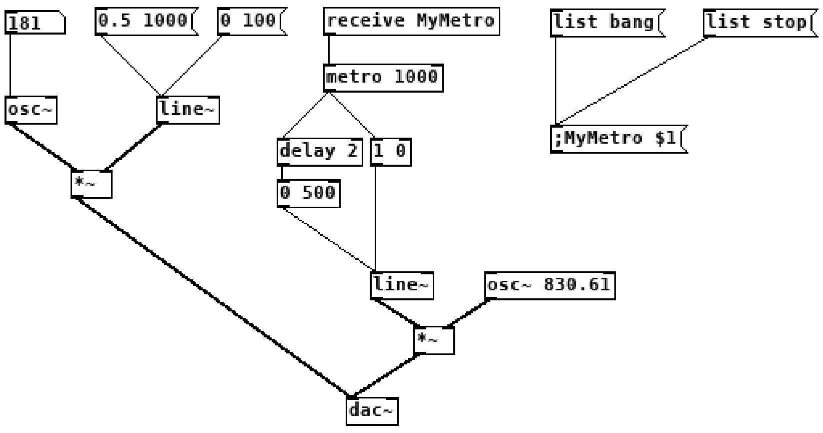
# Demonstration of the interpreter modeling Pure Data

To wrap up the presentation of the interpreter modeling Pure Data, we present a demonstration of its use. We build a simple synthesizer with both frequency and amplitude controllable via events, and use it to play the motif from the main theme of the film “Back To The Future”, composed by Alan Silvestri.

First, we define a few constants corresponding to the frequency in Hertz of some musical notes:

```
cSharp = 554.37
aSharp = 932.33
g      = 783.99
gSharp = 830.61
f      = 698.46
```

Then, we construct the patch that corresponds to the following graph:



```
example = PdPatch (fromList [
  PdAtomBox (PdFloat 0),    -- 0
  PdObj      [PdSymbol "osc~", PdFloat gSharp] 2 1,    -- 1
  PdMsgBox   [PdCmd PdToOutlet (map (PdTAtom ∘ PdFloat) [0.5, 1000])],    -- 2
  PdMsgBox   [PdCmd PdToOutlet (map (PdTAtom ∘ PdFloat) [0,100])],   -- 3
  PdObj      [PdSymbol "line~"] 2 1,   -- 4
  PdObj      [PdSymbol "*~"] 2 1,   -- 5
  PdObj      [PdSymbol "dac~"] 1 0,   -- 6
  PdObj      [PdSymbol "receive", PdSymbol "MyMetro"] 0 1,   -- 7
  PdObj      [PdSymbol "metro", PdFloat 500] 2 1,   -- 8
  PdObj      [PdSymbol "delay", PdFloat 5] 2 1,   -- 9
  PdObj      [PdSymbol "list", PdFloat 0.5, PdFloat 0.1] 2 1,   -- 10
  PdObj      [PdSymbol "list", PdFloat 0, PdFloat 500] 2 1,   -- 11
  PdObj      [PdSymbol "line~"] 1 1,   -- 12
  PdObj      [PdSymbol "osc~", PdFloat (gSharp / 2)] 1 1,   -- 13
  PdObj      [PdSymbol "*~"] 2 1,   -- 14
  PdMsgBox   [PdCmd PdToOutlet
               [PdTAtom (PdSymbol "list"), PdTAtom (PdSymbol "bang")]],   -- 15
  PdMsgBox   [PdCmd PdToOutlet
               [PdTAtom (PdSymbol "list"), PdTAtom (PdSymbol "stop")]],   -- 16
  PdMsgBox   [PdCmd (PdReceiver "MyMetro") [PdTDollar 1]]]   -- 17
  )(fromList [
    ((0,  0) ▷ (1,  0)), ((1,  0) ▷ (5,  0)), ((2,  0) ▷ (4,  0)),
    ((3,  0) ▷ (4,  0)), ((4,  0) ▷ (5,  1)), ((5,  0) ▷ (6,  0)),
    ((7,  0) ▷ (8,  0)), ((8,  0) ▷ (9,  0)), ((8,  0) ▷ (10,0)),
    ((9,  0) ▷ (11,0)), ((10,0) ▷ (12,0)), ((11,0) ▷ (12,0)),
    ((12,0) ▷ (14,0)), ((13,0) ▷ (14,1)), ((14,0) ▷ (6,  0)),
    ((15,0) ▷ (17,0)), ((16,0) ▷ (17,0))]
  )[1,4,5,12,13,14,6]
```

This is the sequence of input events that corresponds to playing the tune:

```
main :: IO ()
main =
  ByteString.putStr $ runPut (putWav output)
  where
  output = genOutput $ runSteps 10000 example [
    (PdEvent 1000 15 [PdSymbol "bang"]),   -- MyMetro bang
    (PdEvent 1010 2 [PdSymbol "bang"]),   -- 0.1 1000
    (PdEvent 1900 3 [PdSymbol "bang"]),   -- 0 100
    (PdEvent 2001 0 [PdSymbol "float", PdFloat cSharp]),
    (PdEvent 2002 2 [PdSymbol "bang"]),   -- 0.1 1000
    (PdEvent 2900 3 [PdSymbol "bang"]),   -- 0 100
    (PdEvent 3001 0 [PdSymbol "float", PdFloat g]),
    (PdEvent 3002 2 [PdSymbol "bang"]),   -- 0.1 1000
    (PdEvent 4660 3 [PdSymbol "bang"]),   -- 0 100
    (PdEvent 4749 2 [PdSymbol "bang"]),   -- 0.1 1000
    (PdEvent 4750 0 [PdSymbol "float", PdFloat gSharp]),
    (PdEvent 4875 0 [PdSymbol "float", PdFloat aSharp]),
    (PdEvent 5000 0 [PdSymbol "float", PdFloat gSharp]),
    (PdEvent 5333 0 [PdSymbol "float", PdFloat f]),
    (PdEvent 5666 0 [PdSymbol "float", PdFloat cSharp]),
    (PdEvent 6000 0 [PdSymbol "float", PdFloat g]),
    (PdEvent 6650 3 [PdSymbol "bang"]),    -- 0 100
    (PdEvent 6745 2 [PdSymbol "bang"]),    -- 0.1 1000
    (PdEvent 6750 0 [PdSymbol "float", PdFloat gSharp]),
    (PdEvent 6875 0 [PdSymbol "float", PdFloat aSharp]),
    (PdEvent 7000 0 [PdSymbol "float", PdFloat gSharp]),
    (PdEvent 7000 16 [PdSymbol "bang"]),    -- MyMetro stop
    (PdEvent 8000 3 [PdSymbol "bang"])]   -- 0 100
```

In Pure Data, the sound card is represented by the `dac~` object. Our interpreter does nat handle actual audio output natively, but we can extract the inlet data from that node from the list of states, and convert it to an audio `wav` file format, that is then sent to standard output.

```
convertData :: PdNodeState → [Integer]
convertData (PdNodeState ins _) =
  let inlet = index ins 0
  in map (λ(PdFloat f) → floor (f ∗ 32768)) inlet
everyOther :: [a] → [a]
everyOther (x : (y : xs)) = x : everyOther xs
everyOther x = x
genOutput x = concat $ everyOther
  $ toList
  $ fmap (λ(PdState _ nss _ _) → convertData $ index nss 6) x
putWav vs =
  let
    riff   = 0 x46464952
    wave = 0 x45564157
    fmts  = 0 x20746d66
    datx  = 0 x61746164
    formatHeaderLen = 16
    fileSize = (44 + (length vs) ∗ 2)
    bitsPerSample = 16
    format = 1
    channels = 1
    sampleRate = 32000
  in do
    putWord32le riff
    putWord32le (fromIntegral fileSize)
    putWord32le wave
    putWord32le fmts
    putWord32le formatHeaderLen
    putWord16le format
    putWord16le channels
    putWord32le sampleRate
    putWord32le (sampleRate * bitsPerSample * (fromIntegral channels) `div` 8)
    putWord16le (((fromIntegral bitsPerSample) * channels) `div` 8)
    putWord16le (fromIntegral bitsPerSample)
    putWord32le datx
    putWord32le (fromIntegral ((length vs) * 2))
    mapM_ (putWord16le ∘ fromIntegral) vs
```

# Appendix B

# Demonstration of the spreadsheet interpreter

We present here a demonstration of the spreadsheet interpreter in use. This appendix is a Literate Haskell program including the complete source code of the demonstration.

This program imports the interpreter defined in Chapter 5 as a module, as well as some standard modules from the Haskell Platform. We also use one additional module for tabular pretty-printing of the output: `Text.PrettyPrint.Boxes`, available from Hackage, the Haskell community's package repository[1].

```
import XlInterpreter
import Data.Char (chr, ord)
import Data.Map.Strict as Map (foldlWithKey, empty, lookup, toList, (!))
import Text.PrettyPrint.Boxes as Box (render, hcat, vcat, text)
import Text.PrettyPrint.Boxes as Alignment (left, right)
```

Running the program produces the following output:

```
  |A        |B        |C        |D        |E        |F        |G
 1|       15|       15|"B"      |       75|       30|      105|     1015
 2|       30|       15|        1|         |         |         |
 3|         |         |         |         |         |         |
 4|         |         |         |         |         |         |
 5|         |         |"#VALUE!"|      115|         |       15|
 6|         |         |         |      130|         |       16|
 7|         |         |         |         |         |         |
 8|         |         |         |         |         |         |"#VALUE!"
 9|         |         |         |         |         |         |
10|       10|         |         |       10|      -20|       30|
11|"10"     |         |         |         |       20|         |
12|False    |"#DIV/0!"|         |         |         |         |
13|True     |"#VALUE!"|         |         |         |         |
14|True     |"#DIV/0!"|         |         |         |         |
```

[1] `https://hackage.haskell.org/package/boxes`

### B.0.1 Formatting

In order to produce a more readable output, we define the instance *Show* for our *XlState* type, using the *Text.PrettyPrint* package to produce tabular outputs.

```
instance Show XlState where
  show (XlState cells values) =
    "\nCells:\n" ++ listCells ++
    "\nValues:\n" ++ tableValues ++
    "\n" ++ show values ++ "\n"
    where
      r_max       = Map.foldlWithKey (λmx (⟨r⟩, _) _ → max r mx) 0 values
      c_max       = Map.foldlWithKey (λmx (_, ⟨c⟩) _ → max c mx) 0 values
      listCells   = Box.render
                      $ Box.vcat Alignment.left
                      $ map Box.text
                      $ map show (Map.toList cells)
      tableValues = Box.render
                      $ Box.hcat Alignment.left
                      $ numsCol : map doCol [0 .. c_max]
      numsCol     = Box.vcat Alignment.right
                      $ map Box.text
                      $ "  " : map show [1 .. (r_max + 1)]
      doCol c     = Box.vcat Alignment.left
                      $ Box.text ['|', chr (c + 65)] :
                        map (λs → Box.text ('|' : doRow s c)) [0 .. r_max]
      lpad m xs   = reverse $ take m $ reverse $ (take m $ repeat ' ') ++ (take m xs)
      doRow r c   = case Map.lookup ((⟨r⟩, ⟨c⟩)) values of
                      Just (XlNumber n) → lpad 9 (num2str n)
                      Just v            → show v
                      Nothing           → ""
```

### B.0.2 A test driver

We construct below a test driver function that runs test cases and compares their results to expected values.

```
runTest :: String → [(XlEvent, XlValue)] → IO ()
runTest name operations =
  let
    env@(XlState cells values) = runEvents (XlWorksheet Map.empty)
                                            (map fst operations)
    value :: String → XlValue
    value a1 = values ! (toRC a1)
    failures = filter (λv → v ≢ Nothing) $ map doCheck operations
      where
        doCheck (op, value) =
          case op of
            XlSetFormula rc fml →
              if values ! rc ≡ value
              then Nothing
              else Just (rc, value)
            XlSetArrayFormula rc_from rc_to fml →
              if values ! rc_from ≡ value
              then Nothing else
              Just (rc_from, value)
  in do
    putStrLn ""
    print name
    print env
    if null failures
    then putStrLn "OK! :-D"
    else
      do
        putStrLn "Failed: "
        print failures
```

We employ a few shortcuts to write down formulas more tersely:

```
str s = XlString s
num n = XlNumber n
err e = XlError e
boo b = XlBool b
lnum n = XlLit (XlNumber n)
lstr s = XlLit (XlString s)
lmtx mx = XlLit (XlMatrix (map (map XlNumber) mx))
lmtxs mx = XlLit (XlMatrix (map (map XlString) mx))
fun f args = XlFun f args
ref a1 = XlRef (toRC a1)
range a1 b2 = XlRng (toRC a1) (toRC b2)
toRC (l : num) = ⦇⟨((read num) − 1)⟩, ⟨((ord l) − 65)⟩⦈
addF rc f v = (XlSetFormula (toRC rc) f, v)
addAF rc_from rc_to f v = (XlSetArrayFormula (toRC rc_from) (toRC rc_to) f, v)
sumSqrt l = num $ foldr (+) 0 (map sqrt l)
```

### B.0.3 The example spreadsheet

We then run the main program, using *runTest* to create a spreadsheet, taking a list of input events as a parameter. In this list, *addF* and *addAF* are events adding formulas and array formulas to cells. The last argument is the expected value. All tests here produce the indicated values.

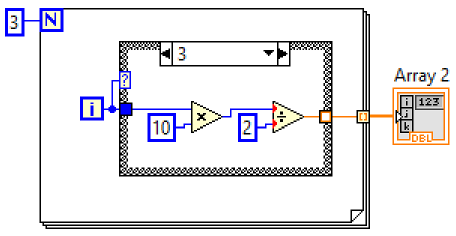
```
main :: IO ()
main =
do
runTest "Example" [
   addF "A1" (lnum 15) (num 15),
   addF "B1" (lnum 0) (num 15),
   addF "A2" (fun "+" [ref "A1", ref "B1"]) (num 30),
   addF "B1" (ref "A1") (num 15),
   addF "C1" (lstr "B") (str "B"),
   addF "C2" (lnum 1) (num 1),
   addF "B2" (fun "INDIRECT" [fun "&" [ref "C1", ref "C2"]]) (num 15),
   addF "D1" (fun "SUM" [range "A1" "B2"]) (num 75),
   addF "E1" (fun "SUM" [range "B1" "B2"]) (num 30),
   addF "F1" (fun "SUM" [range "D1" "E1"]) (num 105),
   addF "D10" (lnum 10) (num 10),
   addF "E10" (lnum (−20)) (num (−20)),
   addF "F10" (lnum 30) (num 30),
   addF "E11" (fun "ABS" [range "D10" "F10"]) (num 20),
   addF "G8" (fun "ABS" [range "D10" "F10"]) (err "#VALUE!"),
   addF "A10" (lnum 10) (num 10),
   addF "A11" (lstr "10") (str "10"),
   addF "A12" (fun "=" [ref "A10", ref "A11"]) (boo False),
   addF "A13" (fun "=" [ref "A10", lnum 10]) (boo True),
   addF "A14" (fun "=" [ref "A13", lnum 1]) (boo True),
   addF "B12" (fun "/" [lnum 1, lnum 0]) (err "#DIV/0!"),
   addF "B13" (fun "=" [ref "G8", ref "B12"]) (err "#VALUE!"),
   addF "B14" (fun "=" [ref "B12", ref "G8"]) (err "#DIV/0!"),
   addF "G1" (fun "+" [lnum 1000, range "A1" "A2"]) (num 1015),
   addF "C5" (range "A1" "A2") (err "#VALUE!"),
   addAF "F5" "F6" (lmtx [[15], [16]]) (num 15),
   addAF "D5" "D6" (fun "+" [range "A1" "A2", lnum 100]) (num 115)]
```

In `http://hisham.hm/thesis` one can find a number of tests using this test driver. These tests document the specific behavior of the interpreter and also serve as a list of corner cases which expose incompatibilities between real-world spreadsheet applications.

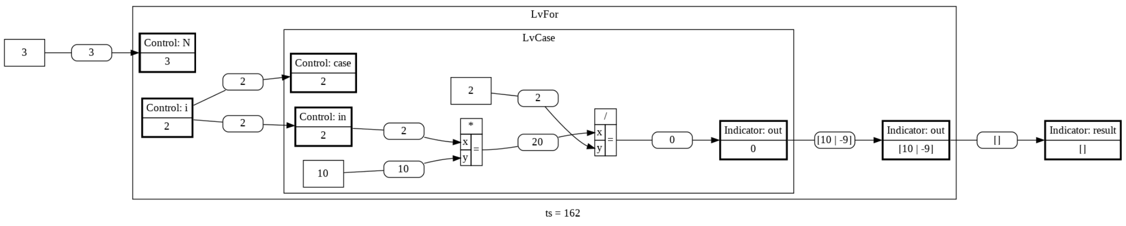
# Appendix C

# Demonstration of the interpreter modeling LabVIEW

We present here a demonstration of the interpreter modeling LabVIEW. We produced several examples, which are available at `https://hisham.hm/thesis/`, along with the resulting outputs of their execution. Here, we present only one of them. This is the test for the "case" structure, represented in LabVIEW this way:



Through the process explained in Section 6.2.6, our interpreter produces a visualization of the execution. This is a frame of the resulting animation, just before performing the division:



The test program for this example is the following.

Again, the implementation uses only standard modules included in the Haskell Platform.

```
import LvInterpreter
import Data.Sequence (fromList, elemIndexL)
import Data.List
import Data.Maybe
import Data.List.Split
```

```
main =
  do
    print vi
    runVI vi
      where vi = randomXY
```

### C.0.4 Program construction

To ease the writing of tests, we construct *LvVI* objects using a convenience function which converts the definition of wires from textual names to the numeric indices expected by the interpreter.

```
data LvStringWire = LvStringWire String String
  deriving Show
wire :: String → String → LvStringWire
wire a b = LvStringWire a b
makeVI :: [(String, LvControl)] → [(String, LvIndicator)]
          → [(String, LvNode)] → [LvStringWire] → LvVI
makeVI ctrls indics nodes stringWires =
  LvVI {
    vCtrls = ctrls,
    vIndics = indics,
    vNodes = nodes,
    vWires = map convert stringWires
  }
  where
    convert :: LvStringWire → LvWire
    convert (LvStringWire src dst) =
      let
        (type_src, srcElem, port'_src) = findElem ctrls   LvC vIndics src
        (type_dst, dstElem, port'_dst) = findElem indics LvI vCtrls  dst
      in
        LvWire (|type_src, srcElem, port'_src|)
               (|type_dst, dstElem, port'_dst|)
    findIndex :: [(String, a)] → String → Maybe Int
    findIndex es name = elemIndex name $ map fst es
    must :: (String → Maybe a) → String → a
    must fn name = fromMaybe (error ("No such entry " ++ name)) (fn name)
    findElem :: [(String, a)] → LvElemType → (LvVI → [(String, b)])
                → String → (LvElemType, Int, Int)
    findElem entries etype elems name
      | isJust $ find (≡ ':') name =
        let
          [elemName, portName] = splitOn ":" name
          elem = (must ∘ flip lookup) nodes elemName
          findPort (LvStructure _ subVi) = must $ findIndex (elems subVi)
          findPort (LvCase subVis)       = must $ findIndex (elems (head subVis))
          findPort (LvFunction _)        = λs → if null s then 0 else read s
          findPort _                     = λs → 0
        in
          (LvN, (must ∘ findIndex) nodes elemName, findPort elem portName)
      | otherwise =
        case findIndex entries name of
        Just i → (etype, i, 0)
        Nothing → findElem entries etype elems (name ++ ":0")
```

## C.0.5 Demonstration of the VI

This is the example displayed in Figure 6.1.

```
testingCase =
  makeVI
    [ -- controls
    ]
    [ -- indicators
      ("result", LvIndicator (LvArr []))
    ]
    [ -- nodes
      ("3", LvConstant (LvI32 3)),
      ("for", LvStructure LvFor (makeVI
        [ -- controls
          ("i", LvAutoControl),
          ("N", LvTunControl)
        ]
        [ -- indicators
          ("out", LvTunIndicator LvAutoIndexing)
        ]
        [ -- nodes
          ("case", LvCase [
            (makeVI
              [ -- controls
                ("case", LvControl (LvI32 0)),
                ("in", LvControl (LvI32 0))
              ]
              [ -- indicators
                ("out", LvIndicator (LvI32 0))
              ]
              [ -- nodes
                ("+", LvFunction "+"),
                ("10", LvConstant (LvI32 10))
              ]
        [ -- wires
          wire "in" "+:0",
          wire "10" "+:1",
          wire "+" "out"
        ]
      ),
      (makeVI
        [ -- controls
          ("case", LvControl (LvI32 0)),
          ("in", LvControl (LvI32 0))
        ]
        [ -- indicators
          ("out", LvIndicator (LvI32 0))
        ]
        [ -- nodes
          ("-", LvFunction "-"),
          ("10", LvConstant (LvI32 10))
        ]
        [ -- wires
          wire "in" "-:0",
          wire "10" "-:1",
          wire "-" "out"
        ]
      ),
      (makeVI
        [
          ("case", LvControl (LvI32 0)),
          ("in", LvControl (LvI32 0))
        ]
        [ -- indicators
          ("out", LvIndicator (LvI32 0))
        ]
        [ -- nodes
          ("*", LvFunction "*"),
          ("/", LvFunction "/"),
          ("10", LvConstant (LvI32 10)),
          ("2", LvConstant (LvI32 2))
        ]
        [ -- wires
          wire "in" "*:0",
          wire "10" "*:1",
          wire "*" "/:0",
          wire "2" "/:1",
          wire "/" "out"
        ]
      )
```

```
          ])
        ]
        [  -- wires
          wire "i" "case:case",
          wire "i" "case:in",
          wire "case:out" "out"
        ]
      ))
    ]
    [  -- wires
      wire "3" "for:N",
      wire "for:out" "result"
    ]
```