

# Introduction to the Cambridge Programming Research Group Lectures

Max Bolingbroke, Stephen Kell, Robin Message, Dominic Orchard

May 2010

## 1 Introduction

Immense productivity can be derived from computer technology by applying computational and algorithmic thinking through programming. Programming languages are the medium in which we express our ideas to aid our own understanding of a problem, to explicate difficult concepts, to communicate with others, to reuse previous solutions and ideas, and (usually!) to compute some result. The design of programming languages is a multi-faceted and deep field; so far there have been thousands of languages and still more are being added to this number.

The following four lectures, presented by members of the Cambridge Programming Research Group (CPRG), scratches the surface of programming language research and development. This introduction provides some discussion of language design but is by no means definitive or exhaustive.

### 1.1 Lecture Plan

1. *Modularity: what, why and how* - Stephen Kell

Most programming languages claim to support some kind of modularity. This lecture will explore several different goals which underly this claim, beginning with foundational literature on modularity and continuing by contrasting various approaches proposed by subsequent research.

2. *Meta Programming and You* - Robin Message

Meta-programming is a general term for code that creates, manipulates or influences other code. This lecture will introduce meta-programming, and look specifically at meta-programming for creating domain specific languages within Java, Ruby and Lisp.

3. *Mathematically Structuring Programming Languages* - Dominic Orchard

This lecture introduces an example of the application of concepts from abstract mathematics to the semantics of programming languages. This is done in the context of using a functional programming language as a meta language to describe and embed another language, giving a semantical definition and an interpreter in one program.

#### 4. *Optimising Functional Programming Languages* - Max Bolingbroke

This lecture discusses the rich equational theory of purely functional programming languages, and shows how it gives rise to program optimisations that are simultaneously simple to implement and extremely powerful.

## 2 A Model of Programming

The act of programming can be discussed in terms of the information flow between actors in a system. The most simple such system comprises three actors: a *programmer*, an *evaluator*, and a *computer*. We consider that research in the fields of semantics, programming languages, types, compilers, interpreters etc. are a function of the information flow between some or all of the actors in this model. Of course, this model is a simplification of the real world, where there may be many programmers, many languages, many evaluators, and many computers interacting.



Figure 1: The simple model

In this simple model, illustrated in Figure 1, information is *forward* flowing: a programmer writes a program in a language  $L$ , which the evaluator converts into a representation for execution on the target computer (the so called *byte-code* representation  $B$ ). The evaluator may be a compiler, translating a program into a binary format comprising low-level primitives of the computer, or an interpreter, converting a program on-the-fly into running code, or even a hybrid of the two, compiling and executing code dynamically.

Program development is usually not so linear and forwards leaning, as the simple model suggests, but usually proceeds in an iterative fashion. Information flowing back to the programmer from the evaluator may drive the revision of code, as the evaluator may reject a program (Figure 2).

The feedback  $F$  may be a number of things:

- Syntax errors

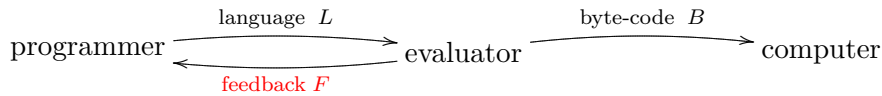


Figure 2: The simple model with evaluator feedback

- Semantic errors: most usually type errors in a statically typed language
- Semantic warnings

For a statically typed language type errors or warnings may be returned from the compiler, but for a dynamically typed language, type errors may occur at runtime.



Figure 3: The simple model with evaluator feedback and computer feedback

A further type of feedback can be added to the model, from the computer to programmer, containing information about how the program is running, not just output, but perhaps debugging information or information that reveals the inner, low-level workings of a program, such as resource utilisation (Figure 3). The semantic distance from computer to programmer is much greater than, say, evaluator to programmer, thus information is usually uncertain and/or unpredictable, requiring more experimental techniques to gain an understanding of behaviour. For some applications resource usage may not be important, but for others resource usage, and patterns of usage, may be extremely important.

Furthermore, some evaluation systems are dynamic, perhaps comprising an interpreter, run-time system, or a JIT (Just-In-Time) compiler. Such evaluators may tune their behaviour based on feedback from the running code, giving another form of execution feedback mechanism between the computer and the evaluator (Figure 4).

## 2.1 Information Content

The program in its byte-code format  $B$  contains all the implementation-level information required for evaluation of a program. The abstract program, in the mind of the programmer, may contain much more information about

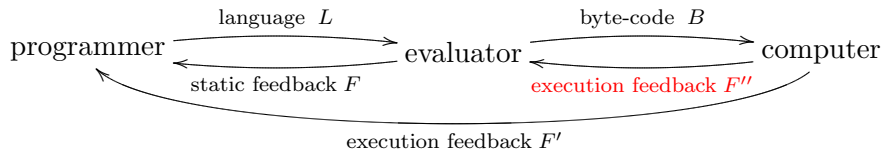


Figure 4: The simple model + evaluator feedback + computer feedback + execution feedback

the deep structure of a program, about its properties, and computational patterns. The information contained in the written program, in language  $L$ , lies somewhere in-between the two extremes of abstract and physical, containing some information relating to the implementation and some relating to the structure of the problem, in proportions depending on the level of information hiding in the language.

The more *high level* a language, the further its syntax and semantics are from the low-level instructions of hardware, thus hiding low-level execution information. The amount of information gained during evaluation may be vast or negligible depending on the level of abstraction in the language i.e. the semantic “distance” between  $B$  and  $L$ .

Information hiding in programming languages is necessary for the productive programming of ever complex programs on ever complex hardware with larger and larger teams of developers. In Lecture 1, abstraction and information hiding will be discussed further. Abstraction with the aim of succinct programming will be a motivating factor in Lectures 2 and 3.

## 2.2 Expressivity and Information Loss

Depending on the language, information about a program’s structure, computational patterns, abstractions, and properties may be lost in the process of transcription from mind to syntax. Further information about the program may be lost in its conversion to byte-code which may have facilitated more efficient implementation via specialised optimisation had the higher-level information been known. For example, certain equational properties may hold for a program, e.g.

$$f(x) + 0 \equiv 0$$

However, in a language with side effects, such as C, it is in general undecidable to decide whether a function is in fact pure (free from side effects). In a language which is inherently pure, with pure constructions, this kind of equational property is guaranteed by the language definition, hence can be used to rewrite a program with a view to optimising its run-time behaviour. Pure languages tend to be much more amenable to a mathematical

treatment, which greatly aids equational reasoning and optimisation. This property will be exploited in Lecture 4, where equational rewrites are used for optimisation, and in Lecture 3 for proving that programs satisfy certain mathematical properties.

There also exists an information impedance between the computer and the programmer. For some applications the programmer may be concerned about execution details such as speed, latency, throughput, and memory usage. However such operational details may be hidden from the programmer i.e. the execution feedback ( $F'$  in Figure 4) is minimal and unpredictable. For example, in lazy languages it can be very hard to reason about space usage at compile time; the information is not well represented in the language. A clear mental model of the execution behaviour of a language aids the programmer in optimising further than algorithmic changes. For some languages, such as C, the distance between the language and the target architecture is significantly small that translation is fairly direct, allowing a good mental model of the mapping from program constructs to execution behaviour. However, for some languages and architectures the distance is significant, leading to a more complicated translation, or run-time system, which may render understanding of execution behaviour intractable for the programmer.

Automatic optimisation of our programs by the evaluation system is usually desirable, particularly for those languages with abstract models that are very different than the target architecture. Unfortunately, this optimisation can compound the issue of a tractable cost-model for a language. Many static program analyses are in general undecidable, thus approximations must be used. However, many such approximated analyses, and transformations from the analysed results, can lead to discontinuous execution behaviour which can be a significant frustration for the programmer who is trying to hand-optimize their program.

### 2.3 Domain Specific Languages

*Domain specific languages* (DSLs) provide constructions that are specialised to a certain domain of use which may be easier to program for a non-expert but may also capture more specialised-information about a program than a general purpose language. Thus, DSLs can provide domain specific expressivity and domain specific optimisation.

*Embedded* domain specific languages are those that can be used inside of another language. The term, embedded domain specific languages, is fairly broad and can even be applied to libraries which provide a kind of language inside of another. DSLs will be discussed further in Lecture 2 in the context of *meta* programming.

### 3 Summary

The development of programming languages, and abstraction away from machine code, has greatly aided software development. Programming languages are a conduit between man and machine, with much of programming language research aiming to improve this interaction and to help us better express our ideas. We can attempt to improve languages for ease of reading, ease of writing, and ease of reasoning, and improve our evaluation systems to use less resources (whether it be processor time, memory, power, etc.) whilst still providing a predictable system. Such facets of programming language design are often non-orthogonal, thus a language designer must trade-off certain improvements for others. Often, a motivating application domain or purpose can help distill which features of a language are most important. This lecture series should give some food for thought in various areas of general programming and programming language design.