In search of types

Stephen Kell

stephen.kell@cl.cam.ac.uk

Computer Laboratory
University of Cambridge

Are we sitting uncomfortably?

"type" = "data type"?

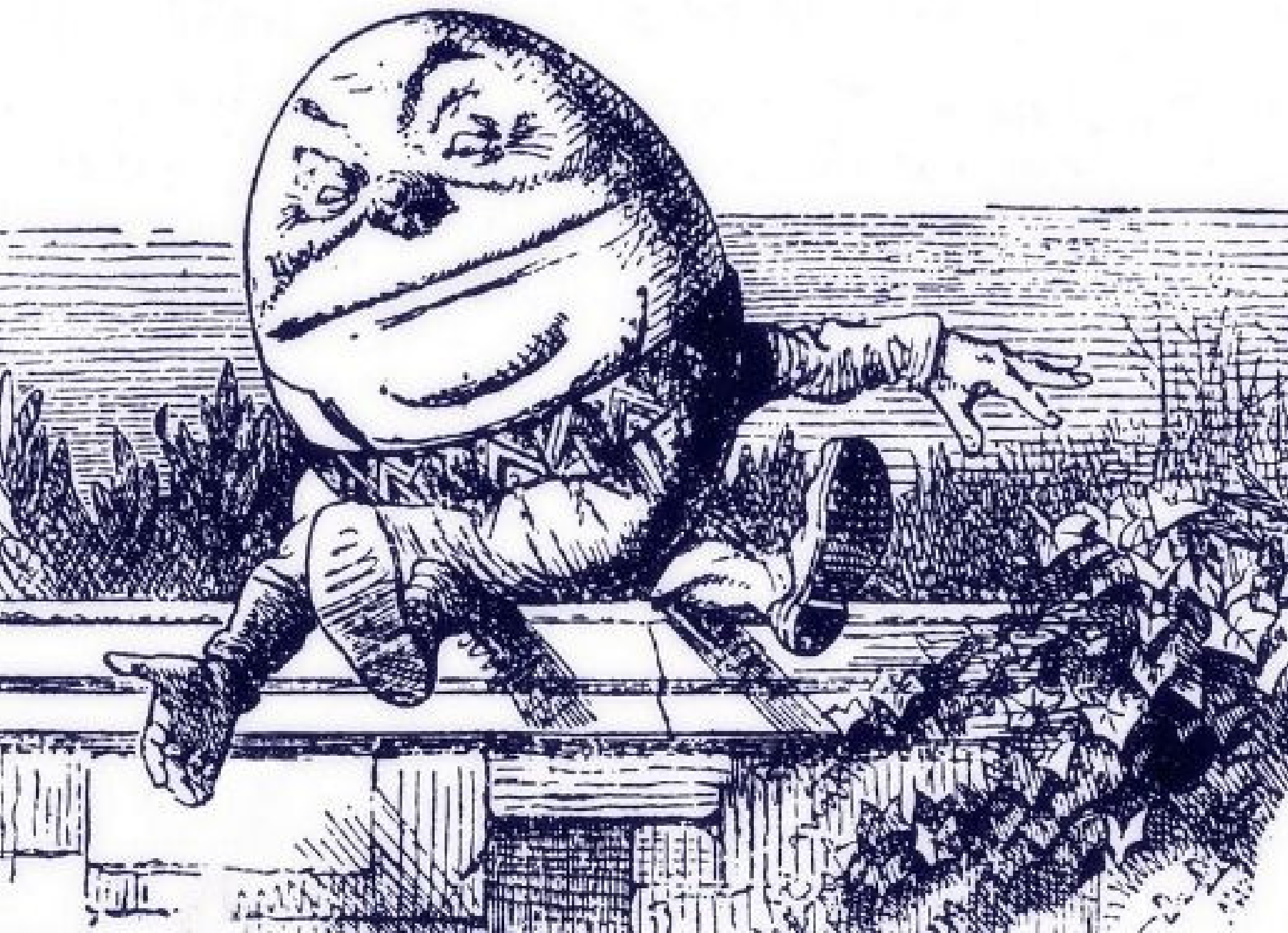
"type system" = ?

{strongly, weakly, dynamically, implicitly, duck, ...}-typed?

"type safety"?

Are we always talking about the same things?

If we're *not*, can we always tell?



Quotation is not endorsement!

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.[1][2]

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 programme we can determine without executing the program".[3]

The essay in a nutshell

- two thought experiments
- two views of abstraction
- a two-pronged history expedition
- a case for change (?)

“Types” and “data types” are essentially different!

“data types”

- a classification of values
- according to what they *model*

[“static”] “types”

- a classification of expressions
- in service of *reasoning*

PL designs often unify the two…

Most languages support multiple data types

Both built-in…

C	int float char arrays pointers functions
ML	int real tuples lists functions
Perl	scalars arrays hashes

Most languages support multiple data types

Both built-in…

C	int float char arrays pointers functions
ML	int real tuples lists functions
Perl	scalars arrays hashes

… and user-defined

C	structs unions enums
ML	ADTs
Perl	modules

Data types $\not\rightarrow$ "typed"

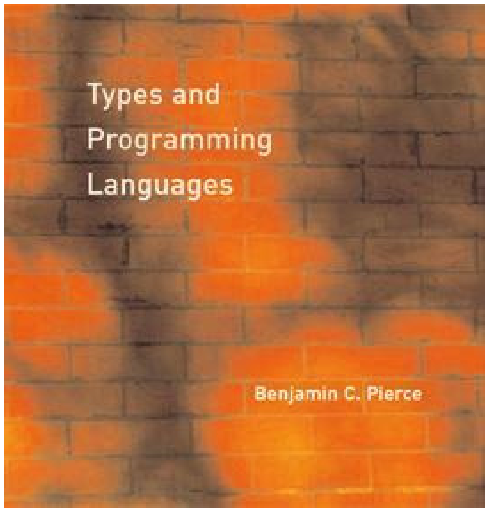
Bizarrely, "typed" does not mean "having > 1 [data] type"

- it's something to do with checking

"Type system" does not mean "system of data types"

- it's something to do with checking
- it's a proof system!
- (. . . unless it's a system of data types)

The "typed = statically checked" position



"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.

…

"Terms like 'dynamically typed' are arguably misnomers and should probably be replaced by 'dynamically checked', but the usage is standard."

Benjamin Pierce
in *Types and programming languages*

Summarising roles of data types in languages

rules about well-formed code

Summarising roles of data types in languages

rules about well-defined executions

rules about well-formed code

It's not just checking

"This storage is for holding integers."

```
int a, b;
```

Not all languages hold us to these statements.

```
int *pi = malloc(sizeof (int));
```

Summarising roles of data types in languages

interface to storage management

rules about well-defined executions

rules about well-formed code

Summarising roles of data types in languages

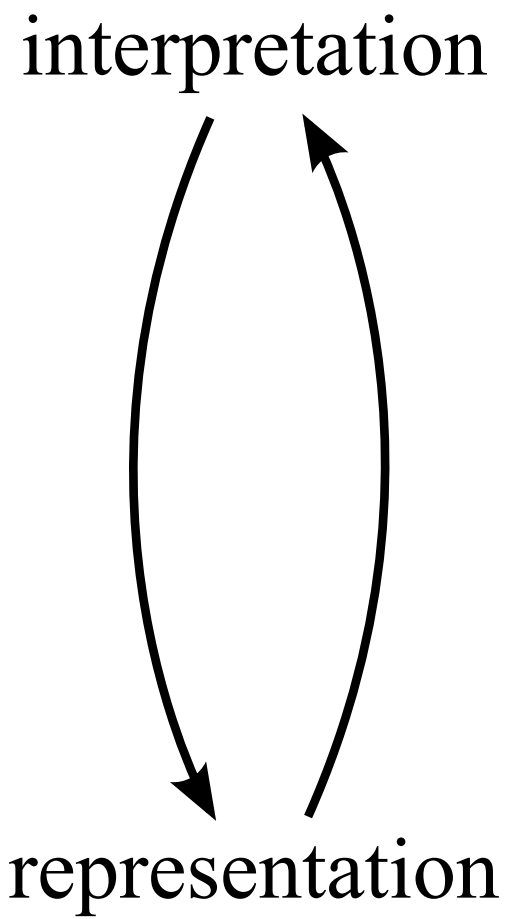
?

interface to storage management

rules about well-defined executions

rules about well-formed code

A fundamental idea



Some languages without [multiple] data types

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

```
choose () {
   if [ −z "$1" ]; then echo $1; else echo $2; fi
}
```

BCPL with slightly friendlier syntax

```
manhattan (x1, y1, x2,  y2)
{
   return abs(x1 − x2) + abs(y1 − y2);
}
```

Thought experiment: data types *minimally*, in BCPL (1)

```
// points are two 32−bit fields in one 64−bit word

manhattan (p1, p2)
{
  return abs(p1>>32 − p2>>32)
  + abs(p1 & ~0>>32 − p2 & ~0>>32);
}
```

Thought experiment: data types *minimally*, in BCPL (2)

```
struct point2d { x:32; y:32; };

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

Thought experiment: data types *minimally*, in BCPL (2)

```
struct point2d { x:32; y:32; };

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

In this language, data types have no role in

- managing storage
- determining operations' well-definedness
- determining programs' well-formedness

Thought experiment: data types *minimally*, in BCPL (2)

```
struct point2d { x:32; y:32; };

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

What do they do?

Thought experiment: data types *minimally*, in BCPL (2)

```
struct point2d { x:32; y:32; };

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

What do they do?

- make explicit *what the data models*
- separate definition from use
- . . . by *factoring out the representation*
- data types are "named" interpretations (really *signed*)

Summarising roles of data types in languages

named interpretations

interface to storage management

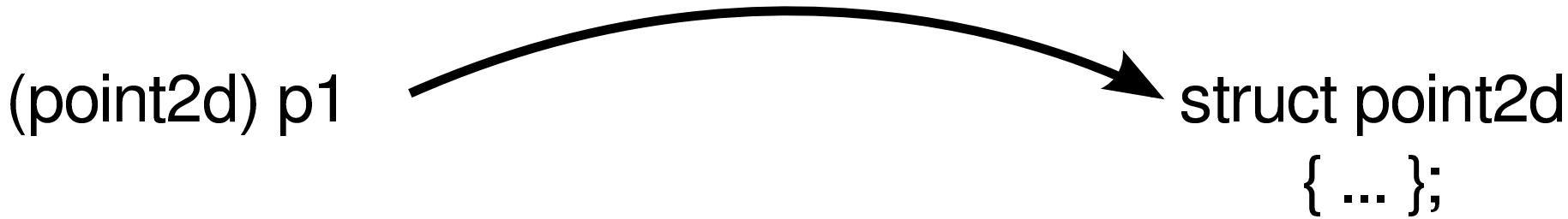
rules about well-defined executions

rules about well-formed code

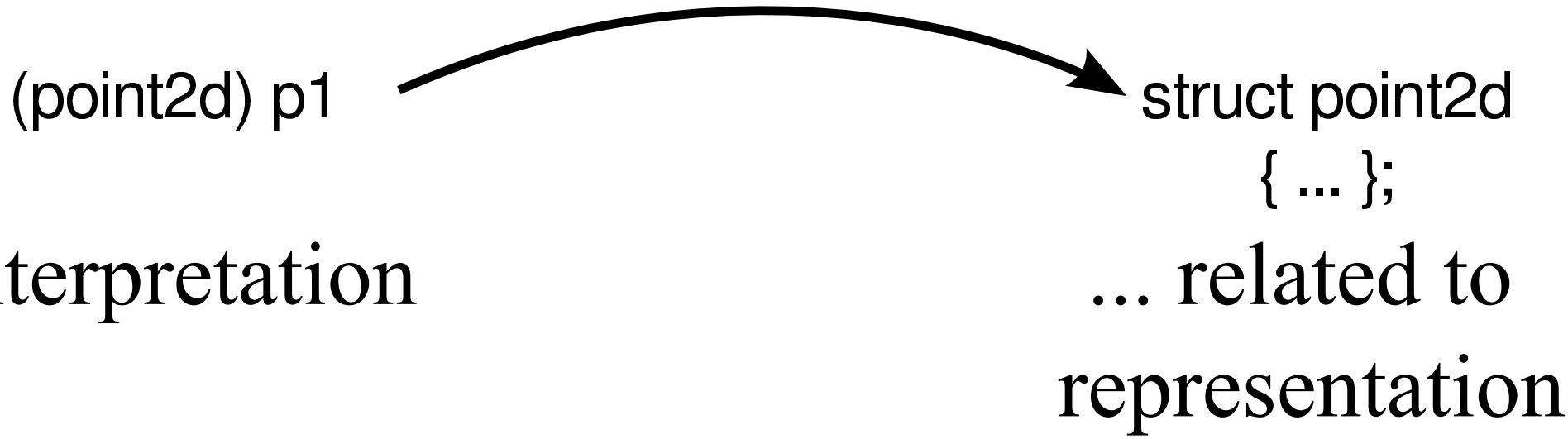
One kind of abstraction

One kind of abstraction

One kind of abstraction



One kind of abstraction



Recap

The essence of data types is def–use separation:

- we *use* an "interpretation"
- the *definition* is a representation
- we could call this *data abstraction*

What is the essence of [static] *type systems*?

- can we have a "type system" *without* data types?

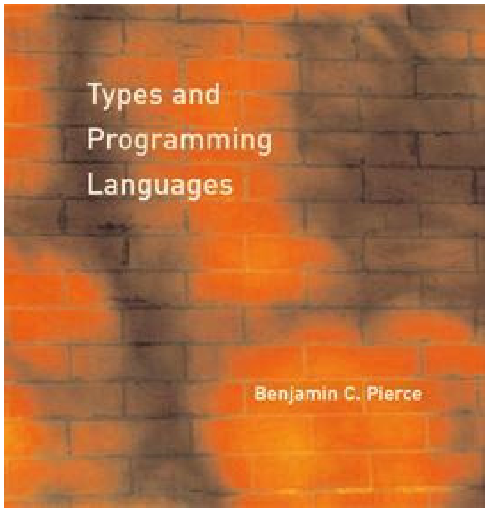
A type system [that works] without data types

```
fn main() {
 let i1 = ~42;
 let i2 = i1;                          // i1 is now invalid
 println!("Answer: {}", *i1); // compile-time error
}
```

This could almost be BCPL!

- with some added *typing rules*
- . . . that enforce linearity of selected data flows
- "linear typing" does not require > 1 data type

“Kinds of values” are not an essential characteristic

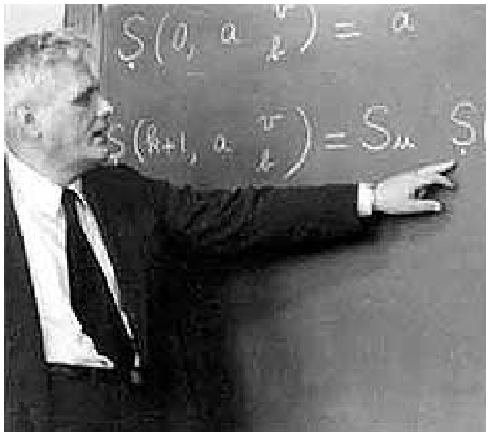


“A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.”

Benjamin Pierce
in *Types and programming languages*

So why do we call them “type systems” anyway?

Type systems as type discipline (1)



"Any finite sequence of primitive symbols is a formula. Certain formulas are distinguished as being well-formed and as having a certain type, in accordance with the following rules: …"

Alonzo Church
A formulation of the Simple Theory of Types
Journal of Symbolic Logic, June 1940

photo: Princeton University. CC-BY 3.0.

Type systems as type discipline (2)

"A type is defined as the range of significance of a propositional function. The division of objects into types is necessitated by the reflexive fallacies which otherwise arise. . . . Whatever contains an apparent variable must be of a [higher] type from the possible values of that variable."

Bertrand Russell
Mathematical logic as based on the Theory of Types
American Journal of Mathematics, July 1908

What is a type discipline?

"Types" classify expressions

- PLs: is E's output a suitable input to E's context?
- Russell: does E's range include its domain?

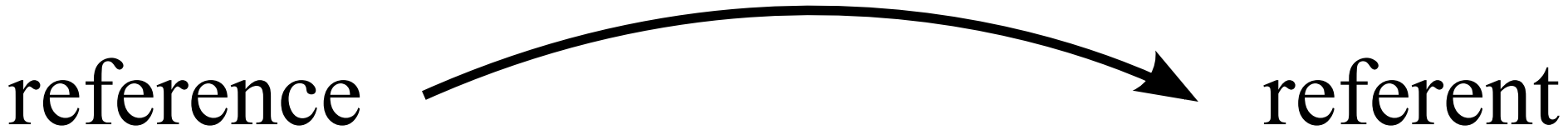
Rules, in terms of types, avoid unwanted constructions

- error states, e.g. "stuck"
- paradoxes

Straw dichotomies: two origins of "type", two mindsets

	"engineering"	"logic"
"types" means	[data] types	[expression] types
heritage	Fortran, Algol, …	λ-calculus, ML, …
goal	creating, maintaining	reasoning

Abstraction as distancing



“This library really provides the right abstractions.”

Abstraction as generality

"The code's notion of numeric quantities is very abstract."

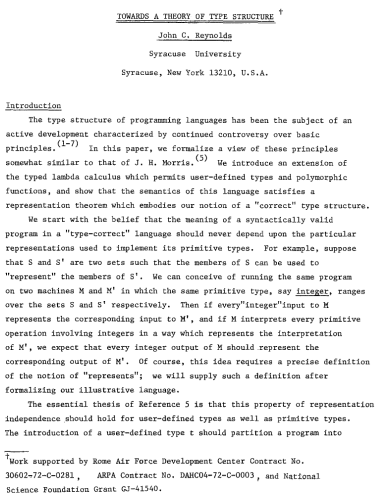
Some interesting code

```
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;
}
```

Meaningful? Useful?

Should it be possible to write this code (at user level)?

A design criterion for languages



TOWARDS A THEORY OF TYPE STRUCTURE †

John C. Reynolds

Syracuse University

Syracuse, New York 13210, U.S.A.

Introduction

The type structure of programming languages has been the subject of an active development characterized by continued controversy over basic principles.(1-7) In this paper, we formalize a view of these principles somewhat similar to that of J. H. Morris.(5) We introduce an extension of the typed lambda calculus which permits user-defined types and polymorphic functions, and show that the semantics of this language satisfies a representation theorem which embodies our notion of a "correct" type structure.

We start with the belief that the meaning of a syntactically valid program in a "type-correct" language should never depend upon the particular representations used to implement its primitive types. For example, suppose that S and S' are two sets such that the members of S can be used to "represent" the members of S'. We can conceive of running the same program on two machines M and M' in which the same primitive type, say integer, ranges over the sets S and S' respectively. Then if every "integer" input to M represents the corresponding input to M', and if M interprets every primitive operation involving integers in a way which represents the interpretation of M', we expect that every integer output of M should represent the corresponding output of M'. Of course, this idea requires a precise definition of the notion of "represents"; we will supply such a definition after formalizing our illustrative language.

The essential thesis of Reference 5 is that this property of representation independence should hold for user-defined types as well as primitive types. The introduction of a user-defined type t should partition a program into

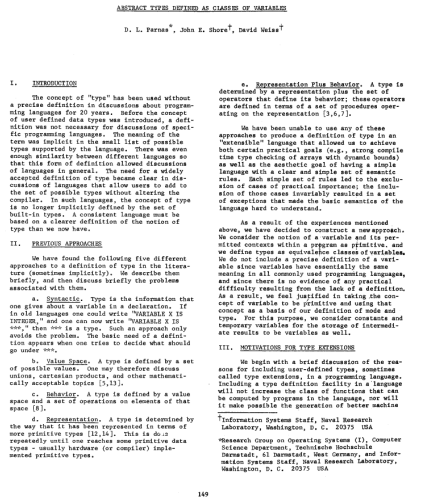
†Work supported by Rome Air Force Development Center Contract No. 30602-72-C-0281, ARPA Contract No. DAHC04-72-C-0003, and National Science Foundation Grant GJ-41540.

"The meaning of a syntactically valid program in a 'type-correct' language should never depend upon the particular representations used to implement its primitive types."

John C. Reynolds
Towards a theory of type structure
Proc. Colloque sur la Programmation, 1974.

Protect abstractions by *enforcement*: CLU, ML, …

A different design criterion for languages



ABSTRACT TYPES DEFINED AS CLASSES OF VARIABLES

D. L. Parnas*, John E. Shore†, David Weiss†

I. INTRODUCTION

The concept of "type" has been used without a precise definition in discussions about programming languages for 20 years. Before the concept of user defined data types was introduced, a definition was not necessary for discussions of specific programming languages. The meaning of the term was implicit in the small list of possible types supported by the language. There was even enough similarity between different languages so that this form of definition allowed discussions of languages in general. The need for a widely accepted definition of type became clear in discussions of languages that allow users to add to the set of possible types without altering the compiler. In such languages, the concept of type is no longer implicitly defined by the set of built-in types. A consistent language must be based on a clearer definition of the notion of type than we now have.

II. PREVIOUS APPROACHES

We have found the following five different approaches to a definition of type in the literature (sometimes implicitly). We describe them briefly, and then discuss briefly the problems associated with them.

a. Syntactic. Type is the information that one gives about a variable in a declaration. If in old languages one could write "VARIABLE X IS INTEGER," and one can now write "VARIABLE X IS ***," then *** is a type. Such an approach only avoids the problem. The basic need of a definition appears when one tries to decide what should go under ***.

b. Value Space. A type is defined by a set of possible values. One may therefore discuss unions, cartesian products, and other mathematically acceptable topics [3,13].

c. Behavior. A type is defined by a value space and a set of operations on elements of that space [8].

d. Representation. A type is determined by the way that it has been represented in terms of more primitive types [12,14]. This is done repeatedly until one reaches some primitive data types - usually hardware (or compiler) implemented primitive types.

e. Representation Plus Behavior. A type is determined by a representation plus the set of operators that define its behavior; these operators are defined in terms of a set of procedures operating on the representation [3,6,7].

We have been unable to use any of these approaches to produce a definition of type in an "extensible" language that allowed us to achieve both certain practical goals (e.g., strong compile time type checking of arrays with dynamic bounds) as well as the aesthetic goal of having a simple language with a clear and simple set of semantic rules. Each simple set of rules led to the exclusion of cases of practical importance; the inclusion of those cases invariably resulted in a set of exceptions that made the basic semantics of the language hard to understand.

As a result of the experiences mentioned above, we have decided to construct a new approach. We consider the notion of a variable and its permitted contexts within a program as primitive, and we define types as equivalence classes of variables. We do not include a precise definition of a variable since variables have essentially the same meaning in all commonly used programming languages, and since there is no evidence of any practical difficulty resulting from the lack of a definition. As a result, we feel justified in taking the concept of variable to be primitive and using that concept as a basis of our definition of mode and type. For this purpose, we consider constants and temporary variables for the storage of intermediate results to be variables as well.

III. MOTIVATIONS FOR TYPE EXTENSIONS

We begin with a brief discussion of the reasons for including user-defined types, sometimes called type extensions, in a programming language. Including a type definition facility in a language will not increase the class of functions that can be computed by programs in the language, nor will it make possible the generation of better machine

†Information Systems Staff, Naval Research Laboratory, Washington, D. C. 20375 USA

*Research Group on Operating Systems (1), Computer Science Department, Technische Hochschule Darmstadt, 61 Darmstadt, West Germany, and Information Systems Staff, Naval Research Laboratory, Washington, D. C. 20375 USA

149

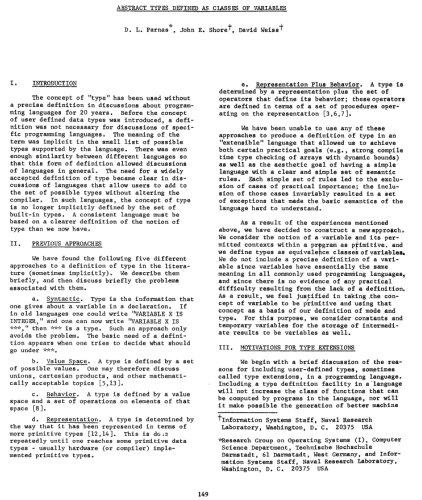
"However nice the aesthetic properties of a language may be, if it forces users to write duplicate programs or forces the code generated to be larger than otherwise necessary… the users of such a language will resort to the **dirtiest of dirty tricks** [when faced with] **time and space constraints**."

Parnas, Shore, Weiss

Abstract types defined as classes of variables

Proc. DADS, 1976

A different design criterion for languages



ABSTRACT TYPES DEFINED AS CLASSES OF VARIABLES

D. L. Parnas*, John E. Shore†, David Weiss†

I. INTRODUCTION

The concept of "type" has been used without a precise definition in discussions about programming languages for 20 years. Before the concept of user defined data types was introduced, a definition was not necessary for discussions of specific programming languages. The meaning of the term was implicit in the small list of possible types supported by the language. There was even enough similarity between different languages so that this form of definition allowed discussions of languages in general. The need for a widely accepted definition of type became clear in discussions of languages that allow users to add to the set of possible types without altering the compiler. In such languages, the concept of type is no longer implicitly defined by the set of built-in types. A consistent language must be based on a clearer definition of the notion of type than we now have.

II. PREVIOUS APPROACHES

We have found the following five different approaches to a definition of type in the literature (sometimes implicitly). We describe them briefly, and then discuss briefly the problems associated with them.

a. Syntactic. Type is the information that one gives about a variable in a declaration. If in old languages one could write "VARIABLE X IS INTEGER," and one can now write "VARIABLE X IS ***," then *** is a type. Such an approach only avoids the problem. The basic need of a definition appears when one tries to decide what should go under ***.

b. Value Space. A type is defined by a set of possible values. One way therefore discuss unions, cartesian products, and other mathematically acceptable topics [3,13].

c. Behavior. A type is defined by a value space and a set of operations on elements of that space [8].

d. Representation. A type is determined by the way that it has been represented in terms of more primitive types [12,14]. This is do.n repeatedly until one reaches some primitive data types - usually hardware (or compiler) implemented primitive types.

e. Representation Plus Behavior. A type is determined by a representation plus the set of operators that define its behavior; these operators are defined in terms of a set of procedures operating on the representation [3,6,7].

We have been unable to use any of these approaches to produce a definition of type in an "extensible" language that allowed us to achieve both certain practical goals (e.g., strong compile time type checking of arrays with dynamic bounds) as well as the aesthetic goal of having a simple language with a clear and simple set of semantic rules. Each simple set of rules led to the exclusion of cases of practical importance; the inclusion of those cases invariably resulted in a set of exceptions that made the basic semantics of the language hard to understand.

As a result of the experiences mentioned above, we have decided to construct a new approach. We consider the notion of a variable and its permitted contexts within a program as primitive, and we define types as equivalence classes of variables. We do not include a precise definition of a variable since variables have essentially the same meaning in all commonly used programming languages, and since there is no evidence of any practical difficulty resulting from the lack of a definition. As a result, we feel justified in taking the concept of variable to be primitive and using that concept as a basis of our definition of mode and type. For this purpose, we consider constants and temporary variables for the storage of intermediate results to be variables as well.

III. MOTIVATIONS FOR TYPE EXTENSIONS

We begin with a brief discussion of the reasons for including user-defined types, sometimes called type extensions, in a programming language. Including a type definition facility in a language will not increase the class of functions that can be computed by programs in the language, nor will it make possible the generation of better machine

†Information Systems Staff, Naval Research Laboratory, Washington, D. C. 20375 USA

*Research Group on Operating Systems (1), Computer Science Department, Technische Hochschule Darmstadt, 61 Darmstadt, West Germany, and Information Systems Staff, Naval Research Laboratory, Washington, D. C. 20375 USA

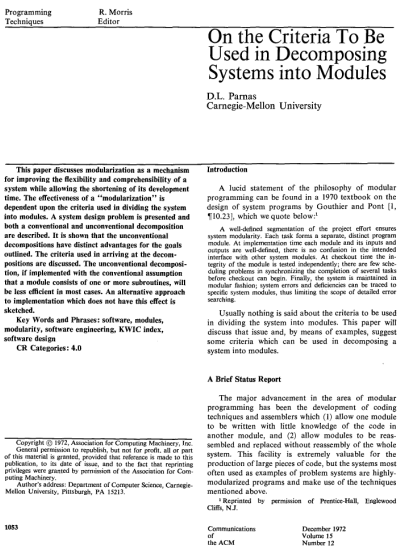
149

"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 users of such a language will resort to the **dirtiest of dirty tricks** [when faced with] **time and space constraints**."

Fortran, C, …, Smalltalk, Python, …

- fewer "dirty tricks", but still *technical debt*
- all mature languages? (Obj, unsafePerformIO, …)

Abstraction to reduce cost



Programming Techniques — R. Morris, Editor

On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University

This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and both a conventional and unconventional decomposition are described. It is shown that the unconventional decompositions have distinct advantages for the goals outlined. The criteria used in arriving at the decompositions are discussed. The unconventional decomposition, if implemented with the conventional assumption that a module consists of one or more subroutines, will be less efficient in most cases. An alternative approach to implementation which does not have this effect is sketched.

Key Words and Phrases: software, modules, modularity, software engineering, KWIC index, software design
CR Categories: 4.0

Author's address: Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213.

Introduction

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [1, ¶10.23], which we quote below:[1]

> A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules. At checkout time the integrity of the module is tested independently; there are few scheduling problems in synchronizing the completion of several tasks before checkout can begin. Finally, the system is maintained in modular fashion; system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching.

Usually nothing is said about the criteria to be used in dividing the system into modules. This paper will discuss that issue and, by means of examples, suggest some criteria which can be used in decomposing a system into modules.

A Brief Status Report

The major advancement in the area of modular programming has been the development of coding techniques and assemblers which (1) allow one module to be written with little knowledge of the code in another module, and (2) allow modules to be reassembled and replaced without reassembly of the whole system. This facility is extremely valuable for the production of large pieces of code, but the systems most often used as examples of problem systems are highly-modularized programs and make use of the techniques mentioned above.

[1] Reprinted by permission of Prentice-Hall, Englewood Cliffs, N.J.

"The formats of control blocks used in queues in operating systems and similar programs must be hidden within a 'control block module'. It is conventional to make such formats the **interfaces** between various modules. Because **design evolution forces frequent changes** on control block formats, such a decision often proves **extremely costly**.

D.L. Parnas

On the criteria to be used in decomposing systems into modules

CACM, December 1972

Data abstraction: some straw dichotomies

	"engineering"	"logic"
"types" means	[data] types	[expression] types
heritage	Fortran, Algol, …	λ-calculus, ML, …
goal	creating, maintaining	reasoning
heuristic	minimise cost	seek guarantees
heroes	Parnas	Reynolds
abstraction is	reference (generality)	generality (reference)
protected by	guidance	enforcement

On understanding… (1)

On Understanding Data Abstraction, Revisited

William R. Cook
University of Texas at Austin
wcook@cs.utexas.edu

Abstract

In 1985 Luca Cardelli and Peter Wegner, my advisor, published an ACM Computing Surveys paper called "On understanding types, data abstraction, and polymorphism". Their work kicked off a flood of research on semantics and type theory for object-oriented programming, which continues to this day. Despite 25 years of research, there is still widespread confusion about the two forms of data abstraction, *abstract data types* and *objects*. This essay attempts to explain the differences and also why the differences matter.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types; D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects

General Terms Languages

Keywords object, class, abstract data type, ADT

1. Introduction

What is the relationship between *objects* and *abstract data types* (ADTs)? I have asked this question to many groups of computer scientists over the last 20 years. I usually ask it at dinner, or over drinks. The typical response is a variant of "objects are a kind of abstract data type".

This response is consistent with most programming language textbooks. Tucker and Noonan [57] write "A class is itself an abstract data type". Pratt and Zelkowitz [51] intermix discussion of Ada, C++, Java, and Smalltalk as if they were all slight variations on the same idea. Sebesta [54] writes "the abstract data types in object-oriented languages... are called classes." He uses "abstract data types" and "data abstraction" as synonyms. Scott [53] describes objects in detail, but does not mention abstract data types other than giving a reasonable discussion of opaque types.

So what is the point of asking this question? Everyone knows the answer. It's in the textbooks. The answer may be a little fuzzy, but nobody feels that it's a big issue. If I didn't press the issue, everyone would nod and the conversation would move on to more important topics. But I do press the issue. I don't say it, but they can tell I have an agenda.

My point is that the textbooks mentioned above are wrong! Objects and abstract data types are not the same thing, and neither one is a variation of the other. They are fundamentally different and in many ways complementary, in that the strengths of one are the weaknesses of the other. The issues are obscured by the fact that most modern programming languages support both objects and abstract data types, often blending them together into one syntactic form. But syntactic blending does not erase fundamental semantic differences which affect flexibility, extensibility, safety and performance of programs. Therefore, to use modern programming languages effectively, one should understand the fundamental difference between objects and abstract data types.

While objects and ADTs are fundamentally different, they are both forms of *data abstraction*. The general concept of data abstraction refers to any mechanism for hiding the implementation details of data. The concept of data abstraction has existed long before the term "data abstraction" came into existence. In mathematics, there is a long history of abstract representations for data. As a simple example, consider the representation of integer sets. Two standard approaches to describe sets abstractly are as an *algebra* or as a *characteristic function*. An algebra has a sort, or collection of abstract values, and operations to manipulate the values[1]. The characteristic function for a set maps a domain of values to a boolean value, which indicates whether or not the value is included in the set. These two traditions in mathematics correspond closely to the two forms of data abstraction in programming: algebras relate to abstract data types, while characteristic functions are a form of object.

In the rest of this essay, I elaborate on this example to explain the differences between objects and ADTs. The

[1] The sort, or carrier set, of an algebra is often described as a set, making this definition circular. Our goal is to define specific set abstractions with restricted operations, which may be based on and assume a more general concept of sets

"Despite 25 years of research, there is still widespread confusion about the two forms of data abstraction, abstract data types and objects. This essay attempts to explain the differences and also why the differences matter."

William R. Cook
On understanding data abstraction, revisited
Onward! 2009

On understanding… (2)

On Understanding Data Abstraction, Revisited

William R. Cook
University of Texas at Austin
wcook@cs.utexas.edu

Abstract

In 1985 Luca Cardelli and Peter Wegner, my advisor, published an ACM Computing Surveys paper called "On understanding types, data abstraction, and polymorphism". Their work kicked off a flood of research on semantics and type theory for object-oriented programming, which continues to this day. Despite 25 years of research, there is still widespread confusion about the two forms of data abstraction, *abstract data types* and *objects*. This essay attempts to explain the differences and also why the differences matter.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types; D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects

General Terms Languages

Keywords object, class, abstract data type, ADT

1. Introduction

What is the relationship between *objects* and *abstract data types* (ADTs)? I have asked this question to many groups of computer scientists over the last 20 years. I usually ask it at dinner, or over drinks. The typical response is a variant of "objects are a kind of abstract data type".

This response is consistent with most programming language textbooks. Tucker and Noonan [57] write "A class is itself an abstract data type". Pratt and Zelkowitz [51] intermix discussion of Ada, C++, Java, and Smalltalk as if they were all slight variations on the same idea. Sebesta [54] writes "the abstract data types in object-oriented languages... are called classes." He uses "abstract data types" and "data abstraction" as synonyms. Scott [53] describes objects in detail, but does not mention abstract data types other than giving a reasonable discussion of opaque types.

So what is the point of asking this question? Everyone knows the answer. It's in the textbooks. The answer may be a little fuzzy, but nobody feels that it's a big issue. If I didn't press the issue, everyone would nod and the conversation would move on to more important topics. But I do press the issue. I don't say it, but they can tell I have an agenda.

My point is that the textbooks mentioned above are wrong! Objects and abstract data types are not the same thing, and neither one is a variation of the other. They are fundamentally different and in many ways complementary, in that the strengths of one are the weaknesses of the other. The issues are obscured by the fact that most modern programming languages support both objects and abstract data types, often blending them together into one syntactic form. But syntactic blending does not erase fundamental semantic differences which affect flexibility, extensibility, safety and performance of programs. Therefore, to use modern programming languages effectively, one should understand the fundamental difference between objects and abstract data types.

While objects and ADTs are fundamentally different, they are both forms of *data abstraction*. The general concept of data abstraction refers to any mechanism for hiding the implementation details of data. The concept of data abstraction has existed long before the term "data abstraction" came into existence. In mathematics, there is a long history of abstract representations for data. As a simple example, consider the representation of integer sets. Two standard approaches to describe sets abstractly are as an *algebra* or as a *characteristic function*. An algebra has a sort, or collection of abstract values, and operations to manipulate the values[1]. The characteristic function for a set maps a domain of values to a boolean value, which indicates whether or not the value is included in the set. These two traditions in mathematics correspond closely to the two forms of data abstraction in programming: algebras relate to abstract data types, while characteristic functions are a form of object.

In the rest of this essay, I elaborate on this example to explain the differences between objects and ADTs. The

[1] The sort, or carrier set, of an algebra is often described as a set, making this definition circular. Our goal is to define specific set abstractions with restricted operations, which may be based on and assume a more general concept of sets.

"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."

What is “type abstraction” anyway?

It’s really about *allowable references*.

- often checked statically, but needn’t be
- separable from any notion of type (witness BCPL)
- could be checked dynamically!

Allowable references, enforced dynamically

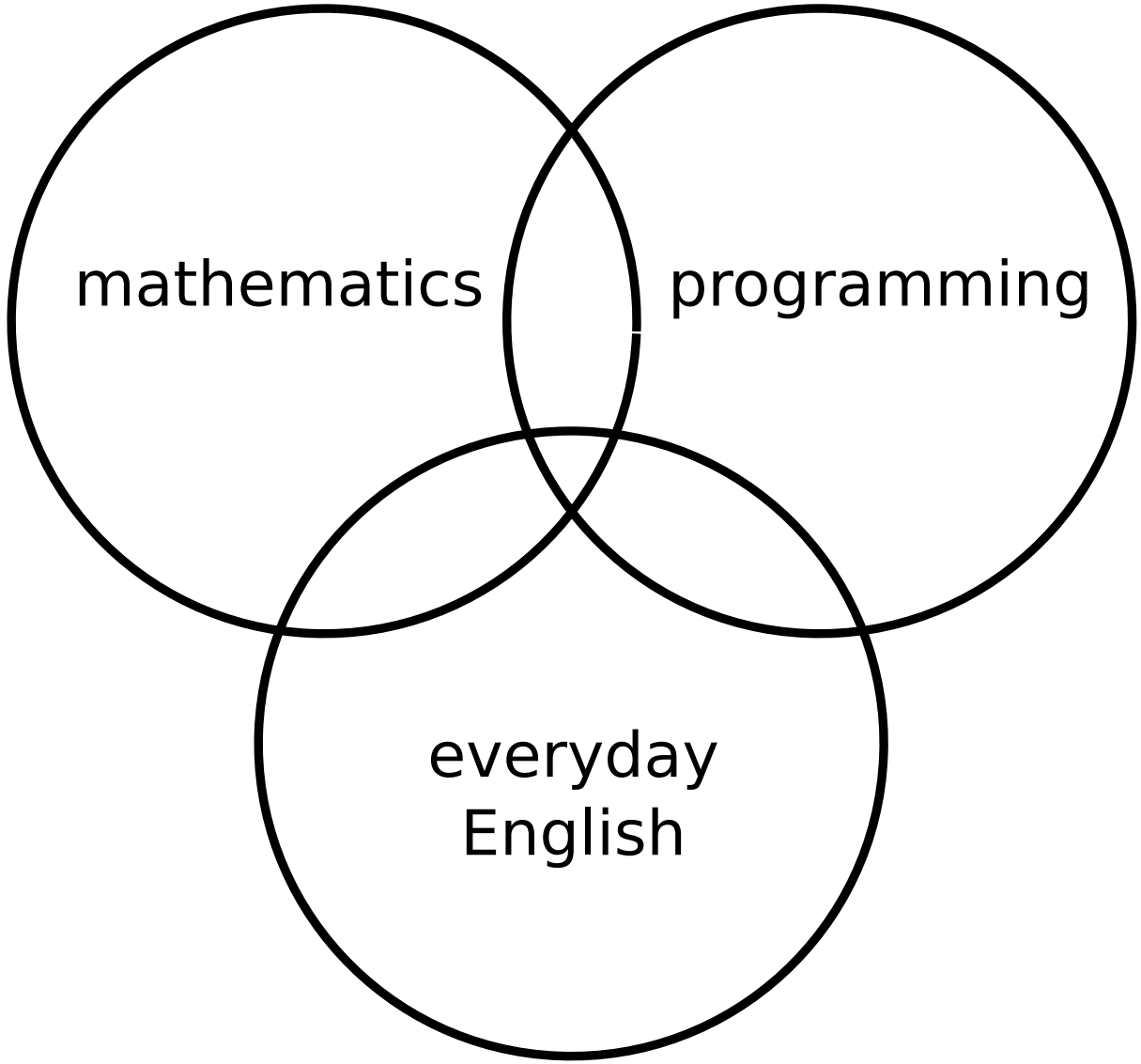
```
struct point2d { int x, y; };

int f(void *p) {
  // ...
  if (cond) {
    return ((point2d*) p)->x;
  }
}
```

In cases where cond is true,

- *p had better be a point2d (correctness)
- f had better be allowed to know this (hiding)

. . . but this is a dynamic property! (undecidability)

An inconvenient pun



The missing link is still missing

Early PL literature uses “type” in everyday English

- pre-Algol 60: “type”, “kind”, “form” [of data]
- afterwards: mostly “type”

Literature on logic inherited Russell’s notion of “type”

- Church, Quine, Robinson, Reynolds. . .

Remarkably little citation cross-over!

- “high-order languages”

How we can do better

Ideas?

Why we need to do better

"Engineering" and "logic" are converging!

- programming and proof tools are converging
  - Coq, Isabelle/HOL
  - Idris, Agda, …
  - Dafny, Whiley, …
- proofs about real programs/systems/languages
  - seL4
  - CompCert, CakeML, …
  - C++ concurrency model, instruction sets, …

Conclusions

The essence of *data types* is as *interpretations*

- abstracted by a def–use relation

Types, from logic, are orthogonal.

- types label expressions, in service of proof rules

Abstraction follows from reference, but is divergent

- “engineering” and “logic” attitudes run deep

Thanks for your attention. Questions?

Maybe we should say “class” instead?

> “[Many] typed object-oriented languages, including Modula-3, C++, Trellis and Simula, are based on the identification of classes and types.”

Cook, Hill, Canning
Inheritance is not subtyping
POPL ’90

What is abstraction?

abstraction

/əbˈstrakʃ(ə)n/

noun

1. the quality of dealing with ideas rather than events.
   "topics will vary in degrees of abstraction"

2. freedom from representational qualities in art.
   "geometric abstraction has been a mainstay in her work"

Origin

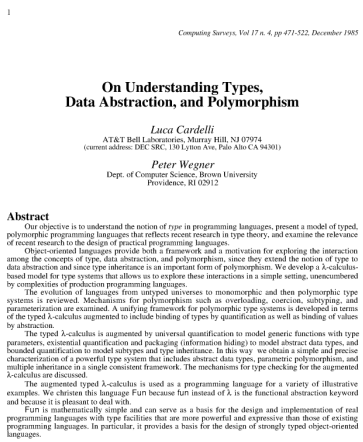
late Middle English: from Latin *abstractio(n-)*, from the verb *abstrahere* 'draw away' (see abstract).

We “draw away” from details, for two reasons:

- structural optimisation (“factoring”)
- generality

These are intertwined, but distinct!

Admitting the arbitrary: a logical catastrophe



1

Computing Surveys, Vol 17 n. 4, pp 471-522, December 1985

On Understanding Types, Data Abstraction, and Polymorphism

Luca Cardelli
AT&T Bell Laboratories, Murray Hill, NJ 07974
(current address: DEC SRC, 130 Lytton Ave, Palo Alto CA 94301)

Peter Wegner
Dept. of Computer Science, Brown University
Providence, RI 02912

Abstract

Our objective is to understand the notion of *type* in programming languages, present a model of typed, polymorphic programming languages that reflects recent research in type theory, and examine the relevance of recent research to the design of practical programming languages.

Object-oriented languages provide both a framework and a motivation for exploring the interaction among the concepts of type, data abstraction, and polymorphism, since they extend the notion of type to data abstraction and since type inheritance is an important form of polymorphism. We develop a λ-calculus-based model for type systems that allows us to explore these interactions in a simple setting, unencumbered by complexities of production programming languages.

The evolution of languages from untyped universes to monomorphic and then polymorphic type systems is reviewed. Mechanisms for polymorphism such as overloading, coercion, subtyping, and parameterization are examined. A unifying framework for polymorphic type systems is developed in terms of the typed λ-calculus augmented to include binding of types by quantification as well as binding of values by abstraction.

The typed λ-calculus is augmented by universal quantification to model generic functions with type parameters, existential quantification and packaging (information hiding) to model abstract data types, and bounded quantification to model subtypes and type inheritance. In this way we obtain a simple and precise characterization of a powerful type system that includes abstract data types, parametric polymorphism, and multiple inheritance in a single consistent framework. The mechanisms for type checking for the augmented λ-calculus are discussed.

The augmented typed λ-calculus is used as a programming language for a variety of illustrative examples. We christen this language Fun because fun instead of λ is the functional abstraction keyword and because it is pleasant to deal with.

Fun is mathematically simple and can serve as a basis for the design and implementation of real programming languages with type facilities that are more powerful and expressive than those of existing programming languages. In particular, it provides a basis for the design of strongly typed object-oriented languages.

"[Representation-dependent code] allows a data representation to be manipulated in ways that were not intended, with **potentially disastrous** results. For example, use of an integer as a pointer **can cause arbitrary modifications** to programs and data."

Cardelli & Wegner
On understanding types, data abstraction and polymorphism
ACM Computing Surveys, December 1985

Data abstraction: some straw dichotomies

	"engineering"	"logic"
"types" means	[data] types	[expression] types
heritage	Fortran, Algol, …	λ-calculus, ML, …
goal	creating, maintaining	reasoning
heuristic	minimise cost	seek guarantees
heroes	Parnas	Reynolds
abstraction is	reference (generality)	generality (reference)
protected by	guidance	enforcement
cost of mistakes	bounded	catastrophic

Definite referential statements are existential

"'The present King of France is not bald' [can be said to mean] 'There is an entity which is now King of France and is not bald'."

Bertrand Russell
On denoting
Mind, 1905

Definite referential statements are existential

"If C is a denoting phrase, say 'the term having the property F', then 'C has property ϕ' means 'one and only one term has the property F, and that one has the property ϕ'. Thus 'the present King of France is not bald' [can be said to mean] 'There is an entity which is now King of France and is not bald'."

Bertrand Russell
On denoting
Mind, 1905