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...

<table>
<thead>
<tr>
<th>C</th>
<th>int float char arrays pointers functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML</td>
<td>int real tuples lists functions</td>
</tr>
<tr>
<td>Perl</td>
<td>scalars arrays hashes</td>
</tr>
</tbody>
</table>
Most languages support multiple data types

Both built-in...

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>ML</th>
<th>Perl</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>int</td>
<td>int</td>
<td>scalars</td>
</tr>
<tr>
<td></td>
<td>float</td>
<td>real</td>
<td>arrays</td>
</tr>
<tr>
<td></td>
<td>char</td>
<td>tuples</td>
<td>arrays</td>
</tr>
<tr>
<td></td>
<td>arrays</td>
<td>lists</td>
<td>hashes</td>
</tr>
<tr>
<td></td>
<td>pointers</td>
<td>functions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>functions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... and user-defined

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>ML</th>
<th>Perl</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>structs</td>
<td>ADTs</td>
<td>modules</td>
</tr>
<tr>
<td></td>
<td>unions</td>
<td>ADTs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>enums</td>
<td>ADTs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADTs</td>
<td></td>
</tr>
</tbody>
</table>
Data types $\mapsto$ “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)
“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
rules about well-defined executions

rules about well-formed code
“This storage is for holding integers.”

```c
int a, b;
```

Not all languages hold us to these statements.

```c
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

interpretation

representation
Some languages without [multiple] data types

LET manhattan (x1, y1, x2, y2) = VALOF
$(
    \text{RESULT IS } \text{abs}(x1 - x2) + \text{abs}(y1 - y2)
)$

choose () {
    if [ -z "$1" ]; then echo $1; else echo $2; fi
}
manhattan (x1, y1, x2, y2)
{
    return abs(x1 - x2) + abs(y1 - y2);
}
Thought experiment: data types *minimally*, in BCPL (1)

```c
// 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)

```c
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 \textit{minimally}, in BCPL (2)

```c
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

\begin{itemize}
  \item managing storage
  \item determining operations’ well-definedness
  \item determining programs’ well-formedness
\end{itemize}
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)

```c
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

reference \rightarrow referent
One kind of abstraction

use $\rightarrow$ definition
(point2d) p1 → struct point2d
{ ... };
One kind of abstraction

(int point2d) p1

interpretation

struct point2d
{
...
};

... related to representation
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?
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?
“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.
“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
<table>
<thead>
<tr>
<th>“types” means</th>
<th>“engineering”</th>
<th>“logic”</th>
</tr>
</thead>
<tbody>
<tr>
<td>heritage</td>
<td>[data] types</td>
<td>[expression] types</td>
</tr>
<tr>
<td>goal</td>
<td>creating, maintaining</td>
<td>reasoning</td>
</tr>
</tbody>
</table>
“This library really provides the right abstractions.”
“The code’s notion of numeric quantities is very abstract.”
Some interesting code

```c
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)?*
“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*

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

“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

<table>
<thead>
<tr>
<th>“types” means</th>
<th>[data] types</th>
<th>[expression] types</th>
</tr>
</thead>
<tbody>
<tr>
<td>heritage</td>
<td>Fortran, Algol, …</td>
<td>λ-calculus, ML, …</td>
</tr>
<tr>
<td>goal</td>
<td>creating, maintaining</td>
<td>reasoning</td>
</tr>
<tr>
<td>heuristic</td>
<td>minimise cost</td>
<td>seek guarantees</td>
</tr>
<tr>
<td>heroes</td>
<td>Parnas</td>
<td>Reynolds</td>
</tr>
<tr>
<td>abstraction is</td>
<td>reference (generality)</td>
<td>generality (reference)</td>
</tr>
<tr>
<td>protected by</td>
<td>guidance</td>
<td>enforcement</td>
</tr>
</tbody>
</table>
“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
“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!
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

- Mathematics
- Programming
- Everyday English
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
/ˈəbˈstrækʃ(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"
What else?

**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!
"[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

<table>
<thead>
<tr>
<th></th>
<th>“engineering”</th>
<th>“logic”</th>
</tr>
</thead>
<tbody>
<tr>
<td>“types” means</td>
<td>[data] types</td>
<td>[expression] types</td>
</tr>
<tr>
<td>heritage</td>
<td>Fortran, Algol, …</td>
<td>λ-calculus, ML, …</td>
</tr>
<tr>
<td>goal</td>
<td>creating, maintaining</td>
<td>reasoning</td>
</tr>
<tr>
<td>heuristic</td>
<td>minimise cost</td>
<td>seek guarantees</td>
</tr>
<tr>
<td>heroes</td>
<td>Parnas</td>
<td>Reynolds</td>
</tr>
<tr>
<td>abstraction is</td>
<td>reference (generality)</td>
<td>generality (reference)</td>
</tr>
<tr>
<td>protected by</td>
<td>guidance</td>
<td>enforcement</td>
</tr>
<tr>
<td>cost of mistakes</td>
<td>bounded</td>
<td>catastrophic</td>
</tr>
</tbody>
</table>
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
“If $C$ is a denoting phrase, say ‘the term having the property $F$’, then ‘$C$ has property $\phi$’ means ‘one and only one term has the property $F$, and that one has the property $\phi$’. 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*