Types

Programming Language Design 2021

Hans Hüttel

1 March 2021

Part I

Learning goals

The learning goals this time

- To understand why the notion of type exists and is useful
- To be able to distinguish between static and dynamic type checking and to understand the rationales behind these two approaches and their limitations
- To understand the various other ways of classifying type systems and apply them to distinguish between different type systems
- To understand commonly occurring type constructs
- To understand the notion of slack

What are types?

The notion of types give us a way of classifying entities in a program.

The notion of type in computer science dates back to the early work on the foundations of mathematics by Russell and Whitehead.

But the idea of course dates back much longer. We know that it does not make sense to perform aritmetic on incompatible values. For instance, the addition 1 kilogram + 1 meter is nonsense (we return to this later!)

Typed vs. untyped

Many programming languages have a notion of type that involves type checking. Examples of typed languages are

- C
- Java
- Scheme
- Haskell

Other languages have no notion of type; they are untyped.

- PHP
- Forth
- Prolog

The theory of types by Russell and Whitehead

Bertrand Russell and Alfred North Whitehead wrote *Principia Mathematica* in 1903 in order to provide mathematics with foundations based on logic.

They did not succeed, but their work was the beginning of mathematical logic and set theory – and of the study of types.

An unpleasant set

$$D = \{x \mid x \notin x\}$$

D is the set of all sets that do not have themselves as an element. But do we have that $D \in D$?

- If $D \in D$, then that implies that $D \notin D$
- If $D \notin D$, then that implies that $D \in D$

The only way out of this mess is to classify sets.

The way out: Russell's theory of types

In set theory, our entities are sets. So a type classifies a set of sets. Each type is a natural number.

- Simple elements, that are not sets, have type 0.
- A set has type n if its elements have type $n-1$.

This means that

$$D = \{x \mid x \notin x\}$$

cannot have a type. For if D had a type n, then D would also have type $n-1$.

The way out: Russell's theory of types

In set theory, our entities are sets. So a type classifies a set of sets. Each type is a natural number.

- Simple elements, that are not sets, have type 0.
- A set has type n if its elements have type $n-1$.

This means that

$$D = \{x \mid x \notin x\}$$

cannot have a type. For if D had a type n, then D would also have type $n-1$.

The morale is that *If an entity is well-typed, it cannot "go wrong"*. This is precisely what types in programming languages are meant to ensure.

But what are types, really?

- We can think of a type T as denoting a set – the collection of entities that have type T.
- Or we can think of a type T as being a specification of the properties that entities of type T have.

Both views have their strong and not so strong points.

Are types sets?

We can think of a type as a set.

- The type **int** represents the set of integers
- The type **bool** represents the set of truth values, $\{\text{true}, \text{false}\}$
- ...

Are types sets?

We can think of a type as a set.

- The type **int** represents the set of integers
- The type **bool** represents the set of truth values, $\{\text{true}, \text{false}\}$
- ...

What about function types? In OCaml and F# we can write

```
datatype ff = I of int | F of (ff → ff)
```

But for any set A, the set of functions from A to A has strictly greater size than A. If A is the set of integers, the size of the set $A \to A$ of functions is that of the real numbers. `ff` appears not be a well-defined set.

Are types specifications?

An entity that has the function type $T_A \to T_R$ is a function that

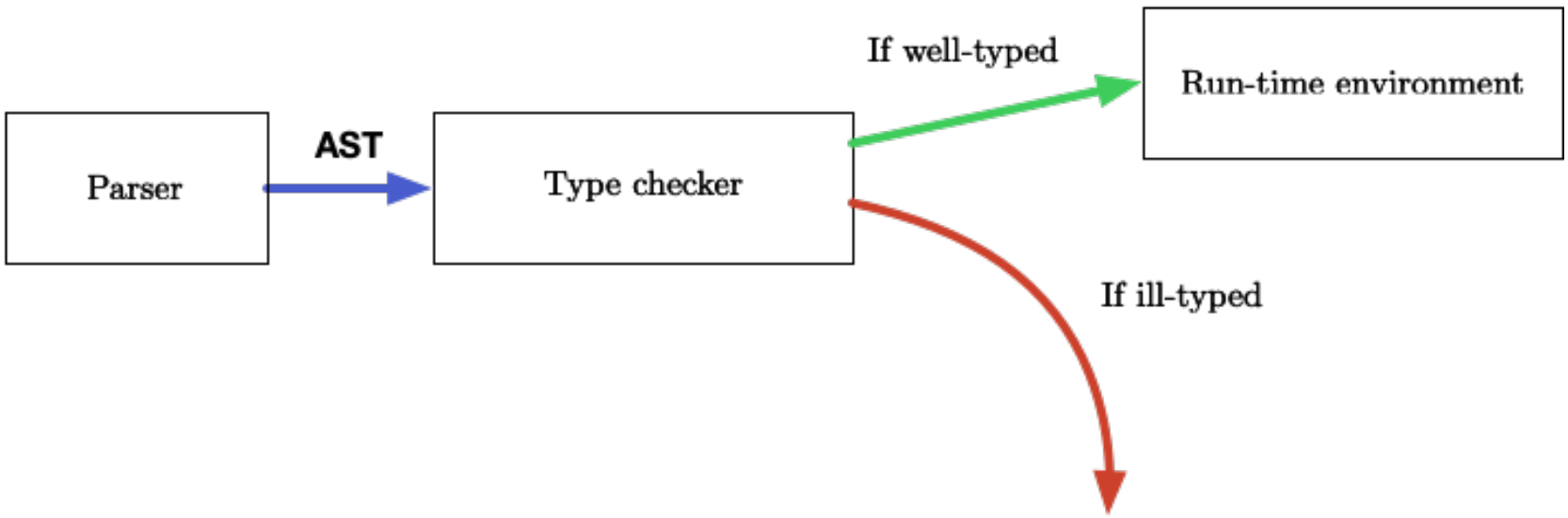
- for any argument that has type T_A
- gives us a result of type T_R

If we think of it in this way, a type is a specification of behaviour.

Part II

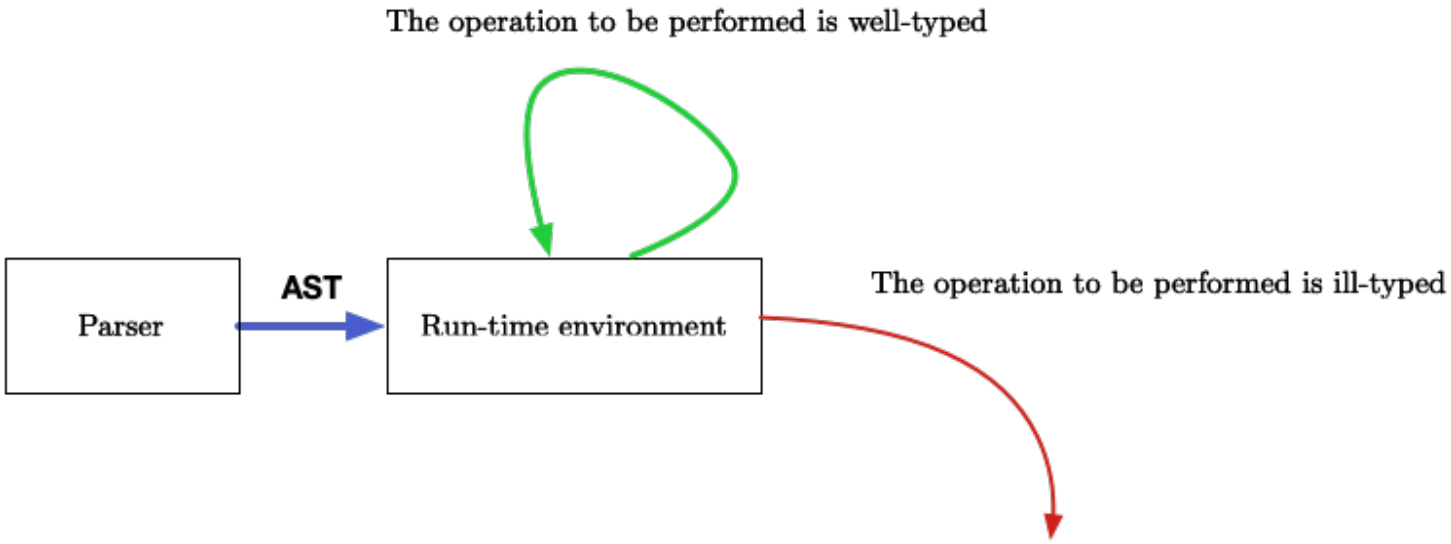
Ways of classifying type disciplines

Static vs. dynamic typing



With static typing, properties are checked at compile time.
Example: F#.

Static vs. dynamic typing



With dynamic typing, properties are checked at run-time. Example: Python.

For and against static typing

Cardelli, Pierce and others argue that

- Type systems can capture important recurring programming errors
- Type systems have methodological advantages for code development; for instance, type annotations can improve readability
- Type systems help with allocation of storage; for instance, decimal numbers and whole numbers should be stored differently in memory.

For and against static typing

Lamport and others argue that

- Types make it more cumbersome to develop programs
- Type systems cannot capture the run-time errors that really matter – such as null dereferencing
- Run-time errors such as "method not available" that are caused by missing type checks are rare

Strong vs. weak typing

There is no precise definition of this distinction.

In 1974, Liskov and S. Zilles defined a strongly-typed language as one in which

> *. . . whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function.*

According to this view,

Strong typing Always verifies if an operation is valid before doing it. Example: Lisp.

Weak typing Does not verify if an operation is valid, assumes it is and just performs it. Example: C.

Strong vs. weak typing

A better notion (certainly, its definition is generally agreed upon) is that of type safety.

A type system is safe wrt. a class of run-time errors if it is the case that whenever a program P is well-typed in the system, it will not lead to a run-time error of this particular form.

Implicit vs. explicit typing

Another important distinction is

Explicit typing Types of entities are given explicitly in the program. Example: Java.

Implicit typing Types of entities are not written in the program but discovered (inferred) at compile time based on how entities are constructed or used. Examples: Haskell, Rust.

How and when are properties verified?

- Dynamically: Equip (tag) values with type information, and check this tag at runtime immediately before operation on value is attempted.
- Statically: By (prior to execution) pseudo-evaluating expressions and statements using types instead of values.

In the session on semantics, we shall see how we can express pseudo-evaluation as type judgements and make our informal understanding of type systems mathematically precise.

Type rules (aka pseudo-evaluation)

If E records the types of the variables that we know, then

$$E \vdash e : T$$

can be read as saying that whenever we know the type bindings in E, then we know that the expression e has type T.

We can then express the conditions of the type system as inference rules such as

$$\frac{E \vdash e_1 : \mathsf{Int} \quad E \vdash e_2 : \mathsf{Int}}{E \vdash e_1 + e_2 : \mathsf{Int}}$$

Much more about this in a later session!

What is the right approach?

Many languages use a combination of the approaches we have just mentioned.

Some properties are checked at compile time and others at run time. Some types are inferred, while others must be specified.

Type conversion

Some languages allow for type conversion, that is, converting a value from one type to another.

- Integer to float (usually safe) or float to integer (can overflow)
- Use of object at supertype (safe)
- Downcast (use of object at subtype), can fail
- Pointer converted to or from number – result may be invalid

Conversion

Some design issues are

- Is conversion implicit (this is called type coercion or do we use an explicit conversion operator?
- Is conversion safe or unsafe? (Can it fail? Is the result valid?)
- Do we change the value representation or just how the value is viewed?

Part III

Simple types

Simple and composite types

Types can be simple or composite.

Atomic types

Atomic types (or base types) are used for values that are not *naturally* divided into components. Some well-known examples are

- Numbers
- Characters
- Booleans
- Symbols
- Enumerated types

Deep down, every values can of course be seen as a sequence of bits, but, for example, a character is not of the same type as its bit pattern.

Numbers

In mathematics, sets of numbers are usually infinite (and can even be uncountable). In programming languages, language designers usually impose restrictions. Among the usual ones are that

- Integers are bounded to, e.g, -2^n to 2^n-1. Many languages support integers that are bounded only by memory.
- Floating-point have limited precision and bounded exponent.
- Fixed-point numbers have limited size and precision.
- Bounds (such as integer range or floating-point precision) may be implementation specific.
- Multiple number types with different bounds can (and usually do) coexist in one language.

Computable reals with unbounded range and precision can be implemented, but "native" language support for these is rare.

Characters

- Different character sets are available: ASCII, ISO 8859-1, Unicode, ...
- Some languages specify exact sets, others just specify a minimal set.
- Ordering by number-code can give different ordering in different character sets/encodings.
- Ordering can take local language conventions into account (for example position of Ö in German/Swedish/Danish/Turkish).

Booleans

Logical values that are either true or false. Truth values can be represented as

- Values of separate type (e.g, in Pascal and F#).
- Integers interpreted as truth values in certain contexts ($0 \equiv$ false, all else true) – used in, e.g., C. Can give issues with equality.
- Any value can be used as boolean, e.g., 0/null object/null pointer/empty string $\equiv$ false, all else true – used in, e.g., JavaScript.

Enumerated types and symbols

Enumerated: Finite list of named values. May have order or numeric code. Some allow parameters: We return to these as sum types.

Symbols: Run-time extensible set of names. Internalised representation (same name implies same address) allows faster equality test than strings.

Enumerated types are typically found in statically typed languages (Pascal, C, ...), while symbols are typically found in dynamically typed languages (LISP, JavaScript, ...).

Part IV

Composite types

Composite types

Composite values are built from multiple components and usually have composite types. Examples include

- Products / tuples
- Records / structs
- Collections:
  - Arrays
  - Lists / streams
  - Sets / multisets
- Unions / Sums
- Function types
- Recursive types

Product / Tuple types

Correspond to cartesian products ($A \times B$, $A \times B \times C$, ...) of sets.

- Components identified by *position*.
- In statically typed languages, different components may be of different type (unlike arrays), but the number of components is fixed by the type. In dynamically types languages, anything goes.
- Components accessed by projection functions or pattern matching.
- Special cases: Empty tuple (`unit`, `void`, `()`), tuple with one element (usually identified with the element type).

Generally supported in functional languages, limited support in other languages.

Record / Struct types

Products where components are identified by *name* rather than by position.

- Suited for products of many components: Easier to remember name than position.
- Order of components may (C) or may not (ML) be specified.
- In some languages (COBOL), field names are globally unique, in other languages (Pascal, C), different records/structs can use overlapping (or even identical) sets of field names.
- Pattern matching can omit irrelevant fields in pattern (in ML by writing ...).
- Records and structs are semantically equivalent to tuples, so many languages support only one of these, or define one in terms of the other.

Arrays

Collection of elements accessed by index. Differences from tuples:

- Size need not be known at compile time. Can sometimes be extended at runtime.
- Usually implemented as contiguous memory area – fast access.
- Index can be a computed value instead of a constant.
- Elements must (in statically types languages) be of same type.
- Can be multi-dimensional. Layout can depend on language.
- Index set can sometimes be other than $1 \ldots n$ or $0 \ldots n$:
  - Subrange such as `-10..10` (Pascal).
  - (Almost) arbitrary type: Associative arrays (Lua). Implemented as hash table or search tree.

Lists and streams

Sequences of elements. Difference from arrays:

- Usually implemented as linked list of elements.
- Easy to add element to front, middle or back. In functional languages usually only in front, so tail can be shared.
- Random access can be slow. Can be made faster with tree-structured implementation.
- Streams are like lists, but generated on demand, and can conceptually be infinite. In lazy languages, lists $\equiv$ streams.

Strings

Strings are sequences of characters.

- Can be implemented as arrays (Pascal, C) or lists (Haskell) of characters, but are often implemented differently than both of these (to allow fast concatenation etc.).
- Operations for value-to-string (show/sprintf) and string-to-value (read/sscanf) are often built in for many types.
- Lexicographic order may be defined for strings even when not for arrays or lists. Order may be based on character-set codes or dictionary order, which may be locale-dependent (e.g, Aa sorted as Å in Danish, and ö as o in German).
- Support for concatenation of strings and extractions of substrings is common, even when these are not supported for arrays.

Sets and multisets

- A set is a collection where order and multiplicity does not matter.
- A multiset is a collection where order does not matter, but multiplicity does.
- Operations for union, intersection, membership test, subset test, emptyness test, ...
- Sets over small domains can be implemented as bit vectors.
- Sets and multisets can both be implemented as search trees or hash tables.

Union and sum types

Value that can be one of several specified types, where the choice is specified by name (field name or constructor name).

- C `union` types: Look like structs, but only one field exists at any one time. Unchecked.
- Pascal variant records: Tag-field determines which field is defined. Checked.
- ML/F#/Haskell datatypes: Constructor name determines alternative. Accessed by pattern matching.
- Java/C#/Swift: Enumerated type with fields.
- OO languages: Subclasses of (possibly abstract) superclass.

Union/Sum types are not needed in dynamically-typed languages – a variable can already assume different types at different times.

Function types

In functional languages, functions are values. Most often, they have arrow types $t_1 \to t_2$. In Haskell we might write

```
fact :: Integer -> Integer

fact 0 = 1
fact n = n * fact (n-1)
```

In Pascal and C, the type syntax is similar to that of function definitions:

```
function twice(function f(x : real) : real;y : real)

double twice(double f(double), double y)
```

Function types

Function type may provide information in addition to argument and result types:

- Which exceptions may be thrown (Java, Swift)
- Whether non-local memory can be read or written (GCC C++)
- Shared/unshared value, ownership (Clean, Mercury, Rust).

Functional values with static scoping and polymorphic types can emulate almost any other type, including products, sums, lists, ...

Recursive types

A recursive type is a type that is partially defined in terms of itself (by referring to its own name).

- Algebraic datatypes in functional languages are recursive.

```
data Tree = Leaf Integer | Branch Tree Tree
```

- In imperative or OO languages, struct or class type with fields can contain pointers/references to elements of the same struct or class type:

```
typedef struct List {
                      int head;
                      struct List *tail
                    } *list;
```

Be careful!

If we have a programming language with lazy evaluation, infinite composite values are allowed.

If not, we must take care in order to avoid having infinite values. Some ways of doing this are by means of

- Reference through a pointer, that may be a null pointer, or
- At least one non-recursive case in a sum type, or
- The recursive type reference occurs inside a list, option, or function type.

Part V

Type equivalence

When are two types the same?

- When they have the same name, or
- When they have identical structure?

The case for name equivalence

Suppose we want to build a system for registering students and their dogs. In C++ we can define the following types.

```
struct student
    {
     char name[100];
     int age;
     float weight;
        }
```

```
struct dog
    {
     char name[100];
     int age;
     float weight;
        }
```

Values of type student have constituents with the same names and types and the constituents of values of type dog. But there is a good reason why we want two distinct types.

The case for structural equivalence

Suppose we want to build a system for manipulating bytes.

```
typedef unsigned char byte;
```

Values of type byte are just characters and we want to be able to use the same operations on bytes that we would use for manipulating characters. Here there is a good reason why we want the byte type not to be distinct from **unsigned char**.

Name equivalence versus structural equivalence

Structural equivalence allows

- Unnamed compound types (e.g., pairs)
- Type inference

Name equivalence allows

- Types that are represented the same way but used differently to be distinguished, e.g., a pair of doubles to be either coordinates or a complex number.
- Recursive types.

Many languages use a combination. For example, ML and Haskell use name equivalence for datatypes and structural equivalence for other types. F# uses name equivalence for classes also.

Units of measure in F#

Floating point and signed integer values in F# can have associated units of measure. This way we can enable the compiler to check that arithmetic relationships have the correct unit.

```
[<Measure>] type m
[<Measure>] type s
[<Measure>] type kg
[<Measure>] type N = kg m / s^2
let G = 6.67408e-11<m^3/(kg s^2)>
let m1 = 1000.0<kg>
let m2 = 2000.0<kg>
let r = 3000.0<m>
let f = G*m1*m2/(r*r)
```

Gives `val f : float<kg m/s^2> = 1.483128889e-11`.

m1 + r is not well-typed – we cannot add meters and kilograms!

Part VI

Type safety

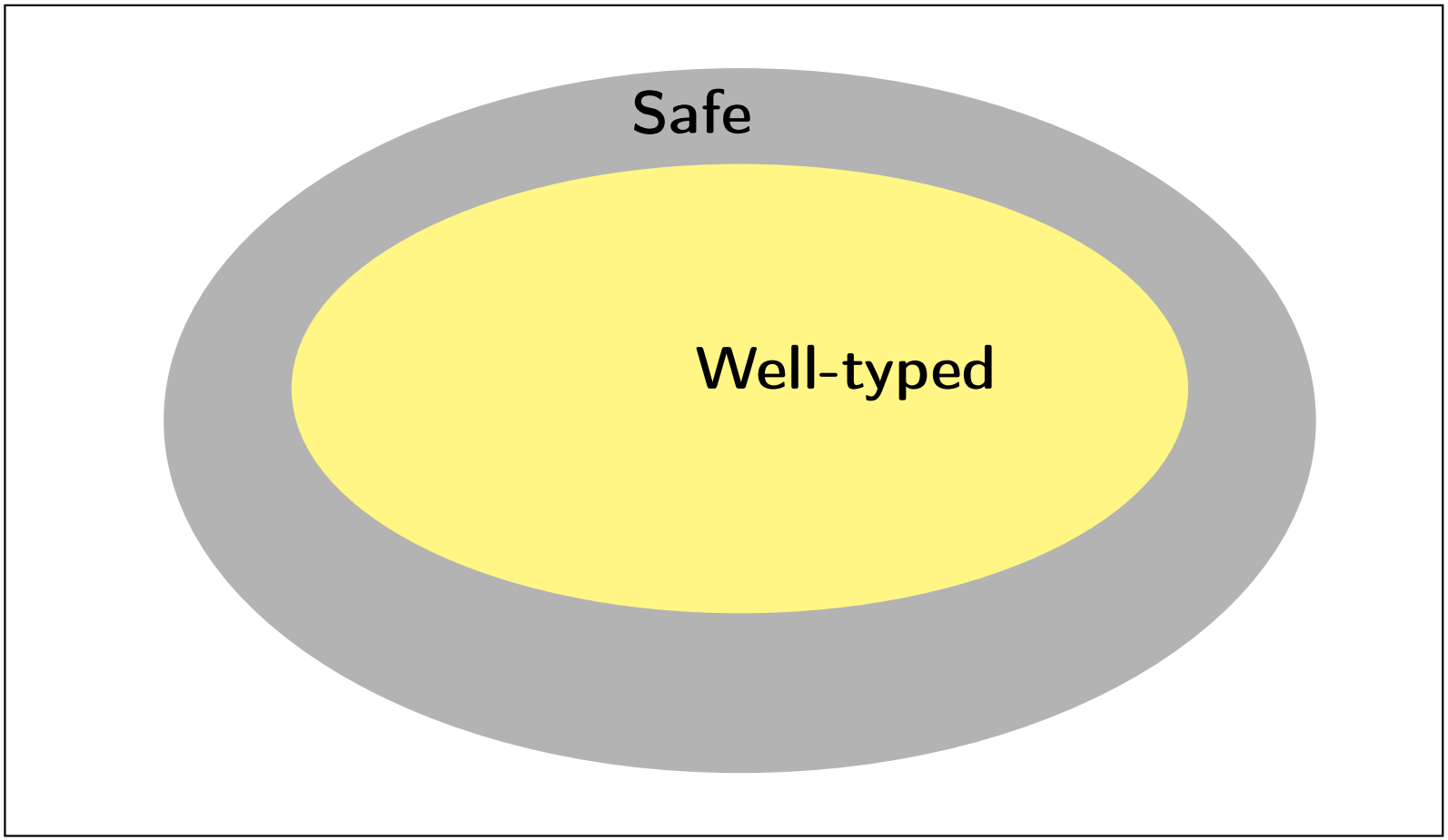
Well-typed programs cannot go wrong

One of the main selling points of static typing is that of type safety. If a program does not contain type errors, then we know that certain run-time errors cannot happen.

Seen in this way, type checking is therefore a form of static program analysis.

But there is a limit to what we can do in this way.

Every reasonable type system has slack



A tiny example of slack

The following statement in the Pascal language

```
program innocent;
var x : integer;
begin
      if 0 = 0 then
                              write('Nothing happens')
                        else
                              x := 1 + true
end
```

is rejected by the Pascal type system but will not lead to a run-time error, as the else-branch in the conditional statement will not be visited.

Dealing with slack

The previous example was a bit far-fetched.

How about

```
id x = x

val myvalue = (id 7, id True)
```

In a simple type system, this tiny Haskell program would not be well-typed – but it ought to be.

The way ahead is to enrich our type system; this is the focus of the session on polymorphism.