In Search of Types

Stephen Kell

Computer Laboratory, University of Cambridge
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
firstname.lastname@cl.cam.ac.uk

Abstract

The concept of "type" has been used without a consistent, precise definition in discussions about programming languages for 60 years.[1] In this essay I explore various concepts lurking behind distinct uses of this word, highlighting two traditions in which the word came into use largely independently: engineering traditions on the one hand, and those of symbolic logic on the other. These traditions are founded on differing attitudes to the nature and purpose of abstraction, but their distinct uses of "type" have never been explicitly unified. One result is that discourse *across* these traditions often finds itself at cross purposes, such as *overapplying* one sense of "type" where another is appropriate, and occasionally proceeding to draw wrong conclusions. I illustrate this with examples from well-known and justly well-regarded literature, and argue that ongoing developments in both the theory and practice of programming make now a good time to resolve these problems.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]; K.2 [*History of Computing*]

Keywords data types, abstraction, type system, type discipline, information hiding

1. Introduction

I feel uneasy when I hear the word "type". Most of us use it rather a lot, and most of us are aware that it can mean several different things. For lack of a better-established alternative, we muddle through. We can usually tell what each other are talking about. But this is, at the very least, philosophically unsatisfying. Where did "types" come from? What are they? What are they useful for?

The word "type" crops up repeatedly in literature about everyday programming practice, and also in some of the most deeply theoretical work on programming languages. It is therefore tempting to believe that there is a deep connection between apparently distinct uses of the word, such as "data types" (abstractions of information as it is manipulated during program execution) and "types" of expression (logical predicates over expressions, used to reason about code before execution). Such a deep connection would justify the Humpty Dumptyish[2] attitude that many take, my sometime self included, of believing that we can be liberal in our use of "type" because it always means what we intend it to mean. But over time, from reading and talking to other researchers, I have slowly come to believe that the connection is much flimsier than I had previously thought. It seems more as though a convenient pun—the sharing of the word "type" between logic, everyday English, and programming—has allowed us to avoid articulating exactly what we mean.

The last 40 years have seen an impressive confluence of theory and practice in programming language design, with the interesting side-effect of taking the sense of "types" originating in symbolic logic and implanting it into engineering traditions. This has overloaded what was already an ambiguous term, and opened up more subtle kinds of confusion arising from the fact that logicians and engineers hold differing latent assumptions about what is fundamental versus what is incidental. This includes attitudes to *cost* and *guarantees*, to *abstraction* generally, and to *types* specifically as they relate to all this. On the surface, it seems everybody is writing about "types", but as we will see, a careful reading reveals how these different attitudes are responsible for various apparent disagreements about in what ways types are useful, essential, harmful or irrelevant.

[1] Any similarity to the opening of a paper by Parnas et al. [1976] is entirely intentional.

[2] In Lewis Carroll's *Through the Looking-Glass*, Humpty Dumpty addresses the question of what words mean in the following way. "When I use a word, it means just what I choose it to mean—neither more nor less."

This is the author's version of the work. The definitive version was published in Onward! '14.

, .
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3210-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/2661136.2661154

In outline, my argument will proceed as follows.

Firstly, I will survey the crowded field of distinct notions of "type" and their underlying purposes in language design. It will be necessary to consider what the essence of a "data type" is (§2), what we mean by "abstraction" (§3), including in what senses abstractions may be protected or violated (§4), and how the concept of "types" from logic relates to these (§5). Most readers will be comfortable with many of the distinctions that I highlight, but I hope that by enumerating them as fully as I can, there will be something of surprise or novelty to most readers.

Secondly, I will show that this overcrowding leads to *overapplication*: we consider one sense of "type" and mistakenly attribute to it some properties that actually derive from another, or from something that does not really concern "types" in any sense. I will review some instances of this in well-known literature (deliberately choosing examples that are justly well-regarded—§6, §7), another in an informal yet influential context (§8), and a collection of smaller observations (§9).

Thirdly, I will survey the history of the word "type" in programming through the last 60 years (§10) and its earlier origins in logic. As I will show, even within computer science, the discourse can be divided into two clear threads, which I call (for want of better terms) the *engineering tradition* and the *logic tradition*. The surprising finding of this survey, at least to its author, is that the issue of unifying the various meanings of "type" is rarely tackled, and, in particular, the sense of the logic tradition appears never to have been explicitly reconciled with that of the engineering tradition.

We begin with a question in the sphere of programming (and not, for now, of logic): what, essentially, is a *data type*?

2. The essence of "data types"

Most practitioners use "type" interchangeably with "data type". In this section I search for an essential property behind this notion, and consider how it has been extended to various practical ends in the sphere of programming.

Information theory gives us a model of communication as transmitting symbols drawn from an alphabet. The selection of a specific symbol, from among the multiple possibilities defined by the alphabet, conveys information [Shannon and Weaver 1949]. Programming fits this model too, since the process of evaluating a program consists of flows of information—often called operations, transitions or reduction steps. In this context, we prefer to call symbols *values* and alphabets *data types*. (That is not to deny that, as we will see, a data type may be defined in terms more abstract than an alphabet of symbols.)

At least two languages which most computer scientists have heard of, and which have seen considerable use, have no notion of *data types*, whether built-in or user-defined. These are the Unix shell [Bourne 1978] and BCPL [Richards 1969].[3] In both languages, a single alphabet implicitly pervades all programs; all values are instances of a single data type. In the Unix shell, this is the character string. In BCPL, it is the machine word. Both languages consequently lack four distinct features, each identifiable as a *role* of "data types".

1. **Named interpretations**—some languages offer a linguistic mechanism for referring to different sets of meanings that a value might have. For example, some language construct might afford the opportunity to say "integer" in one instance and "real" in another. All languages that feature data types have one or more such construct, but neither BCPL nor the shell has any. In languages that do, the role of "integer" and "real" is to identify explicitly, within the program, different higher-level meanings, or *interpretations*, of some symbols. (The inverse of interpretation is *representation*: a symbol represents some higher-level meaning.) "Integer" and "real" may be primitives, or may be defined explicitly within the language, but they necessarily give an explicit identity to an interpretation. I call this kind of language feature "named interpretations". Note that this isn't intended to exclude apparently nameless ways of doing the same job. One language might denote a pair of integers by `pair<int>`, another by (,); I will say both expressions "name" the same interpretation.
2. **Storage contracts**—some languages allow declaring particular storage (say, a variable) as holding values admitting a particular interpretation.[4] We might want to do so, to signify that only symbols (bit-patterns) representing integers will be stored there, and that consequently its contents can be meaningfully interpreted as an integer. I call this kind of *quid pro quo* a *storage contract*. Storage contracts include any facility under which values can be declared as, or tagged with, or otherwise restricted to a named interpretation which bounds what it may (contractually) represent; we then often say the value is "typed".[5] Neither BCPL nor the shell supports this; we can't declare a variable "as" an integer, say. Such facilities are precluded in BCPL and the Unix shell because the language does not fill role 1: we can't name the integer interpretation.
3. **Operational well-definedness over storage**—in some languages, an operation generates a run-time error[6] if its input values are governed by inappropriate storage con-

[3] We could also include assembly languages, since they share all the relevant properties of BCPL.

[4] If "storage" seems too low-level, we could instead say "bound values".

[5] Note that contracts have only the property that *if* they are upheld consistently, *then* some good outcomes result. The extent to which they *are* upheld or enforced varies considerably. We revisit this in §4.

[6] Here "run-time error" includes untrapped errors, like "undefined behaviour" in C, which cause future execution to behave arbitrarily.

tracts. This can be called a "type error". By contrast, neither BCPL nor the Unix shell defines such error conditions, since they have no storage contracts. Note, conversely, that BCPL and the shell (like all languages) feature operations whose definedness is predicated on *other* properties of their input values, besides storage contracts. For example, integer division is defined only for a non-zero divisor, while memory deallocation is defined only for a previously allocated address. In such cases, the associated error conditions are not considered "type errors".

4. **Semantic well-formedness**—in some languages an expression may be considered ill-formed (creating an error *before* execution) according to an analysis of what values it might compute. Neither BCPL nor the shell includes any such analysis. A popular use of such analyses is to ensure the absence of run-time type errors (as defined in the previous paragraph), by including a proof system in the language and requiring that well-formed programs embody checkable proofs of this absence. The properties proved are of a single form: that a computed value always satisfies a given contract, stated as a data type. So it makes sense to call the proof-checking "type checking". (Recall that at the start of this section, we assumed a context where *type* and *data type* are equivalent. "Type checking" must therefore check properties to do with data types. But in other contexts, such as if we were using "type" in the sense from logic, other properties would be in scope; we return to this in §5.)

What are the relationships between the four roles we just identified? Most importantly, each builds on its predecessor. Note also that all roles can be filled to greater or lesser extents. Role 1 might be satisfied by offering an expressive language of data types, or by just a handful of built-ins. Role 2 might be satisfied by very fine-grained storage contracts (like C++, where the structure of objects is fixed at allocation time), or by just in a few places (like Python, where the only kind of contract is an object's class, and there are no restrictions on what fields or local variables may refer to). Role 3 might rule out many operations (as with Pascal) or only a few (as with C). Role 4 might demand proof of the absence of *all* run-time type errors (like ML), most (like Java with generics) or a limited selection (like pre-generics Java).

Instead of saying that a language is "typed", perhaps with a qualifier such as "strongly" or "weakly", it is more meaningful to characterise it along all four dimensions. How expressive is it regarding data types? What kind of storage contracts it can express? How strict or permissive are its operational well-definedness rules (and are any error cases trapped or untrapped)? How completely do its well-formedness criteria prove the absence of errors under these rules? The phrase "type system" is frequently used to refer to the language design details supplying answers to any and all of these questions. But they are all clearly separate, and merit careful distinction. I will therefore avoid saying "type system" (but return to the phrase itself in §5).

The popular labels "statically typed" and "dynamically typed" fall out of the third and fourth roles, as specific regions in the design space. Under a "dynamically typed" language, no well-formedness criteria of this kind are imposed (role 4), but all run-time errors are trapped (role 3). By contrast, "statically typed" languages necessarily impose a non-empty set of well-formedness criteria, but may choose to trap remaining erroneous operations (as with "safe" languages like Java) or neglect to do so (as in C).[7] These two labels are unsatisfying: they are clearly not opposites, and imply a dichotomy where at least two distinct continuums are apparent (in *how strong* the criteria are, and separately, *how completely* the remaining errors are trapped). Moreover, as we will see, it is conceivable for a language to be neither statically nor dynamically typed, yet still to feature data types.

In the rest of this section I argue that the first role is the essence of data types—the one without which we couldn't claim a language to feature data types at all. To do so, we will consider an (imaginary) extension to BCPL which fills this role, and *only* this role, finding that it is useful in isolation.

There is a latent idea of data types in all computation, since in any language we can distinguish between values, hence carve out distinct subsets of an alphabet. We can use composition of multiple values to create more complex alphabets, like tuples or lists. To do these things in BCPL we would write procedures that distinguish different integers or combine multiple integers, creating implicit *encodings* of these more complex concepts. Even functions in BCPL may be viewed as integers, i.e. as lists of machine instructions.[8] But in BCPL or the shell, these encodings are described only implicitly in procedures, not in an explicit definition. Languages supporting "data types" make these definitions *explicit* by providing some specialised construct (often a declarative notation), enabling certain benefits in return.

BCPL is an ancestor of C, and a spiritual descendent of Algol. Suppose we want to compute the Manhattan distance[9] between two points in the (x, y) plane. A BCPL procedure for doing so is the following. We assume our points' coordinates are restricted to non-negative integers.

```
LET manhattan (x1, y1, x2, y2) = VALOF
$(
    RESULTIS abs(x1 − x2) + abs(y1 − y2)
$)
```

Let's devise a simple extension of BCPL, and call it "BCPL with data types". To mark the change, we also switch to a (more familiar) C-like syntax. But our language will be very different from C! We will fill role 1, but not roles

[7] We use "safe" in the sense of [Krishnamurthi and Felleisen 1999], summarised as "unsafe programs don't signal errors; safe programs do".

[8] The fact that many interesting relations on functions are not computable, such as behavioural equivalence, does not detract from this point.

[9] This is the sum of distances in the x and y directions—the distance one would walk in a grid pattern of streets.

2, 3 or 4 (noting that C exhibits all of these, to varying extents). In particular, like BCPL, we will allow variables to be introduced only as uninterpreted words of memory. Unlike BCPL, we will be allowed to name different interpretations that can be made of bit-patterns in the language, like int, float, pointer, etc.. We may as well go a little further and also allow the user to define their own data types (structs, unions, enums, distinct kinds of function, etc.). But we maintain the key restriction that data types never attach to stored (bound) values, nor to expressions. We may only *refer to* data types when specifying operations.

What use are data types in such a language? Like before, let us define a Manhattan distance function in terms of three other operations: $+$, $-$ and abs. Unlike before, we may define a data type to abstract two-dimensional points; call it point2d. To keep things simple, we ensure that every value is one 64-bit word in size, including instances of point2d. It should then be obvious that, like in BCPL, a compiler needn't reason about data types to generate code—each intermediate result is a single word in size.

```
// point is a word, decomposed into two fields of 32 bits
struct point2d { x:32; y:32; };

manhattan(p1, p2)
{
  return abs(((point2d) p1).x − ((point2d) p2).x)
        + abs(((point2d) p1).y − ((point2d) p2).y);
}
```

Our member selection operator, ".", projects out the bits corresponding to the named field and right-shifts them to the least significant position. This operator gives us an *abbreviated* way to say "select the {low, high} 32 bits" from a word representing a point, using its left-hand subexpression to *name* a data type, here point2d.

What if we wanted a version that worked for floating-point coordinates? In BCPL[10] every arithmetic operator is duplicated: those for integer bit-patterns use the usual symbol, and those for floating-point bit patterns are prefixed with #. Assuming that floating-point operations work at single precision, on the lower 32 bits of a word, we write the following.

```
struct point2df { x:32; y:32; };

manhattan_float(p1, p2)
{
  return #abs(((point2df) p1).x #− ((point2df) p2).x)
       #+ #abs(((point2df) p1).y #− ((point2df) p2).y);
}
```

Note that we did not strictly have to define a new data type for points. Our old point2d would suffice, since its fields can contain any 32-bit pattern. However, it was helpful to do so, since the added redundancy serves as documentation: the new name point2df reminds the reader that floating-point coordinates are being used. Similarly, we could have mentioned data types int and float when declaring fields x and y; we avoid this (to avoid inventing extra syntax).

[10] The core BCPL language has no floating-point arithmetic; we follow the conventions used by the "extended" xbcpl and certain other variants.

At this point, our language is starting to creak a little. The ad-hoc distinction between floating-point (#-prefixed) and integer (unprefixed) operators is inadequately extensible, since our language now includes many data types. We can solve this by generalising from the "#" syntax to one which references data types by name. We might end up with code like the following, in a language we'll call BCPL++.

```
struct point2d { x:32; y:32; };
manhattan<int>(p1, p2)
{
  return    abs<int>(((point2d) p1).x −<int> ((point2d) p2).x)
     +<int> abs<int>(((point2d) p1).y −<int> ((point2d) p2).y);
}
manhattan<float>(p1, p2)
{
  return      abs<float>(((point2d) p1).x −<float> ((point2d) p2).x)
     +<float> abs<float>(((point2d) p1).y −<float> ((point2d) p2).y);
}
```

Like in ordinary BCPL, we are still in charge of managing data representations across data movements, such as parameter passing or assignment. We can pass a float bit-pattern where an integer is wanted; conversions are the programmer's responsibility. In plain BCPL, these are the fix operator and its inverse float. In BCPL++ we can do better by grouping such functions in a family convert<*from_t*, *to_t*>, encompassing sign extension (a.k.a. width conversion on signed integers), other arithmetic conversions (like between integer and floating-point), and so on.

The addition of explicitly named (and, in our examples, user-defined) data types to BCPL has clearly helped us somehow. A large part of this help has been in a *documentary* role: by talking about data types, we can explicitly group related things together (like our different manhattan functions), and distinguish distinct things in our code (like our point2df versus point2d example). We have also *abbreviated* the code's expression of certain details, e.g. by avoiding the need to repeat which bits were allocated to our point's x and y fields (though not without adding some syntactic overhead in other places). Note that the program is operationally identical to what we would write in BCPL; there are no more error cases (especially not "type errors"). It defines no storage contracts, so cannot use them to define operational well-definedness criteria. Nor does it define any semantic well-formedness criteria on the basis of what values an expression computes.

Unsurprisingly, the language retains several weaknesses of the original BCPL. It is needlessly hamstrung by the constraint that each operation yield a single word (which had already forced our rather improbable choice of 64-bit words but 32-bit floating point). It is needlessly explicit: we select operations explicitly (like abs<int>), size storage explicitly, and handle conversions explicitly during data movement, when much of this could clearly be automated. It is consequently needlessly repetitive "in the small": in our manhattan function we repeat int many times, to select within the same related family of operations. It is needlessly error-prone: we can easily select the wrong operation, or omit to insert a conversion, and hence obtain meaningless output—whereas we might rather have the program enter a self-explaining error state, or to be rejected before execution. It is needlessly repetitive "in the large"—our two manhattan functions turned out to be almost identical!

BCPL++ brings a small cost of some added syntactic overhead, since our angle-bracket syntax was a bit more verbose than the original # syntax. However, it also brings several benefits. Being more verbose has made it more readable. Naming data types helps document code. Also, named definitions *within* data types themselves improve on numbered byte- and bit-ranges, as with our point2d and its named fields x and y, since the space of names is more cognitively agreeable, and also simply *larger*, than the space of bit-ranges. It therefore conveys more information to the programmer; it is useful even though this information is irrelevant to execution (so is discarded by the compiler).

Adding data types has also made various solutions to BCPL's remaining problems much more apparent (indeed, *so* apparent that no language like BCPL++ has ever existed).

By constraining each *expression* to yield a value of fixed data type and fixed size, we allow the compiler to *automate* tasks that previously fell to the programmer, such as selecting arithmetic operators and conversion operators. Alternatively, we can automate a different way: by omitting this constraint, but looking up these routines dynamically, using *tags* which represent instantiated data types within a run-time system.[11]

By somehow constraining how we program, we could possibly enforce the property that when a bit-pattern *representing* an instance of some data type is created, it can (provably) only ever be consumed by code which *interprets* it under that same data type. Alternatively, we could avoid this constraint, and apply dynamic checks that use tags to check this property (albeit *during* rather than *before* execution).

By exploiting latent *similarity*, e.g. between our two manhattan functions, we could find ways of programming with greater *generality*, such that only one such function need be defined—perhaps by generating code for each kind of data item that manhattan is well-defined over, or perhaps by generating one version of code that can bind dynamically to any such data.

From the foregoing, it should be apparent that the act of naming has provided independent value, both in *documenting* and (in some sense) *abbreviating* our programs. Meanwhile, run-time versus compile-time implementation choices are pervasive, yet are clearly orthogonal to the goal being pursued, modulo the essential difference of earliness (ahead of execution) versus lateness (during execution).[12] Furthermore, automation, generality and checking are all distinct goals—notwithstanding that a *design synthesis* might allow a given language to elegantly tackle more than one at once.

Philosophically, the act of *naming* is often called "denoting" or "reference". The essence of data types is as named interpretations to which we can *refer*. In this sense data types resemble Plato's Forms: they are concepts which recur across multiple places in our program, existing outside the context of any one execution or any one operation.

At this point it is fitting to reveal that BCPL—a self-identifying "typeless" language—should nevertheless be defined in a report which states the following.

> An Rvalue may represent an object of one of the following [nine] types: integer, logical, Boolean, function, routine, label, string, vector, and Lvalue.

In other words, the need to *talk* or *write* about data types, and hence their existence as named interpretations, has proved necessary, albeit only in the meta-language of its documentation, not in the object language. Some more modern languages, like JavaScript or Self, are like BCPL in this regard: they include few or no named data types in the language, but are hard to explain to another human without inventing data types in the meta-language.

So far we have limited ourselves to talking about data types as sets of symbols, which seems overly concrete. Meanwhile, many of the concerns we have just mentioned, such as abbreviation, checking and generality, are bound up in the term "abstraction", which we consider next.

3. The twin essences of abstraction

Parnas et al. [1976] state that "an abstraction is a concept that can have more than one possible realization", adding that "the power of abstraction, in mathematics as well as in programming, comes from the fact that by solving a problem in terms of the abstraction one can solve many problems at once". We will call this view "abstraction as generality".

However, we often talk informally about abstractions as a repertoire of things that we can *refer to*. We might praise a system that "exposes the right abstractions"; we might say an improvement "raises the level of abstraction". This doesn't contradict "abstraction as generality", because each of these abstractions may indeed admit more than one realisation. But the feature we are highlighting is really the fact that *no realisation need be stated explicitly*; it may simply be *referred to*. If a system provides abstractions, it means we do not have to define them ourselves in terms of more concrete

[11] Tags may denote only "concrete" data types, since giving values a single tag necessarily *partitions* them into non-overlapping equivalence classes. These correspond to *instantiable* data types like "integer", "string", "circle" etc.. In other contexts than instantiation, we might want to refer to interpretations which *overlap*, such as "ordinal", "shape", etc., which can be seen as *predicates*, need not induce a partitioning, and are a notable expressiveness threshold in a language's support for role 1.

[12] Of course, there are corresponding effects on time and memory consumption at compile time versus run time.

things. In turn, this *abbreviates* our systems, both syntactically and in the cognitive effort required to express them. It is often this property, not generality *per se*, that is meant by "abstraction". We can call this viewpoint "abstraction as reference". Data types in BCPL++ clearly have this property.

Generality and reference are distinct, but they are not completely orthogonal. Reference tends to create generality, because when we refer to a thing, we only refer to certain properties of it at any time. For example, suppose we are using an instance of a Complex data type implementing complex numbers. At any point, we only invoke certain properties of it—some operation that it supports (invoked explicitly) and its expected behaviour (embodied implicitly our code). With regard to any *other* property of Complex, we could say our code is *abstract*. This is the essence of abstraction: abstraction always has to do with *leaving something out*. In the case of "abstraction as reference", we leave out details which would otherwise be repeated across multiple uses of the same definition. In the case of "abstraction as generality", we leave out details that are different from one instance to another of the same generalisation.

Therefore, even if only abbreviation is our intention, using *reference* means that we are likely to achieve some generality in a happy accident. This is not guaranteed, though—we might, by chance, end up depending on all the properties that define Complex, down to all the operations it supports and all details of its representation. In that case, no generality remains in our code. Although we're unlikely to depend on *every* detail of a data type, it's easy to unintentionally depend on something we'd rather not. We might therefore prefer a system that *protects* a particular degree of generality by forbidding us from writing certain code in certain contexts.

A key insight of Parnas [1972] was that we can turn these two related concepts—reference and generality—to our best advantage only once we know *with respect to what properties* we wish our code to be general. He argued that *change-proneness* was an overriding criterion for such properties, since changes to software account for much of its *cost*. To enforce that most code be kept general with respect to change-prone details, we localise them (the code that embodies them) inside a construct called a module, where they are *hidden*. Hiding means that we forbid external code from *referring* to the change-prone details within the module. The module must also define a less change-prone set of definitions, which we now call an *interface*, around the change-prone internals. Generality *with respect to the identified change-prone properties* is the intention of information hiding. *Abbreviation* of client code may or may not follow; if it does, it is a happy accident. Sometimes, the abstraction inherent in interfaces naturally abbreviates programs. Other times, operations are convoluted by information hiding. For example, if we knew the representation of a mutable stack, we might empty it by clearing its head pointer, whereas otherwise we might be forced to iteratively pop each element.

In summary, we can observe two non-orthogonal kinds of abstraction: generality and reference. Any device for *reference*, including data types, introduces both of these. Generality is both fragile and valuable, such that we might want it to be *protected*. Information hiding mechanisms provide such protection, and are motivated by the fact that changes to software are costly.

4. Abstraction and data types

Let's return to thinking about *data types*. Alphabets of symbols are one way to define data types—but they are not the only way. The essence of data abstraction is the generalisation away from "data" as manifest, stored symbols, to data as (recursively) data-accepting and data-yielding *behaviours*: patterns of information flow, instead of mere symbols. These behaviours include reads and writes of stored values (the base case), but can also be themselves realised by computations (the recursive case). Data abstraction is therefore recursion on the concept of a computation, applied at the granularity of symbols in a naïve "concrete" computing machine like Turing's.[13]

Abstract data types can be realised using information hiding, as with CLU [Liskov and Zilles 1974] and many more recent languages. Recall that information hiding is done in conjunction with some policy that selects modularisation decisions according to some expectations about the *cost* of software development. CLU fixes a particular information hiding policy: hide the *representation* of an abstract data type. In other words, it is assumed that representations are an important class of change-prone decision, and that *generality with respect to representation* is the generality that is sought. (As we will see, this particular criterion is one that was adopted by the burgeoning confluence of mathematical logic and programming language design from the mid-1970s.)

In CLU, this property is not a subject merely to guidance; it is *guaranteed*, in that information hiding is *enforced* by the language. No exceptions can be made, nor alternative modularisations selected, on cost/benefit grounds. One could summarise this philosophy as follows: *there is a hierarchical structure to abstraction; code should target the highest possible level of abstraction (although no higher).*

We see this hierarchical view of abstraction implicit in phrases such as "levels of abstraction". Indeed, Reynolds [1983] would later write that "type structure is a syntactic discipline for enforcing levels of abstraction" (a definition we return to in §5 and §10). The relation between each (abstract) data type and the data type(s) of its representation forms a partial order that is well-known and unique.[14]

[13] This phrase is chosen with intentional similarity to the description by Kay [1996] of Smalltalk objects as a "recursion on the notion of a computer itself".

[14] Equating "partial order" with "hierarchy" is committing something of an abuse, although I follow Parnas [1972] in doing so.

This "fixed policy" contrasts with the writing of Parnas, in which information hiding can be applied to any design decision. *Representation* is merely one design decision that can be hidden; there is no blanket rule. Instead, one must be guided by cost. On hiding representations, Parnas et al. [1976] wrote the following.

> The same program could calculate distance from origin for a point in two-dimensional cartesian space and the magnitude of a complex number whose representation is in terms of real and imaginary parts. However nice the aesthetic properties of a language may be, if it forces users to write duplicate programs or forces the code generated to be larger than otherwise necessary, the language will have difficulty gaining acceptance by organizations with strong cost, time and memory constraints. Under pressure, the users of such a language will resort to the dirtiest of dirty tricks to meet their time and space constraints.... A user should [therefore] be able to... define a set of operations... in terms of [a] representation. The decision to have a representation-dependent program should [however] be an explicit one and the points at which representation dependence is introduced should be easily recognized.

Again, the overriding criterion is one of cost. If constraints demand that writing representation-dependent code delivers the lowest net cost, a language should not prevent it. Indeed, Parnas [1978] had expressed distaste for the phrase "levels of abstraction" for exactly this reason.

> I have not found a relation, 'more abstract than', that would allow me to define an abstraction hierarchy. Although I myself am guilty of using it, in most cases the phrase 'levels of abstraction' is an abuse of language.

In contrast to the hierarchical view, in which abstractions are ordered relative to one another, this is a heterarchical view: many abstractions are possible, and no one ordering is universally applicable. We can paraphrase this philosophy as follows: *there is a heterarchical structure to abstraction; code should target an abstraction that minimises* cost.

One example of code illustrating Parnas's point is the infamous inverse square root code [Eberly 2010].

```
float InvSqrt (float x)
{
    float xhalf = 0.5f*x;
    int i = *(int*)&x;
    i = 0x5f3759df − (i >> 1); // This line hides a LOT of math!
    x = *(float*)&i;
    x = x*(1.5f − xhalf*x*x); // repeat for a better approximation
    return x;
}
```

This code is clearly dependent on the particular bit-pattern format of floating-point values, and also that of integer values. Note that int is not even float's representation—the two are formally unrelated data types. It would not be expressible in a representation-hiding language. We can express it in C because although C offers multiple and user-defined data types as *abstractions*, it does not require that abstractions are *protected*: we can find a way to *violate* them, here by *reinterpreting* the floating-point value's representation. As this code demonstrates, it is *possible* to have such abstraction-violating operations yield meaningful results (a good approximation of the reciprocal of its input's square root, in this case). Respect for abstractions is only a proxy, albeit a useful one, for meaningfulness. This code is, by its nature, an extreme example, and has the disadvantage that it is fragile with respect to change in the two bit-pattern formats. Nevertheless, despite its representation-dependence, it is not a travesty. The value of having a fast approximate reciprocal square root function was evidently very great to the originating project (at least worth the investment of "a lot of math"). Since bit-level representations of integers and floating-point values are not particularly change-prone, it is very plausible that the benefit would justify the cost.

A more hierarchically minded viewpoint, in the style of CLU-like languages, would be that this code can still be written, but only *inside* the compiler or floating-point library. But this is unsatisfactory. Clearly, the authors of those components had not anticipated the need for this code. It is also far from clear that integrating such code would be a cost-optimal decision. While maintaining the code externally incurs some expense to its creators, the cost of integrating it to the library might well be greater overall. Suppose that many approximations existed, each suitable for a different application; integrating each one would likely be too burdensome. So, at least in some cases, such code will continue to exist outside the privileged modules.

Even though C lets us violate the abstraction of floats where we choose to, it would be wrong to claim that it doesn't *offer* the abstraction of floats. This seems to be a controversial point, in that when asking my fellow researchers about "types", it is a popular position that if we have not protected our abstractions, we have not really abstracted. But we have definitely done *something*. It is possible to define abstraction as requiring protection, but doing so would conflate two distinct concepts. Most programmers would recognise floats in C as an abstraction, just as our data types in BCPL++ are abstractions—even if they are not protected. That is not to say that protecting abstractions isn't a valuable thing to do. Generality is fragile: just as entropy tends to destroy order unless work is put in, so the tendency is for generality to be eroded unless special measures are taken. In this sense our two kinds of abstraction are far from symmetric.

Talking about what abstractions *may* or *must* be protected, and by what mechanism, gives us a more precise way to talk about what many people call *type safety*. As with most terms involving "types", it has many different meanings; saying *protection of abstractions* allows us to be more precise. For example, many people would say that C and BCPL++ are both "type-unsafe" languages. However, something clearly distinguishes C from BCPL++ in this regard. Unlike BCPL++, a C compiler will force us to use an out-of-the-ordinary language feature—a pointer cast—to access the representation. We can't just apply the floating-point operations as we normally would. Although it doesn't forbid us from violating the abstraction, it offers something I will call *guidance*. With *guidance-protected abstractions*, you can't write abstraction-violating code the same way, syntactically, as you can write "normal", non-abstraction-violating code. (This is exactly Parnas's "easily recognized" property, from the first extract quoted above.) The other approach is *enforced protection*: you can't write abstraction-violating code at all. We can no doubt attribute some of the continued popularity of C to the fact that it allows us to write abstraction-violating code in the occasional cases where this is justified.[15]

We can also note that within the same language, different abstractions are offered different kinds of protection. In Python, the basic abstraction of objects as dictionaries has enforced protection, in that it is not possible to write code which reinterprets the bits of a dictionary. However, user-defined classes are only offered "guidance"-style protection, in that we can instantiate a class but then delete fields or methods from the created objects. The guidance arises in how doing this deletion requires unusual-looking code: access to the `__dict__` field or use of `del`, say.

5. The essence of type discipline

Most computer scientists of theoretical training use "type" in the sense of Russell or Church, as a property that classifies expressions in a language.

In the previous sections, we talked about "data types" as synonymous with "types". We will require a slightly different definition of "type" in this section. Unlike "data type" in programming, the origin of "type" in logic is well documented. Russell [1908] introduced the simple theory of types as a way of restricting quantification in propositions, hence avoiding various paradoxes. Church [1940] integrated this theory into his λ-calculus and showed that it could derive theorems of primitive recursive systems, including a subset of Peano's arithmetic postulates, with the guarantee that well-typed expressions would always terminate.[16]

All this work occurs firmly in the context of logic, not computation. Church introduces the reduction sequence of formulae as a proof, not a program, and notes that "a complete incorporation of the calculus of λ-conversion into the theory of types is impossible" effectively since many partial recursive functions are not well-typed under this calculus.[17]

[15] This property of C—that even when data has been abstracted, its representation is pervasively available—has been described as amounting to "two type systems" [Krishnamurthi 2003, chapter 28]: one in which state is bytes, and manipulated by selecting lvalue types much like in BCPL++, and the other more conventional approach. Since these "type systems" are not particularly comparable, and and noting the ambiguity we previously highlighted in this phrase (§2), I do not follow this view.

[16] Infinite recursion in Church's calculus is analogous to the divergent nature of paradoxes, such as Russell's: both concern infinite progress-free chains of reduction or deduction.

[17] Of course, types in a type discipline can be considered propositions in some logic—a meta-logic, not the object logic!—such that programs are constructive proofs of those propositions (the Curry–Howard isomorphism). This has no bearing on the arguments this essay.

Until the 1980s, computer science literature in this area typically talked about a *doctrine* of types or, later, a *type discipline*. Nowadays, *type system* has supplanted these, but I prefer "discipline" since it appropriately connotes a *quid pro quo*: in return for obeying certain rules, certain benefits emerge. I also use it in this essay because it is unambiguous: it always refers to a proof system. By contrast, as we noted (§2) in popular usage, "type system" is often used informally to refer to various properties of a programming language design filling any of our four identified roles, and particularly to the language of data types that can be expressed (role 1). This is clearly not a type discipline, nor, by itself, a "type system" in the sense of Pierce [2002].

Russell wrote that "a *type* is defined as the range of significance of a propositional function, *i.e.*, as the collection of functions for which the said function has values". Types are ordered, starting at the "type of individuals", then inductively generated by the rule that "whatever [proposition] contains an apparent variable must be of a different type from the possible values of that variable; we will say that it is of a *higher* type". Types exist to restrict quantification within propositions. It is significant that Russell does not develop a notation for types themselves. Instead, he refers to them indirectly using phrases like "the next higher type than that of x". One type or another is uninteresting; the utility of types comes from their ordering, since it allows the expression of rules which detect meaningless (paradoxical) propositions.

This is very different from our notion of data types. Data types are always something that we could refer to explicitly. Russell's types are embodied implicitly in a rule (concerning the appearance of variables, and how they may be interpreted). Although Church did introduce a notation for types, types in his system fulfil the same role.

What role do type disciplines have in the context of programming? As in logic, they have something to do with forbidding erroneous constructions, i.e. forbidding programs with certain errors. However, they need not concern *data types* at all. To demonstrate this, let's imagine adding a type discipline to the Unix shell without adding data types. Suppose that our type discipline concerns itself with the emptiness of strings, and the checker checks that uppercased shell variables never hold empty strings.[18]

```
MYVAR=a               # ok
MYVAR=a${yourvar}     # ok
yourvar=${MYVAR}      # ok
MYVAR=${yourvar}      # not well-typed
```

The first three assignment expressions are well-typed but the fourth is not, because we cannot allow lowercased variable names to be assigned to uppercased (nonempty) without appending a character. We could fairly easily write down a

[18] For the unfamiliar: expression evaluation in the shell consists of repeated substitution of substrings according to special "expansion" constructs, such as `${MYVAR}` which expands to the (string) value of variable `MYVAR`. Empty expansions of variables can often cause a shell program to "get stuck" by expanding the expression into an invalid state.

formal rule to capture this. This would be a *type discipline* in the sense of Russell (and of Church), but also in the sense of much literature concerned with programming. However, it would be a stretch too far to claim that the shell now has any more data types than the unique string data type it started with. We have not even named the set of empty strings and the set of nonempty strings; we have just applied some rules that embody these concepts implicitly.

Russell was concerned with restricting how sets could be built from other sets, to avoid quantification that would create paradoxes. Logicians say that the resulting system is *sound*. We have just used the same idea to restrict how strings can be built from other strings, to avoid expansions that might get stuck. We have done so at a price: some correct programs, which never construct empty strings, nevertheless are not well-typed. Our language is still "complete" so long as we can always refactor a program until it falls within the discipline. In the following code the assignment is not well-typed, although if we substituted the definition of choose we would quickly obtain a well-typed program.

```
choose () {
  if [ -z "$1" ]; then echo $1; else echo $2; fi
}
MYVAR=$( choose a "${yourvar}" )  # type error!
```

This price was of no concern to Russell, since a convoluted construction of some set is no less valuable than a simple one. It only matters that all meaningful sets can be constructed *somehow*. But of course, this price is of concern to programmers: to say a language is (Turing-) complete says little about its usefulness.

What, then, is the defining feature of type disciplines? The most coherent position is that they are metalinguistic reasoning devices that interpret the object language *syntactically*. Russell's type discipline is defined only in terms of the form of *propositions* or *expressions* in a language. In the case of type disciplines in programming languages, we might observe that these expression are usually classified in terms of data types—named interpretations of values or behaviours—but this is secondary. All type disciplines, however, reason over some kind of inductive structure which we characterise as the syntax of a language. Program syntax, like proposition syntax, is invariably such an inductive structure.

Of course, most programming languages which feature type disciplines *do* use explicitly named interpretations, because they repurpose data type definitions as "types" in the sense of the type discipline. This is a *design synthesis*, as we anticipated when introducing data types: type disciplines are a popular tool for defining "semantic well-formedness" criteria which, as we noted earlier (§2), are naturally applied to establishing *operational well-definedness with respect to data types*. Two popular approaches in this space are summarised by the famous dictums of Milner [1978] and Reynolds [1983]. Milner's statement that "well-typed programs cannot 'go wrong'" emphasises the use of type disciplines directly to prove operational well-definedness. Reynolds's statement that "type structure is a syntactic discipline for enforcing levels of abstraction" emphasises the use of type disciplines to embody information-hiding rules and hence prove that *interpretations* (modelled as abstract data types) are used without knowledge of their *representations* (more concrete data types). and therefore that code will withstand changes to the latter.[19]

6. On understanding "On understanding..."

William Cook's essay [Cook 2009] on data abstraction is a wonderful work that I never hesitate to recommend to anybody. It makes a subtle yet essential point: that objects perform data abstraction using a different mechanism than do abstract data types (ADTs). The former use procedural abstraction, gaining flexibility, whereas the latter use relatively syntactic information-hiding rules, gaining tractability.

Nevertheless it contains a wrong statement about exactly how ADTs may and may not work. The misstatement can be blamed on an *overapplication* of the word "type". In section 4.1, Cook writes the following.

> Abstract data types depend upon a static type system to enforce type abstraction... [whereas] objects can be used to define data abstractions in a dynamically typed language.

However, this is not true. The exact kind of abstraction offered by abstract data types can be provided and enforced in a language with no static type checker, like our BCPL++. Indeed, the way it is enforced in C (covered by Cook earlier in his essay), where file separation is used to hide the representation, makes no use whatsoever of C's type checker. One could even imagine abstract data types in a language with no compiler and only dynamic binding. This could be useful in systems using reflection or run-time code generation against ADTs: when attempting to access details private to the ADT's representation, we might enforce information hiding *dynamically*, at the time of code generation or other reflective access, defining the check in terms of the call stack ("who's calling?"). It could conceivably be useful to do so, for example to ensure that the generated code did not embed details of that representation, which would prevent it from running in the context of a different definition of the same ADT.

The problem seems to be use of the phrase "type abstraction" to refer to the separation of an abstract data type's name from its representation. It's difficult to see what this kind of abstraction has to do with "type" *per se*. It's actually another instance of the same abstraction device we called *reference* in §3: separation of a *named definition* from *uses of* that definition. The extra trick it uses, above ordinary reference, is to forbid certain references. Again, this is informa-

[19] This use is arguably a fifth role of "types", since the consideration of code's amenability to change is somewhat "higher-order" relative to Milner-style criteria concerned merely with error-free operation. However, it can also be considered primarily operational in nature, in the sense that programs found to be ill-formed would be prone to errors at run time—if not immediately, then upon changes to hidden representation details.

tion hiding, as described earlier (§4). Although it is possible to use type disciplines for enforcing information hiding, that is only one way to do so. A possible rejoinder would be to squint and call *any mechanism* for enforcing such referential constraints a "type system", even if enforcement is entirely dynamic. But doing so is clearly not Cook's intention, and would degrade the phrase beyond even the low level of agreed meaning than it currently enjoys.

Cook's use of "type" here can be considered a kind of metonymy, in which the more specific phenomenon "type" is named in place of the more general phenomenon "reference". We humans have an easily observed tendency to introduce metonymy—a special case of *metaphor*—into our use of language, and to do so unconsciously. We will see examples later of various other metonymies using the word "type" that have made their way into print.

Of course, saying all this does not undermine Cook's point in any way: that the two kinds of abstraction are very different. Whereas procedural abstraction is necessarily capable of expressing arbitrary computations, by contrast, "abstraction by reference" is limited to some set of rules about what identifier resolves to what referent in what context. It is interesting to note that there is a continuum here nonetheless. Rules can be viewed as a logic or a machine, can be bound early or late; and their expressiveness varies somewhat.[20]

A theme in our discussion of abstraction was the contrast between *guarantees* and *guidance*, and the distinction of *a priori* fixed policies (like representation hiding) from policies informed by cost. We can trace these two issues in Cook's discussion of the relationship between abstract data types and formal models. Cook writes in section 2.5 that "formal models of abstract data types are based on existential types" (citing Mitchell and Plotkin [1988]) and that "abstract data types have a fundamental model based on existential types… [which is] a solid connection to mathematics". Such type-theoretic models are based on *calculi*, meaning languages designed with the goal of discarding as many details as possible beyond those affecting whatever properties are considered relevant *a priori*. The usual goal of analysing a calculus is to prove that some property holds of all programs expressible in that calculus. If the property proved is type-correctness, the calculus is called "type-sound". More generally, the properties considered over calculi invariably apply abstractions that are characteristic of mathematical logic, and concern *guarantees*: considering expressiveness only as a boolean (*whether* something can be expressed, rather than how succinctly), complexity only in the asymptotic case (e.g. big-O notation), decidability only as a boolean (existence of a computable total function), and so on. By contrast, the differences that Cook emphasises most heavily about ADTs versus objects are practical ones: whether or not different implementations can interoperate; which approaches are more amenable to what wider language designs; how tractable it is to reason about them ahead of time. To me it seems these are more concerned with cost than guarantee. Note that even tractability, here, is a practical property, since practical work in software verification often finds, for example, that exponential cases tend to be polynomial in common practice [Milner 1978; Järvisalo et al. 2012]; that generally undecidable problems turn out to be efficiently decidable in many common cases [Cook et al. 2011]; and so on.

As it happens, we can talk much more directly about what abstract data types are doing by focusing on the device of *reference*, not on type theory. Ironically, although Russell would become the founder of type theory, he had already elucidated the necessary properties of reference in an earlier article [Russell 1905] that concerns the philosophy of language (and has nothing to do with types). He elaborates referential statements of the form "the King of France is bald" into existentially quantified statements: "there exists an individual x such that x is bald and x is the King of France". Abstract data types apply this exact elaboration on *client* code, combined with information hiding restrictions on what properties can be listed after "such that" (restricted to those listed in the specification, and, in particular, excluding any properties of the representation). The connection between existential statements and abstract data types owes to the fundamental device of *reference*, and can be explained outside the context of type theory.

7. "Types" and programming languages

Benjamin Pierce's *Types and Programming Languages* [Pierce 2002] is a wonderful textbook. Very early on, it acknowledges the difficulty of defining such things as "types" or "type systems", but proposes the following definition as a "plausible" starting point.

> A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.

Indeed, the systems which are covered by the substance of the book conform strongly to this definition. However, the same introductory chapter later makes the following remark on the history of "type systems".

> The first type systems in computer science, beginning in the 1950s in languages such as Fortran, were introduced to improve the efficiency of numerical calculations by distinguishing between integer-valued arithmetic expressions and real-valued ones; this allowed the compiler to use different representations and generate appropriate machine instructions for primitive operations.

This statement is true, but it demands an exceptionally careful reading. In particular, it would be false to infer from

[20] In philosophy of language, theories of naming are judged by their ability to attach meaning to complex denoting expressions arising in natural language. The theory of Russell [1905] was criticised by Kripke [1980] for its inability to capture the invariant intension of *names* recurring across different modalities—such as statements about a world where Aristotle had been raised by wolves. Kripke's refinement was in effect a context-sensitive model of names, as mappings from worlds to extensions, analogous to extending a computational interpreter with state, context or scoping.

it that a *type system* (in Pierce's own sense) is necessary in order to "distinguish between integer-valued arithmetic expressions and real-valued ones". Our examples in §2 have already established that this is not true: BCPL expresses this distinction in an ad-hoc form, and our BCPL++ expresses it in a general form by including an explicit notion of data types which can be used to index operations. Pierce correctly limits the scope of "type system" to "distinguishing between integer-valued arithmetic expressions… *[in] the compiler*", where the emphasis I have added is crucial. We must also read "efficiency" very carefully: there is nothing to prevent a BCPL program from generating equally efficient code as that generated from Fortran, so we must include the *human* cost, meaning the efficiency of programming as a process. By assigning types to variables, the compiler can relieve the programmer from managing temporary storage and inserting conversion routines (as discussed in §2), both of which would be necessary in BCPL, hence adding to the opportunity for error. Early Fortran has no non-numeric data—even conditional branching was defined arithmetically—so there is little opportunity for semantically ill-formed expressions. The compile-time analysis does not exist to prove the absence of any particular class of errors, but rather to take a somewhat error-prone task away from the programmer.

My point here, again, is that "type system" and "types" are overapplied terms. Although we can find arguments that Fortran's system belongs alongside the others in the book, this is only by overlooking a host of distinctions. There is a significant difference between how "types" are used in Fortran (as data types, employed so as to take work away from the programmer; roles 1 and 2 from §2) and how they are used in the systems which Pierce's book is about (as types in a type discipline, to guarantee absences of certain errors; roles 3 and 4).

8. Wikipedia

Can we crowdsource a definition of "types"? Wikipedia's article on "data type", as retrieved on 2014/4/10, begins as follows.

> In computer science and computer programming, a *data type* or simply *type* is a classification identifying one of various types of data, such as real, integer or Boolean, that determines the possible values for that type; the operations that can be done on values of that type; the meaning of the data; and the way values of that type can be stored.

This opening is very general, and makes it clear that the issue of "data types" mixes a number of concerns: values, operations, "meaning" and storage. When reading related Wikipedia pages, such as that for "Type system" (retrieved on the same date), one notices how much alternation (use of "or") there is in the text. To make correct statements about the meaning of "type", it becomes necessary to take the union of a large variety of possible meanings. Popular discourse suffers from the weight of these possible meanings, which are confusing for a newcomer and a minefield even for experts. Meanwhile, in the very next paragraph, we read the following.

> Data types are used within type systems, which offer various ways of defining, implementing and using them. Different type systems ensure varying degrees of type safety. Formally, a type can be defined as "any property of a program we can determine without executing the program".

The nested quotation—"any property of a program we can determine without executing the program"—is by Krishnamurthi [2003]. This is no longer talking about data types as introduced in the first paragraph—it is about types in the sense of type disciplines. Recall how we saw in §5 that type disciplines can state and prove properties over programs without introducing new data types. This transition, from the very general earlier definition to a materially different concept, occurs in the article with no acknowledgement that a change has occurred. In so doing, the Wikipedia article has *overapplied* the notion of data types to the distinct notion of types from logic.[21]

Krishnamurthi's definition is interesting because it defines *types* as a specification device—they are *properties* that we wish to be *determined*—but doesn't require that a type discipline is the method used to establish these properties. This reflects a use of "type" that I consider an overapplication: its use to denote *any* specification mechanism for decidable properties. Type disciplines are one of a host of ways by which we can establish properties of programs. Specifically, they are those methods which perform proofs over the inductive structure of program syntax. But we can interpret programs in radically *non*-syntactic ways, for example as transition systems, hence exposing other structural dimensions than that of syntax. None of these is the universally "right" decomposition. Type disciplines owe their tractability but also their limitations to the very fact that they do not transcend syntax. Although we could squint and, consistent with calling any property a type, also call any method for proving such properties a *type system*, as before this would erase useful distinctions and degrade these terms yet further.

9. A voracious appetite

We have started to observe a theme: metonymous uses of the word "type". This turns out to be extremely common. It is almost as if "type" is a word which is hungry to acquire new meanings (which we are all too eager to give it). It appears in a large number of phrases which, on reflection, are not directly to do with types in any of the senses we have seen. I briefly survey some of these here.

[21] Fittingly, since writing this section in early April 2014, this quotation has been removed (as of 2014/4/15), for effectively the reason that I identify here. The presence of this confusion, corrected or otherwise, is still worthy of note. Meanwhile, the second edition of Krishnamurthi's book does not contain the quoted sentence. My comments are directed at the Wikipedia article's inclusion of the quotation without any qualifying context, rather than the book itself, whose second edition is admirably clear and consistent in its use of "type".

Memory safety as "type safety" It is easy to find examples in the literature of "type safety" when what is really being referenced is one or both of two *memory safety* properties: pointers stay within the bounds of the object they started pointing into (spatial memory safety), and that an object is only reclaimed once no live pointers into it remain in circulation (temporal memory safety). The former is typically achieved using dynamic checks (bounds checks, checked downcasts), and the latter using garbage collection. These properties are a cornerstone of how many languages protect their abstractions, including Lisp, Smalltalk and Java. The most obvious links to "types" are that type disciplines can also be used to eliminate certain dynamic checks, and that memory safety is necessary (but not sufficient) to *protect abstractions*, including data types (since a wild pointer can cause arbitrary damage). It is checking properties over memory, not over types, that is essential here.

Late binding as "duck typing" Dynamic language users often refer to "duck typing", but this turns out to be no more than dynamic binding (from named fields and methods to their definition on a target object). This is perceived as a form of "typing" because of an *absence* of the restrictions, stated in terms of data types, which many other languages impose. For example, in Java one might receive a generic object and know that it has a twiddle() method, but one must nevertheless downcast it to a *specific* twiddle()-specifying class before this method can be used. By contrast, so-called "duck typing" means that a value's data type is irrelevant to whether an operation can proceed; the precondition is instead that the methods the client requires can be resolved dynamically at the point of invocation. (Interestingly, variants of the same behaviour which require the client to specify an operand's structural signature, such as that of Go, appear not to attract the label "duck typing"—even though they otherwise remain late-bound and avoid nominal matching against an operand's data type.)

Values as "types" It is fairly easy to find phrases like "passing a value type" to mean "passing a [non-reference] value". This is trivial, but is a common instance of metonymy. Less trivially, the phrase "typestate" [Strom and Yemini 1986] is metonymous because it actually refers to the state of an object, i.e. a mutable value, not the state of a type.[22] The word "type" creeps in because it is convenient to package these state-based specifications within data type definitions. This is not an essential property, however; we could imagine languages which separate these constructs.

"Types" as units of code Smalltalk popularised the structuring of programs into *classes*. A class is a data type, so this promoted the view of software as structured foremost around *data types* rather than procedures. Meanwhile, the use of Reynolds-style type disciplines to enforce information hiding has led to a similar phenomenon in functional languages, most notably the ML module system [MacQueen 1984], in which module interfaces have "types" in the sense of a type discipline, and may or may not be encapsulating something recognisable as an abstract data type. Again, these two are distinct uses of "type" which have converged somewhat coincidentally.

[22] Of course, that does not preclude the use of these state abstractions within type disciplines. This has been a focus of various more recent work on typestate.

Explicitly-structuredness as "typed"-ness *Typed* often refers to the descriptive properties of data types. Data types localise details about representations (symbols)—perhaps concretely (a structure with a described set of fields) or perhaps abstractly (a set of behaviours). In his 2010 SPLASH keynote, Don Syme presented *type providers* in the F# language, a compile-time metaprogramming feature in which a collection of abstract data types can be generated on demand for interfacing with some external data source, such as a third-party web service or database [Syme et al. 2012]. He attributed a key advantage of this system to its being "typed", noting how it helps enable various useful kinds of tooling, such as autocompletion in an editor. "Type providers" could conceivably be called "code providers", where the "code" defines abstract data types, exposes operations primarily at a *storage* level (reading and writing of constituent elements), and the provision of code occurs at or before compile time. (This contrasts with run-time loading systems, such as networked class loaders in Java.) What sense of "typed" was intended? Clearly, this includes the essential sense (§2) of [data] types as abstractions. It also includes the *abstract* part of abstract data types, meaning representation hiding (§4). Meanwhile F# is also a typed language in the logical sense (§5)—but it would be wrong to attribute tooling features like autocompletion, or the benefits of "type providers" generally, to this sense of "typed". Autocompletion and other query-style tool support is certainly more prevalent among such languages, since type disciplines naturally cut down the space of exhaustively-enumerated well-formed results to a tractably small number. But exhaustive search is only one possible implementation of that functionality. (Consider that Google does a creditable job of autocompleting search terms, but natural languages lack type disciplines.) To attribute the benefits of "type providers" to "typed"-ness in the sense of the F# language's type discipline would therefore be metonymous.

10. The missing link—still missing

I have yet to see any work that reconciles the notion of "data type", arising in programming, and that of "type" arising in logic. It's easy to find papers that talk about them as if they are the same thing, but finding one that deals with the question explicitly has so far defeated me. In this section I review some literature that I have uncovered in my search, and which hints towards an answer.

As we noted earlier, the origins of "type" in logic are well known. The origin of "data type" in programming, or just "type" in that context, are less clear. We begin by looking for these origins. Wilkes et al. [1951], in arguably the first book about programming stored-program computers, do not mention "data types", nor use "types" except in its everyday sense. Regarding data, they write about "forms in which numbers and orders are represented within the machine".

Fortran and COBOL both emerged during the 1950s. Neither of their initial definition documents [Backus 1954; Anonymous 1960] uses the word "type" outside its everyday sense. Fortran IV was released in 1962, and its manual *does* talk about data types.[23] Algol 60 was also brewing during the late 1950s. Perlis was among Chipps et al. [1956], who state that "variables are of three types: scalars, vectors and matrices... Two kinds of each variable are required since the computer arithmetic functions in both the fixed and floating point modes"—but they instead use "kind" in other contexts, suggesting only everyday English usage. However, the Algol 58 Preliminary Report does mention "type" (and "arithmetic type", although not "data type"), and the Algol 60 report uses "type" in its familiar sense of "data type". Strachey and Wilkes [1961] use "type" in this same sense without any explanation or qualification. So it seems that the synonymous uses of "type" and "data type" gained common usage in the very early 1960s through the influence of Algol and Fortran.

Around the same time, many of those working in logic—a tradition which already had a well-established notion of type—were concerned with mechanisation of logic, naturally using computers. Influential work from this period included that of Robinson [1965] (at the Argonne National Lab, also contemporaneously with Reynolds) and Pietrzykowski and Jensen [1972]. Through this strand of work, "type" is used in the sense of logic.

Scott [1970] does talk about "data types" in a mathematical context, defining them as partially ordered sets. However, this definition has strong and very specific goals: a "mathematical" (we would now say "denotational") basis for study of programs. Even calculi such as that of Church are insufficiently mathematical, because they lack full abstraction. Instead, Scott & Strachey's work pursued domain-theoretic models of programs, which have (to my knowledge) yet to successfully model large fragments of real programming languages. Meanwhile, other mathematical developments have embarked from the observation that data types correspond to set-theoretic constructs—most notably, those of Hoare [1972] and Martin-Löf [1985]—but still retaining the Church-style view of types as an instrument for syntax-directed reasoning over programs, with no clear link to the other roles which [data] types fill in programming languages.

The late 1970s brought the landmark of ML, a successful programming language based on the lambda calculus. ML achieves an astonishing synthesis in its treatment of types. It starts with the premise of avoiding the need for run-time tags, entailing that the compiler decide, when generating code, how a given stored representation may be manipulated. This quickly leads to the constraint that the set of data types be equal to that of (logical) types in the type discipline. In other words, it is a consequence of ML's *design synthesis* that its data types and type-discipline types must be unified. This remains a remarkable result: the discovery of a language allowing erasability of run-time tags, inferability of typings for most code, while providing a type discipline that accounts even for *polymorphic* functions while remaining remarkably usable in practice. ML and its descendents remain rightly described as a "sweet spot" [Minsky et al. 2013, p. *xv*]. But that very phrase tells us that a wider space exists outside of this specific design synthesis. ML cannot easily encode structures in which the selection of data types is dependent on program input, such as heterogeneous lists, or other patterns associated with "subtyping" in object-oriented languages. Rather, doing so requires means that that effectively reintroduce run-time tags (e.g. using an "any value" algebraic data type). At the other extreme, Bracha [2004] actively advocated type-disciplined programming accommodating divergence between the language of data types and the language of logical types.

Reynolds [1974] talks about "type structure" but always in the sense of a type discipline, stating as a goal that "the meaning of a syntactically valid program in a 'type-correct' language should never depend upon the particular representations used to implements its primitive types". We note immediately that the absoluteness of this statement ("never") and the emphasis on representation-independence puts it at odds with the writing of Parnas et al. [1976]. Indeed, Reynolds [1983] later provided a more general statement of the same idea: "type structure is a syntactic discipline for enforcing levels of abstraction". This bears out several distinctions we have observed: the essential *syntactic* nature of type disciplines (§5), the emphasis on enforcement and on hierarchical "levels of abstraction" (§4), in contrast with Parnas's heterarchical view.

It is a testament to the disconnectedness of the engineering community from that of logic that during the 1970s, very few citations cross between the logic-oriented community and the engineering community. Another testament to this disconnectedness is how the phrase "high-order languages" [Brosgol 1976; Dijkstra 1975] was in use among engineers to describe, essentially, languages with expressive data abstractions. Meanwhile, in logic, "higher-order" had long implied a concern for first-class functions. As an omission in the other direction, Morris [1973] wrote about a pair of distinct purposes of "types", one he called "authentication" and another "secrecy"; two years earlier, Parnas had given the name "information hiding" to secrecy.

The divide between logical mindsets and engineering ones remained alive and well after the end of the 1970s.

[23] ... at least from the earliest revision I found, which was dated 1974.

Cardelli and Wegner [1985] embark on a survey of "types, data abstraction and polymorphism" but move quickly to talking about enforcing correctness and hiding representations. They write that unenforced "type-like" abstractions, by virtue of being prone to occasional violation, are "an illusion"—hence, implicitly, valueless. They talk of "potentially disastrous" and "arbitrary" consequences of such violations as, implicitly, facts that render such a system unacceptable. To a logician, this is understandable. If a system admits the arbitrary, it is called inconsistent, and is of no value. To an engineer though, the idea of *cost* is paramount. Any undesirable eventuality occurs only with a certain probability and incurring a certain cost; any means for preventing such eventualities also has a cost (such as a proof burden).

Recent literature has seen some reflection on these differences. Ostermann et al. [2011] argue convincingly that the emphasis on monotonicity of reasoning, while inevitable from classical logic, continues to be a poor fit for the mental processes of engineering practice. We can immediately connect this with the "hierarchical" versus "heterarchical" distinction from earlier, since monotonic reasoning proceeds along an ordered structure. Having said that, I disagree with the implication that Parnas-style information hiding requires monotonic reasoning, since, as covered in §4, Parnas's writing embraces a heterarchical approach.

It would be nice to rewind history and choose some other word than "type" for some or all of the various meanings it enjoys today. Perhaps we can at least take greater care to qualify our usage—perhaps saying "expression type" when talking about type disciplines, and more consistently including the "data" in "data type". I have begun doing the latter in my writing, ever since I began feeling the unease I mentioned at the start of this essay.

Meanwhile, historically the ingress of ideas from symbolic logic into programming language research has coincided with the egress of cooperation with "systems" research. Gabriel [2012] noted a personal feeling that "in the 1990s it seemed to me that scientists in the programming community pulled back the welcome mat from engineers", and also noted that "before 1990, a person interested in programming could work comfortably both in programming languages and in programming systems, but not so easily after". The bibliographies of the "engineering" papers cited in this section show that programming systems and operating systems research used to be close neighbours. It is unlikely to be a coincidence that it is very hard to prove properties about systems as large as real operating systems—or conversely, to build a functioning operating system that is nevertheless small enough to reason about. Indeed, we have only recently reached a point of being able to do so [Klein et al. 2009]. Perhaps that means the time is right to build a new understanding between communities of logic and engineering. These are not as alien to each other as it might appear. If we succeed in building such an understanding, it will mean learning to share the word "type" in a way that is sensitive to depth and diversity of the concepts lying under it.

Acknowledgments

I'm grateful to Stephen Dolan and Tomas Petricek for fruitful discussions, to Dominic Orchard and Laurence Tratt for comments on a draft, to the many people who've indulged me when I've asked them about "types", and to the library at the Computer Laboratory for its well-organised collection of older literature. This version has benefited from the comments of Peter Sewell, Dominic Mulligan, Jean Pichon, Alan Mycroft, Martin Richards, Jon Crowcroft, Nick Maclaren, Helen Oliver, Benjamin Pierce, Shriram Krishnamurthi, Don Syme, Jessica Leech and the anonymous reviewers. I was supported by EPSRC grant EP/K008528/1, "Rigorous Engineering for Mainstream Systems".

References

Anonymous. COBOL—initial specifications for a COmmon Business Oriented Language. Technical report, Short Range Task Force for the Conference on Data Systems Languages, 1960.

J. Backus. Preliminary report, specifications for the IBM mathematical FORmula TRANslating system, FORTRAN. Technical report, Applied Science Division, IBM, 1954.

S. R. Bourne. UNIX time-sharing system: The UNIX shell. *Bell System Technical Journal*, 57(6):1971–1990, 1978.

G. Bracha. Pluggable type systems, 2004. http://bracha.org/pluggableTypesPosition.pdf. Presented at the OOPSLA Workshop on the Revival of Dynamic Languages; retrieved on 2014/8/17.

B. M. Brosgol. Some issues in data types and type checking. In J. H. Williams and D. A. Fisher, editors, *Design and Implementation of Programming Languages*, volume 54 of *Lecture Notes in Computer Science*, pages 102–130. Springer, 1976.

L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, Dec. 1985.

J. Chipps, M. Koschmann, S. Orgel, A. Perlis, and J. Smith. A mathematical language compiler. In *Proceedings of the 1956 11th ACM National Meeting*, ACM '56, pages 114–117, New York, NY, USA, 1956. ACM.

A. Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

B. Cook, A. Podelski, and A. Rybalchenko. Proving program termination. *Commun. ACM*, 54(5):88–98, May 2011.

W. R. Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 557–572, New York, NY, USA, 2009. ACM.

E. W. Dijkstra. On a language proposal for the Department of Defense, 1975. EWD 514.

D. Eberly. Fast inverse square root (revisited). Web note, 2010. http://www.geometrictools.com/Documentation/FastInverseSqrt.pdf. Retrieved on 2014/3/15.

R. P. Gabriel. The structure of a programming language revolution. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '12, pages 195–214, New York, NY, USA, 2012. ACM.

C. A. R. Hoare. Notes on data structuring. In O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*, pages 83–174. Academic Press Ltd., 1972.

M. Järvisalo, A. Matsliah, J. Nordström, and S. Živný. Relating proof complexity measures and practical hardness of SAT. In M. Milano, editor, *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 316–331. Springer Berlin Heidelberg, 2012.

A. C. Kay. In T. J. Bergin, Jr. and R. G. Gibson, Jr., editors, *History of Programming languages—II*, chapter The Early History of Smalltalk, pages 511–598. ACM, New York, NY, USA, 1996.

G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

S. Kripke. *Naming and Necessity*. Blackwell, Oxford, 1980.

S. Krishnamurthi. *Programming languages: application and interpretation*. e-book, 2003. http://www.cs.brown.edu/%7esk/Publications/Books/ProgLangs/. As generated on 2007/4/26.

S. Krishnamurthi and M. Felleisen. Safety in programming languages. Technical Report TR 99-352, Rice University, 1999.

B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pages 50–59, New York, NY, USA, 1974. ACM.

D. MacQueen. Modules for Standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 198–207. ACM, 1984.

P. Martin-Löf. Constructive mathematics and computer programming. In *Proc. of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*, pages 167–184. Prentice-Hall, Inc., 1985.

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

Y. Minsky, A. Madhavapeddy, and J. Hickey. *Real World OCaml: functional programming for the masses*. O'Reilly, Nov 2013.

J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, July 1988.

J. H. Morris, Jr. Types are not sets. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 120–124, New York, NY, USA, 1973. ACM.

K. Ostermann, P. G. Giarrusso, C. Kästner, and T. Rendel. Revisiting information hiding: Reflections on classical and nonclassical modularity. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 155–178, Berlin, Heidelberg, 2011. Springer-Verlag.

D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

D. L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on software engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press.

D. L. Parnas, J. E. Shore, and D. Weiss. Abstract types defined as classes of variables. In *Proceedings of the 1976 Conference on Data : Abstraction, Definition and Structure*, pages 149–154, New York, NY, USA, 1976. ACM.

B. Pierce. *Types and programming languages*. The MIT Press, 2002.

T. Pietrzykowski and D. C. Jensen. A complete mechanization of (ω)-order type theory. In *Proceedings of the ACM Annual Conference, Volume 1*, ACM '72, pages 82–92, New York, NY, USA, 1972. ACM.

J. C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 408–423, London, UK, 1974. Springer-Verlag.

J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

M. Richards. BCPL: A tool for compiler writing and system programming. In *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*, AFIPS '69 (Spring), pages 557–566, New York, NY, USA, 1969. ACM.

J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, Jan. 1965.

B. Russell. On denoting. *Mind*, 56(14):479–493, 1905.

B. Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30:222–262, 1908. Electronic Edition.

D. Scott. Outline of a mathematical theory of computation. Technical Report PRG-2, Computing Laboratory, University of Oxford, 1970.

C. Shannon and W. Weaver. *A Mathematical Theory of Communication*. University of Illinois Press, 1949.

C. Strachey and M. V. Wilkes. Some proposals for improving the efficiency of ALGOL 60. *Commun. ACM*, 4(11):488–491, Nov. 1961.

R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12:157–171, January 1986.

D. Syme, K. Battocchi, K. Takeda, D. Malayeri, J. Fisher, J. Hu, T. Liu, B. McNamara, D. Quirk, M. Taveggia, W. Chae, U. Matsveyeu, and T. Petricek. F# 3.0: Strongly-typed language support for internet-scale information sources. Technical Report MSR-TR-2012-101, Microsoft Research, September 2012.

M. V. Wilkes, D. J. Wheeler, and S. Gill. *The preparation of programs for an electronic digital computer*. Addison Wesley, 1951.