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 correct method to invoke. Julia's subtyping algorithm is surprisingly complex, and determining 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 normalized unions inside tuples to disjunctive normal form. However, this strategy has two drawbacks: complex type signatures induce space explosion, and interference between normalization and other features of Julia's type system. In this paper, we describe the algorithm that Julia uses to compute subtyping between tuples and unions—an algorithm that is immune to space explosion and plays well with other features of the language. 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 being adopted in mainstream languages. In some cases, such as Julia [7] or TypeScript [2], they are exposed at the source level. In others, such as Hack [1], they are only used internally as part of type inference. As a result, subtyping algorithms between union types are of increasing practical import. The standard subtyping algorithm for this combination of features has, for some time, been exponential in both time and space. An alternative algorithm, linear in space but still exponential in time, has been tribal knowledge in the subtyping community [15]. In this paper, we describe and prove correct an implementation of that algorithm.

We observed the algorithm in our prior work formalizing the Julia subtyping relation [17]. There, we described Julia's subtyping relation as it arose from its decision procedure but were unable to prove it correct. Indeed, we found bugs in the Julia implementation and identified unresolved correctness issues. Contemporary work addresses some correctness concerns [5] but leaves algorithmic correctness open.

Julia's subtyping algorithm [6] is used for method dispatch. While Julia is dynamically typed, method arguments can have type annotations. These annotations allow one method to be implemented by multiple functions. At run time, Julia searches for the most specific applicable function for a given invocation. Consider theese 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)
```


42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The first two methods implement, respectively, multiplicaton of range 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 uses subtyping to decide which of the methods to call at any specific site. The call `1*(1:4)` dispatches to the first, `1*1.1` the second, and `1*1` the third.

Julia offers programmers a rich type language to express complex relationships in type signatures. The type language includes nominal primitive types, union types, existential types, covariant tuples, invariant parametric datatypes, and singletons. Intuitively, subtyping between types is based on semantics subtyping, the subtyping relation between types holds when the sets of values they denote are a subset of one another [7]. We write the set of values represented by a type t as $[\![t]\!]$. Under semantic subtyping, the types t_1 and t_2 are subtypes iff $[\![t_1]\!] \subseteq [\![t_2]\!]$. From this, we derive a *forall-exists* intuition for subtyping: for every value denoted on the left-hand side, there must exist some value on the right-hand side to match it, thereby establishing the subset relation. This simple intuition is, however, complicated to check algorithmically.

In this paper, we focus on the interaction of two features: covariant tuples and union types. These two kinds of type are important to Julia's semantics. Julia does not record return types, so a function's signature consists solely of the tuple of its argument types. These tuples are covariant, as a function with more specific arguments is preferred to a more generic one. Union types are widely used as shorthand to avoid writing multiple functions with the same body. As a consequence, Julia library developers write many functions with union typed arguments, functions whose relative specificity must be decided using subtyping. To prove the correctness of the subtyping algorithm, we first examine typical approaches in the presence of union types. Based on Vouillon [16], the following is a typical deductive system for subtyping union types:

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

While this rule system might seem to make intuitive sense, it does not match the semantic intuition for subtyping. For instance, consider the following judgment:

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

Using semantic subtyping, the judgment should hold. The set of values denoted by a union $[\![\texttt{Union\{}t_1, t_2\texttt{\}}]\!]$ is just the union of the set of values denoted by each of its members $[\![t_1]\!] \cup [\![t_2]\!]$. A tuple $\texttt{Tuple\{}t_1, t_2\texttt{\}}$'s denotation is the set of tuples of the respective values $\{\texttt{Tuple\{}v_1, v_2\texttt{\}} \mid v_1 \in [\![t_1]\!] \wedge v_2 \in [\![t_2]\!]\}$. Therefore, 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 $[\![\texttt{Tuple\{}t', t\texttt{\}}]\!] \cup [\![\texttt{Tuple\{}t'', t\texttt{\}}]\!]$ or equivalently $\{\texttt{Tuple\{}v', v''\texttt{\}} \mid v' \in [\![t']\!] \cup [\![t'']\!] \wedge v'' \in [\![t]\!]\}$. These sets are the same, and therefore subtyping should hold in either direction between the left- and right-hand types. 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 using $\textsc{exist}$L or $\textsc{exist}$R, respectively, 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?

Most prior work addresses this problem by normalization[4, 14, 3], rewriting all types into their disjunctive normal form, as unions of union-free types, *before* building the derivation. Now all choices are made at the top level, avoiding the structural entanglements that cause difficulties. The correctness of this rewriting step comes from the semantic denotational model, and the resulting subtyping algorithm can be proved both correct and complete. Other proposals, such as Vouillon [16] and Dunfield [9], do not handle distributivity. Normalization is used by Frisch et al.'s [10], by Pearce's flow-typing algorithm [13], and by Muehlboeck and Tate in their general framework for union and intersection types [12]. Few alternatives have been proposed, with one example being Damm's reduction of subtyping to regular tree expression inclusion [8].

However, a normalization-based algorithm has two major drawbacks: it is not space efficient, and other features of Julia render it incorrect. The first drawback is caused because normalization can create exponentially large types. Real-world Julia code [17] has types like the following whose normal form has 32,768 constituent union-free types:

```
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 arises because of type-invariant constructors. For example, `Array{Int}` is an array of integers, and is not a subtype of `Array{Any}`. In conjunction with type variables, this makes normalization ineffective. Consider $\texttt{Array\{Union\{}t', t''\texttt{\}\}}$, the set of arrays whose elements are either t' or t''. It wrong to rewrite it as $\texttt{Union\{Array\{}t'\texttt{\}}, \texttt{Array\{}t''\texttt{\}\}}$, as this denotes the set of arrays whose elements are either all t' or t''. A weaker disjunctive normal form, only lifting union types inside each invariant constructor, is a partial solution. However, this reveals a deeper problem caused by existential types. Consider the judgment:

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

It holds if the existential variable T is instantiated with $\texttt{Union\{}t, t'\texttt{\}}$. If types are in invariant-constructor weak normal form, an algorithm would strip off the array type constructors and proceed. However, since type constructors are invariant, the algorithm must test that both $\texttt{Union\{Tuple\{}t\texttt{\}}, \texttt{Tuple\{}t\texttt{\}\}} <: \texttt{Tuple\{}T\texttt{\}}$ and $\texttt{Tuple\{}T\texttt{\}} <: \texttt{Union\{Tuple\{}t\texttt{\}}, \texttt{Tuple\{}t'\texttt{\}\}}$ hold. The first of these can be concluded without issue, producing the constraint $\texttt{Union\{}t, t'\texttt{\}} <: T$. However, this constraint on T is retained for checking the reverse direction, which is where problems arise. When checks the reverse direction, the aglorithm has to prove that $\texttt{Tuple\{}T\texttt{\}} <: \texttt{Union\{Tuple\{}t\texttt{\}}, \texttt{Tuple\{}t'\texttt{\}\}}$, 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 deriving 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-normalization* rewriting must be performed on sub-judgments of the derivation; to the best of our knowledge it is not part of any subtyping algorithm based on ahead-of-time disjunctive normalization.

Julia's subtyping algorithm avoids these problems, but it is difficult to determine how: the complete subtyping algorithm is implemented in close to two thousand lines of highly optimized C code. In this paper, we describe and prove correct 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 use a minimal type language with union, tuples, and primitive types to avoid being drawn into the vast complexity of Julia's type language. This tiny language is expressive enough to highlight the decision strategy and illustrate the structure of the algorithm. Empirical evidence from Julia's implementation suggests that this technique extends to invariant constructors and existential types [17], among others. We expect that the algorithm we describe 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

Formally, our core type language consists of binary unions, binary tuples, and primitive types ranged over by $p_1 \dots p_n$, as shown below:

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

We define subtyping for primitives as the identity, so $p_i <: p_i$.

2.1 Normalization

To explain the operation of the space-efficient algorithm, we first describe how normalization can be used as part of subtyping. Normalization rewrites types to move all internal unions to the top level. The resultant term consists of a union of union-free terms. Consider the following relation:

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

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

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

The top level unions can then be viewed as sets of union-free-types equivalent to each side,

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

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 $\ell_1 <: \ell_2$ is equivalent to checking that for each tuple component t_1 in ℓ_1, there should be an element t_2 in ℓ_2 such that $t_1 <: t_2$. Checking this final relation is straightforward, as neither t_1 nor t_2 may contain unions. Intuitively, this mirrors the rules ([ALLEXIST], [EXISTL/R], [TUPLE]).

A possible implementation of normalization-based subtyping can be written compactly, as shown in the code below. 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 supertype, and to `exist` to check that there is at least one supertype 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 (norm a) (norm b)

let allexist(a:list typ)(b:list 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 norm = function
  | Prim i -> [Prim i]
  | Tuple t t' ->
      map_pair Tuple (cartesian_product (norm t) (norm t'))
  | Union t t' -> (norm t) @ (norm t')
```

However, as previously described, this expansion is space-inefficient. Julia's algorithm is more complicated, but avoids having to pre-compute the set of normalized types as `norm` does.

2.2 Iteration with choice strings

Given a type term such as the following,

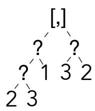
$\texttt{Tuple\{Union\{Union\{}p_2, p_3\texttt{\}}, p_1\texttt{\}}, \texttt{Union\{}p_3, p_2\texttt{\}\}}$

we want an algorithm that checks the following tuples,

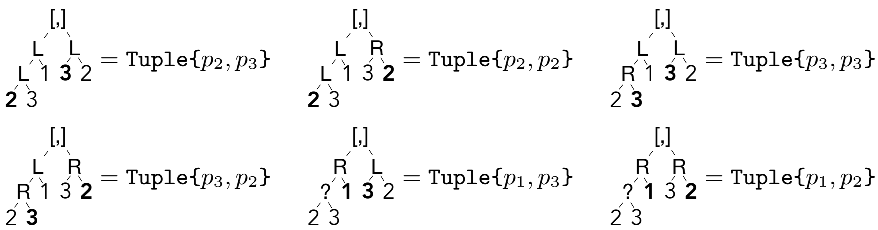
$\texttt{Tuple\{}p_2, p_3\texttt{\}}$, $\texttt{Tuple\{}p_2, p_2\texttt{\}}$, $\texttt{Tuple\{}p_1, p_3\texttt{\}}$, $\texttt{Tuple\{}p_1, p_2\texttt{\}}$, $\texttt{Tuple\{}p_3, p_3\texttt{\}}$, $\texttt{Tuple\{}p_3, p_2\texttt{\}}$

without having to compute and store all of them ahead-of-time. This algorithm should be able to generate each tuple on-demand while still being guaranteed to explore every tuple of the original type's normal form.

To illustrate the process that the algorithm uses to generate each tuple, consider the type term being subtyped. An alternative representation for the term is a tree, where each occurrence of a union node is a *choice point*. The following tree thus has three choice points, each represented as a ? symbol:



At each choice point we can go either left or right; making such a decision at each point leads to visiting 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 $\texttt{LL}\underline{\texttt{L}} \rightarrow \texttt{L}\underline{\texttt{L}}\texttt{R} \rightarrow \texttt{LR}\underline{\texttt{L}} \rightarrow \underline{\texttt{L}}\texttt{RR} \rightarrow \texttt{R}\underline{\texttt{L}} \rightarrow \texttt{RR}$, where the underlined choice is next one to be toggled in that step. 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`s. The next string is $c'\,\texttt{R}\,c_{pad}$, that is to say it retains the prefix c', toggles the `L` to an `R`, and is padded by a sequence of `L`s. The leftover tail c'' is discarded. If there is no `L` in c, iteration terminates.

One step of iteration is performed by calling the `next` function with a type term and a choice string (encoded as a `list` of `choice`s); `next` either returns the next string in the sequence or `None`. Internally, it calls `step` to toggle the last `L` and shorten the string (constructing $c'\,\texttt{R}$). Then it calls 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, whose first element is the padded string and second element is the string left over from the current type. Each union encountered by `pad` in its traversal of the type consumes a character from the input string. Unions explored after the exhaustion of the original choice string are treated as if there was an `L` remaining in the choice string. The first component of the returned value is the original choice string extended with an `L` for every union encountered after exhaustion of the original.

```
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 composed solely 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 are related by subtyping. It does so by iterating over all union-free type terms t_a in `a`, and checking that for each of them, there exists a union-free type term t_b in `b` such that $t_a <: t_b$.

```
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 takes two type terms and choice strings and returns a value of type `res`. A `res` can be either `NotSub`, indicating that the types are not subtypes, or `IsSub(_,_)` when they are subtypes. If the two types are primitives, then they are only subtypes if they are equal. If the types are tuples, they are subtypes if each of their respective elements are subtypes. Note that the return type of `sub`, when successful, holds the unused choice strings for both type arguments. When encountering a union, `sub` follows 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` are checked, otherwise `sub` checks `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

This implementation represents choice strings as linked lists, but this design requires allocation and reversals when stepping. However, the implementation can be made more efficient by using a mutable bit vector instead of a linked list. Additionally, the maximum length of the bit vector is bounded by the number of unions in the type, so it need only be allocated once. Julia's implementation uses this efficient representation.

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. This provides the general framework for which we prove two iterator-based algorithms correct. The first iterator-based algorithm explicitly includes the structure of the type in its state to guide iteration; the second is identical to that of the prior section.

The order in which choice strings iterate through a type term is determined by both the choice string and the type term being iterated over. Rather than directly working with 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 subtyping algorithm that uses these simpler iterators. Finally, we establish a correspondence between tree iterators and choice string iterators. This concludes our proof of correctness and completeness, and details can be found in the Coq mechanization.

The denotational semantics we use for types is as follows:

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

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 requires proving that the `norm` function returns all union-free type terms.

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

Theorem 3 states that the `subtype` relation of Section **??** abides by Definition 1 because it uses `norm` 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.

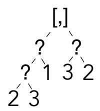
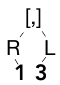
3.2 Subtyping with tree iterators

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 guarantee consistency of iterator state with type structure.

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 can either step to a successor state or 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:*

 , RL $= \texttt{Tuple}\{p_1, p_3\}$  $= \texttt{Tuple}\{p_1, p_3\}$

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 depth-first, right-to-left order. There are four cases to consider:

- A primitive has no successor.
- A tuple steps its second child; if that has no successor step, then it steps its first child and resets the second child.
- An `ILeft` tries to step its child. If it has no successor, then the `ILeft` becomes an `IRight` with a newly initialized child corresponding to the right child of the union.
- An `IRight` also tries to step its child, but is final if its child has no successor.

```
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 a function `inum`, which assigns natural numbers to each state. It simply counts the number of remaining steps in the iterator. To count the total number of union-free types denoted by a type, we use the `tnum` helper function.

```
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 * tnum t2 + inum t2 i2
  | IUnionL t1 t2 i1 => inum t1 i1 + tnum 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* `next` t $it = it'$ *then* `inum` t $it = 1 +$ `inum` t it'.

It is now possible to define an induction principle over `next`. By monotonicity, `next` eventually reaches a final state. For any property of interest, if we prove that it holds for 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 for the final state, and whenever P holds for a successor state it then it holds for its precursor it' where* `next` t $it' = it$. *Then P holds for every iterator state over t.*

Now, we can prove correctness of the subtyping algorithm with tree iterators. We implement subtyping with respect to choice strings in the Coq implementation in a two-stage process. First, we compute the union-free types induced by the iterators over their original types using `here`. Second, we decide subtyping between the two union-free types in `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') => p==p'
  | (Tuple a a', Tuple b b') =>
      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).
```

This version of `sub` differs from the algorithmic implementation to ensure that recursion is well founded. The previous version of `sub` was, in the case of unions, decreasing on alternating arguments when unions were found on either of the sides. In contrast, the proof's version of `sub` applies the choice string to each side first using `here`, a strictly decreasing function that recurs structurally on the given type. This computes the union-free type induced by the iterator applied to the current type. The algorithm then checks subtyping between the resultant union-free types, which is entirely structural. These implementations are equivalent, as they both apply the given choice strings at the same places while computing subtyping; however, the proof version separates choice string application while the implementation intertwines it with the actual subtyping decision.

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 $\{a(i') \,|\, i' \in \mathcal{R}(i)\}$, 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). $|\mathtt{initial}\ a|_a = [\![a]\!]$.

Next, Theorem 5 allows us to show that `exists` of a, b, with i_a and i_b tries 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.** *$\mathtt{exist}\ a\ b\ i_a\ i_b$ 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 `exist` $a\ b\ i_a$ (`initial` b) finds 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.** *$\texttt{allexist}\ a\ b\ i_a$ 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 correctness of the algorithm.

▶ **Theorem 9.** *$\texttt{subtype}\ a\ b$ 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 correct and complete. We start by showing that iterators over choice strings simulate tree iterators. This lets us prove that the choice string based subtyping algorithm is correct by showing that the iterators at each step are equivalent. To relate tree iterators to choice string iterators, we use the `itp` function, which traverses a tree iterator state and linearizes it, producing a choice string using 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, we define an induction principle over choice strings by way of linearized tree iterators. The `next` function in Section 2.2 works by finding the last `L` in the choice string, turning it into an `R`, and replacing the rest with `L`s until the type is 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 $\texttt{next}(\texttt{itp}\ it) = (\texttt{itp}\ it')$.*

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 correctness proofs 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 by `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. 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.** *`allexist` a b (`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.** *`allexist` a b (`pad` t `[]`) 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.** *`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 string based implementation of iterators.

4 Complexity

The worst-case time complexity of Julia's subtyping algorithm and normalization-based approaches is determined by the number of terms that could 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 those 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 it 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 can be done sequentially without needing explicit enumeration.

5 Future work

We plan to handle additional features of Julia. Our next steps will be subtyping for primitive types, existential type variables, and invariant constructors. Adding 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 $\texttt{Union\{}p_2, p_3\texttt{\}}$? 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 when combined with existential variables may create cycles within the subtyping relation.

6 Conclusion

It is likely that subtyping with unions and tuples is always going to be exponential time, as subtyping of regular expression types have been proven to be EXPTIME-complete [11]. However, it need not take exponential space to decide subtyping: we have described and proven correct a subtyping algorithm for covariant tuples and unions that uses iterators instead of normalization. This algorithm uses linear space and allows common patterns, such as testing if a tuple of primitives is a subtype of a tuple of unions, to be handled as a special case of the subtyping algorithm. Finally, based on Julia's experience with the algorithm, we think that it can generalize to rich type languages; Julia supports bounded polymorphism and invariant constructors enabled in part by its use of this algorithm.

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. We would also like to thank Ming-Ho Yee, Celeste Hollenbeck, and Julia Belyakova for their help in preparing this paper. 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 FPCA*, 1991. `doi:10.1007/3540543961_21`.
4. Franco Barbanera and Mariangiola Dezani-Ciancaglini. Intersection and union types. In *Theoretical Aspects of Computer Software TACS*, 1991. `doi:10.1007/3-540-54415-1_69`.
5. Julia Belyakova. Decidable tag-based semantic subtyping for nominal types, tuples, and unions. In *Proceedings of the 21st Workshop on Formal Techniques for Java-like Programs FTFJP*, 2019.
6. Jeff Bezanson. *Abstraction in technical computing*. PhD thesis, Massachusetts Institute of Technology, 2015. URL: `http://dspace.mit.edu/handle/1721.1/7582`.
7. 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`.
8. Flemming M. Damm. Subtyping with union types, intersection types and recursive types. In *Theoretical Aspects of Computer Software TACS*, 1994. `doi:10.1007/3-540-57887-0_121`.
9. Joshua Dunfield. Elaborating intersection and union types. *J. Funct. Program.*, 2014. `doi:10.1017/S0956796813000270`.
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. Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for xml. *ACM Trans. Program. Lang. Syst.*, 2005.
12. Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated subtyping. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018.

13 David J. Pearce. Sound and complete flow typing with unions, intersections and negations. In *Verification, Model Checking, and Abstract Interpretation VMCAI*, 2013. doi:10.1007/978-3-642-35873-9_21.

14 Benjamin Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.

15 Ross Tate. personal communication.

16 Jerome Vouillon. Subtyping union types. In *Computer Science Logic (CSL)*, 2004. URL: https://www.cis.upenn.edu/~bcpierce/papers/uipq.ps, doi:10.1007/978-3-540-30124-0_32.

17 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.