



HAL
open science

Julia's efficient algorithm for subtyping unions and covariant tuples (Pearl)

Benjamin Chung, Francesco Zappa Nardelli, Jan Vitek

► To cite this version:

Benjamin Chung, Francesco Zappa Nardelli, Jan Vitek. Julia's efficient algorithm for subtyping unions and covariant tuples (Pearl). ECOOP 2019 - 33rd European Conference of Object-Oriented Programming, Jul 2019, London, United Kingdom. 10.4230/LIPIcs.ECOOP.2019.6 . hal-02297696

HAL Id: hal-02297696

<https://inria.hal.science/hal-02297696v1>

Submitted on 26 Sep 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

1 Julia’s efficient algorithm for subtyping unions and 2 covariant tuples (Pearl)

3 Benjamin Chung

4 Northeastern University

5 Francesco Zappa Nardelli

6 Inria

7 Jan Vitek

8 Northeastern University & Czech Technical University in Prague

9 — Abstract —

10 The Julia programming language supports multiple dispatch and provides a rich type annotation
11 language to specify method applicability. When multiple methods are applicable for a given call,
12 Julia relies on subtyping between method signatures to pick the correct method to invoke. Julia’s
13 subtyping algorithm is surprisingly complex, and determining whether it is correct remains an open
14 question. In this paper, we focus on one piece of this problem: the interaction between union
15 types and covariant tuples. Previous work normalized unions inside tuples to disjunctive normal
16 form. However, this strategy has two drawbacks: complex type signatures induce space explosion,
17 and interference between normalization and other features of Julia’s type system. In this paper,
18 we describe the algorithm that Julia uses to compute subtyping between tuples and unions—an
19 algorithm that is immune to space explosion and plays well with other features of the language. We
20 prove this algorithm correct and complete against a semantic-subtyping denotational model in Coq.

21 **2012 ACM Subject Classification** Theory of computation → Type theory

22 **Keywords and phrases** Type systems, Subtyping, Union types

23 **Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2019.6

24 **Category** Pearl

25 **1** Introduction

26 Union types, originally introduced by Barbanera and Dezani-Ciancaglini [2], are being
27 adopted in mainstream languages. In some cases, such as Julia [5] or TypeScript [11], they
28 are exposed at the source level. In others, such as Hack [8], they are only used internally as
29 part of type inference. As a result, subtyping algorithms between union types are of increasing
30 practical import. The standard subtyping algorithm for this combination of features has, for
31 some time, been exponential in both time and space. An alternative algorithm, linear in space
32 but still exponential in time, has been tribal knowledge in the subtyping community [15]. In
33 this paper, we describe and prove correct an implementation of that algorithm.

34 We observed the algorithm in our prior work formalizing the Julia subtyping relation [17].
35 There, we described Julia’s subtyping relation as it arose from its decision procedure but were
36 unable to prove it correct. Indeed, we found bugs in the Julia implementation and identified
37 unresolved correctness issues. Contemporary work addresses some correctness concerns [3]
38 but leaves algorithmic correctness open.

39 Julia’s subtyping algorithm [4] is used for method dispatch. While Julia is dynamically
40 typed, method arguments can have type annotations. These annotations allow one method
41 to be implemented by multiple functions. At run time, Julia searches for the most specific
42 applicable function for a given invocation. Consider these declarations of multiplication:

43



© Benjamin Chung, Francesco Zappa Nardelli, Jan Vitek;
licensed under Creative Commons License CC-BY

33rd European Conference on Object-Oriented Programming (ECOOP 2019).

Editor: Alastair F. Donaldson; Article No. 6; pp. 6:1–6:15

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

6:2 Subtyping union types and covariant tuples

```

44
45 * (x::Number, r::Range) = range(x*first(r), ...)
46 * (x::Number, y::Number) = *(promote(x, y)...)
47 * (x::T, y::T) where T <: Union{Signed, Unsigned} = mul_int(x, y)
48

```

49 The first two methods implement, respectively, multiplication of a range by a number and
50 generic numeric multiplication. The third method invokes native multiplication when both
51 arguments are either signed or unsigned integers (but not a mix of the two). Julia uses
52 subtyping to decide which of the methods to call at any specific site. The call `1*(1:4)`
53 dispatches to the first, `1*1.1` the second, and `1*1` the third.

54 Julia offers programmers a rich type language to express complex relationships in type
55 signatures. The type language includes nominal primitive types, union types, existential
56 types, covariant tuples, invariant parametric datatypes, and singletons. Intuitively, subtyping
57 between types is based on semantic subtyping: the subtyping relation between types holds
58 when the sets of values they denote are subsets of one another [5]. We write the set of values
59 represented by a type t as $\llbracket t \rrbracket$. Under semantic subtyping, the types t_1 and t_2 are subtypes
60 iff $\llbracket t_1 \rrbracket \subseteq \llbracket t_2 \rrbracket$. From this, we derive a *forall-exists* intuition for subtyping: for every value
61 denoted on the left-hand side, there must exist some value on the right-hand side to match
62 it, thereby establishing the subset relation. This simple intuition is, however, complicated to
63 check algorithmically.

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

$$\begin{array}{c}
\text{ALLEXIST} \qquad \text{EXISTL} \qquad \text{EXISTR} \qquad \text{TUPLE} \\
\frac{ft' <: t \quad t'' <: t}{\text{Union}\{t', t''\} <: t} \quad \frac{t <: t'}{t <: \text{Union}\{t', t''\}} \quad \frac{t <: t''}{t <: \text{Union}\{t', t''\}} \quad \frac{t_1 <: t'_1 \quad t_2 <: t'_2}{\text{Tuple}\{t_1, t_2\} <: \text{Tuple}\{t'_1, t'_2\}}
\end{array}$$

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

$$76 \text{Tuple}\{\text{Union}\{t', t''\}, t\} <: \text{Union}\{\text{Tuple}\{t', t\}, \text{Tuple}\{t'', t\}\}$$

77 Using semantic subtyping, the judgment should hold. The set of values denoted by a
78 union $\llbracket \text{Union}\{t_1, t_2\} \rrbracket$ is just the union of the set of values denoted by each of its members
79 $\llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$. A tuple $\text{Tuple}\{t_1, t_2\}$ ’s denotation is the set of tuples of the respective values
80 $\{\text{Tuple}\{v_1, v_2\} \mid v_1 \in \llbracket t_1 \rrbracket \wedge v_2 \in \llbracket t_2 \rrbracket\}$. Therefore, the left-hand side denotes the values
81 $\{\text{Tuple}\{v', v''\} \mid v' \in \llbracket t' \rrbracket \cup \llbracket t'' \rrbracket \wedge v'' \in \llbracket t \rrbracket\}$, while the right-hand side denotes $\llbracket \text{Tuple}\{t', t\} \rrbracket \cup$
82 $\llbracket \text{Tuple}\{t'', t\} \rrbracket$ or equivalently $\{\text{Tuple}\{v', v''\} \mid v' \in \llbracket t' \rrbracket \cup \llbracket t'' \rrbracket \wedge v'' \in \llbracket t \rrbracket\}$. These sets are the
83 same, and therefore subtyping should hold in either direction between the left- and right-hand
84 types. However, we cannot derive this relation from the above rules. According to them, we
85 must pick either t' or t'' on the right-hand side using EXISTL or EXISTR, respectively, ending
86 up with either $\text{Tuple}\{\text{Union}\{t', t''\}, t\} <: \text{Tuple}\{t', t\}$ or $\text{Tuple}\{\text{Union}\{t', t''\}, t\} <: \text{Tuple}\{t'', t\}$.
87 In either case, the judgment does not hold. How can this problem be solved?

88 Most prior work addresses this problem by normalization [2, 14, 1], rewriting all types into
89 their disjunctive normal form, as unions of union-free types, *before* building the derivation.
90
91

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 [7], do not handle distributivity. Normalization is used by Frisch et al.'s [9], 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 [6].

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 `Array{Union{t', t''}}`, the set of arrays whose elements are either t' or t'' . It is wrong to rewrite it as `Union{Array{t'}, Array{t''}}`, 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:

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

It holds if the existential variable T is instantiated with `Union{t, t'}`. 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 `Union{Tuple{t}, Tuple{t'}} <: Tuple{T}` and `Tuple{T} <: Union{Tuple{t}, Tuple{t'}}` hold. The first of these can be concluded without issue, producing the constraint `Union{t, t'} <: T`. However, this constraint on T is retained for checking the reverse direction, which is where problems arise. When checking the reverse direction, the algorithm has to prove that `Tuple{T} <: Union{Tuple{t}, Tuple{t'}}`, and in turn either $T <: t$ or $T <: t'$. All of these are unprovable under the assumption that `Union{t, t'} <: T`. The key to deriving a successful judgment for this relation is to rewrite the right-to-left check into `Tuple{T} <: Tuple{Union{t, t'}}`, 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

6:4 Subtyping union types and covariant tuples

140 enough to highlight the decision strategy and illustrate the structure of the algorithm.
141 Empirical evidence from Julia’s implementation suggests that this technique extends to
142 invariant constructors and existential types [17], among others. We expect that the algorithm
143 we describe can be leveraged in other modern language designs.

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

2 A space-efficient subtyping algorithm

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

```
148  
149  
150 type typ = Prim of int | Tuple of typ * typ | Union of typ * typ  
151
```

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

2.1 Normalization

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

158 $\text{Union}\{\text{Tuple}\{p_1, p_2\}, \text{Tuple}\{p_2, p_3\}\} <: \text{Tuple}\{\text{Union}\{p_2, p_1\}, \text{Union}\{p_3, p_2\}\}.$

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

160 $\text{Union}\{\text{Tuple}\{p_2, p_3\}, \text{Union}\{\text{Tuple}\{p_2, p_2\}, \text{Union}\{\text{Tuple}\{p_1, p_3\}, \text{Tuple}\{p_1, p_2\}\}\}$

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

162 $\ell_1 = \{\text{Tuple}\{p_1, p_2\}, \text{Tuple}\{p_2, p_3\}\}$

163 and

164 $\ell_2 = \{\text{Tuple}\{p_2, p_3\}, \text{Tuple}\{p_2, p_2\}, \text{Tuple}\{p_1, p_3\}, \text{Tuple}\{p_1, p_2\}\}.$

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

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

```
175  
176  
177 let subtype(a:typ)(b:typ) = allexist (norm a) (norm b)  
178  
179 let allexist(a:list typ)(b:list typ) =  
180   foldl (fun acc a' => acc && exist a' b) true a  
181
```

```

182 let exist(a:typ)(b:list typ) =
183     foldl (fun acc b' => acc || a==b') false b
184
185 let rec norm = function
186     | Prim i -> [Prim i]
187     | Tuple t t' ->
188         map_pair Tuple (cartesian_product (norm t) (norm t'))
189     | Union t t' -> (norm t) @ (norm t')

```

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

194 2.2 Iteration with choice strings

195 Given a type term such as the following,

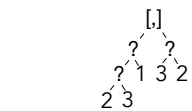
196 `Tuple{Union{Union{p2, p3}, p1}, Union{p3, p2}}`

197 we want an algorithm that checks the following tuples,

198 `Tuple{p2, p3}, Tuple{p2, p2}, Tuple{p1, p3}, Tuple{p1, p2}, Tuple{p3, p3}, Tuple{p3, p2}`

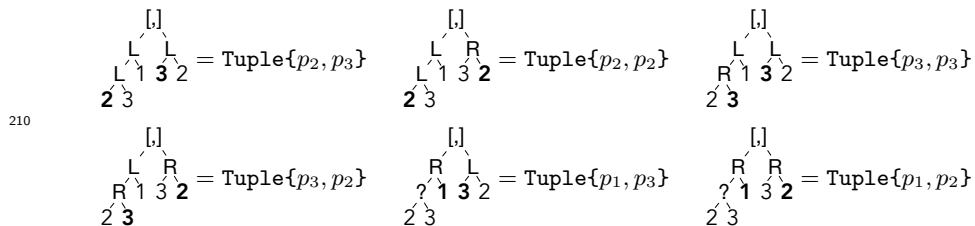
199 without having to compute and store all of them ahead-of-time. This algorithm should be
 200 able to generate each tuple on-demand while still being guaranteed to explore every tuple of
 201 the original type's normal form.

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



207

208 At each choice point we can go either left or right; making such a decision at each point
 209 leads to visiting one particular tuple.



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

215 The iteration sequence in the above example is thus $\underline{\text{LLL}} \rightarrow \underline{\text{LLR}} \rightarrow \underline{\text{LRL}} \rightarrow \underline{\text{LRR}} \rightarrow \underline{\text{RL}}$
 216 $\rightarrow \underline{\text{RR}}$, where the underlined choice is next one to be toggled in that step. Stepping from
 217 a choice string c to the next string consists of splitting c in three, $c' L c''$, where c' can be
 218 empty and c'' is a possibly empty sequence of Rs. The next string is $c' R c_{pad}$, that is to say

6:6 Subtyping union types and covariant tuples

219 it retains the prefix c' , toggles the L to an R, and is padded by a sequence of Ls. The leftover
220 tail c'' is discarded. If there is no L in c , iteration terminates.

221 One step of iteration is performed by calling the `next` function with a type term and a
222 choice string (encoded as a list of choices); `next` either returns the next string in the sequence
223 or `None`. Internally, it calls `step` to toggle the last L and shorten the string (constructing $c'R$).
224 Then it calls on `pad` to add the trailing sequence of Ls (constructing $c'Rc_{pad}$).

```
225  
226  
227 type choice = L | R  
228  
229 let rec next(a:typ)(l:choice list) =  
230     match step l with  
231     | None -> None  
232     | Some(l') -> Some(fst (pad a l'))  
233
```

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

```
237  
238  
239 let step(l:choice list) =  
240     match rev (toggle (rev l)) with  
241     | [] -> None  
242     | hd::tl -> Some(hd::tl)  
243  
244 let rec toggle = function  
245     | [] -> []  
246     | L::tl -> R::tl  
247     | R::tl -> toggle tl  
248
```

249 The `pad` function takes a type term and a choice string to be padded. It returns a pair, whose
250 first element is the padded string and second element is the string left over from the current
251 type. Each union encountered by `pad` in its traversal of the type consumes a character from
252 the input string. Unions explored after the exhaustion of the original choice string are treated
253 as if there was an L remaining in the choice string. The first component of the returned value
254 is the original choice string extended with an L for every union encountered after exhaustion
255 of the original.

```
256  
257  
258 let rec pad t l =  
259     match t,l with  
260     | (Prim i,l) -> ([],l)  
261     | (Tuple(t,t'),l) ->  
262         let (h,tl) = pad t l in  
263         let (h',tl') = pad t' tl in (h @ h',tl')  
264     | (Union(t,_),L::r) ->  
265         let (h,tl) = pad t r in (L::h,tl)  
266     | (Union(_,t),R::r) ->  
267         let (h,tl) = pad t r in (R::h,tl)  
268     | (Union(t,_),[]) -> (L::(fst (pad t [])), [])  
269
```

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

```

273
274
275 let initial(t:typ) = fst (pad t [])
276

```

2.3 Subtyping with iteration

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

```

282
283
284 let subtype(a:typ)(b:typ) = allexist a b (initial a)
285

```

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

```

290
291
292 let rec allexist(a:typ)(b:typ)(f:choice list) =
293   match exist a b f (initial b) with
294   | true  -> (match next a f with
295               | Some ns -> allexist a b ns
296               | None  -> true)
297   | false -> false
298

```

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

```

303
304
305 type res = NotSub | IsSub of choice list * choice list
306
307 let rec exist(a:typ)(b:typ)(f:choice list)(e:choice list) =
308   match sub a b f e with
309   | IsSub(_,_) -> true
310   | NotSub ->
311     (match next b e with
312      | Some ns -> exist a b f ns
313      | None  -> false)
314

```

315 Finally, the `sub` function takes two type terms and choice strings and returns a value of type
316 `res`. A `res` can be either `NotSub`, indicating that the types are not subtypes, or `IsSub(_,_)`
317 when they are subtypes. If the two types are primitives, then they are only subtypes if they
318 are equal. If the types are tuples, they are subtypes if each of their respective elements
319 are subtypes. Note that the return type of `sub`, when successful, holds the unused choice
320 strings for both type arguments. When encountering a union, `sub` follows the choice strings
321 to decide which branch to take. Consider, for instance, the case when the first type term is
322 `Union(t1,t2)` and the second is type `t`. If the first element of the choice string is an `L`, then
323 `t1` and `t` are checked, otherwise `sub` checks `t2` and `t`.

324

6:8 Subtyping union types and covariant tuples

```
325 let rec sub t1 t2 f e =
326   match t1,t2,f,e with
327   | (Prim i,Prim j,f,e) -> if i==j then IsSub(f,e) else NotSub
328   | (Tuple(a1,a2), Tuple(b1,b2),f,e) ->
329     (match sub a1 b1 f e with
330      | IsSub(f', e') -> sub a2 b2 f' e'
331      | NotSub -> NotSub)
332   | (Union(a,_),b,L::f,e) -> sub a b f e
333   | (Union(_,a),b,R::f,e) -> sub a b f e
334   | (a,Union(b,_),f,L::e) -> sub a b f e
335   | (a,Union(_,b),f,R::e) -> sub a b f e
336
```

2.4 Further optimization

338 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

344 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.

345 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.

346 The denotational semantics we use for types is as follows:

$$\begin{aligned} 359 \quad \llbracket p_i \rrbracket &= \{p_i\} \\ 360 \quad \llbracket \text{Union}\{t_1, t_2\} \rrbracket &= \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket \\ 361 \quad \llbracket \text{Tuple}\{t_1, t_2\} \rrbracket &= \{\text{Tuple}\{t'_1, t'_2\} \mid t'_1 \in \llbracket t_1 \rrbracket, t'_2 \in \llbracket t_2 \rrbracket\} \end{aligned}$$

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

363 ► **Definition 1.** *The subtyping relation $t_1 <: t_2$ holds iff $\forall t'_1 \in \llbracket t_1 \rrbracket, \exists t'_2 \in \llbracket t_2 \rrbracket, t'_1 = t'_2$.*

364 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). $t' \in \llbracket t \rrbracket$ iff $t' \in \text{norm } t$.

Theorem 3 states that the `subtype` relation of Section 2.1 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 $a <: b$.

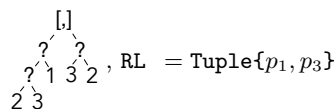
Therefore, normalization-based subtyping is correct against our definition.

3.2 Subtyping with tree iterators

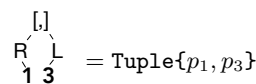
Reasoning about iterators that use choice strings, as described in Section 2.2, is tricky as it requires simultaneously reasoning about the structure of the type term and the validity of the choice string that represents the iterator's state. Instead, we propose to use an intermediate data structure,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



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.

6:10 Subtyping union types and covariant tuples

409 ■ An `IRight` also tries to step its child, but is final if its child has no successor.

410

411

```
412 Fixpoint next(t:Typ)(i:iter t): option(iter t) := match i with
413 | IPrim _ => None
414 | ITuple t1 t2 i1 i2 =>
415   match (next t2 i2) with
416   | Some i' => Some(ITuple t1 t2 i1 i')
417   | None =>
418     match (next t1 i1) with
419     | Some i' => Some(ITuple t1 t2 i' (start t2))
420     | None => None
421     end
422   end
423 | ILeft t1 t2 i1 =>
424   match (next t1 i1) with
425   | Some(i') => Some(ILeft t1 t2 i')
426   | None => Some(IRight t1 t2 (start t2))
427   end
428 | IRight t1 t2 i2 =>
429   match (next t2 i2) with
430   | Some(i') => Some(IRight t1 t2 i')
431   | None => None
432   end
433 end.
434
```

435 An induction principle for tree iterators is needed to reason about all iterator states for a
436 given type. First, we show that iterators eventually reach a final state. This is done with a
437 function `inum`, which assigns natural numbers to each state. It simply counts the number of
438 remaining steps in the iterator. To count the total number of union-free types denoted by a
439 type, we use the `tnum` helper function.

440

441

```
442 Fixpoint tnum(t:Typ):nat :=
443   match t with
444   | Prim i => 1
445   | Tuple t1 t2 => tnum t1 * tnum t2
446   | Union t1 t2 => tnum t1 + tnum t2
447   end.
448
449 Fixpoint inum(t:Typ)(ti:iter t):nat :=
450   match ti with
451   | IPrim i => 0
452   | ITuple t1 t2 i1 i2 => inum t1 i1 * tnum t2 + inum t2 i2
453   | IUnionL t1 t2 i1 => inum t1 i1 + tnum t2
454   | IUnionR t1 t2 i2 => inum t2 i2
455   end.
456
```

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

460 ► **Lemma 4** (Monotonicity). *If $next\ t\ it = it'$ then $inum\ t\ it = 1 + inum\ t\ it'$.*

461 It is now possible to define an induction principle over `next`. By monotonicity, `next` eventually
 462 reaches a final state. For any property of interest, if we prove that it holds for the final state
 463 and for the induction step, we can prove it holds for every state for that type.

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

467 Now, we can prove correctness of the subtyping algorithm with tree iterators. We implement
 468 subtyping with respect to choice strings in the Coq implementation in a two-stage process.
 469 First, we compute the union-free types induced by the iterators over their original types
 470 using `here`. Second, we decide subtyping between the two union-free types in `ufsub`. The
 471 function `here` walks the given iterator, producing a union-free type mirroring its state. To
 472 decide subtyping between the resulting union-free types, `ufsub` checks equality between `Prim`
 473 `s` and recurses on the elements of `Tuples`, while returning false for all other types. Since
 474 `here` will never produce a union type, the case of `ufsub` for them is irrelevant, and is false by
 475 default.

```

476 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 p1 => (here t1 p1)
  | IRight t1 t2 pr => (here t2 pr)
  end.
  
```

```

476 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.
  
```

```

477
478 Definition sub (a b:Typ) (ai:iter a) (bi:iter b) :=
479   ufsub (here a ai) (here b bi).
480
481
  
```

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

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

```

498
499 Definition subtype (a b:Typ) = allexist a b (initial a)
500
501
502 Program Fixpoint allexist (a b:typ) (ia:iter a) {measure (inum ia)} =
  
```

6:12 Subtyping union types and covariant tuples

```

503   exists a b ia (initial b) &&
504     (match next a ia with
505       | Some(ia') => allexist a b ia'
506       | None => true).
507
508 Program Fixpoint exist(a b:typ)(ia:iter a)(ib:iter b)
509                               {measure(inum ib)} =
510   subtype a b ia ib ||
511     (match next b ib with
512       | Some(ib') => exist a b ia ib'
513       | None => false).
514

```

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

520 ► **Lemma 6** (Initial equivalence). $|initial\ a|_a = \llbracket a \rrbracket$.

521 Next, Theorem 5 allows us to show that `exists` of a, b , with i_a and i_b tries to find an iterator
522 state i'_b starting from i_b such that $b(i'_b) = a(i_a)$. The desired property trivially holds when
523 $|i_b|_b = \emptyset$, and if the iterator can step then either the current union-free type is satisfying or
524 we defer to the induction hypothesis.

525 ► **Theorem 7.** *exist a b i_a i_b holds iff $\exists t \in |i_b|_b, a(i_a) = t$.*

526 We can then appeal to both Theorem 7 and Lemma 6 to show that `exist a b i_a (initial b)`
527 finds a satisfying union-free type on the right-hand side if it exists in $\llbracket b \rrbracket$. Using this, we can
528 then use Theorem 5 in an analogous way to `exist` to show that `allexist` is correct up to the
529 current iterator state.

530 ► **Theorem 8.** *allexist a b i_a holds iff $\forall a' \in |i_a|_a, \exists b' \in \llbracket b \rrbracket, a' = b'$.*

531 Finally, we can appeal to Theorem 8 and Lemma 6 again to show correctness of the algorithm.

532 ► **Theorem 9.** *subtype a b holds iff $\forall a' \in \llbracket a \rrbracket, \exists b' \in \llbracket b \rrbracket, a' = b'$.*

533 3.3 Subtyping with choice strings

534 We prove the subtyping algorithm using choice strings correct and complete. We start by
535 showing that iterators over choice strings simulate tree iterators. This lets us prove that
536 the choice string based subtyping algorithm is correct by showing that the iterators at
537 each step are equivalent. To relate tree iterators to choice string iterators, we use the `itp`
538 function, which traverses a tree iterator state and linearizes it, producing a choice string
539 using depth-first search.

```

540
541
542 Fixpoint itp{t:Typ}(it:iter t):choice list :=
543   match it with
544   | IPrim _ => nil
545   | ITuple t1 t2 it1 it2 => (itp t1 it1)++(itp t2 it2)
546   | ILeft t1 _ it1 => Left::(itp t1 it1)
547   | IRight _ t2 it1 => Right::(itp t2 it1)
548   end.

```

550 Next, we define an induction principle over choice strings by way of linearized tree iterators.
 551 The `next` function in Section 2.2 works by finding the last L in the choice string, turning it
 552 into an R, and replacing the rest with Ls until the type is valid. If we use `itp` to translate
 553 both the initial and final states for a valid `next` step of a tree iterator, we see the same
 554 structure.

555 ► **Lemma 10** (Linearized Iteration). *For some type t and tree iterators $it\ it'$, if $next\ t\ it = it'$,
 556 there exists some prefix c' , an initial suffix c'' made up of Rs, and a final suffix c''' consisting
 557 of Ls such that $itp\ t\ it = c'\ Left\ c''$ and $itp\ t\ it' = c'\ Right\ c'''$.*

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

560 ► **Lemma 11** (Step Equivalence). *If it and it' are tree iterator states and $next\ it = it'$, then
 561 $next(itp\ it) = (itp\ it')$.*

562 The initial state of a tree iterator linearizes to the initial state of a choice string iterator.

563 ► **Lemma 12** (Initial Equivalence). $itp(initial\ t) = pad\ t\ []$.

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

571 ► **Lemma 13**. $exist\ a\ b\ (itp\ i_a)\ (itp\ i_b)$ holds iff $\exists t \in |i_b|_b, a(i_a) = t$.

572 With the correctness of `exist` following from the tree iterator definition, we can apply the
 573 same proof methodology to show that `allexist` is correct. In order to do so, we instantiate
 574 Lemma 13 with Lemma 6 and Lemma 12 to show that if $exist\ a\ b\ (itp\ i_a)\ (pad\ t\ [])$ then
 575 $\exists t \in \llbracket b \rrbracket, a(i_a) = t$, allowing us to check each of the exists cases while establishing the
 576 forall-exists relationship.

577 ► **Lemma 14**. $allexist\ a\ b\ (itp\ i_a)$ holds iff $\forall a' \in |i_a|_a, \exists b' \in \llbracket b \rrbracket, a' = b'$.

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

580 ► **Theorem 15**. $allexist\ a\ b\ (pad\ t\ [])$ holds iff $\forall a' \in \llbracket a \rrbracket, \exists b' \in \llbracket b \rrbracket, a' = b'$.

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

582 ► **Theorem 16**. $subtype\ a\ b$ holds iff $\forall a' \in \llbracket a \rrbracket, \exists b' \in \llbracket b \rrbracket, a' = b'$.

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

585 **4** Complexity

586 The worst-case time complexity of Julia's subtyping algorithm and normalization-based
 587 approaches is determined by the number of terms that could exist in the normalized type. In

6:14 Subtyping union types and covariant tuples

588 the worst case, there are 2^n union-free tuples in the fully normalized version of a type that
589 has n unions. Each of those tuples must always be explored. As a result, both algorithms
590 have worst-case $O(2^n)$ time complexity. The approaches differ, however, in space complexity.
591 The normalization approach computes and stores each of the exponentially many alternatives,
592 so it also has $O(2^n)$ space complexity. However, Julia need only store the choice made at
593 each union, thereby offering $O(n)$ space complexity.

594 Julia’s algorithm improves best-case time performance. Normalization always experiences
595 worst-case time and space behavior as it has to precompute the entire normalized type.
596 Julia’s iteration-based algorithm can discover the relation between types early. In practice,
597 many queries are of the form $uft <: \text{union}(t_1 \dots t_n)$, where uft is an already union-free tuple.
598 As a result, all that Julia needs to do is find one matching tuple in $t_1 \dots t_n$, which can be done
599 sequentially without needing explicit enumeration.

600 **5** Future work

601 We plan to handle additional features of Julia. Our next steps will be subtyping for primitive
602 types, existential type variables, and invariant constructors. Adding subtyping to primitive
603 types would be the simplest change. The challenge is how to retain completeness, as a
604 primitive subtype heirarchy and semantic subtyping have undesirable interactions. For
605 example, if the primitive subtype hierarchy contains only the relations $p_2 <: p_1$ and $p_3 <: p_1$,
606 then is p_1 a subtype of $\text{Union}\{p_2, p_3\}$? In a semantic subtyping system, they are, but this
607 requires changes both to the denotational framework and the search space of the iterators.
608 Existential type variables create substantial new complexities in the state of the algorithm.
609 No longer is the state solely restricted to that of the iterators being attempted; now, the
610 state includes variable bounds that are accumulated as the algorithm compares types to
611 type variables. As a result, correctness becomes a much more complex contextually linked
612 property to prove. Finally, invariant type constructors induce contravariant subtyping, which
613 when combined with existential variables may create cycles within the subtyping relation.

614 **6** Conclusion

615 It is likely that subtyping with unions and tuples is always going to be exponential time,
616 as subtyping of regular expression types have been proven to be EXPTIME-complete [10].
617 However, it need not take exponential space to decide subtyping: we have described and
618 proven correct a subtyping algorithm for covariant tuples and unions that uses iterators
619 instead of normalization. This algorithm uses linear space and allows common patterns, such
620 as testing if a tuple of primitives is a subtype of a tuple of unions, to be handled as a special
621 case of the subtyping algorithm. Finally, based on Julia’s experience with the algorithm, we
622 think that it can generalize to rich type languages; Julia supports bounded polymorphism
623 and invariant constructors enabled in part by its use of this algorithm.

624 **Acknowledgments**

625 The authors thank Jiahao Chen for starting us down the path of understanding Julia, and
626 Jeff Bezanson for coming up with Julia’s subtyping algorithm. We would also like to thank
627 Ming-Ho Yee, Celeste Hollenbeck, and Julia Belyakova for their help in preparing this paper.
628 This work received funding from the European Research Council under the European Union’s
629 Horizon 2020 research and innovation programme (grant agreement 695412), the NSF (award

630 1544542 and award 1518844), the ONR (grant 503353), and the Czech Ministry of Education,
631 Youth and Sports (grant agreement CZ.02.1.01/0.0/0.0/15_003/0000421).

632 ——— **References** ———

- 633 1 Alexander Aiken and Brian R. Murphy. Implementing regular tree expressions. In *Functional*
634 *Programming Languages and Computer Architecture FPCA*, 1991. doi:10.1007/3540543961_
635 21.
- 636 2 Franco Barbanera and Mariangiola Dezani-Ciancaglini. Intersection and union types. In
637 *Theoretical Aspects of Computer Software TACS*, 1991. doi:10.1007/3-540-54415-1_69.
- 638 3 Julia Belyakova. Decidable tag-based semantic subtyping for nominal types, tuples, and unions.
639 In *Proceedings of the 21st Workshop on Formal Techniques for Java-like Programs FTFJP*,
640 2019.
- 641 4 Jeff Bezanson. *Abstraction in technical computing*. PhD thesis, Massachusetts Institute of
642 Technology, 2015. URL: <http://dspace.mit.edu/handle/1721.1/7582>.
- 643 5 Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach
644 to numerical computing. *SIAM Review*, 59(1), 2017. doi:10.1137/141000671.
- 645 6 Flemming M. Damm. Subtyping with union types, intersection types and recursive types. In
646 *Theoretical Aspects of Computer Software TACS*, 1994. doi:10.1007/3-540-57887-0_121.
- 647 7 Joshua Dunfield. Elaborating intersection and union types. *J. Funct. Program.*, 2014. doi:
648 10.1017/S0956796813000270.
- 649 8 Facebook. Hack. <https://hacklang.org/>.
- 650 9 Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing
651 set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4), 2008.
652 doi:10.1145/1391289.1391293.
- 653 10 Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for xml.
654 *ACM Trans. Program. Lang. Syst.*, 2005.
- 655 11 Microsoft. Typescript language specification. URL: [https://github.com/Microsoft/](https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md)
656 [TypeScript/blob/master/doc/spec.md](https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md).
- 657 12 Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated
658 subtyping. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018.
- 659 13 David J. Pearce. Sound and complete flow typing with unions, intersections and negations.
660 In *Verification, Model Checking, and Abstract Interpretation VMCAI*, 2013. doi:10.1007/
661 978-3-642-35873-9_21.
- 662 14 Benjamin Pierce. Programming with intersection types, union types, and polymorphism.
663 Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.
- 664 15 Ross Tate. personal communication.
- 665 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.
- 666 17 Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson,
667 and Jan Vitek. Julia subtyping: a rational reconstruction. *Proc. ACM Program. Lang.*,
668 2(OOPSLA), 2018. doi:10.1145/3276483.
- 669