1. Gradual typing [3]

- **Goal:** the programmer can control precisely how much type-checking is done statically versus dynamically.
- **Pros:**
  - Rapid prototyping
  - The code is more flexible and easier to maintain
- **Cons:**
  - Often reduced to an all-or-nothing approach
  - Performance issues

Example [4]:

```plaintext
let succ : Int -> Int = ...
let notb : Bool -> Bool = ...
let f (condition : Bool) (x : ?) : ? =
  if condition then succ x
  else notb x
```

2. Set-theoretic types [1]

- **Goal:** interprete types as sets of values and use set-theoretic operations (union, intersection, etc.) on types.
- **Pros:**
  - Powerful type system (stronger than simple types)
  - Subtyping is reduced to set containment
  - Concepts such as overloading can be precisely typed
- **Cons:**
  - Type reconstruction is undecidable
  - The syntactic overhead hinders development speed

Example:

```plaintext
let f (condition : Bool) (x : (Int | Bool)) :
  (Int | Bool) =
  if condition then succ (Int) x (+ Typecast +)
  else notb (Bool) x
```

3. Motivation

Advantages of mixing gradual types and set-theoretic types

- Finer grained transition between static typing and dynamic typing
- Reduce the syntactic overhead of set-theoretic types
- Stronger static guarantees on gradually-typed programs

Reject ill-typed applications, without the need for explicit casts:

```plaintext
let f (condition : Bool) (x : (Int | Bool) & ?) :
  (Int | Bool) =
  if condition then succ x
  else notb x
```

More precise return types, depending on the type of the previous parameters:

```plaintext
let f : (Bool -> (Int & ?) -> Int)
  & (Bool -> (Bool & ?) -> Bool) =
  fun condition x ->
  if condition then succ x
  else notb x
```

Remark. The previous type is equivalent to the following type:

```
Boolean -> (((Int & ?) -> Int) & ((Bool & ?) -> Bool))
```

4. Semantic interpretation and subtyping

**Syntax.**

```plaintext
\( t \in \text{StaticTypes} ::= t \lor t \land \neg t \lor t \to b \lor \text{Empty} \lor \text{Any} \)
\( \tau \in \text{GradualTypes} ::= \tau \lor \tau \land \neg \tau \lor \tau \to \tau \lor \text{Empty} \lor \text{Any} \)
\( b ::= \text{Int} \lor \text{Bool} \lor \ldots \)
```

**Abstract interpretation [2].** A gradual type is interpreted as a set of static types. Functions can then be lifted from static types to gradual types:

```
GType \xrightarrow{\gamma} GType
\gamma(\tau) = \text{SType}
\gamma(\tau \lor \text{Int}) = \{t \lor \text{Int} | t \in \text{SType}\}
```

Minimal and maximal concretisations are obtained by substituting all occurrences of \( \tau \) by \( \text{Empty} \) or \( \text{Any} \), depending on their position (covariant or contravariant). For example,

```
\max((\tau \to \tau) \lor \tau) = (\text{Empty} \to \text{Any}) \lor \text{Any}
```

**Gradual subtyping** for set-theoretic types reduces to static (semantic) subtyping:

```
\sigma \subseteq \tau \iff \exists (s, t) \in \sigma \times \tau, s \leq t
```

5. Typing applications

By only partially concretizing gradual types we can obtain a finite concretisation function \( \gamma \) and a set-theoretic abstraction function:

```
\alpha : \text{P}(\text{GType}) \to \text{GType}
\alpha(\sigma) = \bigvee_{\sigma \in \mathcal{S}}
```

The domain and result type of an application are defined by lifting their static counterparts using the finite operations.

```
dom : \text{GType} \to \text{SType}
dom(\text{Int} \to \text{Int} \& ?) = \text{Any} \left((\text{Int} \to \text{Int}) \land ? \right) \lor \text{Int} = \text{Int} \land ?
dom(\text{Int} \to \text{Int}) \lor ? = \text{Int} \left((\text{Int} \to \text{Int}) \lor ? \right) \lor \text{Bool} = \text{Bool} \land ?
```

6. Compiling applications

**Two domains are defined** for every function type: a safe domain \( \text{dom}^{s} \) and a possible domain \( \text{dom}^{p} \):

```
\text{dom}^{s}(\tau) = \text{Any} \text{ as } \tau \text{ possibly represents any function.}
\text{dom}^{p}(\tau) = \text{Empty} \text{ as } \tau \text{ is not always a function.}
```

**Casts are inserted** if the type of the argument a function is not always in its safe domain:

- If \( \Gamma f :: (\text{Int} \to \text{Int}) \land ? (\text{Bool} \to \text{Bool}) \) and \( \Gamma x :: ? \)
  - Then \( f x \) compiles to \( f (\text{Int} \lor \text{Bool}) x \)
- If \( \Gamma f :: ? \land (\text{Bool} \lor \text{Int}) \) and \( \Gamma x :: \text{Int} 
  - Then \( f x \) compiles to \( (\text{Int} \to ?) f \) \( x \)

**Results:*

- Compilation is type-preserving
- Every well-typed term without gradual type reduces to a value
- Every well-typed term reduces to a value or a cast error

7. Bibliography


