The *Four Rs* of Programming Language Design

Dominic Orchard
Computer Laboratory, University of Cambridge, UK
`dominic.orchard@cl.cam.ac.uk`

Categories and Subject Descriptors D.1.0 [*Software*]: Programming Techniques—General; I.0 [*Computing Methodologies*]: GENERAL

General Terms Design, Languages

Keywords Programming language design, The Four Rs, Domain-specific languages

"I can learn the poor things *reading*, *writing*, and *'rithmetic*, and counting as far as the rule of three, which is just as much as the likes of them require;" *Lawrie Todd: Or the Settlers in the Woods*, Galt (1832) [4].

Many will be familiar with the old adage that at the core of any child's education should be the *three Rs*: *reading*, *writing*, and *'rithmetic*. The phrase, which appeared first in print in 1825 [12] has been appropriated and parodied at length ("*read*, *reason*, *recite*", "*reduce, reuse, recycle*", etc.). Each permutation has the same purpose: to express succinctly the core tenets of an approach or philosophy.

The *four Rs of programming language design* is another such parody of this old phrase, providing a rubric, or framework, for the design and evaluation of effective programming languages and language features.

Since the very first programming language back in the 1940s [14] *thousands* of programming languages have been developed, representing a broad spectrum of paradigms, perspectives, and philosophies. And yet, there is no single language which is "*all things to all people*".

The *four Rs* were born out of trying to answer a number of questions about the nature of programming languages and programming language design: what makes a programming language effective or ineffective? What should be the core aims of a language designer? How should programming languages and features be compared? Why is there no single "perfect" language? The *four Rs* go some-way towards answering these questions.

Before I reveal the *four Rs*, let's first consider some more foundational questions:

Why programming languages? The development of programming languages has greatly aided software engineering. As hardware and software have grown increasingly complex, programming languages have developed to manage this complexity more effectively, aiding us in expressing ideas and solving increasingly complex problems.

Programming languages provide *abstraction*, by both hiding details and allowing components to be reused, allowing programmers to more effectively manage complexity in software and hardware. While it is in principle possible for any program to be written in machine code, it's hard to imagine some of the larger computer programs we interact with daily being developed in such a way. By building layers of abstraction with languages, increasingly complex systems can be constructed.

What is programming? In essence, programming is a communication process between one or more programmers and one or more computer systems. Programming languages are the medium of this communication.

Programming is not only a communication process, it is also a *translation process*. Each participant in the programming process has an internal language, both programmers and machines. In the case of a machine, the internal language comprises the instructions of the underlying hardware. In the case of a programmer, the internal language is far more nebulous, perhaps comprising natural and formal languages, along with other incorporeal, abstract thoughts.

In any case, a programming language acts as the intermediate language of translation between the participants. *Programming* is the translation from a programmer's internal language to a programming language, and *execution* is the translation from the programming language to the machine's internal language. McCracken, in 1957, captured some of this sentiment, saying "Programming [...] is basically a process of translating from the language convenient to human beings to the language convenient to the computer" where the convenient language for humans was "mathematics or English statements of decisions to be made" [8]. Here we consider the "language convenient to human beings" to be programming languages, bridging the gap between our ideas and the underlying, low-level instructions of a computer system.

Sometimes, programming is more *exploration* than communication. In which case, a programmer explores and learns about a problem by translating their internal thoughts into a program and then re-internalising the result to gain further insight. Again the process is translational.

It is from this view of programming, as a translation, communication, and exploration process, that the *four Rs* are sculpted.

A programming language should improve the *four Rs* of programs: *reading*, *writing*, *running*, and *reasoning*.

These four tenets are both guidelines for language design and research, and criteria for judging a language. They are by no means mutually exclusive, independent, or orthogonal, but are all inter-related. They are also not designed to subsume or replace the

Onward! 2011 October 22–27, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-0941-7/11/10. . . $10.00

plethora of excellent books and research papers on the subject of programming language design.

Reading and *writing* are complementary, representing effective translation from a programmer's internal, mental language to a programming language (writing) and vice versa (reading). *Running* is the execution process (i.e. "running" a program) which is the effective translation from a programming language to the internal language of some hardware system. *Reasoning* covers several activities, such as reasoning about program correctness or resource usage, but is also an intermediate process, carried out by both programmers and machines, aiding reading/writing for humans and running for machines.

There is much that could be said for each of these aspects. In the following I cover a few key points for each, but the concepts covered within each are by no means exhaustive and I strongly encourage the reader to think of their own examples and ideas and to see how they fit within the rubric of the *four Rs*.

Writing and reading are inextricably linked, thus most of what I will say about writing and reading is applicable to the other. I begin with writing, since this act must precede reading.

Writing

"Don't you see that the whole aim of Newspeak is to narrow the range of thought?" [...]

"In Oceania at the present day, Science, in the old sense, has almost ceased to exist. In Newspeak there is no word for 'Science."' *1984*, George Orwell (1941)

~

In linguistics, the *Sapir-Whorf hypothesis* embodies the concept of *linguistic relativity*, postulating that there is a relationship between the characteristics of a language and a person's understanding of the world. At the limit of this hypothesis is the concept that "the structure of anyone's native language strongly influences or fully determines the world-view he will acquire as he learns the language." [1]. The concept is strongly applicable to programming languages (though many question the extent to which it applies to natural languages).

Just as Newspeak in Orwell's 1984 restricts the thoughts of the citizens of Oceania, so too our programming languages restrict the kinds of programs we can think about, or at least write down. Depending on the program, a programming language may facilitate or hinder the process or writing. Ideally, writing should be as natural as possible, such that one can easily express their ideas with greater productivity.

When designing a language we must pay particular attention to the kinds of programs that will be most natural in the language, and conversely those that will be least natural. For example, languages with *algebraic data structures* and *pattern matching* – such as ML, Haskell, Scala – are particularly well suited for writing compilers and interpreters, due to the ease with which tree data-structures can be manipulated. However, efficient systems programming can be difficult in such languages because of the lack of fine-grained memory management (particularly in Haskell with lazy evaluation).

Another aspect of language to be considered for writability is how *succinct* or *verbose* is the syntax of a language. A verbose language might adversely affect productivity by increasing the amount of time it takes to write a program. Iverson calls this issue *economy* within a language, saying, "The utility of a language as a tool of thought increases with the range of topics it can treat, but decreases with the amount of vocabulary and the complexity of grammatical rules which the user must keep in mind." [6]. A verbose or *large* language might significantly impede a programmer's progress.

However, a language which is overly succinct might be just as unproductive as a verbose language, if not more. As an extreme example, consider the SKI calculus which has just three functional combinators: S, K, and I. The language is extremely laconic, yet writing anything but a trivial program by hand is extremely tedious and error-prone.

A useful programming concept, related to that of verbosity-vs-succinctness, is that of *abstraction*, or parameterisability. A language which provides many opportunities for abstraction can be much more effective as a tool for managing complexity, through reuse and detail hiding.

Consider a program that calculates the length of a list. A language which lacks the ability to abstract the concept of a list from its element type might require several instances of the length function for lists of different types (such as in Pascal). However, a language that supports some form of type parameterisation or polymorphism may permit just one definition that can be reused for any list type. Much effort is put into finding new forms of abstraction in programming which can greatly improve programmer productivity.

How *intuitive* a language's syntax is can also have a big effect on writing. A syntax which is so unintuitive that the programmer cannot remember how a particular construct in the language operates, or misunderstands how it operates, poses a big challenge to productivity in writing, as well as reading.

Reading and writing are complimentary. Again, much of what is suggested for writing is applicable to reading and vice versa.

Reading

"You get used to it. I don't even see the code. All I see is blond, brunette, red-head..." Cypher, *The Matrix* (1999)

~

It is often said that a program is written once, but read a thousand times. Furthermore, a program (or part of a program) is often written by one person but read by many. Such is the scale of many modern programming tasks, and development teams, that this situations is inescapable. Even small-scale projects may be accessed by a large number of people, particularly in open-source communities. But how easy is it for us to read a program written by another or even ourselves? A programmer cannot be productive if they must be "part historian, part detective, and part clairvoyant" [2] all at once. Readability of programs is important for productive programming, both individually and in teams. Depending on the program, the source programming language may facilitate or hinder this process of translating a program into one's own internal language.

Ultimately, a programming language must aid human comprehension of a program, such that the programmer has some idea of what it is that the program expresses or computes.

For Cypher, he became so acquainted with the source language of *The Matrix* (the green-scrolling glyphs) that he no longer *saw* the code *per se*, but was able to effortlessly interpret what it represented (unfortunately with a male chauvinist slant). Ideally, our languages should be equally transparent, and the task of acquaintance and fluency short.

Abstraction, discussed for writing, can facilitate reading as it provides structure to a program such that it is easily navigated and understood by comprehension of decomposed sub-programs.

How intuitive a language is again affects writing and reading. To improve readability a language might use illustrative keywords, non-overloaded syntax, and have high compositionality by providing context-independent constructions.

As a very simple and obvious example of non-intuitive syntax, consider an if-then-else construction with the semantics:

$$[\![\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3]\!] = \begin{cases} [\![e_3]\!] & \textit{if}\ [\![e_1]\!] \\ [\![e_2]\!] & \textit{otherwise} \end{cases}$$

The branching behaviour here is the opposite of the implied behaviour of the natural language reading of the construction. The example is rather obvious and extreme, but demonstrates that syntax can be non-intuitive.

Running Reading and writing concerned the translation of concepts in the mind of a programmer into a programming language, and vice versa. *Running* involves the other type of participant in the programming process: machines. Running thus requires a program to be translated into the internal language of the machine (perhaps via some other layers of abstraction e.g. translate a program into LLVM byte-code which is then translated into x86 assembly code).

Performance and *resource usage* are common concerns for programmers. In some cases, minimal resource usage is desirable, but in other cases predictability is important (e.g. for scheduling and resource management).

The *translation distance* between a programming language and the hardware language measures the disparity between the computational models of each. A *low-level* language has a computational model that is close to the architectural model; a *high-level* language has a computational model that is further away from the hardware.

For example, C is considered a low-level language because its computational model is close to the standard von-Neumann architectural model. The translation distance is relatively small compared to, say, the translation distance between Prolog and von-Neumann architectures. Since C is a low-level language (with respect to von-Neumann architectures) it is easier to translate a program to the hardware; translation is closer to a one-to-one mapping. One consequence of a more direct mapping is that there is a more predictable *cost model* for the language, which helps the programmer to understand the resource costs of their program. Another consequence of a more direct mapping is that a certain degree of performance can be achieved with relatively little effort in the compilation/interpretation step.

However, as a result of their proximity to a particular architectural model, lower-level languages often suffer from poor *portability* to other architectures and poor *scalability*, for example, in scaling from one to n processing units. Programs written in a language which is high-level and hardware agnostic can be easier to port to different architectures, and to scale to different resources and hardware configurations, because the programs have not yet made any commitment to any particular, concrete implementation.

Compared to a lower-level language, a higher-level language will likely offer various language invariants and program properties can be more useful for providing significant improvements in the efficiency of a program, even asymptotic changes in complexity. Infelicities of a language, or the presence of implementational details, may obscure program properties from the evaluation system, or even from the programmer. Program properties may be inferred via analysis, but often these properties are in general undecidable, requiring some form of safe approximation for the analysis. Approximations may result in too few optimisation and perhaps an unpredictable cost model, where the compiler becomes an impenetrable, petulant, and tempestuous black-box which the programmer must appease in order for optimisation to be granted.

A well-designed language might instead guarantee certain program invariants (for example purity, pointer safety, or no out-of-bounds access), or encode program properties in a clear way that is decidably inferred, thus guaranteeing efficient evaluation by applying certain optimisations. Using program properties and language invariants to provide more performant programs is an act of *reasoning* by the compiler/interpreter, thus we have begun to stray into the territory of the final aspect of the *four Rs*.

Reasoning

"In the long run it is not advisable to write large concurrent programs in machine-oriented language that permit unrestricted use of store locations and their addresses. There is just no way we will be able to make such programs reliable (even with the help of complicated hardware mechanisms)." *The Architecture of Concurrent Programs*, Hansen (1977) [5]

~

Reasoning incorporates many activities and can be implicit, as a kind of meta-process, facilitating reading, writing, and running. The ability of a programmer to reason about a program, to understand exactly what it does and how it does it, will affect the readability and writability of the language. The ability of a compiler/interpreter to reason about a program will affect the implementation provided and the kinds of optimisation that can be applied to improve resource usage.

Reasoning can also be a separate, intentional process, where a programmer explicitly reasons about the *correctness* of a program or about its *resource usage*.

Reasoning about program correctness has been at the forefront of computer scientist's minds ever since the very first programs [7]. There have been numerous costly blunders in the past that have starkly highlighted the importance of program correctness, particularly for safety critical applications (e.g. medical appliances, transport). Unfortunately, programming languages may obfuscate program (in)correctness; consider for example difficulties in reasoning about aliasing given a language with first-class pointers and arbitrary pointer arithmetic. As Hansen points out, these language features make it even harder to reason about concurrent programs, which are in high demand given the trend towards increasingly parallel architectures.

A well-designed language might prevent programmer's from writing programs with certain kinds of bugs. For example, static typing has been a very successful method of eliminating large classes of programming errors at compile-time (hence the slogan: "well-typed programs can't go wrong" [9]).

In some cases reasoning may be automated, but in others it may rely on human efforts. In both, the design of the language affects the amount of reasoning that can be reasonably performed.

As mentioned for *running*, compiler analysis is a form of reasoning on the part of the machine, where a program is reasoned about in order to decide how to efficiently execute the program. A language with few useful invariants requires a significant reasoning effort on the part of the compiler/interpreter in order to perform higher-level program transformations and optimisation. Since such reasoning will likely be undecidable, a decidable approximation is required, which may complicate the cost model of the language.

Of course, correctness, or having a good mental cost model, may not be primary concerns for some tasks e.g. rapid prototyping or educational programming languages. In this case, a language may be designed with a greater focus on other aspects, perhaps reading and writing.

An Example Consider a language with *static typing* and a *type safety* property – that a well-typed program will not produce any run-time type errors. Types are assigned to expressions, either through a type inference algorithm or explicitly by the programmer. A program will fail to compile if the constraints of types are not met.

Let's consider static typing in the rubric of the *four Rs*. Note, that this is *not* an argument *for* or *against* static typing, but is instead an evaluation process, considering the trade-offs between the different core aspects of languages, embodied in the *four Rs*. You can likely add your own further arguments to this list.

– *Reading* – Static type systems may include *type declarations* or *type signatures* which provide a form of documentation, aiding a reader's comprehension of a program. The amount of information conveyed in a type varies depending on the complexity of a type system. For example, some advanced type systems can describe a range of values within a type (see *refinement types*), thus providing even more detailed information to the programmer. However, as type systems become more complex the *type errors* that are produced become increasingly esoteric and hard to understand.

– *Writing* – The type of an expression can give a contract or specification from which to write. In some languages, for example Haskell, the type of an expression might uniquely determine its value (for example, there is only one function of type $\forall a.a \rightarrow a$, the identity function). However, static typing is necessarily an approximate analysis, thus some safe programs are excluded. As a simple example, consider the following:

  `if` ⟨complex-always-true-expression⟩ `then 0 else "hello"`

  This program will likely cause a type error under many static type systems, even though the false-branch is never taken at run-time.

  For rapid prototyping and fast exploration, static typing can be extremely inconvenient. Many people find it does not provide enough benefits to make it worthwhile, preferring instead the ability to write programs quickly without planning the types and data structures first. Systems such as *gradual typing* try to incorporate both the guarantees of static typing, with the ease of writing provided by dynamic languages.

– *Running* – Static typing allows run-time type checks to be removed, and by statically inferring the representation of a value various other kinds of storage or alignment optimisations can be applied. Further, as the types of operations are known, and are strict, the domains and ranges of computations can be matched, allowing equational rewriting to be safely applied.

– *Reasoning* – Static typing has proven to be an extremely successful form of verification in programming and can eliminate whole classes of run-time errors. Eliminating these errors, and restricting the kind of programs that can be written, greatly improves reasoning abilities for a language, and consequently greatly improves reading, writing, and running. By restricting the sets of values that a computation produces, a programmer has more information about what a program does, as various cases are statically ruled out.

Trade-offs and Domain-Specific Languages Since the *four Rs* are non-orthogonal, a careful balance must be struck, perhaps with certain inevitable trade-offs. A language design might focus on one, two, or three of the tenets, but we should be aware of the effect, positive or negative, on the other aspects.

For example, a high-level language might be easy to read, write, and reason about the correctness of, but could be difficult to run efficiently and predictably, thus difficult for the programmer to develop a good mental cost model for the language.

Additionally, a language might enhance some of the *four Rs*, but only for a small subset of programs. As discussed, certain classes of program are natural to express in a language, but another class of program might be equally unnatural. Language designers should be aware of this issue and consider for different classes of program which of *four Rs* the language promotes or inhibits.

In some cases it is appropriate to design a language to target a particular class of programs, providing a *domain-specific language* [13]. With a domain-specific language it is often possible to improve all of the *four Rs at once*, but only for a small class of programs. The last example here will be of a domain-specific language.

Another Example This example is from my own work on designing a domain-specific language, called Ypnos, for data-parallel array programming, mainly for scientific computing [10]. It is primarily a *pure* functional language with respect to input/output side-effects and state, which is an extremely useful language invariant for parallel programming. Secondly it has no general-indexing operation on arrays. Instead, *relative indexing* is provided by a special form of pattern-matching, called a *grid pattern*, that describes data access on arrays without any integer indices. Grid patterns provide variable bindings to values in an array, where the relative lexical position of variable binders to a current element binder determines to which element they are bound.

The following snippet defines and applies a function `f` in Ypnos, which is used to compute the average of an element and its neighbours:

```
f | _  t _ | = (t+l+c+r+b)/5.0
  | l @c r |
  | _  b _ |

arrayB = run f arrayA
```

The code between `f` and `=` is a grid pattern, where five variables, `t`, `l`, `r`, `b`, and `c` are bound. Underscore is a wild-card pattern.

The code is analogous to the following C code:

```
for (i = 1; i < (n-1); i++) {
   for (j = 1; j < (m-1); j++) {
     c = A[i][j];
     t = A[i][j-1];
     b = A[i][j+1];
     l = A[i-1][j];
     r = A[i+1][j];
     B[i][j] = (t+l+c+r+b)/5.0;
   }
}
```

Thus, `f` in the Ypnos program is a parameter to the higher-order `run` combinator, which applies the function at every index within the array. The `@` symbol in the grid pattern denotes which variable is to be bound to the *current* element (i.e. the index `[i][j]` in the C code). The indices of the other binders are calculated based on their relative position to the current element. For example, in this grid pattern, `t` is one line above the current element and is in the same column, thus it binds the element at index `[i][j-1]`; `l` is on the same line but in the preceding column, thus it binds the element at index `[i-1][j]`, and so on.

The only way to write elements to an array is via the `run` combinator, and a few other related combinators with a similar operation, which computes values for an entire array at once.

The language is very restricted, but within this domain it is powerful and effective as we'll see by considering it within the rubric of the *four Rs*.

– *Reading* – The grid pattern syntax makes it easy to read and understand programs which have much array-based data access, as code is free from indexing expressions which, due to their relative similarity to each other, can clutter code and confuse the reader. Ypnos programs are free from a commitment to a particular implementation, allowing the actual mathematical problem and solution to be focused on and understood, as opposed to the implementational details of a particular solution.

  One criticism of grid patterns might be that binding each element to a variable adds a layer of indirection, as a variable must be "looked-up" in the grid pattern to ascertain which element it binds to.

– *Writing* – The syntax of Ypnos is very small, with just a handful of combinators, the grid pattern syntax, and another special syntax for specifying boundary conditions. The grid pattern syntax is easy to write (up to a certain pattern size, perhaps greater than a 7×7 stencil is awkward), and there is less chance for making a mistake compared to using integer indexes on arrays. The language is hardware agnostic, and thus the programmer does not have to consider architectural or implementational details.

– *Running* – Due to the lack of side effects, and the lack of random updates, a parallel execution can be guaranteed. Due to the very restricted grid pattern syntax, the data access pattern of a grid can be statically determined, without any analysis other than parsing. This decidable, static data-access information is used to enforce the invariant that a well-typed Ypnos program cannot cause out-of-bounds errors thus run-time bounds checks can be eliminated [11]. Since parallelisation and bounds-check elimination is guaranteed by the language invariants, execution of Ypnos programs is not only efficient, but also predictable. The static grid pattern information also permits various other optimisations such as data layout optimisations.

– *Reasoning* – The lack of side effects, the grid pattern syntax, and the static type-system, provide the programmer with various correctness guarantees (including the out-of-bounds free property mentioned above). It is then easier to reason about the mathematical numerical correctness of the specific problem at hand as the language is free from other sources of error that might affect numerical accuracy.

By its restriction to a particular domain, Ypnos can improve the *four Rs* for structured, aggregate, array programming. If a programmer tries to write a program in Ypnos that is not within the domain they will quickly find themselves writing ugly code that is slow to execute. Thus the language gives a guide as to what is efficient: if it easy to write down, it will be efficiently executed.

More* R*s or* [A-Z]*s? There are surely many other important aspects of programming language design which have not been mentioned here which you might have in mind. The *four Rs* are sufficiently broad that other aspects fit within this framework, usually having an affect on, or relying upon, more than one of the tenets.

For example, *refactoring* was suggested to me as another possible *R*. Refactoring fits within the *four Rs*, providing a way to improve the readability of a program, and the writability of future programs through exposing common code. The ability of a language to be refactored relies on other properties of the language, for example, how easy is it reason about the equivalence of two programs?

Learnability – i.e. how easily, or how quickly, a language can be learnt – is another aspect of programming language design that has been suggested. Learnability fits into the *four Rs* as part of the ability of a programmer to read, write, and reason about their programs. A language that is hard to learn will be, at least initially, hard to read, write, and reason about.

I would encourage the reader to think of other useful aspects of programming languages and see how they fit into the *four Rs*.

Concluding Remarks

"Everything becomes clearer once you express it in the proper language." *Schild's Ladder*, Egan (2004) [3]

Language designers are forever searching for the "proper language" for a domain, task, or class of programs, such that everything – such as correctness, optimisations, abstractions, high-level structures, solutions to a problem, etc. – becomes clearer when the language is used. The *four Rs* capture the key aspects of programming language's affect on programs, in its ability to improve *reading*, *writing*, *running*, and *reasoning*.

These are four important areas that we need to consider when designing effective languages. The *four Rs* are not meant to argue whether a language is "good" or "bad", or to unilaterally promote one over the other. Instead, the *four Rs* provide a framework for thinking critically about the effectiveness of languages and language features.

Trade-offs between the *four Rs* have given a broad and beautiful spectrum of programming languages over the last 70 years. I await further languages with excitement and hope that with better languages everything might become clearer.

Acknowledgments I'd like to thank William Cook, Richard Gabriel, David West, and the rest of the participants at the Onward! 2011 Essay workshop for their advice, discussions, and feedback. Thanks are also due to Ellie Beagley, Robin Message, Alan Mycroft, Tomas Petricek, and James Willmoth for helpful discussions and comments on a draft of this essay, and to the Cambridge Programming Research Group for stimulating discussions.

Any remaining infelicities and errors are entirely my own.

References

[1] BROWN, R. Reference: in memorial tribute to Eric Lenneberg. *Cognition 4* (1976), 125–154.

[2] CORBI, T. Program understanding: Challenge for the 1990s. *IBM Systems Journal 28*, 2 (1989), 294–306.

[3] EGAN, G. *Schild's Ladder*. HarperCollins, 2004.

[4] GALT, J. *Lawrie Todd: Or the Settlers in the Woods*. Lawrie Todd: Or the Settlers in the Woods. Richard Bentley, 1832.

[5] HANSEN, P. *The architecture of concurrent programs*. Prentice-Hall, Inc., 1977.

[6] IVERSON, K. Notation as a tool of thought. *ACM SIGAPL APL Quote Quad 35*, 1-2 (2007), 2–31.

[7] JONES, C. The early search for tractable ways of reasoning about programs. *Annals of the History of Computing, IEEE 25*, 2 (2003), 26–49.

[8] MCCRACKEN, D. *Digital computer programming*. John Wiley & Sons, 1957.

[9] MILNER, R. A theory of type polymorphism in programming. *Journal of computer and system sciences 17*, 3 (1978), 348–375.

[10] ORCHARD, D., BOLINGBROKE, M., AND MYCROFT, A. Ypnos: Declarative, Parallel Structured Grid Programming. In *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming* (2010), ACM, pp. 15–24.

[11] ORCHARD, D., AND MYCROFT, A. Efficient and Correct Stencil Computation via Pattern Matching and Static Typing. *Electronic Proceedings in Theoretical Computer Science 66* (2011), 68–92.

[12] TIMBS, J. *The Mirror of literature, amusement, and instruction*, vol. 5. J. Limbird, 1825.

[13] VAN DEURSEN, A., KLINT, P., AND VISSER, J. Domain-specific languages: An annotated bibliography. *ACM Sigplan Nortices 35*, 6 (2000), 26–36.

[14] ZUSE, K. Über den allgemeinen Plankalkül als Mittel zur Formulierung schematisch-kombinativer Aufgaben. *Archiv der Mathematik 1*, 6 (1948), 441–449.