Julia's efficient algorithm for subtyping unions and covariant tuples

Benjamin Chung
Northeastern University

Francesco Zappa Nardelli
Inria

Jan Vitek
Northeastern University & Czech Technical University in Prague

Abstract

The Julia programming language supports multiple dispatch and provides a rich type annotation language to specify method applicability. When multiple methods are applicable for a given call, Julia relies on subtyping between method signatures to pick the method to invoke. Julia's subtyping algorithm is surprisingly complex, and deciding whether it is correct remains an open question. In this paper, we focus on one piece of this problem: the interaction between union types and covariant tuples. Previous work that addressed this particular combination of features did so by normalizing types to a disjunctive normal form ahead-of-time. Normalization is not practical due to space-explosion for complex type signatures and to interactions with other features of Julia's type system. Our contribution is a description of the algorithm implemented in the Julia run-time system. This algorithm is immune to the space-explosion and expressiveness problems of standard algorithms. We prove this algorithm correct and complete against a semantic-subtyping denotational model in Coq.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Type systems, Subtyping, Union types

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.23

1 Introduction

Union types, originally introduced by Barbanera and Dezani-Ciancaglini [4], are increasingly being used in mainstream languages. In some cases, such as Julia [6] or TypeScript [2], they are exposed at the source level. In others, such as Hack [1], they are only used internally for type inference. We describe a space-efficient technique for computing subtyping between types in the presence of distributive unions, arising from the Julia programming language. In our previous work on formalizing the Julia subtyping algorithm [15], we described the subtyping relation but were unable to describe the subtyping algorithm or prove it correct. Indeed, we found bugs and were left with unresolved issues.

Julia's subtyping algorithm [5] is an important part of its semantics. Julia is a dynamically typed language where methods are annotated with type signatures to enable multiple dispatch. During program execution, Julia must determine which method to invoke at each call site; for this, it searches the most specific applicable method (according to subtyping) that applies for a given invocation. The following snippet shows three declarations of multiplication:

```
*(x::Number, r::Range)  = range(x*first(r),...)
*(x::Number, y::Number) = *(promote(x,y)...)
*(x::T, y::T) where T <: Union{Signed,Unsigned} =  mul_int(x,y)
```

The first two methods implement, respectively, the case where a range is multiplied by a number and generic numeric multiplication. The third method invokes native multiplication

when both arguments are either signed or unsigned integers (but not a mix of the two). Julia will use subtyping to decide which of the methods will be invoked for any specific invocation.

To allow the expression of complex relationships in type signature, Julia offers programmers a rich type language. Julia's types include nominal single subtyping, union types, existential types, covariant tuples, invariant parametric datatypes, and singletons. These are all widely used in libraries, but pose challenges for subtyping. Julia's design [6] was inspired by semantic subtyping [9], and while its types do not precisely abide by a semantic set-theoretic intuition, its subtyping algorithm must handle many of the same problems.

This paper documents the first steps towards proving the correctness of Julia's subtyping algorithm. We focus on the interaction of two features: union types and covariant tuples. Tuples are used to represent function signatures, as Julia does not record return types. They are covariant, as a function with more specific arguments is preferred to a more generic one. Union types are used as shorthand to avoid writing multiple functions with the same body. Rules for subtyping union types and covariant tuples have been known for a long time. Based on Vouillon [14], the following is a typical deductive system:

$$\textsc{allexist}\ \frac{t' <: t \qquad t'' <: t}{\texttt{Union\{}t', t''\texttt{\}} <: t} \qquad \textsc{exist}\mathrm{L}\ \frac{t <: t'}{t <: \texttt{Union\{}t', t''\texttt{\}}} \qquad \textsc{exist}\mathrm{R}\ \frac{t <: t''}{t <: \texttt{Union\{}t', t''\texttt{\}}} \qquad \textsc{tuple}\ \frac{t_1 <: t_1' \qquad t_2 <: t_2'}{\texttt{Tuple\{}t_1, t_2\texttt{\}} <: \texttt{Tuple\{}t_1', t_2'\texttt{\}}}$$

While this rule system makes sense, it does not match the semantic intuition for subtyping. If we think of types as sets of values [13], we would expect that a union type would be analogous to a set theoretic union. Similarly, we would then expect that two types would be subtypes if their sets of values were subsets. Therefore, when a union type appears on the left-hand side of a judgment, *all* its components must be subtypes of the right-hand side; when a union type appears on the right-hand side of a judgment, there must *exist* a component that is a supertype of the left-hand side. The above system of rules violates these ideas. Consider the following judgment:

$$\texttt{Tuple\{Union\{}t', t''\texttt{\},}t\texttt{\}} \ <: \ \texttt{Union\{Tuple\{}t', t\texttt{\},Tuple\{}t'', t\texttt{\}\}}$$

Using the semantic subtyping intuition, the judgment should hold. We write the set of values denoted by the type t as $[\![t]\!]$. The left hand side denotes the values $\{\texttt{Tuple\{}v', v''\texttt{\}} \mid v' \in [\![t']\!] \cup [\![t'']\!] \wedge v'' \in [\![t]\!]\}$, while the right hand side denotes $[\![\texttt{Tuple\{}t', t\texttt{\}}]\!] \cup [\![\texttt{Tuple\{}t'', t\texttt{\}}]\!]$. Obviously, the sets are the same. However, we cannot derive this relation from the above rules. According to them, we must pick either t' or t'' on the right hand side, ending up with either $\texttt{Tuple\{Union\{}t', t''\texttt{\},}t\texttt{\}} <: \texttt{Tuple\{}t', t\texttt{\}}$ or $\texttt{Tuple\{Union\{}t', t''\texttt{\},}t\texttt{\}} <: \texttt{Tuple\{}t'', t\texttt{\}}$. In either case, the judgment does not hold. How can this problem be solved?

Some early work [4, 13, 3] focused normalization to decide distributive subtyping between union types, while [7] explores how to reduce subtyping to regular tree expression inclusion. Other designs, as Vouillon [14] or Dunfield [8] do not handle distributivity. These days normalization remains popular, and, for instance, it is relied upon by Frisch et al. work on semantic subtyping [10], by Pearce flow-typing algorithm [12], and by Muelbrock and Tate in their general framework for union and intersection types [11]. Normalization entails rewriting all types into their disjunctive normal form, as unions of union-free types, *before* building the derivation. This lifts all choices to the top level, avoiding the structural entanglements that cause trouble. The correctness of this rewriting step comes from the semantic-subtyping denotational model [10], and the resulting subtype algorithm can be proved both correct and complete. However, this algorithm has two major drawbacks: it is not space efficient, and it does not interact well with the other features of Julia.

The first drawback is that normalization can lead to exponentially bigger types. Real-world Julia code has types like the following [15] whose normal form has 32,768 constituent union-free types, making it impractical to store or to compute with:

```
Tuple{Tuple{Union{Int64, Bool}, Union{String, Bool}, Union{String, Bool},
            Union{String, Bool}, Union{Int64, Bool}, Union{String, Bool},
            Union{String, Bool}, Union{String, Bool}, Union{String, Bool},
            Union{String, Bool}, Union{String, Bool}, Union{String, Bool},
            Union{String, Bool}, Union{String, Bool}, Union{String, Bool}}, Int64}
```

The second drawback of normalization is that it does not interact well with other features of the type system. In particular, Julia supports invariant constructors, which are incompatible with union normalization. For example, the type `Array{Int}` is an array of integers, and is not a subtype of `Array{Any}`. This seemingly simple feature, in conjunction with type variables, makes normalization ineffective. Consider the type $\texttt{Array\{Union\{}t', t''\texttt{\}\}}$. This type denotes the set of arrays whose elements are either of type t' or t''. It would be incorrect to rewrite it as $\texttt{Union\{Array\{}t'\texttt{\}, Array\{}t''\texttt{\}\}}$, as this latter type denotes the set of arrays whose elements are either all of type t' or all of type t''. A weaker disjunctive normal form, only lifting union types inside each invariant constructor, can circumvent this problem. However, doing so only reveals a deeper problem caused by the presence of both invariant constructors and existential types. This is illustrated by the following judgment:

$$\texttt{Array\{Union\{Tuple\{}t\texttt{\}, Tuple\{}t'\texttt{\}\}\}} \;<:\; \exists T\,.\,\texttt{Array\{Tuple\{}T\texttt{\}\}}$$

This judgment holds if we set the existential $T = \texttt{Union\{}t, t'\texttt{\}}$. Since all types are in weak normal form, an algorithm based on the standard system of judgment rules would strip off the array type constructors and proceed. However, since type constructors are invariant on their arguments, it must first test that the relation holds in the original order (e.g. that $\texttt{Union\{Tuple\{}t\texttt{\}, Tuple\{}t\texttt{\}\}} <: \texttt{Tuple\{}T\texttt{\}}$) and in the reverse order (that $\texttt{Tuple\{}T\texttt{\}} <: \texttt{Union\{Tuple\{}t\texttt{\}, Tuple\{}t'\texttt{\}\}}$). It is in this combined check that we run into problems. The subtype check conducted in the original order can be concluded without issue, producing the constraint $\texttt{Union\{}t, t'\texttt{\}} <: T$. However, this constraint on T is stored for checking the reversed direction of subtyping, which is where the problems arise. When we check the opposite subtype order, we end up having to prove that $\texttt{Tuple\{}T\texttt{\}} <: \texttt{Union\{Tuple\{}t\texttt{\}, Tuple\{}t'\texttt{\}\}}$ and in turn either $T <: t$ or $T <: t'$. All of these are unprovable under the assumption that $\texttt{Union\{}t, t'\texttt{\}} <: T$. The key to derive a successful judgment for this relation is to rewrite the right-to-left check into $\texttt{Tuple\{}T\texttt{\}} <: \texttt{Tuple\{Union\{}t, t'\texttt{\}\}}$, which is provable. This *anti-normalisation* rewriting must be performed on sub-judgments of the derivation, and to the best of our knowledge it is not part of any subtype algorithm based on ahead-of-time disjunctive normalisation. As a result, straightforward normalization, even to a relaxed normal form, is incompatible with the full Julia type system.

The complete Julia subtype algorithm is implemented in close to two thousand lines of highly optimized C code. This paper addresses only one part of that algorithm, the technique used to avoid space explosion while dealing with union types and covariant tuples. This is done by defining an iteration strategy over type terms, keeping a string of bits as its state. The space requirement of the algorithm is bounded by the number of unions in the type terms being checked. We prove in Coq that the algorithm is correct and complete with respect to a standard semantic subtyping model.

We have chosen a minimal language with union, tuples, and primitive types to avoid being drawn into the vast complexity of Julia's type algebra. This tiny language is expressive enough to highlight the decision strategy, and make this implementation technique known to a wider audience. The full Julia implementation shows that this technique extends to to invariant constructors and existential types [15], among others. We expect that it can be leveraged in other modern language designs.

Our mechanized proof is available at: `benchung.github.io/subtype-artifact`.

2 A space-efficient subtyping algorithm

Let us focus on a core type language consisting of binary unions, binary tuples and primitive types ranged over by $p_1 \ldots p_n$ where primitive type subtyping is identity, $p_i <: p_i$.

```
type typ =   Prim of int  | Tuple of typ * typ  | Union of typ * typ
```

2.1 Normalization

Using normalization to determine subyping entails rewriting tuples so that unions occur at the top level. Consider the following query:

$$\texttt{Union\{Tuple\{}p_1,p_2\texttt{\}},\texttt{Tuple\{}p_2,p_3\texttt{\}\}} \;<:\; \texttt{Tuple\{Union\{}p_2,p_1\texttt{\}},\texttt{Union\{}p_3,p_2\texttt{\}\}}$$

The term on the left is normal form, but the right term needs to be rewritten as follows:

$$\texttt{Union\{Tuple\{}p_2,p_3\texttt{\}},\texttt{Union\{Tuple\{}p_2,p_2\texttt{\}},\texttt{Union\{Tuple\{}p_1,p_3\texttt{\}},\texttt{Tuple\{}p_1,p_2\texttt{\}\}\}\}}$$

Given normalized types, one more step of rewriting gives us union-free lists of tuples,

$$\ell_1 = \{\texttt{Tuple\{}p_1,p_2\texttt{\}},\texttt{Tuple\{}p_2,p_3\texttt{\}}\}$$

and

$$\ell_2 = \{\texttt{Tuple\{}p_2,p_3\texttt{\}},\texttt{Tuple\{}p_2,p_2\texttt{\}},\texttt{Tuple\{}p_1,p_3\texttt{\}},\texttt{Tuple\{}p_1,p_2\texttt{\}}\}.$$

determining whether $\ell_1 <: \ell_2$ boils down to checking that for each element in ℓ_1 there should be an element in ℓ_2 such that the tuples are subtypes. Intuitively this mirrors the above defined rules ([ALLEXIST], [EXISTL/R], [TUPLE]). As for Julia's algorithm, the intuition is that one can avoid normalization by iterating over the original type terms and visiting every one of the elements of ℓ_1 and ℓ_2 without having to materialize those sets. The remainder of this section explains how this is done.

A possible implementation of normalization-based subtyping can be written compactly. The `subtype` function takes two types and returns true if they are related by subtyping. It delegates its work to `allexist` to check that all normalized terms in its first argument have a super-type, and to `exist` to check that there is at least one super-type in the second argument. The `norm` function takes a type term and returns a list of union-free terms.

```
let subtype(a:typ)(b:typ) = allexist (normalize a) (normalize b)

let allexist(a:typ)(b:typ) =
  foldl (fun acc a' => acc && exist a' b) true a

let exist(a:typ)(b:list typ) =
  foldl (fun acc b' => acc ||  a==b') false b

let rec normalize = function
  | Prim i -> [Prim i]
  | Tuple t t' ->
      map_pair Tuple (cartesian_product (normalize t) (normalize t'))
  | Union t t' -> (normalize t) @ (normalize t')
```

2.2 Iteration with Choice Strings

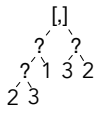
Given a type term such as the following,

$\texttt{Tuple\{Union\{Union\{}p_2, p_3\texttt{\}}, p_1\texttt{\}}, \texttt{Union\{}p_3, p_2\texttt{\}\}}$

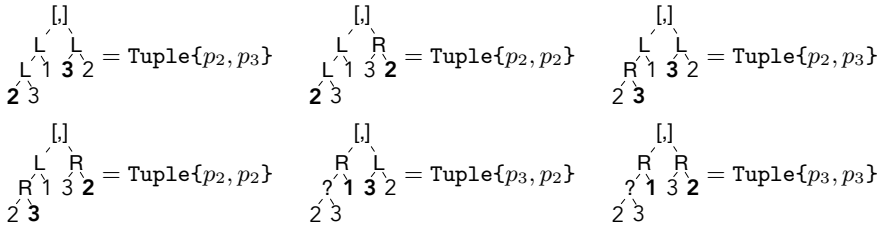
we are looking for an iteration sequence that will yield the following tuples,

$\texttt{Tuple\{}p_2, p_3\texttt{\}}$, $\texttt{Tuple\{}p_2, p_2\texttt{\}}$, $\texttt{Tuple\{}p_1, p_3\texttt{\}}$, $\texttt{Tuple\{}p_1, p_2\texttt{\}}$, $\texttt{Tuple\{}p_3, p_3\texttt{\}}$, $\texttt{Tuple\{}p_3, p_2\texttt{\}}$.

An alternative representation for the term is a tree, where each occurence of union node is a *choice point*. The following tree thus has three choice points.



At each choice point we can go either left or right, making such a decision at each points leads to visit one particular tuple.



Each tuple is uniquely determined by the original type term t and a choice string c. In the above example, the result of iteration through the normalized, union-free, type terms is defined by the strings `LLL`, `LLR`, `LRL`, `LRR`, `RL`, `RR`. The length of each string is bounded by the number of unions in a term.

The iteration sequence in the above example is thus $\texttt{LL}\underline{\texttt{L}} \to \texttt{L}\underline{\texttt{L}}\texttt{R} \to \texttt{LR}\underline{\texttt{L}} \to \underline{\texttt{L}}\texttt{RR} \to \texttt{R}\underline{\texttt{L}} \to \texttt{RR}$. Stepping from a choice string c to the next string consists of splitting c in three, $c'\,\texttt{L}\,c''$, where c' can be empty and c'' is a possibly empty sequence of `R`. The next string is $c'\,\texttt{R}\,c_{pad}$, that is to say it retains the prefix c', toggles `L` to `R`, and is padded by a sequence of `L`s. If there is no `L` in c, iterations has terminated.

One step of iteration is performed by calling the `next` function with a type term and a choice string. `next` either returns the next string in the sequence or `None`. Internally, it calls on `step` to toggle the last `L` and shorten the string (constructing $c'\,\texttt{R}$). Then it call on `pad` to add the trailing sequence of `L`s (constructing $c'\,\texttt{R}\,c_{pad}$).

```
type choice = L | R

let rec next(a:typ)(l:choice list) =
  match step l with
    | None -> None
    | Some(l') -> Some(fst (pad a l'))
```

The `step` function delegates the job of flipping the last occurrence of `L` to `toggle`. For ease of programming, it reverses the string so that `toggle` can be a simple recursion without an accumulator. If the given string has no `L`, then `toggle` returns empty, and `step` returns `None`.

```
let step(l:choice list) =
  match rev (toggle (rev l)) with
  | [] -> None
  | hd::tl -> Some(hd::tl)

let rec toggle = function
  | [] -> []
  | L::tl -> R::tl
  | R::tl -> toggle tl
```

The `pad` function takes a type term and a choice string to be padded. It returns a pair, the first element is the padded string and the second is remaining string. `pad` traverses the term, visiting both side of each tuple, and for unions it uses the given choice string to direct its visit. Each union encountered consumes a character out of the input string, once the string is fully consumed, any remaining unions are treated as if there was a `L`. The first component of the returned value is the choice given as argument extended with a number of `L` corresponding to the number of unions encountered after string ran out.

```
let rec pad t l =
   match t,l with
   | (Prim i,l) -> ([],l)
   | (Tuple(t,t'),l) ->
      let (h,tl) = pad t l in
      let (h',tl') = pad t' tl in (h @ h',tl')
   | (Union(t,_),L::r) ->
      let (h,tl) = pad t r in (L::h,tl)
   | (Union(_,t),R::r) ->
      let (h,tl) = pad t r in (R::h,tl)
   | (Union(t,_),[]) -> (L::(fst(pad t [])),[])
```

To obtain the initial choice string, the string only composed of `L`s, it suffices to call `pad` with the type term under consideration and an empty list. The first element of the returned tuple is the initial choice string. For convenience, we define the function `initial` for this.

```
let initial(t:typ) = fst (pad t [])
```

2.3 Subtyping with Iteration

Julia's subtyping algorithm visits union-free type terms using choice strings to iterate over types. The `subtype` function takes two type terms, `a` and `b`, and returns true if they they are related by subtyping. It does so by iterating over all union-free type terms in `a`, and checking that for each of them, there exists a union-free type term in `b` that is a super-type.

```
let subtype(a:typ)(b:typ) = allexist a b (initial a)
```

The `allexist` function takes two type terms, `a` and `b`, and a choice string `f`, and returns true if `a` is a subtype of `b` for the iteration sequence starting at `f`. This is achieved by recursively testing that for each union-free type term in `a` (induced by `a` and the current value of `f`), there exists a union-free super-type in `b`.

```
let rec allexist(a:typ)(b:typ)(f:choice list) =
  match exist a b f (initial b) with
  | true -> (match next a f with
             | Some ns -> allexist a b ns
             | None -> true)
  | false -> false
```

Similarly, the `exist` function takes two type terms, `a` and `b`, and choice strings, `f` and `e`. It returns true if there exists in `b`, a union-free super-type of the type specified by `f` in `a`. This is done by recursively iterating through `e`. The determination if two terms are related is delegated to the `sub` function.

```
type res = NotSub | IsSub of choice list * choice list

let rec exist(a:typ)(b:typ)(f:choice list)(e:choice list) =
  match sub a b f e with
  | IsSub(_,_) -> true
  | NotSub ->
     (match next b e with
      | Some ns -> exist a b f ns
      | None -> false)
```

Finally, the `sub` function take two type terms and two choice strings and return a value of type `res` which can be `NotSub` to indicate that the types are not subtypes or `IsSub(_,_)` when they are. If the two types are primitives, then they are only subtypes if they are equal. If the types are tuples, they are subtypes is both of their elements are subtypes. Note that the return type of `sub`, when successful, hold the unused choice strings for both type arguments. When confronted with a union, `sub` will follow the choice strings to decide which branch to take. Consider for instance the case when the first type term is `Union(t1,t2)` and the second is type `t`, if the first element of the choice string is an `L`, then `t1` and `t` will be checked, otherwise `sub` will check `t2` and `t`.

```
let rec sub t1 t2 f e =
  match t1,t2,f,e with
  | (Prim i,Prim j,f,e) -> if i==j then IsSub(f,e) else NotSub
  | (Tuple(a1,a2), Tuple(b1,b2),f,e) ->
     (match sub a1 b1 f e with
      | IsSub(f', e') -> sub a2 b2 f' e'
      | NotSub -> NotSub)
  | (Union(a,_),b,L::f,e) -> sub a b f e
  | (Union(_,a),b,R::f,e) -> sub a b f e
  | (a,Union(b,_),f,L::e) -> sub a b f e
  | (a,Union(_,b),f,R::e) -> sub a b f e
```

2.4 Further optimization

We have presented an implementation that used lists to represent choice strings. It thus required allocation when adding elements to the list and for reversing the list. In Julia, choice strings are represented by bit vectors of size bounded by the number of unions in each type term. Once that size is known and the bit vector is created, no further allocation is required.

3 Correctness and Completeness of Subtyping

To prove the correctness of Julia's subtyping we take the following general approach. We start by giving a denotational semantics for types from which we derive a definition of semantic subtyping. Then we easily prove that a normalization-based subtyping algorithm is correct and complete. Rather than directly working with the notion of choice strings as iterators over types, we start with a simpler structure, namely that of iterators over the trees induced by type terms. We prove correct and complete a subtype algorithm that uses these simpler iterators. Finally, we establish a correspondence between tree iterators and choice list iterators. This concludes our proof of correctness and completeness, details can be found in the Coq mechanization.

The denotational semantics we use for types is as follows:

$$\begin{aligned} [\![p_i]\!] &= \{p_i\} \\ [\![\texttt{Union}\{t_1, t_2\}]\!] &= [\![t_1]\!] \cup [\![t_2]\!] \\ [\![\texttt{Tuple}\{t_1, t_2\}]\!] &= \{\texttt{Tuple}\{t_1', t_2'\} \mid t_1' \in [\![t_1]\!], t_2' \in [\![t_2']\!]\} \end{aligned}$$

We define subtyping as follows, if $[\![t]\!] \subseteq [\![t']\!]$, then $t <: t'$. This leads to the definition of subtyping in our restricted language.

▶ **Definition 1.** *The subtyping relation* $t_1 <: t_2$ *holds iff* $\forall t_1' \in [\![t_1]\!], \exists\, t_2' \in [\![t_2]\!], t_1' = t_2'$.

The use of equality for relating types is a simplification afforded by the structure of primitives.

3.1 Subtyping with Normalization

The correctness and completeness of the normalization-based subtyping algorithm of Section 2.1 requires proving that the `normalize` function returns all union-free type terms.

▶ **Lemma 2** (NF Equivalence). $t' \in [\![t]\!]$ *iff* $t' \in$ *normalize* t.

Theorem 3 states that the `subtype` relation of Section 2.1 abides by Definition 1 because it uses `normalize` to compute the set of union-free type terms for both argument types, and directly checks subtyping.

▶ **Theorem 3** (NF Subtyping). *For all* a *and* b, *subtype* a b *iff* $a <: b$.

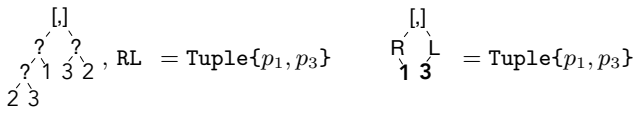
Therefore, normalization based subtyping is correct against our definition.

3.2 Subtyping with Tree Iterators

Reasoning about iterators that use choice strings, as described in Section 2.2, is tricky as it requires simultaneously reasoning about the structure of the type term and the validity of the choice string that represents the iterator's state. Instead, we propose to use an intermediate data structure called a tree iterator to tie the two together and thus makes reasoning simpler.

A tree iterator is a representation of the iteration state embedded in a type term. Thus a tree iterator yields a union-free tuple, and given a type term, a tree iterator can either step to a successor state or is a final state. Recalling the graphical notation of Section 2.2, we can represent the state of iteration as a combination of type term and a choice or, equivalently, as a tree iterator.

Choice string: *Tree iterator:*



This structure-dependent construction makes tree iterators less efficient than choice strings. A tree iterator must have a node for each structural element of the type being iterated over, and is thus less space-efficient than the simple choices-only strings. However, it is easier to prove subtyping correct for tree iterators first.

Tree iterators depend on the type term they iterate over. The possible states are `IPrim` at primitives, `ITuple` at tuples, and for unions either `ILeft` or `IRight`.

```
Inductive iter: Typ -> Set :=
| IPrim : forall i, iter (Prim i)
| ITuple : forall t1 t2, iter t1 -> iter t2 -> iter (Tuple t1 t2)
| ILeft : forall t1 t2, iter t1 -> iter (Union t1 t2)
| IRight : forall t1 t2, iter t2 -> iter (Union t1 t2).
```

The `next` function for tree iterators steps in a depth-first, right-to-left order. We have four cases to worry about. For a primitive type, there is no successor state. A tuple steps its second child; if that has no successor step, then it steps its first child and reset the second child. When given a `ILeft` or an `IRight` it tries step it only child. If the child has no successor, an `ILeft` steps to an `IRight` and its child is set to the right child of the corresponding node in the type term.

```
Fixpoint next(t:Typ)(i:iter t): option(iter t) := match i with
  | IPrim _ => None
  | ITuple t1 t2 i1 i2 =>
    match (next t2 i2) with
    | Some i' => Some(ITuple t1 t2 i1 i')
    | None =>
      match (next t1 i1) with
      | Some i' => Some(ITuple t1 t2 i' (start t2))
      | None => None
      end
    end
  | ILeft t1 t2 i1 =>
    match (next t1 i1) with
    | Some(i') => Some(ILeft t1 t2 i')
    | None => Some(IRight t1 t2 (start t2))
    end
  | IRight t1 t2 i2 =>
    match (next t2 i2) with
    | Some(i') => Some(IRight t1 t2 i')
    | None => None
    end
  end.
```

An induction principle for tree iterators is needed to reason about all iterator states for a given type. First, we show that iterators eventually reach a final state. This is done with function `inum` that assigns natural numbers to each state. It simply counts the number of

remaining steps in the iterator, using `tnum` to count the total number of union-free types denoted by some type `t` as a helper.

```
Fixpoint tnum(t:Typ):nat :=
  match t with
  | Prim i => 1
  | Tuple t1 t2 => tnum t1 * tnum t2
  | Union t1 t2 => tnum t1 + tnum t2
  end.

Fixpoint inum(t:Typ)(ti:iter t):nat :=
  match ti with
  | IPrim i => 0
  | ITuple t1 t2 i1 i2 => inum t1 i1 * total_num t2 + inum t2 i2
  | IUnionL t1 t2 i1 => inum t1 i1 + total_num t2
  | IUnionR t1 t2 i2 => inum t2 i2
  end.
```

This function then lets us define the key theorem needed for the induction principle. At each step, the value of `inum` decreases by 1, and since it cannot be negative, the iterator must therefore reach a final state.

▶ **Lemma 4** (Monotonicity). *If* $\mathit{next}\ t\ it = it'$ *then* $\mathit{inum}\ t\ it = 1 + \mathit{inum}\ t\ it'$.

It is not possible to define an induction principle over `next`. By monotonicy, `next` eventually reaches a final state. For any property of interest, if we prove that it holds of the final state and for the induction step, we can prove it holds for every state for that type.

▶ **Theorem 5** (Tree Iterator Induction). *Let P be any property of tree iterators for some type t. Suppose P holds of the final state, and whenever P holds of a successor state it then it holds of its precursor it' where* $\mathit{next}\ t\ it' = it$. *Then P holds of every iterator state over t.*

Now, one can prove correctness of the subtyping algorithm with tree iterators. We implement subtyping with respect to choice lists in the Coq implementation by deciding subtyping between the two union-free types induced by the iterators over the two original types, avoiding having to prove termination of the combined algorithm. The decision procedure implemented in `sub` first uses `here` on both types and their iterators to pick the union-free components of each original type given by their iterators, then calls `ufsub` to decide union-free subtyping between them.

This procedure needs two helpers, `here` and `ufsub`. The function `here` walks the given iterator, producing a union-free type mirroring its state. To decide subtyping between the resulting union-free types, `ufsub` checks equality between `Prim` s and recurses on the elements of `Tuple` s, while returning false for all other types. Since `here` will never produce a union type, the case of `ufsub` for them is irrelevant, and is false by default.

```
Fixpoint here (t:Typ)(i:iter t):Typ :=
  match i with
  | IPrim i => Prim i
  | ITuple t1 t2 p1 p2 => Tuple (here t1 p1) (here t2 p2)
  | ILeft t1 t2 pl => (here t1 pl)
  | IRight t1 t2 pr => (here t2 pr)
  end.

Fixpoint ufsub(t1 t2:Typ) :=
```

```
  match (t1, t2) with
  | (Prim p, Prim p') => Nat.eqb p p'
  | (Tuple a a', Tuple b b') =>
      andb (ufsub a b) (ufsub a' b')
  | (_, _) => false
  end.

Definition sub (a b:Typ) (ai:iter a) (bi:iter b) :=
    ufsub (here a ai) (here b bi).
```

Versions of `exist` and `allexist` that use tree iterators are given next. They are similar to the string iterator functions of Section 2.2. `exist` tests if the subtyping relation holds in the context of the current iterator states for both sides. If not, it recurs on the next state. Similarly, `allexist` uses its iterator for a in conjunction with `exist` to ensure that the current left hand iterator state has a matching right hand state. We prove termination of both using Lemma 4.

```
Definition subtype(a b:Typ) = allexist a b (initial a)

Program Fixpoint allexist (a b:typ)(ia:iter a) {measure(inum ia)} =
   exists a b ia (initial b) &&
      (match next a ia with
       | Some(ia') => allexist a b ia'
       | None => true).

Program Fixpoint exist(a b:typ)(ia:iter a)(ib:iter b)
                                        {measure(inum ib)} =
   subtype a b ia ib  ||
      (match next b ib with
       | Some(ib') => exist a b ia ib'
       | None => false).
```

The denotation of a tree iterator state $\mathcal{R}(i)$ is the set of states that can be reached using `next` from i. Let $a(i)$ indicate the union-free type produced from the type a at i, and $|i|_a$ is the set $\{a(i') \mid i' \in \mathcal{R}(i)\}$, the union-free types that result from states in the type a reachable by i. This lets us prove that the set of types corresponding to states reachable from the initial state of an iterator is equal to the set of states denoted by the type itself.

▶ **Lemma 6** (Initial equivalence). $|\mathit{initial}\ a|_a = [\![a]\!]$.

Next, Theorem 5 allows us to show that `exists` of a, b, with i_a and i_b will try to find an iterator state i'_b starting from i_b such that $b(i'_b) = a(i_a)$. The desired property trivially holds when $|i_b|_b = \emptyset$, and if the iterator can step then either the current union-free type is satisfying or we defer to the induction hypothesis.

▶ **Theorem 7.** $\mathit{exist}\, a\, b\, i_a\, i_b$ *holds iff* $\exists t \in |i_b|_b, a(i_a) = t$.

We can then appeal to both Theorem 7 and Lemma 6 to show that $\texttt{exist}\, a\, b\, i_a\, (\texttt{initial}\, b)$ will find a satisfying union-free type on the right hand side if it exists in $[\![b]\!]$. Using this, we can then use Theorem 5 in an analogous way to `exist` to show that `allexist` is correct up to the current iterator state.

▶ **Theorem 8.** $\mathit{allexist}\, a\, b\, i_a$ *holds iff* $\forall a' \in |i_a|_a, \exists b' \in [\![b]\!], a' = b'$.

Finally, we can appeal to Theorem 8 and Lemma 6 again to show the subtyping algorithm to be correct.

▶ **Theorem 9.** *subtype a b holds iff* $\forall a' \in [\![a]\!], \exists b' \in [\![b]\!], a' = b'$.

3.3 Subtyping with Choice Strings

We prove the subtyping algorithm using choice strings by showing that string iterators simulate tree iterators. To relate tree iterators to choice string iterators, we use the `itp` function, which traverses a tree iterator state and produces a choice string using a depth-first search.

```
Fixpoint itp{t:Typ}(it:iter t):choice list :=
   match it with
   | IPrim _ => nil
   | ITuple t1 t2 it1 it2 => (itp t1 it1)++(itp t2 it2)
   | ILeft t1 _ it1 => Left::(itp t1 it1)
   | IRight _ t2 it1 => Right::(itp t2 it1)
   end.
```

Next, in order to show that the choice string iteration order is exhibited when linearizing tree iterators into choice strings. The `next` function in Section 2.2 worked by finding the last `L` in the choice string, turning it into an `R`, and replacing the rest with `L`s until the type was valid. If we use `itp` to translate both the initial and final states for a valid `next` step of a tree iterator, we see the same structure.

▶ **Lemma 10** (Linearized Iteration). *For some type t and tree iterators it it', if $\texttt{next}\, t\, it = it'$, there exists some prefix c', an initial suffix c'' made up of* `R`*s, and a final suffix c''' consisting of* `L`*s such that $\texttt{itp}\ t\ it = c'\ \texttt{Left}\, c''$ and $\texttt{itp}\ t\ it' = c'\ \texttt{Right}\, c'''$.*

We can then prove that stepping a tree iterator state is equivalent to stepping the linearized versions of the state using the choice string `next` function.

▶ **Lemma 11** (Step Equivalence). *If it and it' are tree iterator states and $\texttt{next}\ it = it'$, then $\texttt{next}(\texttt{itp}\ it) = (\texttt{itp}\ it')$.*

Finally, the initial state of a tree iterator linearizes to the initial state of a choice string iterator.

▶ **Lemma 12** (Initial Equivalence). $\texttt{itp}(\texttt{initial}\, t) = \texttt{pad}\, t\ \texttt{[]}$.

The functions `exist` and `allexist` for choice string based iterators are identical to those for tree iterators (though using choice string iterators internally), and `sub` is as described in Section 2.2. The proofs of correctness for the choice string subtype decision functions use the tree iterator induction principle (Theorem 5), and are thus in terms of tree iterators. By Lemma 11, however, each step that the tree iterator takes will be mirrored precisely `itp` into choice strings. Similarly, the initial states are identical by Lemma 12. As a result, the sequence of states checked by each of the iterators is equivalent with `itp`.

▶ **Lemma 13.** $\texttt{exist}\ a\ b\ (\texttt{itp}\ i_a)(\texttt{itp}\ i_b)$ *holds iff* $\exists t \in |i_b|_b, a(ia) = t$.

With the correctness of `exist` following from the tree iterator definition, we can apply the same proof methodology to show that `allexist` is correct when over translated tree iterators. In order to do so, we instantiate Lemma 13 with Lemma 6 and Lemma 12 to show that if `exist a b (itp ia) (pad t [])` then $\exists \texttt{t} \in [\![\texttt{b}]\!], \texttt{a}(\texttt{ia}) = \texttt{t}$, allowing us to check each of the exists cases while establishing the forall-exists relationship.

▶ **Lemma 14.** $\texttt{allexist}\ a\ b(\texttt{itp}\ i_a)$ *holds iff* $\forall a' \in |i_a|_a, \exists b' \in [\![b]\!], a' = b'$.

We can then instantiate Lemma 14 with Lemma 12 and Lemma 6 to show that `allexist` for choice strings ensures that the forall-exists relation holds.

▶ **Theorem 15.** $\texttt{allexist}\ a\ b\ (\texttt{pad}\ t\ \texttt{[]})$ *holds iff* $\forall a' \in [\![a]\!], \exists b' \in [\![b]\!], a' = b'$.

Finally, we can prove that subtyping is correct using the choice string algorithm.

▶ **Theorem 16.** $\texttt{subtype}\ a\ b$ *holds iff* $\forall a' \in [\![a]\!], \exists b' \in [\![b]\!], a' = b'$.

Thus, we can correctly decide subtyping with distributive unions and tuples using the choice list based implementation of iterators.

4 Complexity

Worst case time complexity of Julia's subtyping algorithm and normalization-based approaches is determined by the number of terms that would exist in the normalized type. In the worst case, there are 2^n union-free tuples in the fully normalized version of a type that has n unions. Each of these tuples must always be explored. As a result, both algorithms have worst-case $O(2^n)$ time complexity. The approaches differ, however, in space complexity. The normalization approach computes and stores each of the exponentially many alternatives, so also has $O(2^n)$ space complexity. However, Julia need only store the choice made at each union, thereby offering $O(n)$ space complexity.

Julia's algorithm improves best-case time performance. Normalization always experiences worst case time and space behavior as it has to precompute the entire normalized type. Julia's iteration-based algorithm can discover the relation between types early. In practice, many queries are of the form $\mathit{uft} <: \mathit{union}(t_1 ... t_n)$ where uft is an already union-free tuple. As a result, all that Julia needs to do is find one matching tuple in $t_1 ... t_n$, which could be handled quickly by a fast path.

5 Conclusion

We have described and proven correct a subtyping algorithm for covariant tuples and unions that use iterators instead of normalization. This algorithm is able to decide subtyping in the presence of distributive semantics for union over tuples.

Future work is to handle some additional features of the Julia language. Our next steps will be subtyping for primitive tpyes, existential type variables, and invariant constructors. Adding a subtyping to primitive types would be the simplest change. The challenge is how to retain completeness, as a primitive subtype heirarchy and semantic subtyping have undesirable interactions. For example, if the primitive subtype hierarchy contains only the relations $p_2 <: p_1$ and $p_3 <: p_1$, then is p_1 a subtype of $\texttt{Union}\{p_2, p_3\}$? In a semantic subtyping system, they are, but this requires changes both to the denotational framework and the search space of the iterators. Existential type variables create substantial new complexities in the state of the algorithm. No longer is the state solely restricted to that of the iterators being attempted; now, the state includes variable bounds that are accumulated as the algorithm compares types to type variables. As a result, correctness becomes a much more complex contextually-linked property to prove. Finally, invariant type constructors induce contravariant subtyping, which in the context of existential type variables has the potential to create cycles within the subtyping relation. As a consequence, the termination of our algorithm comes into question even if the language is otherwise limited to avoid provable non-termination.

Acknowledgments

The authors thank Jiahao Chen for starting us down the path of understanding Julia, and Jeff Bezanson for coming up with Julia's subtyping algorithm. This work received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement 695412), the NSF (award 1544542 and award 1518844), the ONR (grant 503353), and the Czech Ministry of Education, Youth and Sports (grant agreement CZ.02.1.01/0.0/0.0/15_003/0000421).

References

1. Hack. https://hacklang.org/. Accessed: 2019-01-11.
2. Typescript language specification. URL: https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md.
3. Alexander Aiken and Brian R. Murphy. Implementing regular tree expressions. In *Functional Programming Languages and Computer Architecture*, 1991. doi:10.1007/3540543961_21.
4. Franco Barbanera and Mariangiola Dezani-Ciancaglini. Intersection and union types. In *International Symposium on Theoretical Aspects of Computer Software*, 1991. doi:10.1007/3-540-54415-1_69.
5. Jeff Bezanson. *Abstraction in Technical Computing*. PhD thesis, Massachusetts Institute of Technology, 2015.
6. Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1), 2017. doi:10.1137/141000671.
7. Flemming M. Damm. Subtyping with union types, intersection types and recursive types. In *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*, pages 687–706, 1994. doi:10.1007/3-540-57887-0_121.
8. Joshua Dunfield. Elaborating intersection and union types. *J. Funct. Program.*, 24(2-3):133–165, 2014. doi:10.1017/S0956796813000270.
9. Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping. In *Symposium on Logic in Computer Science (LICS)*, 2002. doi:10.1109/LICS.2002.1029823.
10. Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4), 2008. doi:10.1145/1391289.1391293.
11. Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated subtyping. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018. doi:10.1145/3276482.
12. David J. Pearce. Sound and complete flow typing with unions, intersections and negations. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, pages 335–354, 2013. doi:10.1007/978-3-642-35873-9_21.
13. Benjamin Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.
14. Jerome Vouillon. Subtyping union types. In *Computer Science Logic (CSL)*, 2004. doi:10.1007/978-3-540-30124-0_32.
15. Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. Julia subtyping: A rational reconstruction. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018. doi:10.1145/3276483.