Why Types Matter

Zheng Gao

CREST, UCL
Russell’s Paradox

Let $R$ be the set of all sets that are not members of themselves. Is $R$ a member of itself?

- If so, this contradicts with $R$’s definition
- If not, by definition, $R$ should contain itself

Formalism in naïve set theory:

Let $R = \{ x \mid x \not\in x \}$, then $R \in R \iff R \not\in R$
The Barber Paradox

There is a town with a male barber who shaves all and only those men who do not shave themselves. **Who shaves the barber?**
The Barber Paradox

There is a town with a male barber who shaves all and only those men who do not shave themselves. **Who shaves the barber?**

- If the barber does not shave himself, according to the rule he must shave himself.
The Barber Paradox

There is a town with a male barber who shaves all and only those men who do not shave themselves. **Who shaves the barber?**

- If the barber does not shave himself, according to the rule he must shave himself.

- If he does shave himself, according to the rule he will not shave himself.
The Barber Paradox

There is a town with a male barber who shaves all and only those men who do not shave themselves. Who shaves the barber?

- If the barber does not shave himself, according to the rule he must shave himself.
- If he does shave himself, according to the rule he will not shave himself.

Naïve set theory contains contradiction.
Types to the Rescue

Constructs a hierarchy of types.

Any object is built only from those of higher types, which prevents circular referencing.

1) a barber as a citizen of the town, who shaves himself

and

2) a barber as a professional, who shaves others

are of different types.
Type Theory

An alternative to set theory as a foundation for mathematics, in which each term has a type

Simply typed $\lambda$-calculus is one of the many forms of type theory, which consists of

- Base types
- Only one type constructor, $\rightarrow$, used to model the type of functions
The Evolution of Types
The Evolution of Types

Initially, types are a mechanism to avoid self-reference
The Evolution of Types

Initially, types are a mechanism to avoid self-reference.

In λ-calculus, types are a method for describing computations.
The Evolution of Types

Initially, types are a mechanism to avoid self-reference.

In λ-calculus, types are a method for describing computations.

In early ML languages, types are sets of all possible values that a computation can produce.
The Evolution of Types

Initially, types are a mechanism to avoid self-reference.

In λ-calculus, types are a method for describing computations.

In early ML languages, types are sets of all possible values that a computation can produce.

In effect systems and monads, types are sets of values and the computation’s side effects.
The Evolution of Types

Initially, types are a mechanism to avoid self-reference.

In λ-calculus, types are a method for describing computations.

In early ML languages, types are sets of all possible values that a computation can produce.

In effect systems and monads, types are sets of values and the computation’s side effects.
Type System

A tractable method that assigns types to syntactic phrases that compose a program, and automatically checks whether the usage of these phrases comply with their types.

An over-approximation of the run-time behaviour of program terms.
Static & Dynamic Type Checking
Static & Dynamic Type Checking

Static type checking

Source Code → Compilation → Executable → Execution
Static & Dynamic Type Checking

Early error detection

Static type checking

Source Code → Compilation → Executable → Execution
Static & Dynamic Type Checking

- Early error detection
- Increased run-time efficiency

Source Code → Compilation → Executable → Execution
Static & Dynamic Type Checking

- Early error detection
- Increased run-time efficiency
- Better documentation

Diagram:
- Source Code → Compilation → Executable → Execution

Static type checking

[Image of flowchart with nodes labeled Source Code, Compilation, Executable, Execution, and an arrow indicating the flow from Source Code to Compilation to Executable to Execution.]
Static & Dynamic Type Checking

Static type checking:
- Early error detection
- Increased run-time efficiency
- Better documentation

Dynamic type checking

Source Code → Compilation → Executable → Execution
Static & Dynamic Type Checking

Static type checking:
- Early error detection
- Increased run-time efficiency
- Better documentation

Dynamic type checking:
- Reduced implementation overhead

Source Code → Compilation → Executable → Execution
Static & Dynamic Type Checking

- Early error detection
- Increased run-time efficiency
- Better documentation

- Reduced implementation overhead
- Better expressibility

Source Code → Compilation → Executable → Execution

Static type checking

Dynamic type checking
Why We Care

Generally, almost all real-world programming languages have type systems which offers multiple benefits.

Specifically for GI/GP, type systems have the promise to guide the search and avoid the construction of invalid individuals.
Java’s Static Type Checking

Suppose we have:

class A {
    A me() {
        return this;
    }

    public void doA() {
        System.out.println("Do A");
    }
}

class B extends A {
    public void doB() {
        System.out.println("Do B");
    }
}

class C extends A{
    public void doC() {
        System.out.println("Do C");
    }
}
Java’s Static Type Checking

```java
new B().me().doB();

new B().me().doA();

((B) new B().me()).doB();

((C) new B().me()).doC();
```
Java’s Static Type Checking

```java
new B().me().doB();

new B().me().doA();

((B) new B().me()).doB();

((C) new B().me()).doC();
```

Illegal. Compiler thinks `new B().me()` returns an object of class A, but at runtime, this returns an objects of class B.
Java’s Static Type Checking

Illegal. Compiler thinks new B().me() returns an object of class A, but at runtime, this returns an object of class B.

Legal.
Java’s Static Type Checking

Illegal. Compiler thinks new B().me() returns an object of class A, but at runtime, this returns an objects of class B.

new B().me().doB();

new B().me().doA();

((B) new B().me()).doB();

((C) new B().me()).doC();

Legal.
Java’s Static Type Checking

*Illegal*. Compiler thinks `new B().me().doB()` returns an object of class `A`, but at run-time, this returns an object of class `B`.

*Legal.*

*Legal.*

*Legal.* But throws cast exception at run-time.
Hindley Milner’s Type System

One of the most famous type systems for the typed λ-calculus with parametric polymorphism:

- A fast (nearly linear time) algorithm that automatically infer types of the constructs from their usage

- A set of typing rules, e.g.

\[
\frac{\Gamma \vdash e_0 : \tau \rightarrow \tau' \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0 \ e_1 : \tau'} \quad [\text{App}]
\]
HM Example

Let us assume that we have a function myFunc of type:

\[
\text{myFunc : ADT} \rightarrow \text{int}
\]

And we want to infer the type of a function someFunc

\[
\text{someFunc (x) + myFunc (x)}
\]
Step One

\texttt{someFunc \( (x) \) + myFunc \( (x) \)}
Step One

\[ \text{someFunc} (x) + \text{myFunc} (x) \]

\[ x : \alpha \]
Step Two

\[ \text{someFunc} (x) + \text{myFunc} (x) \]

\[ x : \alpha \quad \text{ADT} \longrightarrow \text{int} \]
Step Two

\[ \text{someFunc}(x) + \text{myFunc}(x) \]

\[ x : \alpha \]

\[ \text{ADT} \rightarrow \text{int} \]

\[ \alpha = \text{ADT} \]
Step Two

someFunc (x) + myFunc (x)

\[ x : \alpha \]
\[ \alpha = \text{ADT} \]
\[ x : \text{ADT} \]
Step Three

\[ \text{someFunc}(x) + \text{myFunc}(x) \]

\[ x : \text{ADT} \]

\[ \text{ADT} \rightarrow \text{int} \]
Step Three

\[ \text{someFunc} (x) + \text{myFunc} (x) \]

\[ x : \text{ADT} \quad \rightarrow \quad \text{int} \]

\[ + : \text{int} \rightarrow \text{int} \rightarrow \text{int} \]
Step Three

\[ \text{someFunc}(x) + \text{myFunc}(x) \]

\[ x : \text{ADT} \]

\[ \text{someFunc} : \text{ADT} \rightarrow \text{int} \]

\[ + : \text{int} \rightarrow \text{int} \rightarrow \text{int} \]
Polymorphism

The provision of a single interface to entities of different types
Parametric Polymorphism

Generic programming in programming languages

```java
class List<T> {
    class Node<T> {
        T elem;
        Node<T> next;
    }
    Node<T> head;
    int length() { ... }
}
List<B> map(Func<A,B> f, List<A> xs) {
    ...
}
```

Rank-N polymorphic function is a function whose parameters are Rank-(N-1) polymorphic
Ad Hoc Polymorphism

Function overloading in programming languages

```plaintext
function Add( x, y : Integer ) : Integer;
begin
    Add := x + y
end;

function Add( s, t : String ) : String;
begin
    Add := Concat( s, t )
end;
```
Inclusion Polymorphism

**Inheritance** creates inclusion polymorphism (subtyping)

```
abstract class Animal {
    abstract String talk();
}

class Cat extends Animal {
    String talk() {
        return "Meow!";
    }
}

class Dog extends Animal {
    String talk() {
        return "Woof!";
    }
}
```
Inclusion Polymorphism

**Inheritance** creates inclusion polymorphism (subtyping)

```java
abstract class Animal {
    abstract String talk();
}

class Cat extends Animal {
    String talk() {
        return "Meow!";
    }
}

class Dog extends Animal {
    String talk() {
        return "Woof!";
    }
}
```

Cat < Animal

Dog < Animal
HM Limitations

- Limited to rank 1 parametric polymorphism
- Does not support ad hoc polymorphism
- No notion of subtyping
Suppose we have subtyping $B < A$, any function that takes arguments of type $A$ is expected to takes arguments of type $B$ as well.

\[ \text{someFunc} (x) + \text{myFunc} (x) \]

$$x : \alpha \quad \text{ADT} \rightarrow \text{int}$$

$$\alpha = \text{ADT} \quad ???$$

$$x : \text{ADT} \quad ???$$
Limitation Example One

Suppose we have subtyping $B < A$, any function that takes arguments of type $A$ is expected to take arguments of type $B$ as well.

\[
some\text{Func}\ (x) + \text{myFunc}\ (x)
\]

\[
\begin{align*}
x : \alpha & \quad \text{ADT} \rightarrow \text{int} \\
\alpha & = \text{ADT} ??? \\
x & : \text{ADT} ???
\end{align*}
\]

$\alpha$ could be any subtype of ADT
In HM, an assumption set may contain at most one typing assumption for an construct.

The operator <, for example, has types:

```plaintext
char ⟷ char ⟷ bool
int ⟷ int ⟷ bool
```

But it does not have the type:

```plaintext
∀α.α ⟷ α ⟷ bool
```

So any single typing is either too narrow or too wide.
Intersection Types

Allow a term to have multiple types by introducing a type constructor \( \land \), a universal type \( \omega \) used for untypable constructs, and the following typing rules:

\[
\begin{align*}
M : (\sigma_1 \land \sigma_2) & \quad M : (\sigma_1 \land \sigma_2) \\
M : \sigma_1 & \\
M : \sigma_2 & \\
M : (\sigma_1 \land \sigma_2) &
\end{align*}
\]

(\(\land I\))

In practice, intersection types enable function overloading.
Union Types

The dual notion of intersection types, which introduces a type constructor \( \triangledown \) and similar typing rules.

In C / C++, union types are the construct `union`
Consider the following code snippet in C++:

```c++
typedef struct {
    char c;
    bool b;
} ADT;

typedef union {
    int i;
    ADT a;
} unionType;

void foo(unionType x, int y) {};
void foo(unionType x, float y) {};
```

The type of function `foo` would be:

\[
((\text{int} \lor \text{ADT}) \rightarrow \text{int} \rightarrow \text{void}) \land ((\text{int} \lor \text{ADT}) \rightarrow \text{float} \rightarrow \text{void})
\]
Example

Consider the following code snippet in C++:

```c++
typedef struct {
    char c;
    bool b;
} ADT;

typedef union {
    int i;
    ADT a;
} unionType;

void foo(unionType x, int y) {};
void foo(unionType x, float y) {};
```

The type of function `foo` would be:

```
((int ∨ ADT) → int → void) ∧ ((int ∨ ADT) → float → void)
```
Consider the following code snippet in C++:

typedef struct {
  char c;
  bool b;
} ADT;

typedef union {
  int i;
  ADT a;
} unionType;

void foo(unionType x, int y) {};
void foo(unionType x, float y) {};

The type of function *foo* would be:

\[
((\text{int} \lor \text{ADT}) \rightarrow \text{int} \rightarrow \text{void}) \land ((\text{int} \lor \text{ADT}) \rightarrow \text{float} \rightarrow \text{void})
\]
Example

Consider the following code snippet in C++:

```c++
typedef struct {
    char c;
    bool b;
} ADT;

typedef union {
    int i;
    ADT a;
} unionType;

void foo(unionType x, int y) {};
void foo(unionType x, float y) {};
```

The type of function `foo` would be:

\(((\text{int} \lor \text{ADT}) \rightarrow \text{int} \rightarrow \text{void}) \land ((\text{int} \lor \text{ADT}) \rightarrow \text{float} \rightarrow \text{void}))\)
Example

Consider the following code snippet in C++:

```c++
typedef struct {
  char c;
  bool b;
} ADT;

typedef union {
  int i;
  ADT a;
} unionType;

void foo(unionType x, int y) {};
void foo(unionType x, float y) {};
```

The type of function `foo` would be:

\[(\text{int } \lor \text{ ADT}) \rightarrow \text{ int } \rightarrow \text{ void}) \land (\text{int } \lor \text{ ADT}) \rightarrow \text{ float } \rightarrow \text{ void})\]
A general tool that automatically replaces certain types, together with the corresponding operations if necessary, of a program with new ones.
Potential Applications

Reducing energy consumption

Precision tracking and improvement for FP programs

New mutation operators in GI/GP

Taint analysis

Symbolic execution

Auto-transplantation
Intersection Types in Retype

We use intersection types to cleanly model function overloading, because Retype may generate new overloads of an existing operator.

Consider the following code snippet:

```c
int main() {
    int a, b;
    b = a + 2;
    a = b + ext_func(a);
    return 0;
}
```

**Assumption:** an external function `ext_func` of type `int ⟷ int`  
**Objective:** retype `int` to ADT
Intersection Types in Retype

We use intersection types to cleanly model function overloading, because Retype may generate new overloads of an existing operator.

Consider the following code snippet: 

```c
int main() {
    int a, b;
    b = a + 2;
    a = b + ext_func(a);
    return 0;
}
```

**Assumption:** an external function `ext_func` of type `int → int`

**Objective:** retype int to ADT
Intersection Types in Retype

We use intersection types to cleanly model function overloading, because Retype may generate new overloads of an existing operator.

Consider the following code snippet:

**Assumption:** an external function ext_func of type int \(\rightarrow\) int

**Objective:** retype int to ADT

```
int main() {
    int a, b;
    b = a + 2;
    a = b + ext_func(a);
    return 0;
}
```

**Before retyping**

\[+ : \text{int} \rightarrow \text{int} \rightarrow \text{int}\]
Intersection Types in Retype

We use intersection types to cleanly model function overloading, because Retype may generate new overloads of an existing operator.

Consider the following code snippet:

```
int main() {
    int a, b;
    b = a + 2;
    a = b + ext_func(a);
    return 0;
}
```

**Assumption:** an external function `ext_func` of type `int --> int`

**Objective:** retype `int` to ADT
Intersection Types in Retype

We use intersection types to cleanly model function overloading, because Retype may generate new overloads of an existing operator.

Consider the following code snippet:

```c
int main() {
    ADT a, b;
    b = a + 2;
    a = b + ext_func(a);
    return 0;
}
```

Assumption: an external function `ext_func` of type `int` → `int`

Objective: retype `int` to `ADT`

After retyping

- `+: ADT` → `ADT` → `ADT`
- `+: ADT` → `int` → `ADT`