Simply Typed Lambda Calculus with Subtyping

Rayane Bouafia
School of Computer Science
McGill University
261119286

William Melançon
School of Computer Science
McGill University
260963759

***Abstract*—Simply Typed Lambda Calculus (STLC) is a foundational model in programming language theory, known for its simplicity and strong typing discipline. However, its rigid typing rules can restrict practical software development. To address this limitation, we introduce subtyping into STLC, allowing for more flexible type assignments following Liskov's Substitution Principle. We implemented an algorithmically driven type system supporting subtyping, and then expanded it with joins and meets, based on *Types and Programming Languages* by Pierce [13]. Our implementation addresses the complexity introduced by subtyping in both theoretical and practical aspects, particularly highlighting challenges in algorithmic type checking. This paper discusses our implementation details, encountered challenges, and the historical and practical significance of subtyping.**

I. Introduction

Simply Typed Lambda Calculus (STLC) offers a robust theoretical framework for reasoning about functional languages. Despite its elegance, STLC is limited by its strict type matching requirement: it is required that every function argument exactly matches the declared parameter type. For example, a function that takes as input a record with less fields (e.g. `{ x : Bool }`) cannot directly accept records with additional fields (e.g. `{ x : Bool , y : Bool }`) even if intuitively, it is perfectly safe to do so. Subtyping addresses this rigidity. It is the formalization of the idea that something more specific is suitable where something more general is needed.

The idea behind subtyping traces its roots back to the early history of programming languages. In the late 1960s, the development of Simula 67 [1], often regarded as the first object-oriented language, introduced the idea of class-based inheritance. Although Simula 67 did not formally distinguish subtyping from inheritance, it laid out an important concept: subclass instances could be used wherever superclass instances were expected.

The formal notion of subtyping emerged explicitly in the 1980s. Barbara Liskov and Jeannette Wing, in their work on data abstraction and safe substitution [2], [3], articulated what is now called the Liskov Substitution Principle or Safe Substitution Principle: a guideline for when it is safe to substitute instances of one type for another. This was formally stated in their 1994 paper as follows:
"*Subtype Requirement*: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T [3]."

At around the same time, researchers such as Luca Cardelli and Peter Wegner initiated the study of type systems with explicit subtyping relations [4]. Cardelli's 1984 development of the Amber language marked an important milestone, systematically introducing subtype polymorphism into type theory [5].

During the 1980s and 1990s, as object-oriented programming gained widespread popularity, subtyping became less of an academic notion and more of a fundamental language feature. Languages like C++ [6], Eiffel [9] and later Java [7] integrated subtyping through class hierarchies and interfaces. In Java, for instance, a class `B` that `extends` a class `A` is treated as a subtype, enabling instances of `B` to be used wherever instances of `A` are expected [7]. Eiffel in particular emphasizes adherence to Liskov's principle by integrating subtyping with a strong contract-based system [8], [12].

Beyond nominal inheritance, languages like OCaml and TypeScript explore structural subtyping: where a value's type is determined by its structure rather than its nominal declaration, bringing even greater flexibility and enabling polymorphism based purely on shape [10], [11]. Today, subtyping remains a cornerstone of programming-language design in most mainstream languages.

This paper outlines our implementation strategy, describing the algorithmic approach to subtyping and detailing the implementation of key concepts, including function types, product types, record types, conditionals, and type operations such as joins and meets. Our algorithmic approach is adapted from its presentation in *Types and Programming Languages* by Pierce [13].

II. Development

As mentioned above, our project is based on *Types and Programming Languages* [13], which explores subtyping in two passes: first, a specification-level presentation with a simple, declarative (naive) subtyping relation, and second, an algorithmic presentation that refines the relation into a syntax-directed form suitable for implementation.

This naive implementation is useful for definitions and for proving type safety. However, in practice we can't actually use these rules because `T-Sub` and `S-Trans` are non-syntax-directed, meaning that when trying to figure out whether `S <: T`, there is no clear direction in which to proceed.

Specifically, `T-Sub` states that if you already have a type `S` for a term `t`, and you know that `S` is a subtype of some other type `T`, then you can safely consider `t` as having type `T`. The issue, however, is that the term `t` here is completely general; there's nothing about it that guides you on when exactly to apply this rule.

More obviously, to check if `S` is a subtype of `T` using `S-Trans`, we must first guess some intermediate type `U`. But there are infinitely many types we could guess, and no clear guideline about how to select one.

The algorithmic implementation is the one we implemented.

A. William

My part of the implementation mostly concerned the backbone of the code: this included the metatypes `tp` and `tm`, and the code for `subtype` and `typeof`, barring the subcases for conditionals and `TmFst`/`TmSnd`/`TmProj` in `typeof`. Rayane was still involved in key implementation decisions, however, specifically for `tp` and `tm`. These, as well my contributions `subtype` and `typeof` will be discussed in the subsections below.

1) Basic implementation & related decisions: Since our project mainly focuses on subtyping, we chose to focus on basic features which have a strong overlap with subtyping, as well as inspiring ourselves from the TAPL implementation (see chapter 17) of subtyping in STLC. The features and types I added to the implementation which are of interest include:

1) The `Top` type. This represents a maximal type, e.g. the type which has all other types as subtypes. This is often encountered in object-oriented languages as the `Object` type, and while it does not have large consequences on our subtyping system it was necessary in `join` for the subcase where both input types did not have any subtyping relation (see Rayane's section for more details on `join`).
2) Lambda expressions and arrow types (labeled `Func` in the code). Notably, following TAPL's [13, p. 221] initiative, our `Lambda` type (note: this is a type, but recall that it is simply a wrapper type that represents a lambda expression) requires us to specify the type of the input which would otherwise not be inferrable with our basic metatypes.
3) `Nat`, `Zero`, `Succ`, and `Float`. While we did not implement any example functions (such as `sum`) to test the interactions between nats and floats, we still chose to include this type as a way of verifying our subtyping implementation works. One unorthodox choice we made was to implement the representation of a `Float` variable as a string (e.g. `Float of string`); this was done to mirror the way that `Nat` does not wrap an actual positive `int`, but a "meta-representation" of its variable (through `Zero` and `Succ`) so as to avoid directly using OCaml's implementation of `ints` or `floats` and instead writing our own code for conversion from a meta-representation to a usable value.
4) Tuples and records, and their corresponding product and record types, respectively. We chose to follow the book's implementation for record types, specifically that the order of the fields does not matter. Rather, as long as two records contain the same number of fields, with the same labels, and the same respective types to each label, their types are considered to be equal.

2) The `subtype` function: The implementation of `subtype` was relatively straightforward, but three subcases stand out as more interesting (in the following examples, `Func/Prod (s1, s2)` and `TpRec l2` are the terms we are assuming to be subtypes):

```
1) (Func (s1, s2), Func (t1, t2)) ->
       subtype t1 s1 && subtype s2 t2

2) (Prod (s1, s2), Prod (t1, t2)) ->
       subtype s1 t1 && subtype s2 t2

3) (TpRec l2, TpRec l1) ->
         List.for_all (fun (l1i, t1i) ->
             try
```

```
      let t2i = List.assoc l1i l2
      in subtype t2i t1i
    with Not_found -> false)
  l1
```

The first and second subcases correspond to the arrow type and product type respectively and display an interesting symmetry. For both, the outputs follow a *covariant* subtyping relation, meaning the direction of the subtyping relation is preserved: `s2 <: t2`. However, for the arrow type, the inputs follow a *contravariant* subtyping relation, meaning the direction of the subtyping relation is reversed: `t1 <: s1` (product types maintain the covariant rule for inputs). This may initially seem unintuitive for arrow types, but it follows naturally from one of the main principles of subtyping: replaceability. Say we have two functions $f : \alpha \to \beta$ and $g : \alpha' \to \beta'$, and we assume that g's type is a subtype of f and thus it can be used anywhere f can be used. g must be able to accept all inputs to f; it does not matter however if g's domain is "larger", because we only require it to accept at minimum all of f's inputs. Thus, we require that $\alpha <: \alpha'$. On the other hand, g's outputs must also be usable anywhere f's outputs are usable; it cannot output anything with a type outside of the co-domain of f. Thus, $\beta' <: \beta$.
The third subcase requires a bit of background. We have taken the following definition of subtyping for records from TAPL [13, pp.183-184]: given two records `R1` with type α and `R2` with type β, we say that $\beta <: \alpha$ if and only if the following conditions are met:

1) `R2` contains *at least* the same number of fields as `R1` (crucially, it can contain more!).
2) All of the labels in `R1` must also appear in `R2`.
3) For each label $l_i : T_i \in$ `R1`, we require that the corresponding label $l'_i : T'_i \in$ `R2` be such that $T'_i <: T_i$.

Once again, there is a slightly unintuitive property here, namely the fact that `R2` can be larger than `R1`, but if viewed once again under the lens of replaceability, it follows logically: to use `R2` in the place of `R1`, it must contain at least all of its labels, but it doesn't matter if it contains more because we assume that any place `R1` is used will not refer to labels not contained within `R1`.
Finally, here is a quick explanation for the code (a very elegant implementation, also from TAPL [13, p. 222]). The `for_all` function from the OCaml `List` module takes a predicate and a list as input (our records are underlyingly implemented as lists), applies the predicate to each of the list's elements, and only returns true if the predicate returned true for all elements. As we are trying to check if `TpRec l2`'s type is a subtype of `TpRec l1`'s type, for each field `l1i` in `l1`, we check if it exists in `l2` (using a `try` block to catch the error if it is not found), and then check if `t2i` (the type of `l2i`) is a subtype of `t1i` (the type of `l1i`).

3) typeof: While `typeof` required much more verbose code than `subtype`, its implementation was more immediately intuitive. The cases of interest I handled here were function application and records (the code was left out here this time as it falls outside the margins of the paper, and the implementation is quite simple). For `App`, only a slight adjustment needed to be made, which is that given some function $f : \alpha \to \beta$ applied to some term $t : \gamma$, we can accept t as an input if $\alpha <: \gamma$, while in our previous implementation of `typeof` we required $t : \beta$. For records, we implemented their type as a `(string * tp) list`, and so we simply iterate over the inputted record, keep the labels, and apply `typeof` to their values to replace them with the corresponding type.

B. Rayane

As mentioned above, both authors were involved in key implementation decisions. I corrected errors in Will's implementation, and I implemented `TmFst`, `TmSnd`, and `TmProj`.

My main contribution was the implementation of conditionals, joins and meets [13, Chapter 16.3]. In STLC, constructs like *if-expressions* require both branches to have the same exact type. In STLC with Subtyping, it is now desirable for the two branches to have different but related types. If the branches share a common structure, the expression is still safe and meaningful. For example:

```
t_then = { x = true ,  y = false }
(* has type  { x : Bool ,
y : Bool } *)
t_then = { x = false ,  z = true }
(* has type  { x : Bool ,
z : Bool } *)
```

In STLC with Subtyping, every {`x : Bool, y : Bool`} and every {`x : Bool, z : Bool`} can be used where {`x : Bool`} is needed. That type is referred to as the minimal type, and the `join` is what we use to find it. The join of two types S and T, written $S \vee T$ is the "smallest" supertype that both branches' types can be safely promoted to. It is important to highlight that "smallest" is in regard to the subtyping relation:

it may not be the biggest "physically" (number of fields in the record, for instance) but it is the most informative and therefore smaller. This is explained further below. To implement `join` fully, we also need `meet`. The meet of two types S and T, written $S \wedge T$, is the "largest" type that is a subtype of both S and T. Again, "largest" refers to the subtyping relation: it is the most specific type that both S and T can safely be downcast to. Again, this is explained further below.

First, in both `join` and `meet`, the base cases check for subtyping: if one type is already a subtype of the other, the answer is trivially the supertype (for join) or the subtype (for meet).

Now we go through the cases where the types differ. The `join` of `Nat` and `TpFloat` is `TpFloat`, because any integer can be safely promoted to a float, and floats form a supertype. The meet of `Nat` and `TpFloat` is `Nat`, because any shared value must be valid as a `Nat`, which is more specific. For products, joining two product types means recursively joining their respective components. The intuition is that two pairs are compatible if their corresponding parts can be promoted to compatible types. Similarly for meeting two product types.

Doing the functions was a significant challenge I faced, as it requires `meet` to be implemented, but I initially thought I did not need `meet`, so significant time was spent trying to make functions work without meets, in vain. `meet` is needed because as mentioned previously, function types are contravariant in the domain and covariant in the codomain. To `join` two functions, their domains must be narrowed (meet) — meaning the function can accept less — while their results are widened (join) — meaning the function may return more general results. Therefore: `Func(meet s1 t1, join s2 t2)`. For the `meet` of two functions, the behavior is reversed: their domains are widened (join) — accepting more kinds of inputs — while their codomains are narrowed (meet) — returning more specific outputs: `Func(join s1 t1, meet s2 t2)`. Finally, records: for joins, we are taking the fields present in both, dropping any fields that are not present in both. For meets, we are merging all fields from both.

And here lies the biggest challenge I faced: the counterintuitive nature of the language used to describe joins and meets. The reason is that types are understood as sets of values — not as sets of fields, features, or structures. If a type is the set of all values it accepts, then from that point of view, a join combines two types by taking the union of their sets of possible values: it must accept anything that either type would accept. The meet combines two types by taking the intersection of their value sets: it accepts only values common to both types. Thus, theoretically, joins are unions of values, and meets are intersections of values. Joins are "smaller" because we are looking for the smallest type that is above both inputs — the minimal common generalization. Meets are "larger" because we are looking for the greatest (largest) type that is below both inputs — the maximal common specialization.

However, when implementing joins and meets, especially for types like records or functions, the situation feels reversed because we work with the shape of the type, not its set of values. Structurally, when joining two records, you often must keep only the common fields (an intersection of structure) to find a type that generalizes both. When meeting two records, you merge all fields (a union of structure) to find a type that specializes both. Similarly, for function types, the join of two functions needs to meet their input types (restricting the domain — intersection-like behavior) and join their output types (union-like behavior), and the meet of two functions needs to join their input types (loosening the domain — union-like behavior) and meet their output types (intersection-like behavior). As a result, in real implementations, `joins` feel like intersections (of structure), and `meets` feel like unions (of structure), even though the underlying type-theoretic definitions are based on unions and intersections of sets of values. This apparent reversal happens because supertypes have fewer structural requirements (they accept more values), and subtypes have more structural requirements (they accept fewer values), making "being more general" look structurally smaller and "being more specific" look structurally larger. All of this is a complicated concept to articulate, and something I struggled with during the implementation part.

III. Conclusion

A. *Further Exploration*

Unlike many other extensions seen in class, subtyping is quite unique in the sense that it is not orthogonal to other extensions but instead interacts with most of them non-trivially [13, p.222]. Were we to continue this project, we would focus on adding the following features:

1) Sum types and lists: as these behave similarly to product types and records respectively, their implementation would be quite straightforward.
2) Recursive types: given that recursion is integral to most work done in computer science. Handling

subtyping between recursive types introduces significant complexity, as it requires reasoning about types that are "infinitely unfolded." It is not immediately obvious how to handle subtyping when the types are "intertwined" as recursive types are, this would be an interesting avenue to investigate.

3) Bottom Type: Currently, our system includes the Top type, a maximal type comparable to `Object` in Java, which is a useful abstraction and a technical convenience in polymorphic systems. Future work would involve extending the system to include a Bottom type (`Bot`) as well. `Bot` represents an empty type with no values, useful for expressing computations that never return (such as exceptions or diverging programs). However, `Bot` complicates typechecking significantly and adjustments to the typing and subtyping algorithms would be necessary to preserve type safety [13, Chapter 15.4, 16.4].
4) Up-casting and down-casting: Another natural extension would be the incorporation of explicit typecasting mechanisms, as seen in many modern programming languages such as Java and Python. Upcasts are straightforward: they allow a value to be safely viewed as a supertype and can be checked statically by the typechecker. Downcasts, on the other hand, are riskier: they assign a more specific (subtype) type to a value and require dynamic runtime checks to ensure that the value actually conforms to the target type.
   Practically, downcasting would allow us to implement more flexible functions. For instance, as mentioned earlier, we could implement a sum function that can handle both `ints` and `floats` and any combination of them by downcasting the `int`. We could also, rather than handling this in the `sum` function specifically, write a `typecast` function which, given a list of variables and typing constraints, determines if typecasting is possible and execute it correctly if so.
5) Miscellaneous: There are numerous other extensions that could further enrich the system, which we briefly mention here. Adding let bindings and unit types would complete the basic functional programming core.
   Support for more traditional data structures such as heaps, trees, and other recursive structures would further demonstrate the system's ability to model complex programs.
   We could also extend the system with richer variant types, allowing tagged union types with subtyping rules supporting width, depth, and permutation. Similarly, exceptions could be modeled elegantly using a combination of variant types and the Bottom type (Bot).
   More sophisticated base types could also be introduced, such as subtyping relations like Bool ¡: Nat, to capture relationships between primitive types.
   In handling mutable state, the type system for references and arrays could be refined: while basic references must be invariant to ensure safety, a more precise model distinguishes between read-only and write-only capabilities.
   Finally, we could add explicit intersection and union types, allowing programs to handle values that belong to multiple types at once or to one of several types, making the type system even more flexible.
   All of these extensions, while powerful, would introduce significant additional complexity both theoretically and in implementation.

B. Related Work

Since Liskov and Cardelli's landmark papers, and the rising popularity of object-oriented programming, subtyping has increasingly been recognized as a practical feature of programming languages, which naturally lead to formal study of its properties. We present here a few developments in the last few decades on the topic of subtyping.

1) Amadio & Cardelli 1993 [14]: A formal treatment of subtyping and recursive types in STLC. The authors give a formal definition of types, subtyping, and recursive types, and then build an extension for recursive types in STLC (the paper also contains more chapters on various topics related to recursive types which are outside of the scope of this paper). To solve the issue of recursive types unfolding to match other types potentially infinitely many times (at least theoretically), and taking the avenue of the (at the time) new subject of structural matching, they represent the types as infinite trees. The paper was quite high-level for us, but from our general understanding it seems that they give an algorithm for converting recursive types into a tree representation, and then define an ordering relation on the trees which corresponds to the subtyping relation for their respective types. Despite its graduate-level theoretical computer science contents, the first two chapters (e.g. up to the STLC implementation) are

more approachable and would be a great place to start for implementing recursive types, as mentioned in the previous section.

2) Fluet & Pucella 2004 [15]: This paper focuses on the ”phantom-types” technique for subtyping. Phantom types are ”datatypes in which type constraints are expressed using type variables that do not appear in the datatype cases themselves” [16], hence the term ”phantom”. According to the authors, traditional Hindley-Milner type systems such as Standard ML can be used to enforce the subtyping relationship, but by using phantom types any finite subtyping hierarchy can be encoded. They give a general procedure for applying this technique to subtyping, give an analysis of different classes of encodings for different kinds of subtyping hierarchies (for example, a binary tree hierarchy as opposed to a lattice hierarchy), and then present a type-preserving translation from a type system with subtyping to a type system with let-bounded polymorphism.
3) System F<: is an extension of System F with bounded quantification. In System F<: , type variables can be given upper bounds, allowing types to be quantified not just over all types, but over types that are subtypes of a given bound. While System F<: is powerful, it remains primarily a theoretical model due to its undecidable typechecking problem. [13, Chapter 26, 28]

C. Final Remarks

Subtyping has become a cornerstone of modern type systems because it naturally aligns with programmer intuition about data abstraction, modularity, and code reuse. It allows different components of a program to interact flexibly through general interfaces, while enabling specific implementations to evolve independently. This flexibility makes libraries more reusable and complex systems more manageable. Without subtyping, programmers would face far stricter boundaries between components, resulting in rigid, overly explicit type systems that would hamper software design.

While subtyping appears conceptually intuitive at first — especially through its naive declarative formulation using simple rules — the historical development and our experience implementing it highlight how much more intricate its full formalization and execution truly are. Building a complete algorithmic typing and subtyping system with support for conditionals required restructuring typing judgments, managing subtyping obligations explicitly, and ensuring properties like determinism and soundness. Moreover, subtyping rarely stands alone: it intersects with many other features of a language, including polymorphism, type inference, functions, and modularity, often amplifying their complexity. This transition from specifications to implementation captures why subtyping, despite its apparent intuitiveness, remains one of the most subtle and demanding aspects of programming language theory.

References

[1] O. J. Dahl and K. Nygaard, *SIMULA 67 Common Base Language*, Publ. S-22. Oslo, Norway: Norwegian Comput. Center, 1970.

[2] B. Liskov, “Data abstraction and hierarchy,” in *Proc. ACM SIGPLAN Conf. Object-Oriented Program., Syst., Lang. Appl. (OOPSLA)*, Oct. 1987, pp. 17–34.

[3] B. Liskov and J. Wing, “A behavioral notion of subtyping,” *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1811–1841, Nov. 1994.

[4] L. Cardelli and P. Wegner, “On understanding types, data abstraction, and polymorphism,” *ACM Comput. Surv.*, vol. 17, no. 4, pp. 471–523, Dec. 1985.

[5] L. Cardelli, “Amber,” in *Combinators, Lambda Calculus and Functional Programming*, J.-P. Jouannaud, Ed. Berlin, Germany: Springer, 1984, pp. 21–47.

[6] ISO/IEC 14882:1998, *Programming Languages — C++*. Geneva, Switzerland: Int. Org. Standardization, 1998.

[7] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java Language Specification*, 4th ed. Upper Saddle River, NJ, USA: Addison-Wesley, 2013.

[8] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, 1997.

[9] B. Meyer, *Eiffel: The Language*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1992.

[10] INRIA and The OCaml Consortium, *The OCaml System, Release 5.2: Documentation and User’s Manual*. Paris, France: INRIA, 2024. [Online]. Available: https://ocaml.org/docs

[11] Microsoft, *TypeScript Handbook*, ver. 5.4. Redmond, WA, USA: Microsoft Corp., Feb. 2024. [Online]. Available: https://www.typescriptlang.org/docs

[12] B. Meyer, “Applying ‘Design by Contract’,” *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992.

[13] B. C. Pierce, *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002.

[14] R. Amadio and L. Cardelli, ”Subtyping Recursive Types”, *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 4, pp. 575–631, 1993, Accessed: Apr. 21, 2025. [Online]. Available: http://lucacardelli.name/papers/srt.pdf

[15] M. Fluet and R. Pucella, ”Phantom Types and Subtyping,” *arXiv (Cornell University)*, Jan. 2004, doi: https://doi.org/10.48550/arxiv.cs/0403034.

[16] J. Cheney and R. Hinze. ”First-Class Phantom Types”. Accessed: Apr. 21, 2025. [Online]. Available: https://www.researchgate.net/publication/213885544_First-Class_Phantom_Types