On Julia's efficient algorithm for subtyping union types and covariant tuples (Pearl)

Anonymous Authors
Anonymous Affiliation

Abstract

We describe the algorithm implemented in the Julia programming language runtime to decide subtyping on a simple type system with union types, covariant tuples, and literals. This algorithm is immune from the space-explosion and expressiveness problems of standard algorithms that normalise types into disjunctive normal form ahead-of-time. 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

1 Subtyping union types and covariant tuples

Union types are increasingly common in mainstream languages. In some cases, as Julia [2] or TypeScript [4], they are exposed at the source level. In others, such as Hack [1], they are only used internally when performing type inference. In all cases, they play a key role in the expressiveness of the language.

Rules for subtyping union types have been known for a long time. Following Vouillon [6] they may be written as:

$$\frac{t_1 <: t \qquad t_2 <: t}{\texttt{Union\{}t_1, t_2\texttt{\}} <: t} \qquad\qquad \frac{t <: t_1}{t <: \texttt{Union\{}t_1, t_2\texttt{\}}} \qquad \frac{t <: t_2}{t <: \texttt{Union\{}t_1, t_2\texttt{\}}}$$

These rules are asymmetrical. Following from set-theoretic intuitions, and as made formal by semantic subtyping models, when a union type appears on the left-hand side of a subtype judgment, then *all* its components must be subtypes of the right-hand side; whan a union type appears on the right-hand side of a subtype judgment, then there must *exist* a component that is a supertype of the left-hand side.

It has also been known for a long time that, in the presence of covariant tuples, the above rules are not complete with respect to a semantic subtyping model [5]. Let us recall the subtype rule for covariant tuples:

$$\frac{t_1 <: t_1' \qquad t_2 <: t_2'}{\texttt{Tuple\{}t_1, t_2\texttt{\}} <: \texttt{Tuple\{}t_1', t_2'\texttt{\}}}$$

In a semantic subtyping model, covariant types should be *distributive* with respect to unions; that is, the following (and the opposite) derivation should hold:

$$\texttt{Tuple\{Union\{}t_1, t_2\texttt{\}}, t\texttt{\}} \quad <: \quad \texttt{Union\{Tuple\{}t_1, t\texttt{\}}, \texttt{Tuple\{}t_2, t\texttt{\}\}}$$

Here the rule for tuples cannot be applied, and a derivation must immediately pick either the first or second component of the union type; as a result, it is impossible to complete the derivation. The standard approach to solve this conundrum is to *rewrite all types into their disjunctive normal form* (DNF), that is as unions of union-free types, *before* building the derivation. The correctness of this rewriting step is justified by the semantic-subtyping denotational model, as in [3], and the resulting subtype algorithm can be proved both correct and complete. However, this standard algorithm based on ahead-of-time normalization has *two major drawbacks.*

The first drawback is that the normalization phase rewrites types into potentially *exponentially bigger* types. This is a problem in practice. Previous work [7] instrumented Julia's subtype decision procedure and when executing library code, routinely observed queries involving types as the one below:

```
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}
```

Normalizing this type before attempting to decide subtyping is not a realistic option. In practice, such types are often matched against structurally simpler types like

```
Tuple{Tuple{Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any, Any},
      Any}
```

(where in Julia `Any` is the supertype of all types); in these cases, it is possible to decide subtyping in linear time without ahead-of-time normalisation.

The second drawback of the ahead-of-time normalisation phase is that it does not interact well with other type system features. For instance, *invariant constructors* make lifting all union types to the top level unsound. Consider the type $\texttt{Array\{Union\{}t_1, t_2\texttt{\}\}}$, where `Array` is an invariant unary constructor. This type denotes the set of arrays whose elements are either of type t_1 or t_2; it would be incorrect to rewrite it as $\texttt{Union\{Array\{}t_1\texttt{\}}, \texttt{Array\{}t_2\texttt{\}\}}$, as this latter type denotes the set of arrays whose elements are either all of type t_1 or all of type t_2. A weaker disjunctive normal form, only lifting union types inside each invariant constructor, can circumvent this problem. However, doing so only to reveals a deeper problem in the presence of both invariant constructors and *existential types.* Consider the judgment below:

$$\texttt{Array\{Union\{Tuple\{}t_1\texttt{\}}, \texttt{Tuple\{}t_2\texttt{\}\}\}} \quad <: \quad \exists T\,.\,\texttt{Array\{Tuple\{}T\texttt{\}\}}$$

This judgment holds by taking $T = \texttt{Union\{}t_1, t_2\texttt{\}}$. Since all types are in weak normal form, a standard algorithm would initially perform the left-to-right subtype check due to the outer invariant constructor. This step would generate the constraint $T >: \texttt{Union\{}t_1, t_2\texttt{\}}$. As a consequence, the subsequent right-to-left check due to the invariant constructor fails. Indeed this requires proving that $\texttt{Tuple\{}T\texttt{\}} <: \texttt{Union\{Tuple\{}t_1\texttt{\}}, \texttt{Tuple\{}t_2\texttt{\}\}}$, which in turns attempts to prove either $T <: t_1$ or $T <: t_2$, both unprovable under the assumption that $T >: \texttt{Union\{}t_1, t_2\texttt{\}}$ The key to derive a successful judgment for this relation is to rewrite the right-to-left check into $\texttt{Tuple\{}T\texttt{\}} <: \texttt{Tuple\{Union\{}t_1, t_2\texttt{\}\}}$, which is provable. This is a sort of *anti-normalisation* rewriting that must be performed on sub-judgments of the derivation, and to the best of our knowledge it is not part of any subtype algorithm based on ahead-of-time disjunctive normalisation.

In this *pearl paper* we describe the keys ideas used by the subtype algorithm implemented in the Julia language to deal with union types and covariant tuples, and we will argue that these avoid the two drawbacks above. To avoid being drawn in the vast complexity of Julia type algebra, we focus on a minimal language featuring union types, covariant tuples, and literals. This tiny language is expressive enough to highlight the decision strategy, and make this implementation technique known to a wider audience. While Julia implementation shows that this technique extends, among others, to invariant constructors and existential types [7], we expect that it can be leveraged in many other modern language designs. Additionally we prove in Coq that the algorithm is correct and complete with respect to a standard semantics subtyping model.

2 A space-efficient algorithm

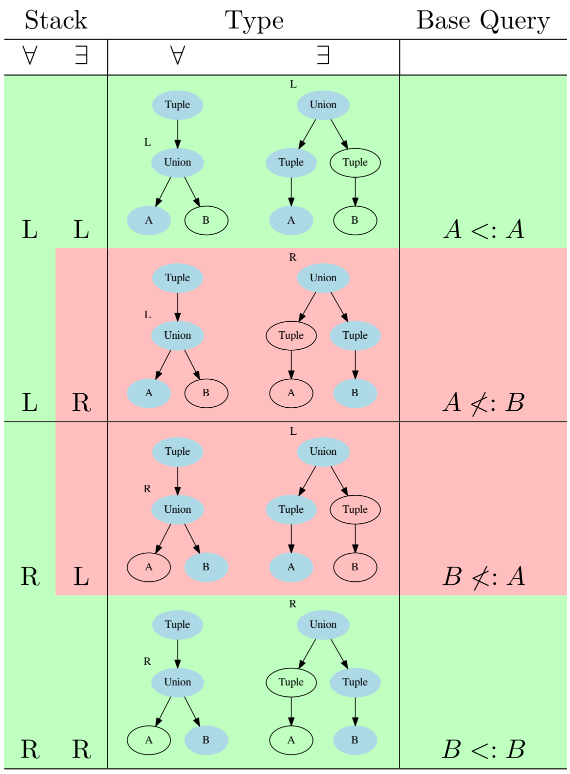
The greatest difficulty of subtyping union types is searching the entire space of possible forall/exist quantifications of union types. We have seen that syntax-directed rules need ahead-of-time normalisation in the presence of covariant tuples.

The algorithm we propose for deciding subtyping works by iterating through choices made at unions along paths complete through the two types In this section, we will describe the implementation of the algorithm in OCaml and will present our proof of correctness in the following section.

We focus on a core type language composed of binary unions, covariant binary tuples, and atom types:

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

Atom types are singletons with respect to subtyping, and are ranged over by $A, B, \ldots$. Abusing the notation, we will assume that we also have unary tuples; these can be easily encoded.



$\texttt{Tuple\{Union\{}A, B\texttt{\}\}} <: \texttt{Union\{Tuple\{}A\texttt{\}}, \texttt{Tuple\{}B\texttt{\}\}}$

Figure 1 Subtyping decision procedure example

As a running example we use the distributivity example from the introduction:

$$\texttt{Tuple\{Union\{}A, B\texttt{\}\}} <: \texttt{Union\{Tuple\{}A\texttt{\}}, \texttt{Tuple\{}B\texttt{\}\}}$$

In Figure 1 we sketch the corresponding execution of the subtyping decision procedure. Recall that to check if the relation holds, the algorithm needs to ensure that for every choice that could be made at a union types on the left hand side of the judgment there is a set of choices for the union types on the right hand side such that subtyping holds. In this example, the relation holds: no matter if we choose A or B on the left hand side inside the covariant tuple, we can always pick a matching type on the right hand side. The key challenge for a subtyping algorithm not based on ahead-of-time normalisation is to enumerate every possible choice of the components of the unions appearing in the left and right hand sides of the judgments.

To do this, we base our approach on iterators. The `st_choice` type represents whether the algorithm takes the left (first) or the right (second) component of a union type:

```
type st_choice = Left | Right
```

The algorithm stores the choice made at each union at the present iteration in *two stacks of* `st_choice`*s*, one for each side of the subtype relation. Each `st_choice` stack can be seen as the state of an iterator that enumerates every union-type induced alternative in a given type. One stack will be used to ensure that all *forall* choices have been explored; the other stack to search for a matching *exist* choice. In figure 1, these are depicted as the ∀ and ∃ stacks.

It is possible to define both an initial element and next state function for iterators defined using `st_choice` stacks.

```
let rec initial_choice (a:typ) =
  match a with
  | Atom i -> []
  | Tuple(t1,t2) -> (initial_choice t1) @ (initial_choice t2)
  | Union(l,r) -> Left::(initial_choice l)
```

We choose the initial value for our iterators to be the state where, every time a union type is encountered in the structure of the type, the iterator takes the `Left` choice (as implemented in `initial_choice` – as we shall see when we will discuss the `extend_list` function, empty lists are expanded on demand with the appropriate `Left` choices).

Given this definition for the initial state, we can then define the iteration function. We break the iteration function, `next_state`, up into two helper functions. The operation of these helpers is illustrated in figure 2. Here, starting from the initial state (where the iterator takes the `Left` choice at the root of the type), `last_left_to_right` converts the final `Left` choice into a `Right` choice and truncates the remaining choices.

```
let rec last_left_to_right
     (acc:st_choice list) (ll : st_choice list option) = function
  | Left::tl -> last_left_to_right (Left::acc) (Some (Right::acc)) tl
  | Right::tl -> last_left_to_right (Right::acc) ll tl
  | [] -> option_map List.rev ll
```

For example, given the type $\texttt{Union\{Union\{}A, B\texttt{\}}, C\texttt{\}}$ and the choice list `LL`, `last_left_to_right` will produce the choice list `LR` (`Left` and `Right` are shortened here with `L` and `R`); if given `LR`, it will produce `R` (as the final left choice is at the top-level and it truncates the remainder of the choice list).

In Figure 2, `last_left_to_right` produced a partial path, as the right child of the root union is a union itself. To solve this, `extend_list` finds where the new choice list runs out and fills it out with left choices to be valid with respect to the type, producing a valid state. `extend_list` pads the list out to take the left choice at this child union, returning the path to validity.

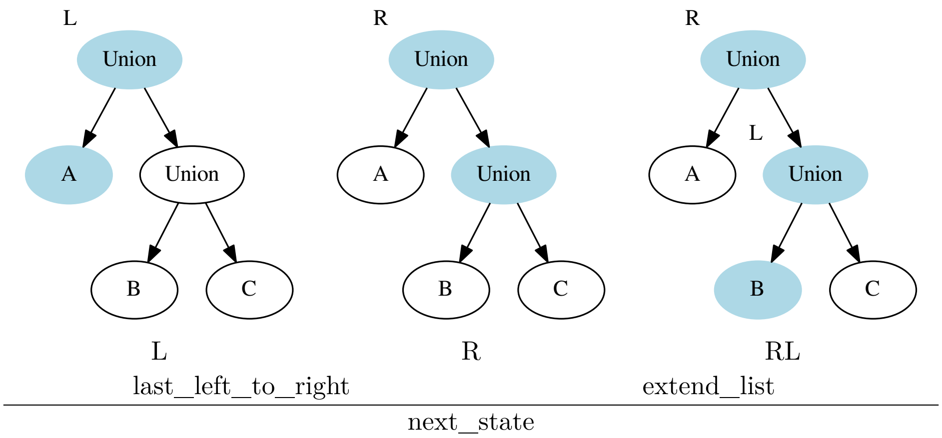


Figure 2 State-stepping operation for choice lists

```
let rec extend_list (a:typ) (ls:st_choice list) = match (a,ls) with
  | (Atom i, _) -> ([], ls)
  | (Tuple(t1,t2), _) -> let (hd,tl) = extend_list t1 ls in
                         let (hd2,tl2) = extend_list t2 tl in
                         (hd @ hd2, tl2)
  | (Union(l,r), Left::rs) -> let (hd,tl) = extend_list l rs in (Left::hd,tl)
  | (Union(l,r), Right::rs) -> let (hd,tl) = extend_list r rs in (Right::hd,tl)
  | (Union(l,r), []) -> (Left::initial_choice l,[])
```

Finally, the `next_state` function combines these operations into a single step operation. It will take the deepest alternative choice at a union using `last_left_to_right`, then add however many left choices are required to make the path valid using `extend_list`. The complete `next_state` operation shown in figure 2 takes the choice stack L and the type `Union{`A`,Union{`B,C`}}` and produces the choice stack RL. To do so, it takes the left choice, converts it to a right, then pads the list out with lefts until the path is valid with respect to the type.

```
let rec next_state (a:typ) (ls:st_choice list) =
  option_map fst (option_map (extend_list a) (last_left_to_right [] None ls))
```

Now that we have defined the core iterator infrastructure, we show how it is used to decide subtyping queries. The algorithm proceeds by maintaining two iterators (one for each side of the subtyping judgment) over choices-at-unions, checking that for every instantiation of the left hand type there exists an instantiation of the right hand type. We will first define the fundamental subtyping relation, used to decide subtyping relationships when given instantiations of the left and right hand types, then describe the algorithm that iterates through those types.

```
type st_res =
  | Subtype of st_choice list * st_choice list
  | NotSubtype
```

The type `st_res` represents the results of a single base subtype query. The query can either succeed, in which case it provides the unused portion of the input stacks, or fail.

```
let rec base_subtype (a:typ) (b:typ) (fa:st_choice list) (ex : st_choice list)
```

```
  match (a,b,fa,ex) with
  | (Atom i, Atom j, _, _) -> if i == j then Subtype(fa, ex) else NotSubtype
  | (Tuple(ta1, ta2), Tuple(tb1, tb2), _, _) ->
     (match base_subtype ta1 tb1 fa ex with
      | Subtype(cfa, cex) -> base_subtype ta2 tb2 cfa cex
      | NotSubtype -> NotSubtype)
  | (Union(a1,a2),b,choice::fa,ex) ->
              base_subtype (match choice with Left -> a1 | Right -> a2) b fa ex
  | (a,Union(b1,b2),fa,choice::ex) ->
              base_subtype a (match choice with Left -> b1 | Right -> b2) fa ex
```

The function base_subtype is responsible for using two paths—one for the left and right hand sides of the subtyping relation—and checking that the subtype relation holds with respect to those paths. Given a basic type—an atom or a tuple—it will either check equality or recur respectively. Given a union type, it will choose the component of the union type following the choice stacks and continue recursively. The function then returns an instance of st_res, which gives the remaining choice stacks if successful or nothing otherwise.

Finally, to check subtyping, we need to iterate through both types in the correct quantification order. Subtyping holds if, for every instantiation of the left there exists an instantiation of the right hand side of the subtyping judgment such that the subtype relation holds. To check this, we use a simple brute force approach sitting atop the choice stack iterator infrastructure described previously. The algorithm is implemented in two functions. ex_subtype checks that there exists an instantiation of the right-hand-side for a given left-hand-side. fa_ex_subtype uses ex_subtype to check that for every instantiation of the LHS, there is an instantiation of the RHS such that the subtype relation holds.

```
let rec ex_subtype (a:typ)(b:typ)(fa:st_choice list)(cex : st_choice list) =
  match base_subtype a b fa cex with (* is a <: b *)
  | Subtype(_,_) -> true (* there exists a subtype *)
  | NotSubtype ->
     (match next_state b cex with (* step exists-env *)
      | Some ns -> ex_subtype a b fa ns (* continue *)
      | None -> false) (* no subtype; exit *)
```

If the current choice — given by cex — is a supertype of the given a according to base_subtype, then ex_subtype has found a valid instantiation of b. Therefore, there exists an instantiation of b that is a supertype of a and the result should be true. Otherwise, ex_subtype will use the iteration operation, next_state, to continue iterating through choices for b until it either finds an instantiation that is a supertype or runs out of instantiations.

```
let rec fa_ex_subtype (a:typ)(b:typ)(cfa:st_choice list) =
  match ex_subtype a b cfa (initial_choice b) with (* a <: b wrt path? *)
  | true -> (match next_state a cfa with
             | Some ns -> fa_ex_subtype a b ns (* continue *)
             | None -> true) (* all subtypes; is subtype *)
  | false -> false (* exists a non-subtype; not subtype *)
```

The function fa_ex_subtype is similar; it checks that for every instantiation of a, there exists an instantiation of b such that subtyping holds. Checking for an instantiation of b is done using ex_subtype, while fa_ex_subtype maintains an iterator for a.

The operation of fa_ex_subtype and ex_subtype as well as their calls to base_subtype can be seen in figure 1. In the $\forall$ column, we see the current state maintained by fa_ex_subtype.

The ∃ column shows the state maintained by `ex_subtype` as it is called by `fa_ex_subtype`. In the example, for a forall-list of L, we an find an exists-list of L such that `base_subtype` holds. Similarly, for a forall-list of R, we can find an exists-list instantiation of R such that subtyping holds. Therefore, for every instantaition of the left-hand-side, there exists an instantiation of the right-hand-side such that subtyping holds and subtyping holds for the type as a whole.

```
let rec subtype (a:typ) (b:typ) = fa_ex_subtype a b (initial_choice a)
```

Finally, `subtype` checks if `a` is a subtype of `b`. It seeds `fa_ex_subtype` with the initial choice for `a`'s iterator, which then checks if for every instantiation of `a` there exists an instantiation of `b` such that subtyping holds.

We have presented our subtyping algorithm using lists of choices. In a practical implementation, however, these lists of choices can be efficiently implemented (without allocation) by means of bit sets. This is the approach taken in the Julia implementation of this algorithm. With this optimization, the needed memory to decide a subtyping judgment is linear in the total number of unions in the given types; the algorithm needs no allocation beyond that of the choice stacks themselves.

3 Correctness and completeness

To prove correctness of our algorithm, we begin by formally specifying correctness for subtyping. We then show that two subtyping algorithms— based on structural iterators and choice lists—are correct with respect to this definition.

We base our definition of subtyping on a denotational semantics for types. We reduce types in the type language including unions to sets of types in the type language without unions through a simple transformation.

$$
\begin{aligned}
[\![\texttt{A}]\!] &= \{A\} \\
[\![\texttt{Union\{}t_1, t_2\texttt{\}}]\!] &= [\![t1]\!] \cup [\![t2]\!] \\
[\![\texttt{Tuple\{}t_1, t_2\texttt{\}}]\!] &= \{\texttt{Tuple\{}t_1', t_2'\texttt{\}} | t_1' \in [\![t_1]\!], t_2' \in [\![t_2']\!]\}
\end{aligned}
$$

Using this denotational semantics for types-with-unions, we can define subtyping as if $[\![t_1]\!] \subseteq [\![t_2]\!]$, then $t_1 <: t_2$. Equivalently, we can state this as definition 1, which is canonicalized in our Coq proof as the `NormalSubtype` relation.

▶ **Definition 1** (Subtyping Correctness). *A subtyping relation* $<:$ *is correct if* $t_1 <: t_2$ *iff* $\forall t_1' \in [\![t_1]\!], \exists t_2' \in [\![t_2]\!], t_1 <: t_2$.

Proving a subtyping algorithm sound and complete is therefore equivalent to producing a function of type `forall t1 t2:type, {NormalSubtype t1 t2} + {~NormalSubtype t1 t2}`; that is, is able to decide whether two types are subtypes or not.

We will begin by describing and proving correct a version of the algorithm that uses explicit type-structural iterators. We will then show the choice stack-based algorithm correct by proving equivalence between structural iterators and choice stacks. In doing so, we will derive an induction principle for structural iterators (and, as an extension, for choice stacks).

3.1 Iterators

The iterator-based implementation is directly equivalent (as will be shown later) to the choice-stack based implementation presented previously in OCaml. However, it retains type structure information inside of the iterator state.

```
Inductive TypeIterator: type -> Set :=
| TIAtom : forall i, TypeIterator (atom i)
| TITuple : forall t1 t2, TypeIterator t1 -> TypeIterator t2 -> TypeIterator (tuple t1 t2)
| TIUnionL : forall t1 t2, TypeIterator t1 -> TypeIterator (union t1 t2)
| TIUnionR : forall t1 t2, TypeIterator t2 -> TypeIterator (union t1 t2).
```

The `TypeIterator` structure follows the structure of the type being iterated over. Choices at unions are represented as either an instance of `TIUnionR` or `TIUnionL`. This structure then lets us trivially define a function that extracts the current type at the iterator's position:

```
Fixpoint current (t:type)(ti:TypeIterator t):type :=
match ti with
| TIAtom i => atom i
| TITuple ti1 ti2 p1 p2 => tuple (current ti1 p1) (current ti2 p2)
| TIUnionL ti1 ti2 pl => (current ti1 pl)
| TIUnionR ti1 ti2 pr => (current ti2 pr)
end.
```

We can then define a function that produces the initial iterator state for a given type:

```
Fixpoint start_iterator (t:type):TypeIterator t :=
  match t with
  | (atom i) => TIAtom i
  | (tuple t1 t2) => TITuple t1 t2 (start_iterator t1) (start_iterator t2)
  | (union t1 t2) => TIUnionL t1 t2 (start_iterator t1)
  end.
```

Next, we can define a step function that takes one state and either steps it to the next state or indicates that no such next state exists.

```
Fixpoint next_state (t:type)(ti:TypeIterator t) : option (TypeIterator t) :=
  match ti with
  | TIAtom i => None
  | TITuple ti1 ti2 p1 p2 =>
    match (next_state ti2 p2) with
    | Some np2 => Some(TITuple ti1 ti2 p1 np2)
    | None =>
      match (next_state ti1 p1) with
      | Some np1 => Some(TITuple ti1 ti2 np1 (start_iterator ti2))
      | None => None
      end
    end
  | TIUnionL ti1 ti2 pl =>
    match (next_state ti1 pl) with
    | Some npl => Some(TIUnionL ti1 ti2 npl)
    | None => Some(TIUnionR ti1 ti2 (start_iterator ti2))
    end
  | TIUnionR ti1 ti2 pr => option_map (TIUnionR ti1 ti2) (next_state ti2 pr)
  end.
```

With these definitions, we can then prove a basic form of correctness with respect to the denotational or normalization semantics:

▶ **Theorem 2** (Correctness of iterators). `Remaining t (start_iterator t) (clauses t)`
Every type in $[\![t]\!]$ *will be explored using* `next_step` *from* `start_iterator t`*.*

Proof. The `Remaining` predicate relates iterators to the list of types that remain to be iterated, so the Coq theorem statement indicates that the initial state of the iterator for type t has every clause in the normalized version of t remaining to be iterated.

We proceed by induction on t. The cases for atomic types and unions follow from the IH trivially. We prove the theorem for tuples correct by case analyzing on the number of clauses induced by the first element in the tuple, then identifying the next element produced by the iterator from the tuple.

See `iterator_has_clauses` in the Coq proof for full details. ◀

`next_state` returns `Some s` if there is some successor state `s` to the current, and `None` if the given iterator state is terminal. It will go left-to-right through unions, and will explore 2-tuples by iterating through the choices on the right for each choice on the left. We can then define an induction principle for type iterators based on `next_state`:

▶ **Theorem 3.**
```
Definition iter_rect
  (t:type) (P:TypeIterator t -> Set)
           (pi: forall it, next_state t it = None -> P it)
           (ps : forall it' it'', P it'' -> next_state t it' = Some it'' -> P it')
           (it : TypeIterator t) : P it
```
For any type `t` *and proposition* `P`*, and if:*

- `P` *holds for an iterator that has no next state (e.g. is done)*
- *if* `P` *holds for the* following *iterator state* `it`*, then* `P` *holds for the* preceeding *iterator state* `it'`*.*

Then `P` *holds for all iterators for type* `t`

Proof. Proving the induction principle for type iterators relies on the `iternum` function, which decides the number of steps remaining in the iterator before termination. The proof proceeds by simultaneous case analysis on the number of remaining states and whether the iterator step function can produce a successor state from the present state.

If the iteration number is not yet 0, and if there is a successor state, then we simply appeal to the induction hypothesis and continue on. If there is no successor state but the iteration number is nonzero or vice versa, then by lemma (`iternum_monotonic`, taking an iterator step decrements the iteration number) contradiction. Finally, if there is no next step and the iteration number is 0, then we have reached the base case and terminate.

For full details, see the Coq definition of `iter_rect`. ◀

Using `iter_rect`, we can implement and prove correct equivalent functions to `ex_subtype`, `fa_ex_subtype`, and `subtype` as described in the OCaml implementation.

```
Definition exists_iter(a b : type) :
  ({ t | InType t b /\ BaseSubtype a t } +
   {forall t, InType t b -> ~(BaseSubtype a t) }).
```

`exists_iter` is equivalent to the choice-stack based `ex_subtype`, and determines if there exists some denotationally-contained type in `b` that is a supertype of the given `a`. Internally, it is implemented in the same way as `ex_subtype`, though using `iter_rect` to iterate through every iterator state.

```
Definition forall_iter (a b : type) :
  { forall t, In t (clauses a) -> exists t', InType t' b /\ (BaseSubtype t t')} +
  { exists t, In t (clauses a) /\ forall t', InType t' b -> ~ (BaseSubtype t t')}.
```

`forall_iter` is to `fa_ex_subtype` what `exists_iter` is to `ex_subtype`. Like `exists_iter` it implements the same decision procedure as `fa_ex_subtype` (and internally relies upon `exists_iter`), though through the abstraction of `iter_rect`.

Finally, we can define a decidable function (called `subtype` in the proof) that decides whether two types are subtypes or not. `subtype` simply invokes `forall_iter` to decide subtyping.

```
Definition subtype(a b:type) : {NormalSubtype a b} + {~NormalSubtype a b}.
  destruct (forall_iter a b).
  - left. [...]
  - right. [...]
Defined.
```

Therefore, using iterators, we can decide whether subtyping holds for any two types in our language. We will now show an equivalence between iterators and stacks-of-choices, allowing for more efficient implementation.

3.2 Stacks

To show that the choice-stack based algorithm is correct, we reduce it to the already-shown-correct iterator-based algorithm for deciding subtyping. To do so, we show an equivalence between choice stacks and iterators, then prove correctness of the subtyping algorithms.

In the context of the Coq proof, we use the type `st_context` to refer to a choice stack. In Coq, this is represented as a list of boolean values, with false representing a left choice and true representing a right choice at a specific union.

To show equivalence between the iterator-based and stack-based algorithm, we need to first prove two properties:

- iterators are convertible to equivalent choice lists;
- stepping an iterator is equivalent to stepping a choice list.

We define an iterator and a choice stack to be equivalent if, when applied to the same type, they select the same subset of that type. To describe this, we define `lookup_path` which looks up what type is selected by a given choice stack.

```
Fixpoint lookup_path(t:type)(p:st_context) : type * st_context :=
  match t, p with
  | atom i, _ => (t, p)
  | tuple t1 t2, _ =>
    let (r1,p1) := lookup_path t1 p in
    let (r2,p2) := lookup_path t2 p1 in
    (tuple r1 r2, p2)
  | union l r, false::rs => lookup_path l rs
  | union l r, true::rs => lookup_path r rs
  | _, nil => (t, nil)
  end.
```

`lookup_path` is notable in that it both returns the selected type as well as whatever of the choice stack remains once it reaches a leaf. This is needed in order to be able to traverse types that contain tuples, whose left branches will potentially be given a longer choice stack then necessary.

Next, we can convert iterators to paths in the straightforward manner, as implemented by `iterator_to_path`:

```
Fixpoint iterator_to_path(t:type)(it:TypeIterator t):st_context :=
  match it with
  | TIAtom _ => nil
  | TITuple t1 t2 it1 it2 => (iterator_to_path t1 it1) ++ (iterator_to_path t2 it2)
  | TIUnionL t1 _ it1 => false :: (iterator_to_path t1 it1)
  | TIUnionR _ t2 it1 => true :: (iterator_to_path t2 it1)
  end.
```

`iterator_to_path` simply traverses the iterator in order, appending onto the output choice stack whatever choice the iterator makes at that union. This illustrates the equivalence between iterators and choice stacks; choice stacks are simply iterators with the structural information removed.

Using the combination of `lookup_path` and `iterator_to_path`, we can then show the first correctness property that we need to prove that the algorithm using choice stacks is correct:

▶ **Lemma 4** (Iterator to path is correct). `Lemma itp_correct : forall t it, current t it = fst (lookup_path t (iterator_to_path t it)).`

For every type `t` *and type iterator* `it`*, the iterator's current type* `current t it` *is equal to the result of looking up the conversion of* `it` *to a choice stack.*

Proof. See `itp_correct` in the Coq proof. ◀

Stepping in the Coq implementation is implemented identically to the OCaml implementation. It only remains to show that this step operation (called `step_ctx` in Coq) is correct with respect to the iterator `next_state`.

▶ **Lemma 5** (Correctness of step_ctx). `forall t it, step_ctx t (iterator_to_path t it) = (option_map (iterator_to_path t) (next_state t it)).`

For every type `t` *and type iterator* `it`*, stepping the choice-list equivalent of* `it` *will produce the same result as converting the result of stepping* `it`*.*

Proof. See `list_step_correct` in the Coq proof. ◀

Now, with the relevant properties proven, we can implement and prove correct `ex_subtype` and `fa_ex_subtype` in Coq. The function names are the same, as are the implementations up to the addition of a fuel parameter (which is shown to be unnecessary).

▶ **Lemma 6** (Correctness of existential subtype checking with choice stacks). `forall a b it, (exists pf, exists_iter_inner a b it = inleft pf) <-> exists n, ex_subtype a b (iterator_to_path b it) n = Some true.`

For every two types `a` *and* `b`*, the iterator-based algorithm* `exists_iter_inner` *will produce a proof that* `a` *is a subtype of* `b` *if and only if there is an integer* `n` *such that* `ex_subtype` *given* `n` *fuel runs producing true.*

Proof. See `ex_sub_corr_eq` in the Coq proof. ◀

▶ **Lemma 7** (Correctness of forall-exists subtype checking with choice stacks).
```
forall a b it,
  (exists pf, forall_iter_inner a b it = left pf) <->
   exists n, fa_ex_subtype a b (iterator_to_path a it) n = Some true.
```
For every two types `a` *and* `b`*, the iterator-based algorithm* `forall_iter_inner` *will produce a proof that* `a` *is a subtype of* `b` *if and only if there is an integer* `n` *such that* `fa_ex_subtype` *given* `n` *fuel runs producing true.*

Proof. See `fa_sub_corr_eq` in the Coq proof. ◀

The choice stack-based algorithm therefore is provably equivalent to the iterator-based algorithm, and is thus correct.

4 Performance Analysis

The algorithm improves upon normalization in two key ways:

- lazily exploring possible clauses, obviating the need to store a fully normalized type;
- enabling fast paths that avoid the exploration of the full choice space.

In the worst case, our algorithm has the same big-O time complexity. However, lazily exploring the choice space allows us to require worst-case polynomial space, in comparison to normalization's exponential space complexity. Similarly, the algorithm enables optimizations that offer best (and typical) case time complexity improvements from exponential to linear time.

Worst case time complexity of both subtyping algorithms is determined by the number of clauses that would exist in the normalized type. In the worst case, (a tuple of unions), each union begets a different clause in the normalized type. Consider `Tuple{Union{`A, B`}, Union{`C, D`}}`, which will normalize to `Union{Tuple{`A, C`}, Tuple{`A, D`}, Tuple{`B, C`}, Tuple{`B, D`}}` generating a new tuple for each choice for every contained union. As a result, there are worst-case 2^n tuples in the fully normalized version of a type that has n unions.

In order to ensure correctness, each of these tuples (or choices at unions) must always be explored. As a result, both the algorithm we present here and normalization will have worst-case $O(2^n)$ time complexity. The approaches differ, however, in space complexity. The normalization approach computes and stores each of the exponentially many alternatives, so also has $O(2^n)$ space complexity. However, the algorithm we discuss need only store the choice made at each union, thereby offering $O(n)$ space complexity.

The algorithm we discuss also can improve best-case time performance. Normalization will necessarily be $o(2^n)$ due to computation of the entire normalized type. However, the lazy subtyping algorithm need only make one choice before discovering that a subtype relation exists in the best-case, giving $o(n)$ performance. Moreover, computing type choices lazily enables fast-paths to short circuit full exploration of choice alternatives.

This is important for Julia due to a common programming idiom. Many Julia library developers write signatures of the form `Tuple{Union{`A, B`}, Union{`C, D`}}` to indicate that their method can take any of the named types. When deciding dispatch against these methods, Julia will frequently check if a tuple containing solely concrete (instantiable) types is a subtype of the tuple of unions. If Julia used normalization, this would always be exponential on the number of unions that appeared in the argument list as this is the above mentioned worst-case exponential complexity case. However, its use of the lazy algorithm enables it frequently identify the best alternative and short circuit before having to explore much of the choice possibility space.

5 Conclusion

We have presented an algorithm for deciding subtyping relationships between types that consist of atomic types, tuples, and unions. This algorithm is able to decide subtyping relationships in the presence of distributive semantics for union types without needing normalization (and therefore using linear space) and without additionally constraining type system features.

References

1. Hack. `https://hacklang.org/`. Accessed: 2019-01-11.
2. 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`.
3. 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`.
4. Anders Hejlsberg. Introducing typescript. URL: `https://channel9.msdn.com/posts/Anders-Hejlsberg-Introducing-TypeScript`.
5. Benjamin Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.
6. Jerome Vouillon. Subtyping union types. In *Computer Science Logic (CSL)*, 2004. `doi:10.1007/978-3-540-30124-0_32`.
7. Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. Julia subtyping: a rational reconstruction. *PACMPL*, 2(OOPSLA):113:1–113:27, 2018. URL: `https://doi.org/10.1145/3276483`, `doi:10.1145/3276483`.