WPI-CS-TR-99-38 December 1999

Abstraction in Programming — working definition

by

John N. Shutt

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Abstraction in Programming — working definition

John N. Shutt
jshutt@cs.wpi.edu
Computer Science Department
Worcester Polytechnic Institute
Worcester, MA 01609

December 1999

Abstract

This paper investigates the nature of the process in programming by which a new *level of abstraction* is constructed by building upon an existing one. The process is called *abstraction*. The purpose of the investigation is to provide an informal characterization of abstraction, as a conceptual foundation for subsequent development of a mathematical theory of abstraction.

Contents

List of Definitions

1 Introduction

This paper is the first step toward developing a mathematical model of the process in programming by which a new *level of abstraction* is constructed by building upon an existing one. The process is called *abstraction.* The goal of the paper is to develop a more detailed characterization of abstraction — not a mathematically rigorous characterization, which this paper necessarily precedes, but an informal working definition and conceptual framework that, once established, could serve as a conceptual foundation for subsequent development of mathematical theory.

The expression "level of abstraction" has been around for some time. It has been in use in programming at least since the mid-1960's (e.g., [Land66]), well before "abstraction" became a prominent buzzword in programming language design.[1] Conceivably, "level of abstraction" may, like "bug" for a system flaw [Tenn91], predate computers entirely.

§2 provides a broad perspective on the treatment of *abstraction* in existing literature, both within and without computer science. §3 develops the proposed 'informal working definition' of *abstraction.* Concluding comments are made in §4.

2 Other senses of 'abstraction'

This section considers treatments of *abstraction* in previous (primarily academic) work, covering a wide range of subjects. It is intended to provide a fair sample of the existing literature, but is by no means a comprehensive survey, which given the amount of material involved was deemed impractical.

In study of the existing literature, the definition of the term *abstraction* can easily become a serious stumbling block. Researchers often either don't define the word, or worse, provide a definition that doesn't accurately reflect the way they actually use it. To combat this difficulty, the sampling in this section particularly emphasizes the differences between the various definitions, and meanings, of *abstraction.*

2.1 Metaphysics

The basic scholarly meaning of "abstraction" belongs to the philosophical subject of metaphysics, and is actually one of the primary meanings of the term in common English usage (in contrast to many philosophical terms that depart rather abruptly from everyday speech). An excellent articulation of the common metaphysical sense of the word occurs in the authoritative second edition of Webster's unabridged dictionary [Webs50]:[2]

[1] The apotheosis of "abstraction" as a buzzword dates approximately to the late 1970's.

[2] This definition is easily the most lucid and complete I have encountered. It dates back, with minor rewording, well into the nineteenth century, possibly far enough to have been written personally by Noah Webster. Evidence in the phrasing suggests direct study of the works of John Locke

> **Definition 2.1 Abstraction**: ... 3. *Metaph.* Act or process of leaving out of consideration one or more qualities of a complex object so as to attend to others. □

The entry goes on to elaborate that abstraction encompasses both the act of considering a single object independent of some of its qualities, and the act of considering a quality or qualities independent of any particular objects.

Some philosophers use the term *analysis* for consideration of an object independent of some qualities, reserving *abstraction* for consideration of a quality independent of any object [Abst11]. Webster's broader definition is more in keeping with the generality intended here.

2.1.1 Universals

Any philosophical discussion of abstraction is centrally concerned with *universals*. A universal is, in essence, an idea with some degree of generality. (Formal definitions of *universal* quickly get mired in philosophical jargon.) The nature of universals was the only aspect of metaphysics — other than the question of the existence of God — that got much attention in Europe during the middle ages. There are three schools of thought:

Realism is based on Plato, and says that universals are real. In its extreme form, it says that *only* universals are real, while material objects are *not* real.

Nominalism is the opposite of realism. Traditionally credited to William of Ockham, it says that universals have no existence.[3] In the extreme form of this theory (more extreme than, in particular, Ockham's position), material objects cannot be similar to each other; when several material objects are called by the same name, the only thing they have in common is that they are called by the same name.

Conceptualism is an intermediate ground, notably advocated by John Locke. It says that universals exist, but only as concepts.

2.2 Symbolic logic

Symbolic logic recognizes three abstraction operators, each of which binds free variables in the expression to which it is applied.

The *class abstraction operator* for variable x, denoted $\widehat{x}$, binds free variable x in a boolean expression, to denote the class of objects x for which the expression is true.[4]

---

and, possibly, Aristotle.

[3] Ockham wasn't the first advocate of nominalism, but he was a very effective one. Being very good at what he did has served his reputation in good stead over the centuries. He didn't invent the Principle of Economy (*entities should not be multiplied unnecessarily*), either, but he wielded it so incisively that to this day it is called Occam's Razor. (See [Copl63].)

[4] The notation used here for class and relational abstraction was adapted from Frege by Russell; the standard notation for function abstraction is traditionally credited to Alonzo Church, [Chur41]. The examples here are borrowed from [Quin47].

For example, $\widehat{x}(x \notin x)$ specifies the class of all objects x that are not elements of themselves.

The *relational abstraction operator* for variables x and y, denoted $\widehat{x}\widehat{y}$, binds free variables x and y in a boolean expression, to denote the relation consisting of pairs $\langle x, y\rangle$ for which the expression is true. For example, $\widehat{x}\widehat{y}(\exists z)((z \in \mathbb{Z}) \wedge (x \times z = y))$ specifies the set of all pairs of integers $\langle x, y\rangle$ such that y is a multiple of x.

The *functional abstraction operator* for variable x, denoted λx, binds free variable x in an expression, to denote the function whose value on input x is denoted by the expression. For example, $\lambda x(x^2)$ specifies the function that squares its input.[5]

2.3 Artificial Intelligence

In automated problem-solving, it is often desirable to simplify a given problem by temporarily disregarding some details. This technique, called *abstraction*, has been recognized for decades, and has actually been implemented in numerous systems; but only recently has a general mathematical foundation been proposed for it, as in (for example) [Giun92]. Similarly to the intent of the current work, [Giun92] begins by explicating the concept to be mathematized, and then uses that explanation as a blueprint in developing the mathematical model. The following definitions are summarized from that paper.

Definition 2.2 (AI, informal) Abstraction:

1. *The process of mapping a representation of a problem*, called (following historical convention) the "ground" representation, *onto a new representation*, called the "abstract" representation, which:
2. *helps deal with the problem in the original search space by preserving certain desirable properties* and
3. *is simpler to handle* as it is constructed from the ground representation by "throwing away details". □

Definition 2.3 (AI, formal) A formal system Σ is a triple $\langle \Lambda, \Delta, \Omega\rangle$ where Λ is the language, Ω is the set of axioms, and Δ is the deductive machinery of Σ. The set of sentences (theorems) that can be deduced from Ω via Δ is $\mathrm{TH}(\Sigma) \subseteq \Lambda$.

An abstraction, written $f : \Sigma_1 \Rightarrow \Sigma_2$, is a pair of formal systems $\langle \Sigma_1, \Sigma_2\rangle$ with languages Λ_1 and Λ_2 respectively and an effective total function $f_\Lambda : \Lambda_1 \to \Lambda_2$.

Abstraction $f : \Sigma_1 \Rightarrow \Sigma_2$ is *TI* (*Theorem Increasing*) iff $\alpha \in \mathrm{TH}(\Sigma_1)$ implies $f_\Lambda(\alpha) \in \mathrm{TH}(\Sigma_2)$; *TD* (*Theorem Decreasing*) iff $\alpha \notin \mathrm{TH}(\Sigma_1)$ implies $f_\Lambda(\alpha) \notin \mathrm{TH}(\Sigma_2)$; and *TC* (*Theorem Constant*) iff it is both TI and TD. □

[5] The use of a dot after the functional abstraction operator, although commonplace today, was reserved in [Chur41] as a shorthand for functions of multiple arguments. Thus $\lambda xy.M = \lambda x(\lambda y M)$.

These definitions simplify the problem of mathematizing an informally conceived process by separating it into two parts — a purely mechanical operation (Part 1 of the informal definition), and its motivation (Parts 2–3). In the formal definition, an *abstraction* per se models only the purely mechanical operation, and consequently needn't meet the motivational criteria. Aspects of the motivation are addressed by the subsequent definition of well-behavedness properties for instances of the operation (Theorem Increasing, etc.).

2.4 Program semantics

Full abstraction, in the realm of formal semantics of programs, refers in general to a mapping between semantic models that occurs without loss of information. This general description actually covers two slightly different uses.

1. When comparing programming languages, a *fully abstract translation* from language $\mathcal{L}_1$ to language $\mathcal{L}_2$ is a mapping of terms in $\mathcal{L}_1$ to terms in $\mathcal{L}_2$ that preserves partial ordering of terms based on their observable consequences. (For the rigorous mathematical definition, see [Riec91].)

   Conceptually, the existence of a fully abstract translation from $\mathcal{L}_1$ to $\mathcal{L}_2$ indicates that $\mathcal{L}_1$ is 'at least as abstract' as $\mathcal{L}_2$, since all behavioral information represented by terms in $\mathcal{L}_1$ is also represented by corresponding terms in $\mathcal{L}_2$.

2. When considering a single programming language, a denotational semantics of the language is *fully abstract* if, whenever two programs have distinct denotations, they also have distinct operational semantics [Schm86].

Another commonly used term, in both formal program semantics and applied program translation, is *abstract interpretation.* An abstract interpretation of a program p is a processing of p that determines some aspects of the semantics of p without performing all the details of the computation. The purpose of such an exercise is, in general, to answer decidable questions about the semantics of p without directly confronting the halting problem. Typical examples of abstract interpretation are *strictness analysis* (which identifies terms that definitely denote $\perp$) and *totality analysis* (which identifies terms that definitely do not denote $\perp$).

2.5 Programming language design

In the field most directly relevant to the current work, programming language design, the term *abstraction* is used with a fairly high degree of consistency in meaning. Most researchers would agree that the following are all examples of abstraction.

- Use of symbolic names for constants/variables.
- Definition of procedures and functions (*procedural* or *functional abstraction*).

- Definition of new data types, especially when packaged with associated operations (*data abstraction*).
- Definition of new control structures (*control abstraction*).[6]

There is some variation in usage between authors. For example, [Gabr89] adds to the list *syntactic abstraction*, by which is meant the use of syntax macros. [Guar78] remarks that built-in facilities of a programming language can be abstractions, whereas most authors only explicitly discuss user-defined abstractions. But minor variations aside, these four items form essentially the complete canon of abstraction in programming language design.

Many authors in this field who work extensively with abstraction also offer an explicit definition of the term. Here are some examples:

> Abstraction is the generalization from a collection of objects that all of the items in the collection share some properties that are important for a given purpose. — [Guar78, p. 4]

> Programming languages are notational systems that facilitate programming ... by providing concise notations ... together with facilities for defining new notation. We call these latter features *definitional mechanisms* or *abstraction mechanisms.* — [Hilf83, p. 1]

> ***Abstraction*** in programming is the process of identifying common patterns that have systematic variations; an abstraction represents the common pattern and provides a means for specifying which variation to use. — [Gabr89, p. 2-8]

The first of these definitions seems to be drawn from the metaphysical tradition (§2.1). The second is almost as general as the definition that will be proposed here in §3 (Definition 3.1). The third appears to *require* that all abstractions be parametric ("have systematic variations"). None of the three definitions fits particularly well with the four canonical forms of abstraction; the parametric definition from [Gabr89] doesn't even gracefully encompass the use of symbolic names, and *none* of the definitions offers any obvious reason for the canonical list to be exhaustive.

Yet in practice, all three quoted works actually use the term *abstraction* for essentially just the canonical forms. Evidently, the authors' understanding of the term was properly drawn from their experience of the canon, and the explicit definitions are merely explanatory (if not patently aesthetic) rather than predictive.

2.6 Comments

Recognizing the breadth of variation in definitions of *abstraction* is a great help to understanding the extant literature on the subject. But are any of those definitions

[6]Researchers surveying the subject usually mention control abstraction mostly to remark that it is grossly underutilized. See for example [Guar78, Gabr90].

relevant to the current general study of the construction of new levels of abstraction?

One tempting connection to programming practice that ought to be quashed sooner rather than later is the suggestive occurrence of the word *object* in Definition 2.1 (§2.1). Remember that that definition hails from metaphysics. Notwithstanding occasional claims by some of the more ardent OOP enthusiasts, the word *object* in modern programming is a specialized technical term for a particular kind of abstract data structure. It isn't at all the common English usage of the word, and shouldn't be expected to correspond too closely to that usage. If an analogy with OOP should eventually emerge as a natural feature of a general theory of abstraction, that would be an interesting result; but attempting to *build in* such an analogy would only compromise the generality of the theory. So for the moment, analogies with OOP will be avoided.

That said, the metaphysical sense of *abstraction* is still a tempting candidate for use in the current work. Of all the definitions enumerated above, Definition 2.1 comes closest to capturing what all the others have in common; presumably it is the ancestral meaning from which the others have evolved. As noted in §2.5, it *has* been proposed as a general definition for programming language design [Guar78].

It also isn't difficult to view some of the more common abstractive programming techniques as examples of the metaphysical definition. Enforced information hiding, in which some of the attributes of a module — or any other kind of program entity — are not available outside a certain scope, is evidently considering an object independent of some of its qualities. Any polymorphic use of data types/classes is, at least in most programming languages, considering qualities independent of any particular objects.

This sort of reasoning by cases, however, is a trap. Showing that the metaphysical definition is *consistent* with certain existing techniques doesn't detract from its credibility, but it can never positively support it, either. For a general theory of the subject, what is needed is a definition whose logical completeness is obvious, and from which specialized cases can be identified by logical partitioning, with no possibility that anything could have been omitted.

There is, moreover, a fundamental problem in applying the metaphysical definition to the general process of constructing levels of abstraction. Metaphysical abstraction is a process of leaving out information. How, then, can it describe a *constructive* process? One might argue that whatever is constructed was already implicitly present; but in that case the opposite problem occurs: If nothing new has been added, then it isn't clear than anything has been left out, either.

Another candidate that deserves some mention is that proposed for programming language design by [Hilf83] (see §2.5) — 'facilities for defining new notation' are called 'abstraction mechanisms'. This is, in a sense, the complement of the metaphysical definition: It readily encompasses the construction of new program entities, which the metaphysical definition stumbled over; but it doesn't clearly address the issue of information hiding, which was the metaphysical definition's particular forte.

Since we don't want a piecewise definition, we can't simply splice together the metaphysical and notational definitions. What is needed is a fresh approach — which is what §3 provides.

3 Abstraction in programming

This section develops a subjective model of abstraction in programming, as a yardstick against which the merits and demerits of rigorous mathematical models can be judged. Following a strategy akin to that noted in §2.3, the working definition itself addresses only the purely mechanical operation of abstraction in programming.

3.1 Working definition

Any source code element — say, a function declaration in an HLL (High-Level Language) — may be thought of as modifying the programming language in which it occurs. A standard programming language is, then, simply the starting point from which a source text diverges by a series of modifications. This view has been called, aptly, *programming as language development* [Chri88]. It is well represented in the literature of programming methodology, e.g., [Dahl72, Wino79], and forms the basis for an entire family of mathematical formalisms called *adaptable grammars* [Shut93, Part I].

Based on the principle of programming as language development, one might formulate a definition like this:

> **Abstraction**: The act or process of transforming one programming language into another. □

(This is, again, a purely mechanical characterization.)

This definition misses something of the flavor of abstraction, at least of the sort we have been considering. Abstraction such as the function declaration mentioned above is a transformation that occurs "from within". That is, the transformation uses facilities that were available in the programming language before the transformation. Do all abstractions (in programming) have this property? It will be argued later in the section that yes, they can be assumed to have, without loss of generality.

> **Definition 3.1 (in programming) Abstraction**: The act or process of transforming one programming language into another by means of facilities available in the former language. The two languages are called respectively the *domain* and *codomain* of the abstraction. □

Some readers may be initially disconcerted — the author certainly was, despite having years to mentally prepare himself — by the explicit depiction of a malleable programming language, constantly shifting at the whim of the programmer. It is more usual to think of a programming language as something vast and monolithic, the fixed medium of discourse for entire software modules comprising possibly tens or hundreds of thousands of lines of source code *each*. Defining a new one usually takes years of labor by a standards committee.[7]

One might conclude that the term "programming language" is being used in the definition with an unconventional meaning — in which case, some other term ought to be substituted that doesn't carry so much excess conceptual baggage. But in fact, the conventional meaning is exactly what *is* wanted, baggage and all. The malleable programming language only appears unfamiliar because we have been looking at it on an unfamiliar scale. Note that each abstraction is induced by a source text (the "means ... available in the [domain language]").[8] The domain language of the abstraction may be the output of a standards committee, or it may be the codomain of some other abstraction; but *either way*, it is the *fixed medium of discourse* for the inducing text — be it a dozen-line function declaration or a million-line module declaration. How monolithic this makes the domain language look will depend on the text length.

3.2 Incremental vs. radical abstraction

Discussions of abstraction in programming usually distinguish between *fixed* abstractions provided by the core language, and *user-defined* abstractions that must be constructed by the programmer; e.g., [Guar78, Hilf83]. The fixed/user-defined distinction per se is not suited to the current work, because it primarily concerns *when* abstraction took place — before or after the current programmer came into the picture — rather than *how* it took place, which is of principal interest here. An analogous distinction is made here between *radical* and *incremental* abstraction.

Definition 3.2 Incremental abstraction: An abstraction that modifies a programming language gradually, by means of selective changes. □

An HLL function declaration is an incremental abstraction; it modifies the language by adding a new operator, but the original language remains otherwise intact.

Definition 3.3 Radical abstraction: An abstraction that replaces one programming language with another wholesale, by means of explicit computation. □

[7] There may be a lesson here about doing things by committee, but that wasn't the point I was trying to make.

[8] For purposes of informal discussion, it will be assumed that the means of abstraction is a string of symbols. This simplification is unnecessary in general; however, symbol strings are the *canonical* means of abstraction, and alternative means will always have similar formal properties.

The canonical example of radical abstraction is an HLL compiler written in an assembly language, which radically abstracts from (apparently — but see below) the assembly language to the HLL.

Under Definition 3.1, the domain of an abstraction is the language whose facilities are actually used to accomplish the transformation. This didn't seem overly controversial in §3.1, because the only abstractions considered there were incremental. The domain and codomain of an incremental abstraction are self-evident (modulo we're working without mathematically rigorous definitions, of course).

Radical abstraction, however, poses an interesting problem.

Example 3.1 Suppose a compiler f, written in C, compiles Scheme programs into assembly language. Obviously, f is a radical abstraction — in fact, it is just the sort of thing that Definition 3.3 was *designed* for. But how, exactly, does this scenario square with the model of abstraction presented in §3.1?

Since the effect of f is to let the programmer write programs in Scheme, it appears (weasel words already!) that the codomain of f is Scheme. What, then, should be considered the domain of f? C? Assembly language?

This is, of course, a trick question: Not enough information has been provided. Before settling on an answer to the question, we will have to extend the basic assumptions of the scenario; and before we do *that*, we will first have to examine more closely the implications of the assumptions we have already made. ◇

3.3 Domain, codomain, and means

An abstraction should not necessarily be thought of as a function in the set-theoretic sense. Even if languages were to be formally defined as sets[9], the act of mutating one set to form another does not necessarily define a mapping from *elements* of the one to *elements* of the other. Nevertheless, conventional notation $f\colon A \to B$ will be used here to mean that an abstraction f has domain A and codomain B.

Section 3.1 claimed there would be no loss of generality by assuming that *every* abstraction is achieved "by means of facilities available in [its domain]". In the case of incremental abstraction, as already noted, the "means" is a string permitted by the domain language. Notation $\mathit{meansof}(f)$ will be used here to denote the text string that induces an abstraction f.[10] Evidently $\mathit{meansof}(f)$ is defined at least whenever f is incremental.

[9] Most definitions of programming language involve more internal structure than the word *set* entails. In the treatment of abstraction in [Shut99b], programming languages will in fact be defined as sets; but that exception will serve mostly to illustrate why it shouldn't be done.

[10] The assumption that $\mathit{meansof}(\cdot)$ is single-valued, i.e., a function, is mathematically useful independent of the current assumption that the means of abstraction is a symbol string. For example, there is no apparent difficulty in positing enough distinct abstractions to have one for each permissible string; but if it were desirable, for some reason, to associate multiple alternative strings with each abstraction, one could interpret this by saying that the means of abstraction is a set of strings.

Abstraction is composable. That is, if $f\colon A \to B$ and $g\colon B \to C$, it makes sense to talk about the composite abstraction $g \circ f\colon A \to C$. When g and f are both induced by text strings, $g \circ f$ is simply the effect on A of the concatenation of the source code elements associated with f and g; that is, $\mathit{meansof}(g \circ f) = \mathit{meansof}(f)\,\mathit{meansof}(g)$. Composition of abstractions is evidently associative.[11]

An abstraction f may be represented diagrammatically as an arrow, labeled by its inducing text, between two points representing its domain and codomain. The simple generic form of such a diagram would be

$$\begin{array}{c}\text{domain}\\ \text{language}\end{array} \xrightarrow{\quad \mathit{meansof}(f) \quad} \begin{array}{c}\text{codomain}\\ \text{language}\end{array}$$

Example 3.1 (continued) The source code for Scheme compiler f is written in C, and therefore induces an *incremental* modification to the C programming environment. Thus,

$$\begin{array}{c}\text{C}\\ \text{language}\end{array} \xrightarrow{\quad \langle\text{compiler source}\rangle \quad} \begin{array}{c}\text{modified C}\\ \text{language}\end{array}$$

Similarly, using compiler f, a Scheme programmer may construct a new program by incremental modification of the initial Scheme environment.

$$\text{Scheme} \xrightarrow{\quad \langle\text{program source}\rangle \quad} \begin{array}{c}\text{modified}\\ \text{Scheme}\end{array}$$

Obviously, the 'modified C language' node in the first diagram is not the same as the 'Scheme' node in the second; so somehow we have to get from one to the other. This missing bridge between the two nodes is qualitatively different from the abstractions shown in the previous diagrams: it's *radical.* Since radical abstraction f must be identified with some part of this path, and the C-to-Scheme bridge is conveniently available, we choose to call the bridge f.

$$\begin{array}{c}\text{C}\\ \text{language}\end{array} \xrightarrow{\quad \langle\text{compiler source}\rangle \quad} \begin{array}{c}\text{modified C}\\ \text{language}\end{array} \xrightarrow{\quad \mathit{meansof}(f) \quad} \text{Scheme} \xrightarrow{\quad \langle\text{program source}\rangle \quad} \begin{array}{c}\text{modified}\\ \text{Scheme}\end{array}$$

[11] A formal prerequisite for any means of abstraction will be that it have an associative binary operation, so that $\mathit{meansof}(\cdot)$ can map into it while preserving associative composition of abstractions. Note that sets of symbol strings, alluded to in the previous footnote, have associative binary concatenation.

The situation is strikingly suggestive of category theory. Programming languages are objects, abstractions are arrows between them. Arrows have associative composition, and each object has an identity arrow: the null abstraction, induced in the canonical example by a text string of length zero. $\mathit{meansof}(\cdot)$ is a functor.

This diagram supposes, of course, that *meansof* is defined on this f at all.

What does f accomplish? Intuitively, f should indicate that the C source code has come to an end, and then *execute* it — perhaps by compiling it and running the resultant binary image. In the real world, typically, these operations would be performed in the command shell of an operating system. This suggests a solution: Let string $meansof(f)$ belong to a meta-language, with its own inducing texts and abstractions, within which both C and Scheme (and, presumably, assembly language) are embedded.

Framing the entire example in the context of a meta-language might produce something like the following.

initial OS state	
	⟨command to compile following C source code⟩
C language	
	⟨C source code for Scheme compiler⟩
modified C language	
	⟨terminator for C source code⟩
second OS state	
	⟨command to compile following Scheme source code⟩
Scheme	
	⟨Scheme source code⟩
modified Scheme	
	⟨terminator for Scheme source code⟩
third OS state	

Figure 1: Meta-language for radical abstraction

The radical abstraction f from the previous diagram appears here as the composition of two abstractions, 'terminator for C source code' and 'command to compile following Scheme source code'.

It is now possible to answer the original question: The domain of f is the modified C language induced by the compiler source code. Ironically, though, that question is no longer particularly interesting; it's now dwarfed by issues of meta-language, semantics, and the relationship between abstraction and computation. □

3.4 Grokking abstraction

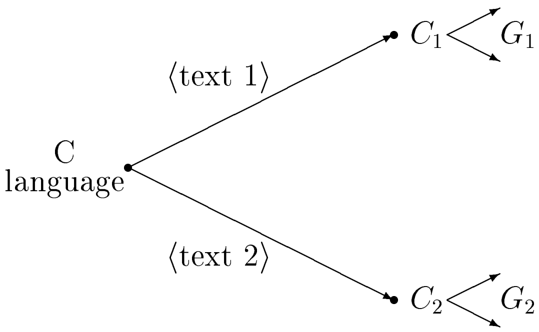
There are (as already implied) several lessons to be learned from the resolution of Example 3.1.

For one thing, the distinction between incremental and radical abstraction, which was originally framed in terms of intention ("selective changes" versus "explicit computation"), is closely related to semantic shifts between different conceptual levels of language. Such shifts of level also occur *within* a typical programming language as well, at levels ranging from vast modules and packages to small nested loops and conditionals. The obvious conclusion is that the distinction between incremental and radical abstraction may be merely one of subjective scale. A more subtle inference is that hierarchical ("context-free") phrase structure may play a pivotal role in abstraction, at least as it occurs in artificially constructed language systems.

Another important point concerns equivalence of programming languages.

Consider the (infinite) directed graph of all abstractions radiating outward from a central point — say, from the 'C language' depicted in Figure 1. Each legal C program text induces an abstraction from the root node to some modified C language. One such language is depicted in the Figure, along with a chain of four additional abstractions leading to further languages beyond it. There are also infinitely many other abstractions, and languages, branching outward from that modified C language; and the same must be true of other modified C languages in the graph.

Consider any two such modified C languages, C_1 and C_2. Each of C_1, C_2 may be the starting point for an infinite web of additional abstractions. Evidently, if some text t_3 induces an abstraction with domain C_1, but *does not* induce an abstraction with domain C_2, then these two languages cannot truly be said to have identical semantics, because there is an observable difference in their consequences. In fact, we have the following general principle: Let G_1 be the subgraph of all abstractions and languages reachable from C_1; and similarly G_2 from C_2.



Then languages C_1 and C_2 cannot have identical semantics unless G_1 and G_2 are isomorphic; that is, unless G_1 and G_2 have the same shape and the same inducing texts on corresponding abstractions.

This principle can be taken even further. From the example of compiler f, we already know that the meta-language of Figure 1 contains some information about the runtime behavior of C programs. So why shouldn't we choose the meta-language so that it contains *all* information about the runtime behavior of C programs? Interactive behavior, for example, can be described readily by interleaving substrings describing input and output; and so on. Isomorphism of reachable subgraphs may then be considered, in principle, a *sufficient* as well as necessary criterion for the semantic equivalence of programming languages.

The $meansof(\cdot)$ operator already implicitly casts abstraction as a purely syntactic phenomenon.[12] What has now emerged is that, by doing so, we do not have to give up on semantic abstraction — because we are now in a position to view semantics itself as a purely syntactic phenomenon.

The dividing line between syntax and semantics has always been a sticky issue in computer science. Between context-free syntax and run-time semantics, there is a no man's land sometimes called *static semantics*.[13] Programming language abstraction issues usually fall squarely into no man's land.

What we are doing here, in effect, is consigning to syntax not only the disputed territory — no man's land — but the entire battlefield.

In retrospect, one can see that this outcome was inevitable. A theory of abstraction that applies only to syntax, or only to semantics, would have to be judged profoundly incomplete; and a piecemeal theory that handles the two cases separately must always be suspected of lacking generality. A single uniform theory requires a single uniform view of the subject matter. That the uniform view appears rather syntactic is natural but not really partisan; it only explicitly acknowledges the metalanguage that semanticists use to address their subject. (If anyone can rightfully claim victory here, it is the semioticians.)

4 Conclusion

This paper has constructed an informal model of the process by which new levels of abstraction are built from old ones.

The problem of construction has been simplified by separating the nuts-and-bolts view of abstraction as a mechanical operation, from higher-level motivational issues. The current work focuses on the mechanics of abstraction. Abstraction is defined as the transformation of one programming language into another by means of facilities available in the domain language (Definition 3.1). Each complete system of abstractions is modeled as a directed graph (or, more precisely, a category) whose nodes are languages and whose edges are abstractions.

On closer consideration of the mechanical model, it appears that in practice (on artificially constructed languages) there is a definite hierarchical structure to the abstraction process. By extending the hierarchy upward, the entire semantics of arbitrary computations can be captured within the model, so that abstraction ultimately subsumes computation.

The purpose of the informal model is to clarify what it is that a formal Theory of Abstraction should capture. The next step will be to identify particular features that

[12]The syntactic flavor of $\mathit{meansof}(\cdot)$ is partly an artifact of the current emphasis on symbol strings; but since all means of abstraction must have a concatenation-like operation, the impression of syntax will tend to linger on into more general mathematical treatments.

[13]Critics maintain, in polite or not-so-polite phrases, that only an ignoramus would mistake context-dependent syntax for any kind of semantics. For a taste of the debate, see [Meek90]. The simple misunderstanding that ultimately causes the debate is neatly cleared up in [Knut90].

a formal mathematical model must have in order to capture various aspects of the informal concept. This agenda will be pursued, at least initially, in [Shut99b], which will investigate the merits and demerits of a formal theory of abstraction in which programs are modeled as flat strings of symbols.

References

[Abst11] "Abstraction", *Encyclopedia Britannica*, 11th edition, 1911.

This entry is about abstraction in philosophy. Corresponding entries from the early 1970's are primarily concerned with symbolic logic; but in 1911, the apotheosis of symbolic logic was still in the future.

[Chri88] Henning Christiansen, "Programming as language development", *datalogiske skrifter* no. 15, Roskilde University Centre, February 1988.

Christiansen's licentiat thesis (equivalent to Ph.D.) is made up of this and six other separate papers. This is the one that ties all of them together.

[Chur41] Alonzo Church, *The Calculi of Lambda-Conversion*, Annals of Mathematics Studies, Princeton: Princeton University Press, 1941.

77 thrilling pages. Heavy reading, but could be much worse.

[Copl63] Frederick Copleston, S. J., *A History of Philosophy*, Volume III: *Ockham to Suárez*, Westminster, Maryland: The Newman Press, 1963.

[Dahl72] O.-J. Dahl and C. A. R. Hoare, "Hierarchical Program Structures", in O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming* [*A.P.I.C. Studies in Data Processing* 8], Academic Press, 1972, pp. 208–220.

The notion of programming as language development is recommended.

[Gabr89] Richard P. Gabriel, editor, "Draft Report on Requirements for a Common Prototyping System", *Sigplan Notices* 24 no. 3 (March 1989), pp. 93ff.

The draft report has its own internal pagination separate from that of the issue; only the first page of the report also has issue pagination (as page 93).

[Gabr90] Richard P. Gabriel and Guy L. Steele Jr., Editorial: "The Failure of Abstraction", *Lisp and Symbolic Computation* 3 no. 1 (January 1990), pp. 5–12.

Quotes a lengthy (nearly 200 word) definition of abstraction from [Gabr89]. Lists three advantages of abstraction: (1) abbreviation, (2) opacity, and (3) locality. Details nine failures of abstraction.

The first six are "failures of human spirit to push the concept of abstraction to its maximum extent": (1) During modification, locality disappears. (2–4) Lack of control, communication, and process abstractions. (5) These omissions cause programs to be written at varying levels of abstraction. (6) Failure to abstract over time.

The last three are "failures of the people who design and use programming languages": (7) Lack of support for documentation and other mechanisms for learning about an abstraction. (8) Abstractions are often selected for performance reasons. (9) Difficulty of reapplying abstractions to new problems.

[Giun92] Fausto Giunchiglia and Toby Walsh, "A theory of abstraction", *Artificial Intelligence* 57 nos. 2–3 (October 1992), pp. 323–389.

Proposes a simple and elegant mathematical foundation for abstraction in AI.

[Guar78] Loretta Rose Guarino, "The Evolution of Abstraction in Programming Languages", *CMU-CS-78-120*, Department of Computer Science, Carnegie-Mellon University, Pittsburg, Pennsylvania, 22 May 1978.

One of the few works on programming abstraction as a whole that is not blatantly biased toward one particular programming language. The bibliography is helpful in identifying relevant commentary by notables in the field.

The author has published in recent years under the name Loretta Guarino Reid.

[Hilf83] Paul N. Hilfinger, *Abstraction Mechanisms and Language Design*, Cambridge, Massachusetts: The MIT Press, 1983.

The Ada programming language is the setting for, and focus of, this treatise. Winner of the 1982 ACM Distinguished Dissertations Award. Originally presented as the author's thesis — CMU, 1981.

[Knut90] Donald E. Knuth, "The Genesis of Attribute Grammars", in Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Ap-*

plications [*International Conference WAGA*] [*Lecture Notes in Computer Science* 41], New York: Springer-Verlag, 1990, pp. 1–12.

For the reader interested (as myself) in the evolution of concepts in grammars and programming language design, this paper is a treasure.

The entire dispute over "static semantics" is explained away so quickly — without ever even mentioning it by name — that it makes all the egregious polemics on the subject look deservedly ridiculous.

[Land66] P. J. Landin, "The Next 700 Programming Languages", *Communications of the ACM* 9 no. 3 (March 1966) [*Proceedings of the ACM Programming Languages and Pragmatics Conference*, San Dimas, California, August 8–12, 1965], pp. 157–166.

[Meek90] Brian Meek, "The Static Semantics File", *Sigplan Notices* 25 no. 4 (April 1990), pp. 33–42.

Although this article was, in part, an invitation for followup letters to the editor, I can only find two such responses. This may be because the original article was so passionate, despite a token disclaimer about objectivity. On a related note, Meek expresses the bizarre opinion that the idea that van Wijngaarden grammars are difficult is bizarre.

The first response is from David Gries (25 no. 7, July 1990), who embraces Meek's opinion that the term "static semantics" is an abomination, repeats the basic elements of sloppy thinking in Meek's paper, and expresses great relief at discovering that perhaps he (Gries) was not personally responsible for visiting the hideous and revulsive term "static semantics" upon the unsuspecting and defenseless world.

The second response doesn't even relate to static semantics. M. Douglas McIlroy (25 no. 12, December 1990) corrects Meek by pointing out that the `++` operator in C is not a consequence of the PDP-11 architecture at all, but actually predates it (through the ancestor language BCPL).

[Quin47] W. V. Quine, *Mathematical Logic*, Harvard University Press, 1947.

[Riec91] Jon G. Riecke, "Fully Abstract Translations between Functional Languages", in *POPL '91*: *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 21–23, 1991, pp. 245–254.

[Schm86] David A. Schmidt, *Denotational Semantics: A Methodology for Language Development*, Boston: Allyn and Bacon, 1986.

Keep your eyes open when using this book, because it contains many, many mathematical errors.

[Shut93] John N. Shutt, "Recursive Adaptable Grammars", M.S. Thesis, Worcester Polytechnic Institute, Worcester, MA, 10 August 1993; Emended 1998.

Part I is a survey of other grammar formalisms, and I recommend it. Part II, about RAGs, is a old formulation, now superseded by [Shut99a].

[Shut99a] John N. Shutt, "Recursive Adaptable Grammars", technical report WPI-CS-TR-99-03, Worcester Polytechnic Institute, Worcester, MA, January 1999.

[Shut99b] John N. Shutt, "S-Expressiveness and the Abstractive Power of Programming Languages", technical report WPI-CS-TR-99-39, Worcester Polytechnic Institute, Worcester, MA, December 1999.

[Tenn91] Edward Tenner, "Revenge Theory", *Harvard Magazine* 93 no. 4 (March–April 1991), pp. 27–30.

[Webs50] *Webster's New International Dictionary of the English Language*, second edition, unabridged, 1950.

The authority of Webster's Second on American English is comparable to that of the (original) Oxford English Dictionary on the English variety of the language. Note that Webster's Third is, to say the least, not a worthy successor. Would you trust a dictionary that defines "infer" and "imply" as synonyms?

[Wino79] Terry Winograd, "Beyond Programming Languages", *Communications of the ACM* 22 no. 7 (July 1979), pp. 391–401.