Subtyping: Overview and Implementation

CS5218: Principles of Program Analysis

Chen Jingwen
National University of Singapore
jingwen.chen@u.nus.edu

1 ABSTRACT

Type systems has been a topic of heavy research in programming language theory for decades, with implementations in different languages across paradigms. In object oriented languages, class based type systems describe a hierarchy between objects using a concept known as subtyping. Subtyping describes relations between types, allowing terms of a type to be used safely in places where terms of another type is expected. This increases the flexibility of existing type systems by increasing expressiveness while maintaining well-typedness. In this technical report, we will give an overview of the motivation and formalism of subtyping systems. We will also provide a Haskell implementation of record datatype subtyping in a functional toy language to demonstrate a typechecking algorithm. Lastly, we will explore related work in the field in the decades following its conception.

2 INTRODUCTION

Type systems are a central feature of programming languages and they come in different manifestations, stemming from decades of research in programming language theory. Despite implementation differences, the one thing in common is to prevent entire classes of erroneous program behaviours from happening.

Cardelli's 1996 paper, Type Systems [8], provides a high level overview on the subject, ranging from first order type systems (a la Simply Typed Lambda Calculus) to System `F<:`. A collection of type system flavours is summarised in the following list:

(1) First order (System F_1): Pascal, Algol68
(2) Second order (System F_2): ML, Haskell, Modula-3
(3) Object-oriented (OO), class-based: Java, C++
(4) Dependent types: Idris, Agda
(5) Dynamic: Smalltalk, Ruby

Zooming in on typed object-oriented systems, the dominating feature is the *type hierarchy* via *subclasses*, which describes a relation of *attribute and behaviour inheritance* between objects. However, the variety of implementations and optimizations of subclassing systems makes understanding and formalising this relation a difficult endeavour. *Subtyping* is a popular approach to formalise such relations, and it was a concept introduced by Cardelli with his seminal paper "A Semantics of Multiple Inheritance" [5].

The purpose of this report is to present a notion of typing and subtyping, an exploration on the evolution of research around subtyping in the past decades, and an implementation of a static typechecker for a language with a subtyping system. In the next section, we will describe what subtyping is and the benefits that it brings to a type system. Next, we will formalize the inference rules for a language with subtyping, alongside a concrete implementation of a typechecker in a toy programming language with subtyping. Finally, we will explore the related work in the field of subtyping.

3 SUBTYPING

3.1 Overview

Traditionally, type systems rigidly restrict the usage of typed terms in locations based on the idea of type equality. This results in programs where it is not obvious why typechecking fails, even though it seems natural in both syntax and semantics. For example, the function application

```
(fn (x: float) => x + 2) (2 :: int)
```

seems that it should be well-typed as the `+` operation permits additions with operands of both `int` and `float` types, but since they are not equal under classical type systems, the expression is ill-typed.

Subtyping is the formalisation of relations between types with the goal of allowing terms of a type to be used in locations where terms of other types are expected. For example, we want to allow operations that are defined on reals, $\mathbb{R}$, to work with integers, $\mathbb{Z}$, by forming a subtype relation between $\mathbb{Z}$ and $\mathbb{R}$. Pierce described this to be the *principle of safe substitution* in his book, Types and Programming Languages [26], where terms of one type can be *safely substituted* in place of a term of another type without incurring runtime type errors. This is also known as Liskov's substitution principle, where safety is asserted on behavioural properties in subtype relations [20].

As a core concept utilised in object-oriented (OO) languages for the hierarchical organization of objects, subtyping is often implemented in the form of *subclassing*. Classes, the templates that objects are instantiated from, are ordered hierarchically in the form of *superclasses* and *subclasses*.

In "A Semantics of Multiple Inheritance", Cardelli analysed popular OO languages (Simula, Lisp & Smalltalk) in their representations of objects [5], and decided to focus on representing objects as *records*, which is essentially the *Cartesian product* type with labels. The main advantage of using the concept of records (with functional components) was for simplicity, because it is possible to statically determine the fields of a record at compile-time, hence removing any need for dynamical analysis at runtime for invalid field accesses. He then presented the grammar and semantics of a static, strongly typed functional language with multiple inheritance, and record and variant subtyping. He also provided sound type inference and type checking algorithms on the semantics.

In a later paper, Type Systems [8], he described subtyping as a form of set inclusion, where terms of a type can be understood as elements belonging to a set, and the subtyping relation is understood as the subset relation between these sets of elements.

3.2 Implementation of FunSub

Before diving into an overview of the fundamental subtyping concepts, we will first describe a Haskell implementation of a toy language, FunSub, and a static typechecker for checking well-typedness in the presence of subtypes.

The purpose of this toy language is to provide a concrete realisation of the formal definitions that we will explore in the later sections. We will attach snippets of implementations alongside the definitions where applicable.

FunSub is a superset of Fun [10], which is in turn an implementation of Church's Simply Typed Lambda Calculus [13] that includes the record datatype and a modified style of explicit typing (also known as *ascriptions*).

The source code for FunSub is available online [12].

3.2.1 Usage. Assuming Haskell is installed and the user is in the project directory, running the following command will invoke the typechecker:

```
runhaskell Main.hs examples/typed_expressions.fun
```

This produces an output similar to the following:

```
..
[Expression]: (fn x :: (Int -> Int) => 2)
[Typecheck][OK]: (Int -> Int)

[Expression]: (fn x :: (Int -> Int) => true)
[Typecheck][FAIL]: Type mismatch: expected Int, got Bool
..
```

3.2.2 Architecture. The components of the language comprises of a REPL/reader, lexer, parser and typechecker. Some examples of FunSub expressions are stored in the `examples/` folder.

`Main.hs` is the entry point that takes in the filename for a file containing FunSub expressions. The lexer (`Lexer.hs`) defines reserved tokens and lexemes, and the parser (`Parser.hs`) uses Parsec on the tokens to generate an abstract syntax tree (AST) defined in `Syntax.hs`. Lastly, the AST is checked for well-typedness with rules defined in the typechecker (`Typecheck.hs`).

3.2.3 Syntax.

$c \in$ `Const`	constants
$a, x \in$ `Ident`	alphanumeric identifiers
$t ::=$ `Int`	integers
$\mid$ `Bool`	booleans
$\mid t_1$ `->` t_2	arrows
$\mid \{a_1 : t_1, \ldots, a_n : t_n\}$	records
$e ::= c$	constant
$\mid x$	variable
$\mid$ (`fn` x `::` (t) `=>` e)	function abstraction
$\mid e_1\ e_2$	function application
$\mid \{a_1 = e_1, \ldots, a_n = e_n\}$ `::` t	record
$\mid e.a$	record projection

3.2.4 Records. Records are data structures in the form of finite and unordered associations from *labels* to *values*, with each pair described as a *field*. To extract a value from a field, we use the projection notation $e.a$ where a is the label of a field in the record e.

The purpose of having records in our toy language is for demonstrating subtyping relations on a composite data structure, unlike `Int` and `Bool`.

3.2.5 Explicit types. To simplify implementation, we chose to use an explicit typing notation (`expr :: type`) to assert the types for functions and records, in place of a type inference algorithm.

3.2.6 Example. The following is a valid expression in the FunSub syntax:

```
(fn x :: ({ a: Int, b: Int } -> { a: Int }) => x)
  { a = 2, b = 2, c = true } :: { a: Int, b: Int, c: Bool }
```

In type systems without subtyping, this function application will not typecheck because the record argument type does not match the parameter type, and the parameter type does not match the function body type even though it is an identity function. However in our subtyping implementation, we are able to use concepts such as *variance* (§3.5), *width* and *depth* record subtyping (§3.4) to make this expression well-typed.

3.2.7 Implementation: AST. The AST is a direct translation of the syntax defined in §3.2.3.

```
-- Syntax.hs
data Ty = IntTy
        | BoolTy
        | ArrowTy Ty Ty
        | RcdTy [(String, Ty)]
          deriving (Show, Eq)

data Expr = I Int Ty
          | B Bool Ty
          | Var String
          | Fn String Expr Ty
          | FApp Expr Expr
          | Rcd [(String, Expr)] Ty
          | RcdProj Expr Expr
            deriving (Show, Eq)
```

3.2.8 Implementation: Typechecker. The two main functions of the typechecker are `typecheck` and `isSubtype`. Their type signatures are defined as follows:

```
-- Typecheck.hs
newtype TypeEnv = TypeEnv (Map String Ty)

isSubtype :: Ty -> Ty -> Bool
typecheck :: TypeEnv -> Expr -> Either String Ty
```

`isSubtype` takes in two types and recursively determines if the first type is a subtype of the second, using the inference rules defined in §3.3.

`typecheck` takes in a type environment (a mapping of variables to types) and an AST, and recursively determines if the expression is well-typed. If it is ill-typed, an error message describing the issue

is bubbled up and handled in `Main.hs`. If it is well-typed, the exact type is returned to the caller.

The exact implementation of these functions will be in the following section.

3.3 Rules

We will now introduce the formal rules of subtyping, as detailed by Cardelli and subsequently summarised by Pierce [5, 6, 8, 26].

3.3.1 Subsumption.

$$\frac{\Gamma \vdash t : \alpha \quad \alpha <: \tau}{\Gamma \vdash t : \tau}\ \texttt{Subsumption}$$

The subsumption rule formally introduces subtyping into a type system. It states that if there exists some term t of type α, and that α is a subtype of another type τ, then t is also of type τ. For the rest of the paper, we will use the symbol `<:` to mean "*is a subtype of*".

We can view this rule from another perspective by loosely equating a type as a *set* of elements, and subtypes as some *well-defined* subsets, and translating the subtype relation `<:` to the subset relation $\subseteq$. The subsets are well-defined because there are certain rules on how they are defined. Using this perspective, it is clear that any element belonging to a set is also an element in its superset(s).

This rule permits a term to have at least two types: its own type, and supertype(s) of its own type. Cardelli describes this idea as *multiple inheritance*, where objects are no longer bound to having only one superclass [5].

This is a stark difference from classical type systems where every term has only one unique type. The reduction in rigidity has obvious benefits; terms of a type τ can now be used in locations where terms of the supertype(s) of τ are expected.

3.3.2 Reflexivity. The reflexive property follows naturally from type subsumption; every type can be used in locations where it is expected. Hence, for any type ι, the type is a subtype of itself.

$$\frac{}{\iota <: \iota}\ \texttt{Refl}$$

The implementation is trivial, and is the catchall match clause for the `isSubtype` function:

```
isSubtype ty1 ty2 = ty1 == ty2
```

3.3.3 Transitivity. If type τ_a is a subtype of τ_b, and τ_b is a subtype of τ_c, then τ_a is a subtype of τ_c.

$$\frac{\tau_a <: \tau_b \quad \tau_b <: \tau_c}{\tau_a <: \tau_c}\ \texttt{Trans}$$

There is no corresponding implementation for this as it is not syntax-directed: the consequent of the rule is too general; it can be applied everywhere with an unbounded number of choices for τ_b [26]. It is also proven that subtyping algorithm is sufficiently decidable with the syntax-directed rules for individual types and without the need for `Trans` (and `Refl`) [26].

3.4 Records

The introduction of subtyping requires all new type definitions to have well-defined subtyping behaviour to maintain consistency with the subsumption rule. Hence, by introducing the record data type, we need to define exactly *how* records are ordered in the relation. To determine whether a record is a subtype of another, there are two approaches to determine this structurally: *width*-wise and *depth*-wise.

3.4.1 Width subtyping. As defined by `Subsumption`, we want to be able to promote a record to its supertype without losing any information. Hence, the subtype should contain at least the same (common fields with identical types), if not *more*, information with its supertype.

$$\frac{\Gamma \vdash {T_i}^{i \in i..n+k}}{\left\{a_i : {T_i}^{i \in i..n+k}\right\} <: \left\{a_i : {T_i}^{i \in i..n}\right\}}\ \texttt{RecordWidthSub}$$

For example, a record of type `{ a: Int, b: Bool }` can be substituted in places where `{ a: Int }` is expected (e.g. projection of `record.a`), simply by ignoring the `b: Bool` field.

3.4.2 Depth subtyping. In cases where the labels of the record fields are identical, we compare the types of the corresponding fields and ensure that they conform to the subtype relation:

$$\frac{{S_i <: T_i}^{i \in i..n}}{\left\{a_i : {S_i}^{i \in i..n}\right\} <: \left\{a_i : {T_i}^{i \in i..n}\right\}}\ \texttt{RecordDepthSub}$$

3.4.3 Permutation independence. Since records are unordered, the subtype relation of the two records should be independent on how the fields are ordered, as long as common fields conform to the relations.

$$\frac{\left\{a_i : {S_i}^{i \in i..n}\right\} \text{ is a permutation of } \left\{b_i : {T_i}^{i \in i..n}\right\}}{\left\{a_i : {S_i}^{i \in i..n}\right\} <: \left\{b_i : {T_i}^{i \in i..n}\right\}}\ \texttt{RecordPermSub}$$

3.4.4 Syntax-directed rule. Like `Trans`, it is not clear which of the three rules above should be used as the syntactical structure is the same in every consequent. Hence, there is a syntax-directed rule in which all three rules, `RecordWidthSub`, `RecordDepthSub` and `RecordPermSub`, are combined. This leaves no ambiguity on rule selection:

$$\frac{\left\{a_i : {S_i}^{i \in 1..n}\right\} \subseteq \left\{b_j : {T_j}^{j \in 1..m}\right\} \quad a_i = b_j \rightarrow T_j <: S_i}{\left\{b_j : {T_j}^{j \in 1..m}\right\} <: \left\{a_i : {S_i}^{i \in i..n}\right\}}\ \texttt{RecordSub}$$

The implementation can be derived from `RecordSub` by performing a predicate check where the first record type has all of the fields in the second record type, and then recursively typechecking the common fields such that the field type in the first record is a subtype of the corresponding field in the second record:

```
isSubtype (RcdTy xs) (RcdTy ys) = all sub ys
  where sub (lbl, ty) = case lookup lbl xs of
                          Just ty2 -> isSubtype ty2 ty
                          Nothing -> False
```

3.5 Functions

Like the record type constructor, we need to define how functions relate to each other in the subtype relation, as our toy language allows functions to be used as arguments.

3.5.1 Function abstraction subtyping. This states that a function of type $\alpha \rightarrow \tau$ is a subtype of a function of type $\alpha' \rightarrow \tau'$ if $\alpha' <: \alpha$ and $\tau <: \tau'$.

$$\frac{\alpha' <: \alpha \quad \tau <: \tau'}{\alpha \to \tau <: \alpha' \to \tau'}\ \texttt{FunctionSub}$$

To aid with the understanding of this relation, we need to introduce the notion of *variance*.

The function type constructor, $\to$, has the types flipped on the left, but not on the right. A way to look at this is by understanding the circumstances under which the functions are called: when calling the function of type $\alpha \to \tau$, it should accept only arguments with *more* informative types, i.e. a subtype $\alpha' <: \alpha$, as it might use information that are not available in supertypes of α. On the other hand, when the function is fully evaluated, it will return a term of type τ, thus the caller of the function will also accept any supertype τ' of τ by type subsumption, hence the order of the types is kept.

This subtyping relation defined on the function caller/callee relationship allows functions of a subtype to be safely substituted for their supertypes.

The concept of flipping a relation from the antecedent to the consequent of a rule is called *contravariance* (or contravariant relation). If the order is kept, it is called *covariance* (or covariant relation).

The implementation is recursively defined as follows:

```
isSubtype (ArrowTy a b) (ArrowTy x y) =
  isSubtype x a && isSubtype b y
```

3.5.2 Function application. To extend the function application rule in the presence of subtypes, we need to use the subsumption rule. It follows that if a function f takes in a value of type τ, and α is a subtype of τ, then terms of type α can also be applied to f.

$$\frac{\Gamma \vdash f : \tau \to \beta \quad \alpha <: \tau \quad \Gamma \vdash v : \alpha}{\Gamma \vdash f v : \beta}\ \texttt{FunctionApp}$$

We can use this definition to directly implement our function application typechecker:

```
typecheck env (FApp fn arg) =
  case typecheck env fn of
    Left err -> Left err
    Right (ArrowTy t1 t2) ->
      case typecheck env arg of
        Left err -> Left err
        Right argType ->
          if isSubtype argType t1
            then Right t2
            else Left $ "Type of argument is not
              a subtype of the parameter"
    Right t -> Left $ "Expected Arrow type, got "
      ++ show t
```

3.6 Putting the Typechecker together

Now that we defined the subtyping relation on every type in our language FunSub, we can make use of the typechecker to check the well-typedness of expressions.

3.6.1 Basic types. Basic types are simple enough for the typechecker to reconstruct its type, so there is no need for explicit typing:

```
[Expr]: 1
[Typecheck] [OK]: Int

[Expr]: a
[Typecheck] [FAIL]: a is not defined

[Expr]: true
[Typecheck] [OK]: Bool
```

3.6.2 Record types. The types of the record fields are checked to match its explicit type:

```
[Expr]: { a = 2, b = 3 } :: { a: Int, b: Int }
[Typecheck] [OK]: { a: Int, b: Int }

[Expr]: { a = 2, b = true } :: { a: Int, b: Int }
[Typecheck] [FAIL]: Incorrect record type
```

Record types can be ascribed with its supertype, but not its subtype:

```
[Expr]: { a = 2, b = 3 } :: { a: Int }
[Typecheck] [OK]: { a: Int }

[Expr]: { a = 2, b = 3 } :: { }
[Typecheck] [OK]: {  }

[Expr]: { a = 2, b = 3 } :: { a: Int, b: Int, c: Int }
[Typecheck] [FAIL]: Incorrect record type
```

They can also be nested:

```
[Expr]: { a = { d = 4 } :: { d : Int }, b = 3 }
   :: { a : { d : Int } }
[Typecheck] [OK]: { a: { d: Int } }

[Expr]: { a = { d = 4 } :: { d : Int }, b = 3 }
   :: { a : { c : Int } }
[Typecheck] [FAIL]: Incorrect record type
```

3.6.3 Function types. The Arrow type is ascribed between the parameter and the function body. It is recursively checked to match the parameter and body types:

```
[Expr]: (fn x :: (Int -> Int) => x)
[Typecheck] [OK]: (Int -> Int)

[Expr]: (fn x :: (Int -> Int) => y)
[Typecheck] [FAIL]: y is not defined

[Expr]: (fn x :: ({ a: Bool } -> Int) => x.a)
[Typecheck] [FAIL]: Type mismatch: expected Int, got Bool

[Expr]: (fn x :: ({ a: Int } -> Int) => x.b)
[Typecheck] [FAIL]: Unable to lookup field b in { a: Int }
```

Identity functions can promote the type of the argument to its supertype, but not subtype:

```
[Expr]: (fn x :: ({ a: Int, b: Int } -> { a: Int }) => x)
[Typecheck] [OK]: ({ a: Int, b: Int } -> { a: Int })

[Expr]: (fn x :: ({ a: Int } -> { a: Int, b: Int }) => x)
[Typecheck] [FAIL]: Type mismatch: expected
    { a: Int, b: Int }, got { a: Int }
```

3.6.4 Function application. The type of the argument needs to be a subtype of the paramter of the function, as defined in the rule `FunctionApp`:

```
[Expr]: (fn x :: ({ a: Int } -> Int) => x.a)
  { a = 2 } :: { a: Int }
[Typecheck] [OK]: Int

[Expr]: (fn x :: ({ a: Bool } -> Bool) => x.a)
  { a = 2 } :: { a: Int }
[Typecheck] [FAIL]: Invalid argument of type { a: Int }
  is not a subtype of the parameter type { a: Bool }
```

Revisiting the function application example in §3.2.6, we can now verify that it type checks, as the type of the argument is a subtype of the parameter:

```
[Expr]: (fn x :: ({ a: Int, b: Int } -> { a: Int }) => x)
  { a = 2, b = 2, c = true } :: { a: Int, b: Int, c: Bool }
[Typecheck] [OK]: { a: Int }
```

Since functions are first-class, we can pass functions in as arguments:

```
[Expr]: (fn x :: ((Int -> Int) -> Int) => x 2)
    (fn y :: (Int -> Int) => y)
[Typecheck] [OK]: Int
```

The contravariant and covariant properties of function subtyping can also be observed in the following example of applying a function to an identity function:

```
[Expr]: (fn x :: (
    ({ a: Int, b: Int } -> { a: Int }) ->
    ({ a: Int, b: Int } -> { a: Int })) => x)
        (fn y :: ({ a: Int } -> { a: Int, b: Int })
        => { a = y.a, b = 2 } :: { a: Int, b: Int })
[Typecheck] [OK]: ({ a: Int, b: Int } -> { a: Int })
```

With this application, the left hand side of the type of the argument, `{ a: Int }`, is demoted to a subtype, `{ a: Int, b: Int }` via the contravariant property. On the other hand, the return value of the argument is promoted from `{ a: Int, b: Int }` to a supertype `{ a: Int }` via the covariant property.

4 RELATED WORK

Since Cardelli's introduction of subtyping [5], there has been decades of research poured into the study of subtyping systems. In this section, we will explore some of these related work from a bird's eye view.

The common theme of these research involves inserting subtyping into classical type systems, and dealing with the complications that arises from it.

4.1 Classification of subtyping systems

There are two major classifications of subtyping systems: *structural* and *nominal*.

4.1.1 Structural. The idea of structural subtyping was first described by Cardelli in "Structural Subtyping and the Notion of Power Type" [6]. He describes the idea that subtype relations can be applied to *all* type constructions, not unlike what we implemented in the earlier section for record and function types, by deriving the relation and type rules solely from the *structure* of the types.

Cardelli showed that for every type, four rules can be derived:

- Type *formation* for defining well-typedness of a type in an environment
- Type *introduction* for annotating a term with its the type
- Type *elimination* for manipulating a term to some other type (e.g. record projection)
- Type *subtyping* for defining subtype relations over a type

The defining advantage of this system is the lack of the need for theorem proving, since types match purely based on syntactical structures. Types can also be compared regardless on *when* and *how* they are constructed, since such information is independent from the relation definitions.

A downside of a system that aims to provide all types with structural subtype relations is that its typechecking algorithm on recursive structures was shown to be undecidable [6] as the tradeoff for more typing expressiveness.

Structural type system are popular in functional programming languages derived from lambda calculus, due to the paradigm's structural language features, e.g. Haskell & ML.

4.1.2 Nominal. Unlike structural systems where names are mere aliases to types, nominal systems delegates the declaration of subtype relations to the programmer, such as the case of class hierarchies using inheritance (e.g. `class A extends B..`) in Featherweight Java, [19] and Java by extension. This brings the advantage that the declaration of one type being a subtype of another is sufficient for the program to decide, at both compile and run time, which type names are related to each other in a global record without having to delve into the structure of the type.

Pierce describes a major advantage of nominal systems to be [26] the ease of defining recursive types: since types are defined nominally, recursive types can easily handle occurrences of its own type name within its structure. There is almost no requirements on *when* a type is defined; a recursive type name reference is no different from a reference to another type name.

However, Pierce also explains that this inherently creates a dependency to some global state whenever information about some type relation is needed, unlike structural types where types are

closed and carry enough information for the type system to determine relations in an isolated manner [26].

There are clear upsides and downsides to either subtyping system, hence there have been efforts by Malayeri and Aldrich to create a type system using concepts from both camps, resulting a core hybrid subtyping calculus [21].

4.2 Recursive subtypes

Introducing subtyping to recursive data structures raises a couple of complications as detailed by Amadio and Cardelli [4]: the approach on how to define a subtype relation between two recursive types, as well as determining whether a recursive term has a type.

To define a subtype relation on the structure of recursive types, the paper describes an algorithm that represents the types in comparison using cyclic linked structures (or graphs) in memory, and a trail of visited node pairs between the two structures. Wherever a recursive call is encountered during a traversal, an edge is created back to the root address of the structure, and the algorithm has the choice on whether to unfold into a larger linked structure or perform a structural case comparison with the current nodes using information from the trail, hence ensuring termination.

The paper also showed that there exists a unique coercion mapping between any two (recursive) types in the subtype relation, and described an algorithm that can infer a least type for terms using such coercions.

4.3 System F with subtyping

Building on Girard-Reynold's work on the second-order lambda calculus System F, Cardelli et al. extended it with the notion of subtyping, resulting in a new type system called System `F<:` described in 1991 [9], and then with an implementation in 1993 [7].

In order to represent subtypes in parametric polymorphic type systems with type schemes, the universal quantifiers for type schemes need to be augmented with bounded type quantifiers in the form of a subtype relation, i.e. a type scheme $\forall X.B$ with a bounded type quantifier results in $\forall X<:A.B$ for some types X, A, B [8].

4.4 Type inference

Type inference algorithms have been a staple in typed languages as it relieves the programmer from verbose typing notations. However, popular algorithms such as Algorithm $\mathcal{W}$ [15] cannot be used alongside subtyping with the absence of the property that every term is uniquely typed. A compatible algorithm introduced by Cardelli [5] uses the *join* and *meet* concepts from order theory for an algorithm to derive the unique *principal types* of each term.

Furthermore, parametric polymorphic type systems built around the Hindley-Milner type inference algorithm [15, 18, 23] have decidable type inference via the unification algorithm. However, by introducing subtyping, the typing constraint is changed from one that is based on *equality* ($\tau_1 = \tau_2$) to one that is based on a *partial order* on types ($\tau_1 <: \tau_2$). This renders the unification algorithm useless and makes such a type inference algorithm undecidable in systems with subtyping.

There has been attempts at bringing decidable type inferencing into parametric polymorphic type systems with subtyping. Dolan's recent work on designing such an algorithm uses influences from data flow analysis on a core calculus of ML with subtyping, MLsub, and has gained notable traction [17]. He made further progress in his Ph.D. thesis, Algebraic Subtyping, where he described a framework for type inference on ML-style type systems using an algebraic approach called the theory of *biunification* [16].

4.5 Subtyping effect systems

Effect systems are extensions of type systems by statically capturing side effects resulting from function applications, such as input/output and randomness, via control flow analysis [27]. Using the subsumption concept, Tang et al. [28] implemented a *subeffect* system, a type system with subtyping and a partially-ordered inclusion relation over side effects, along with a sound and complete type reconstruction algorithm to remove the need for explicit typing.

4.6 Type system for Scala

Scala has a rich and complex type system that attempts to unify the mature object-oriented paradigm of Java with functional programming concepts. Odersky, Scala's creator, made significant efforts in developing Scala's type system by bringing in aspects of modern type system research, such as subtypes, compound types, nested classes and refined types [25].

Cremet et al. created a minimal core calculus for typechecking some parts of the Scala language in Featherweight Scala, which includes decidable typechecking for subtypes [14].

However, Scala's type inference algorithm is not complete (it has a limited form of *local* type inference) as explicit type annotations are still needed in places where the inference algorithm has insufficient information to derive a principal type [2].

4.7 Type system for Erlang

One of the other ways that subtyping found its way into industry practice was via Erlang. Erlang was originally designed to be a functional and untyped language for fault-tolerant and distributed systems applications. Marlow and Wadler [22] made efforts to implement a type system with inference for the language, where they found that a subtyping type system based around the one developed by Aiken and Wimmers [3] was the best fit for the language. The alternatives included the Hindley-Milner type system, where types are constrained by equality instead of an ordered relation, as well as *row variables* with soft typing systems.

Erlang organizes its types into a complete lattice, with the `any()` ($\top$) type and `none()` ($\bot$) types forming the upper and lower bounds of the type hierarchy, and type unions are typed as the greatest lower bound of its types [1].

4.8 Prototypal subtyping

Implementing static and strong type systems for complex dynamic languages is under heavy research. There has been recent development on static type inference on dynamic prototypal languages like JavaScript by Chandra et al. [11], where subtyping relationships are based on a classification of *prototypal property* on types. For prototypal types, the subtype relation is simply the original prototype inheritance where child types contains all attributes defined in the parent's prototype, whereas non-prototypal types uses structural subtyping.

5 FUTURE WORK

Subtyping is a mature field of research, but with the discovery and invention of new type systems, it remains as a consistently interesting additional feature, bringing difficult problems by removing the guarantee of uniquely typed terms through type subsumption.

Our implementation of the typechecker is functional but primitive, and its syntax is verbose with the explicit typing littered throughout expressions. We aim to implement implicit typing by using a type reconstruction algorithm described in Cardelli's paper [5] and further elaborated with the idea of type inferencing using *minimal* types from Mitchell's work [24].

6 SUMMARY

In this report, we explained the practical motivations for adding subtyping into classical type systems. The flexibility gained in implementing subtyping have led to interesting methods of structuring and organizing information in a programming language.

We also listed the formal rules for subtyping on basic, record, and function types. Using these rules, we implemented a typechecker on a toy functional language with subtyping, FunSub, using Haskell, and demonstrated its functionality with concrete examples.

Lastly, we gave an overview of related research that conceptualised from the inclusion of subtyping into classical type systems and programming languages.

REFERENCES

[1] 2017. Erlang – Type and Functions Specifications. (Apr 2017). http://erlang.org/doc/reference_manual/typespec.html

[2] 2017. Local Type Inference – Scala Documentation. (Apr 2017). http://docs.scala-lang.org/tutorials/tour/local-type-inference.html

[3] Alexander Aiken and Edward L Wimmers. 1993. Type inclusion constraints and type inference. *Proceedings of the conference on Functional programming languages and computer architecture* (1993), 31–41. DOI: https://doi.org/10.1145/165180.165188

[4] Roberto Amadio and Luca Cardelli. 1993. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems* 15, 4 (1993), 575–631. DOI: https://doi.org/10.1145/155183.155231

[5] Luca Cardelli. 1984. A semantics of multiple inheritance. *Information and Computation* 76, 2-3 (feb 1984), 138–164. DOI: https://doi.org/10.1016/0890-5401(88)90007-7

[6] Luca Cardelli. 1988. Structural subtyping and the notion of power type. *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '88* (1988), 70–79. DOI: https://doi.org/10.1145/73560.73566

[7] Luca Cardelli. 1993. An implementation of F<:.

[8] Luca Cardelli. 1996. Type systems. *Comput. Surveys* 28, 1 (mar 1996), 263–264. DOI: https://doi.org/10.1145/234313.234418

[9] Luca Cardelli, John C Mitchell, Simone Martini, and Andre Scedrov. 1991. An Extension of System F with Subtyping. *Proceedings of TACS'91* 526, 80 (1991), 750–770. DOI: https://doi.org/10.1006/inco.1994.1013

[10] Luca Cardelli and Peter Wegner. 1985. On understanding types, data abstraction, and polymorphism. *Comput. Surveys* 17, 4 (dec 1985), 471–523. DOI: https://doi.org/10.1145/6041.6042

[11] Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. 2016. Type Inference for Static Compilation of JavaScript (Extended Version). *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2016* (aug 2016), 410–429. DOI: https://doi.org/10.1145/2983990.2984017 arXiv:1608.07261

[12] Jingwen Chen. 2017. jin/subtyping. (apr 2017). https://github.com/jin/subtyping

[13] Alonzo Church. 1940. A Formulation of the Simple Theory of Types. *The Journal of Symbolic Logic* 5, 2 (1940), 56–68. DOI: https://doi.org/10.2307/2266170 arXiv:cs/9301107

[14] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. 2006. A Core Calculus for Scala Type Checking. *MFCS (Mathematical Foundations of Computer Science)* 4162 (2006), 1–24. DOI: https://doi.org/10.1007/11821069

[15] Luis Damas and Robin Milner. 1982. Principal Type Schemes for Functional Programs. In *ACM Symposium on Principles of Programming Languages POPL Albuquerque New Mexico*, Vol. Alburquerq. 207–212. DOI: https://doi.org/10.1145/582153.582176

[16] Stephen Dolan. 2016. Algebraic Subtyping. September (2016).

[17] Stephen Dolan and Alan Mycroft. 2017. Polymorphism, subtyping, and type inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages - POPL 2017*. ACM Press, New York, New York, USA, 60–72. DOI: https://doi.org/10.1145/3009837.3009882

[18] Roger Hindley. 1969. The principle type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.* 146, December (1969), 29–60. DOI: https://doi.org/10.1090/S0002-9947-1969-0253905-6

[19] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems* 23, 3 (2001), 396–450. DOI: https://doi.org/10.1145/503502.503505

[20] Barbara Liskov and Jeannette M. Wing. 1994. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems* 16, 6 (1994), 1811–1841. DOI: https://doi.org/10.1145/197320.197383

[21] Donna Malayeri and Jonathan Aldrich. 2008. Integrating nominal and structural subtyping. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5142 LNCS (2008), 260–284. DOI: https://doi.org/10.1007/978-3-540-70592-5_12

[22] Simon Marlow and Philip Wadler. 1997. A practical subtyping system for Erlang. *ACM SIGPLAN Notices* 32, 8 (1997), 136–149. DOI: https://doi.org/10.1145/258949.258962 arXiv:arXiv:1011.1669v3

[23] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (dec 1978), 348–375. DOI: https://doi.org/10.1016/0022-0000(78)90014-4

[24] John C. Mitchell. 1991. Type inference with simple subtypes. *Journal of Functional Programming* 1, 03 (1991), 245–285. DOI: https://doi.org/10.1017/S0956796800000113

[25] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. An Overview of the Scala Programming Language. *System* Section 2 (2004), 1–130. DOI: https://doi.org/10.1145/1706356.1706358

[26] Benjamin Pierce. 2002. *Types and programming languages*. MIT Press, Cambridge, Mass.

[27] Olin Shivers. 1988. Control flow analysis in scheme. *ACM SIGPLAN Notices* 23 (1988), 164–174. DOI: https://doi.org/10.1145/960116.54007

[28] Yan Mei Tang and Pierre Jouvelot. 1995. Effect systems with subtyping. In *Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation - PEPM '95*. ACM Press, New York, New York, USA, 45–53. DOI: https://doi.org/10.1145/215465.215552