Pragmatic Semantic Subtyping

LILY BROWN, ANDY FRIESEN, and ALAN JEFFREY, Roblox, USA

This paper presents the view of subtyping Luau programming language. This system has been deployed as part of the Luau programming language, used by millions of users of Roblox Studio.

CCS Concepts: • **Software and its engineering** → **Semantics**.

ACM Reference Format:
Lily Brown, Andy Friesen, and Alan Jeffrey. 2024. Pragmatic Semantic Subtyping. *Proc. ACM Program. Lang.* 7, ICFP, Article ?? (September 2024), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Luau is a scripting language used by Roblox creators in the IDE tool in Roblox Studio. In 2022 there were more than 4 million creators, Fig 1 [7], which is the largest user base of Semantic Subtyping. In [1] we discuss why Luau uses Semantic Subtyping:

> *Semantic subtyping interprets types as sets of values, and subtyping as set inclusion [3]. This is aligned with the* minimize false positives *goal of Luau non-strict mode, since semantic subtyping only reports a failure of subtyping when there is a value which inhabits the candidate subtype, but not the candidate supertype. For example, the program:*
>
> ```
> local x : CFrame
>   = CFrame.new()
> local y : Vector3 | CFrame
>   = (math.random() < 0.5 ? CFrame.new() : Vector3.new())
> local z : Vector3 | CFrame
>   = x * y
> ```
>
> *cannot produce a run-time error, since multiplication of* `CFrame`*s is overloaded:*
>
> ```
> ((CFrame, CFrame) -> CFrame) & ((CFrame, Vector3) -> Vector3)
> ```
>
> *In order to typecheck this program, we check that that type is a subtype of:*
>
> ```
> (CFrame, Vector3 | CFrame) -> (Vector3 | CFrame)
> ```
>
> *In the previous, syntax-driven, implementation of subtyping, this subtype check would fail, resulting in a false positive. We have now released an implementation of semantic subtyping, which does not suffer from this defect. See our technical blog for more details [5].*

In Luau, we use a variant of semantic subtyping [3, 4, 6]. The important properties of semantic subtyping are:

- there is a set $\mathcal{D}$ of semantic values,
- each type T has a semantics $[\![T]\!] \subseteq \mathcal{D}$,
- `unknown` and `never` types are interpreted as $\mathcal{D}$ and $\emptyset$,
- union and intersection types are interpreted as set union and intersection, and
- subtyping $T <: U$ is interpreted as $[\![T]\!] \subseteq [\![U]\!]$.

Authors' address: Lily Brown; Andy Friesen; Alan Jeffrey, Roblox, San Mateo, CA, USA.

2475-1421/2024/9-ART?? $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

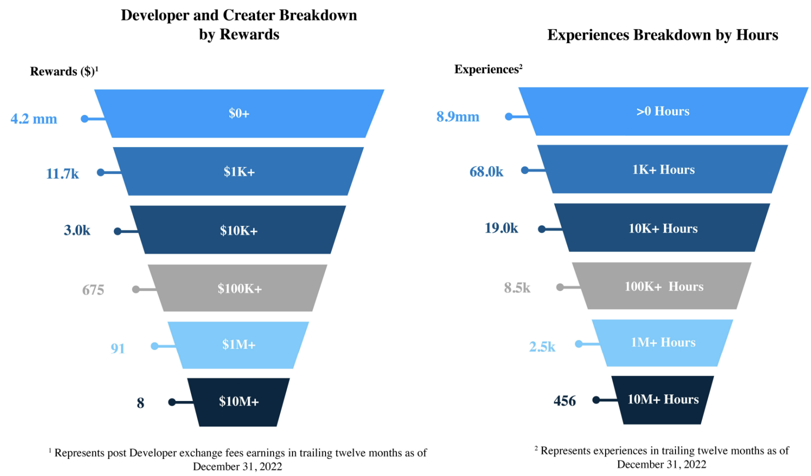

Fig. 1. Creators numbers in 2022

The off-the-shelf presentation of semantic subtyping is *set theoretic* [3, §2.5]:

$$[\![T_1]\!] \subseteq [\![T_2]\!] \text{ if and only if } \mathcal{E}[\![T_1]\!] \subseteq \mathcal{E}[\![T_2]\!]$$

where the most important case for $\mathcal{E}[\![T]\!]$ is function types:

$$\mathcal{E}[\![S \to T]\!] = \mathcal{P}(\mathcal{D}^2 \setminus ([\![S]\!] \times (\mathcal{D} \setminus [\![T]\!])))$$

The set theoretical requirement has some consequences:

All functions types $(\mathsf{never} \to T)$ *are identified.* Consider

$$\begin{aligned}
&\mathcal{E}[\![\mathsf{never} \to T_1]\!] \\
&\quad = \mathcal{P}(\mathcal{D}^2 \setminus ([\![\mathsf{never}]\!] \times (\mathcal{D} \setminus [\![T_1]\!]))) \\
&\quad = \mathcal{P}(\mathcal{D}^2 \setminus (\emptyset \times (\mathcal{D} \setminus [\![T_1]\!]))) \\
&\quad = \mathcal{P}(\mathcal{D}^2) \\
&\quad = \mathcal{P}(\mathcal{D}^2 \setminus (\emptyset \times (\mathcal{D} \setminus [\![T_2]\!]))) \\
&\quad = \mathcal{P}(\mathcal{D}^2 \setminus ([\![\mathsf{never}]\!] \times (\mathcal{D} \setminus [\![T_2]\!]))) \\
&\quad = \mathcal{E}[\![\mathsf{never} \to T_2]\!]
\end{aligned}$$

in particular, this means we cannot define a semantics-preserving function $\mathsf{apply}(T, U)$ such that:

$$\mathsf{apply}(S \to T, U) = T \text{ when } U <: S$$

because there is a nasty case where S is uninhabited. In this presentation, the apply function used in the rule for function application:

$$\frac{\begin{array}{c} D_1 : (\Gamma \vdash M : T) \\ D_2 : (\Gamma \vdash N : U) \end{array}}{\mathsf{app}(D_1, D_2) : (\Gamma \vdash M(N) : \mathsf{apply}(T, U))}$$

so we have to accept that in a set-theoretic model, the type rule for function application has corner cases for uninhabited types.

$$\begin{array}{rcl} v & ::= & s \mid (a \mapsto r) \\ a & ::= & () \mid (v) \\ r & ::= & \mathsf{diverge} \mid \mathsf{check} \mid (v) \end{array}$$

Fig. 2. Semantic values

Union does not distributed through function types. Semantic subtyping gives a natural model of overloaded functions as intersections of arrows, for example the Roblox API for matrices include an overloaded function which supports multiplication of both 2D (`CFrame`) and 1D (`Vector3`) matrices:

```
CFrame.__mul : ((CFrame, CFrame) -> CFrame) & ((CFrame, Vector3) -> Vector3)
```

Overloaded functions are a key part of the Roblox API, and we might expect that all function types can be presented as overloaded functions (that is intersections of arrows). We can do that it we can distribute union through arrow:

$$[\![(S_1 \to T_1) \cup (S_2 \to T_2)]\!] = [\![(S_1 \cap S_2) \to (T_1 \cup T_2)]\!]$$

For example:

$$[\![(\texttt{number?} \to \texttt{number}) \cup (\texttt{string?} \to \texttt{string})]\!] = [\![\texttt{nil} \to (\texttt{number} \cup \texttt{string})]\!]$$

Unfortunately, set-theoretic models do not allow union to distributed through intersection, for example:

$$\begin{array}{rcl} \{(1, \texttt{nil}), (\texttt{"hi"}, \texttt{nil})\} & \in & \mathcal{E}[\![\texttt{nil} \to (\texttt{number} \cup \texttt{string})]\!] \\ \{(1, \texttt{nil}), (\texttt{"hi"}, \texttt{nil})\} & \notin & \mathcal{E}[\![\texttt{number?} \to \texttt{number}]\!] \\ \{(1, \texttt{nil}), (\texttt{"hi"}, \texttt{nil})\} & \notin & \mathcal{E}[\![\texttt{string?} \to \texttt{string}]\!] \end{array}$$

This is why type normalization for function types in set-theoretic models uses a conjunctive normal form of unions of intersections of functions e.g. [6, §4.1.2].

Set-theoretic mode support negatived types. In addition, Luau does not support negation of all types, but only negation of *test types* [**?**], which simplifies the model, by not requiring arbitrary type negation. In particular, since the model does not support negation of function types, the normal form for function types is just overload functions, not combinations of positive and negative function types.

Conclusions of this paper. In summary there is a trade-off in semantic subtyping:

- *set-theoretic* models, which are closer to the set-theoretic model of functions, and
- *pragmatic* models, which drop the set-theoretic requirement, and in return a) do not have corner cases on the type of function application when the argument has uninhabited type, and b) have overloaded functions (that is intersections of arrows) as the normal for function types.

Luau chooses to adopt a pragmatic semantic subtyping model.

This paper shows how Luau pragmatic model is defined, and how it supports benefits of pragmatic models.

2 FORMAL TREATMENT OF CORE LUAU

2.1 Semantic Values for Core Luau

In this presentation, we present the minimal core of Luau, which supports scalars and functions. This presentation ignores tables, mutable features, and object objects. We will ignore the details of scalar types, and assume that:

- there are scalar types, ranged over by b, such as `nil`, `boolean`, `number` and `string`,
- there are scalar values, ranged over by s, such as `nil`, `true`, `false`, numbers and string literals, and
- each scalar type s has a set of scalar values $\langle\!\langle s \rangle\!\rangle$, such as:

$$\langle\!\langle \mathtt{nil} \rangle\!\rangle = \{\mathtt{nil}\} \quad \langle\!\langle \mathtt{boolean} \rangle\!\rangle = \{\mathtt{true}, \mathtt{false}\} \quad \langle\!\langle \mathtt{number} \rangle\!\rangle = \{0, 1, \dots\} \quad \cdots$$

The types we are considering are:

$$S, T ::= b \mid \mathtt{unknown} \mid \mathtt{never} \mid S \rightarrow T \mid S \cap T \mid S \cup T$$

which are:

- the *scalar types* b
- the *top type* `unknown`,
- the *bottom type* `never`,
- a *function* type $S \rightarrow T$,
- an *intersection* type $S \cap T$, and
- a *union* type $S \cup T$.

To give a semantic subtyping, we first declare the domain $\mathcal{D}$ of *semantic values*, given by the grammar v of Figure 2. Semantic values are:

- *scalar values* s, and
- *function values* $a \mapsto r$, modeling a function that can can map an argument a to a result r.

For example:

- `true` and `false` are values in `boolean`,
- `true` and `false` and `nil` are values in the optional type `boolean` $\cup$ `nil`, and
- $(\mathtt{true}) \mapsto (\mathtt{false})$ is a value in the function type $\mathtt{boolean} \rightarrow \mathtt{boolean}$.

Scalar and error-suppressing values are relatively straightforward, but functions are trickier. The case where a type-correct argument is supplied and a type-correct result is returned is clean, for example:

$$((\mathtt{true}) \mapsto (\mathtt{false})) \in [\![\mathtt{boolean} \rightarrow \mathtt{boolean}]\!]$$

But there is also the case where a type-incorrect argument is supplied, in which case there is no guarantee what is returned, for example:

$$((5) \mapsto (37)) \in [\![\mathtt{boolean} \rightarrow \mathtt{boolean}]\!]$$

The type-correctness guarantee for results applies when a type-correct argument is provided:

$$((\mathtt{true}) \mapsto (37)) \notin [\![\mathtt{boolean} \rightarrow \mathtt{boolean}]\!]$$

Those examples consider cases where one value is supplied as an argument, and one is returned, but Luau allows other cases. Luau, as is common in most functional languages, allows functions to diverge (modeled in this semantics as $a \mapsto \mathtt{diverge}$)., for example:

$$((\mathtt{true}) \mapsto \mathtt{diverge}) \in [\![\mathtt{boolean} \rightarrow \mathtt{boolean}]\!]$$

and:

$$((5) \mapsto \mathtt{diverge}) \in [\![\mathtt{boolean} \rightarrow \mathtt{boolean}]\!]$$

Luau allows functions to check arguments, (modeled in this semantics as $a \mapsto \mathtt{check}$ when a checked fails), for example:

$$((5) \mapsto \mathtt{check}) \in [\![\mathtt{boolean} \rightarrow \mathtt{boolean}]\!]$$

but:

$$((\mathtt{true}) \mapsto \mathtt{check}) \notin [\![\mathtt{boolean} \rightarrow \mathtt{boolean}]\!]$$

$$\begin{array}{rcl}
[\![b]\!] & = & \langle\!\langle b \rangle\!\rangle \\
[\![\mathtt{unknown}]\!] & = & \mathcal{D} \\
[\![\mathtt{never}]\!] & = & \emptyset \\
[\![S \to T]\!] & = & \{a \mapsto (w) \mid w \in [\![T]\!]\} \cup \\
 & & \{(v) \mapsto r \mid v \in [\![S]\!]^{\complement}\} \cup \\
 & & \{a \mapsto \mathsf{diverge}\} \cup \\
 & & \{() \mapsto \mathsf{check}\} \\
[\![S \cap T]\!] & = & [\![S]\!] \cap [\![T]\!] \\
[\![S \cup T]\!] & = & [\![S]\!] \cup [\![T]\!]
\end{array}$$

Fig. 3. Semantics of types as sets of values

Luau allows functions to be called without any arguments (modeled in this semantics as $() \mapsto r$) for example:

$$(() \mapsto (\mathtt{false})) \in [\![\mathtt{boolean} \to \mathtt{boolean}]\!]$$

and:

$$(() \mapsto \mathsf{diverge}) \in [\![\mathtt{boolean} \to \mathtt{boolean}]\!]$$

and:

$$(() \mapsto \mathsf{check}) \in [\![\mathtt{boolean} \to \mathtt{boolean}]\!]$$

The restriction on zero-argument function calls is that they are allowed to return a check (since they have been passed the wrong number of arguments) but they are not just allowed to return arbitrary nonsense:

$$(() \mapsto (5)) \notin [\![\mathtt{boolean} \to \mathtt{boolean}]\!]$$

At this point we have introduced the semantic values used by the Luau type system, and can turn the semantics of types, from which semantic subtyping follows.

2.2 Semantics of Core Luau Types

The semantics of Luau types are given in Fig 3. This semantics is presented mechanically in Agda in [2], and we will give the most important results here.

For example, two of the important rules are for functions, in the case where functions are called with argument values, and return result values. The rules are:

- **Type-incorrect argument:** if $v \notin [\![S]\!]$ then $((v) \mapsto r) \in [\![S \to T]\!]$
- **Type-correct result:** if $w \in [\![T]\!]$ then $(a \mapsto (w)) \in [\![S \to T]\!]$

This is the same as the semantics of Coppo types [?] as used in the fully abstract semantics of Lazy Lambda Calculus [?] using Domain Theory In Logical Form [?].

$$((v) \mapsto (w)) \in [\![S \to T]\!] \text{ if and only if } (v \in [\![S]\!]) \Rightarrow (w \in [\![T]\!])$$

In order to give a constructive presentation of the semantics, rather than usual negative presentation of $v \notin [\![S]\!]$, we give a positive presentation $v \in [\![S]\!]^{\complement}$, as given in Fig 4.

It is routine to check that $[\![S \cap T]\!]^{\complement}$ is a constructive presentation of $\mathcal{D} \setminus [\![S \cap T]\!]$.

Lemma 2.1. $(v \in [\![T]\!]^{\complement})$ *if and only if* $(v \notin [\![T]\!])$.

Proof. An proof by injunction on T showings that $[\![T]\!]$ is the negative of $[\![T]\!]^{\complement}$. □

Moreover there is a decision procedure for $v \in [\![T]\!]$ or $v \in [\![T]\!]^{\complement}$.

$$
\begin{array}{rcl}
[\![b]\!]^{\complement} & = & \langle\!\langle b \rangle\!\rangle^{\complement} \\
[\![\mathtt{unknown}]\!]^{\complement} & = & \emptyset \\
[\![\mathtt{never}]\!]^{\complement} & = & \mathcal{D} \\
[\![S \to T]\!]^{\complement} & = & \{s\} \cup \\
& & \{() \mapsto (w) \mid w \in [\![T]\!]^{\complement}\} \cup \\
& & \{(v) \mapsto (w) \mid v \in [\![T]\!] \mid w \in [\![T]\!]^{\complement}\} \cup \\
& & \{(v) \mapsto \mathsf{check} \mid v \in [\![T]\!]\} \\
[\![S \cap T]\!]^{\complement} & = & [\![S]\!]^{\complement} \cup [\![T]\!]^{\complement} \\
[\![S \cup T]\!]^{\complement} & = & [\![S]\!]^{\complement} \cap [\![T]\!]^{\complement}
\end{array}
$$

Fig. 4. Complemented semantics of types as sets of values

.

Lemma 2.2. $(v \in [\![T]\!]) \vee (v \in [\![T]\!]^{\complement})$.

Proof. An proof by injunction on T, that for any v, either $v \in [\![T]\!]$ or $v \in [\![T]\!]^{\complement}$. □

2.3 Properties of Semantic Subtyping

From the semantics of types as sets of semantic values, semantic subtyping $S <: T$ is a proof than if $v \in [\![S]\!]$, then $v \in [\![T]\!]$. Constructively this a dependent function:

$$(S <: T) \text{ if and only if } \forall v \,.\, (v \in [\![S]\!]) \to (v \in [\![T]\!])$$

for example `number` $<:$ `number?` since:

$$\forall v \,.\, (v \in [\![\mathtt{number}]\!]) \to (v \in [\![\mathtt{number?}]\!])$$

Subtyping can be viewed as a dependent function from $[\![T]\!]^{\complement}$ to $[\![S]\!]^{\complement}$

Lemma 2.3. $(S <: T)$ *if and only if* $\forall v \,.\, (v \in [\![T]\!]^{\complement}) \to (v \in [\![S]\!]^{\complement})$

Proof. For "if", for any v, if $v \in [\![S]\!]$, then by Lemma 2.2 either $v \in [\![T]\!]$ or $v \in [\![T]\!]^{\complement}$. In the first case, $v \in [\![T]\!]$ as needed. In the second case $v \in [\![T]\!]^{\complement}$ and so by "if" hypothesis $v \in [\![S]\!]^{\complement}$, but by Lemma 2.1 we have a contradiction from $v \in [\![S]\!]$ and $v \in [\![S]\!]^{\complement}$. So we have established that $S <: T$.

For "only if", for any v, if $v \in [\![T]\!]^{\complement}$, then by Lemma 2.2 either $v \in [\![S]\!]$ or $v \in [\![S]\!]^{\complement}$. In the first case $v \in [\![S]\!]$ and so by "only if" hypothesis, $v \in [\![T]\!]$, but by Lemma 2.1 we have a contradiction from $v \in [\![T]\!]$ and $v \in [\![T]\!]^{\complement}$. In the second case, $v \in [\![s]\!]^{\complement}$ as needed. So we have established that $\forall v \,.\, (v \in [\![T]\!]^{\complement}) \to (v \in [\![S]\!]^{\complement})$. □

More interestingly is the constructive presentation of *anti-subtyping* $S \not<: T$. Normally this is presented negatively, but it can be read constructively since $S \not<: T$ is witnessed by a value v where $v \in [\![S]\!]$ but $v \in [\![T]\!]^{\complement}$.

$$(S \not<: T) \text{ if and only if } \exists v \,.\, (v \in [\![S]\!]) \wedge (v \in [\![T]\!]^{\complement})$$

for example `number?` $\not<:$ `number` since we can pick our witness v to be `nil`:

$$\mathtt{nil} \in [\![\mathtt{number?}]\!] \quad \mathtt{nil} \in [\![\mathtt{number}]\!]^{\complement}$$

Now, by Lemma 2.1, it is direct that $S \not<: T$ is a contradiction of $S <: T$:

Lemma 2.4. $(S \not<: T) \rightarrow \neg(S <: T)$

Proof. The is a witness for $S \not<: T$, $v \in [\![S]\!]$ and $v \in [\![T]\!]^{\complement}$, and so $S <: T$ and $v \in [\![S]\!]$ gives $v \in [\![T]\!]$. So Lemma 2.1 gives $v \in [\![T]\!]^{\complement}$ and $v \in [\![T]\!]$ are a contradiction is required. □

Unfortunately, this does not give a decision procedure for subtyping, for the usual reason that it tricky to build an algorithm for checking semantic subtyping, which requires type normalization [?]. We will return to this in §2.6.

It is direct to show that <: is transitive.

Lemma 2.5. $(S <: T) \wedge (T <: U) \rightarrow (S <: U)$

Proof. The there must be f maps an argument with type $v \in [\![S]\!]$ to a result with type $v \in [\![T]\!]$, and there must be g maps an argument with type $v \in [\![T]\!]$ to a result with type $v \in [\![U]\!]$. So $f; g$ maps an argument with type $v \in [\![S]\!]$ to a result with type $v \in [\![U]\!]$. □

More interestingly there is a dual property for $\not<:$. Classically this is the same as transitivity, just stated in terms of $\not<:$ rather than $<:$. But constructively this is a choice function, that states that if $S \not<: U$ then for any T we have a witness for either $S \not<: T$ or $T \not<: U$. For example `number?` $\not<:$ `number` (witnessed by `nil`), so for a mid-point `string` we have `nil` $\in [\![$`string`$]\!]^{\complement}$ which means we chose the constructive anti-subtype `number?` $\not<:$ `string` witnessed by `nil` $\in [\![$`number?`$]\!]$ and `nil` $\in [\![$`string`$]\!]^{\complement}$.

Lemma 2.6. $(S \not<: U) \rightarrow (S \not<: T) \vee (T \not<: U)$

Proof. $(S \not<: U)$ must have a witness v where $v \in [\![S]\!]$ and $v \in [\![U]\!]^{\complement}$. Now by Lemma 2.2 (the decision procedure for type semantics) we either have $v \in [\![T]\!]$ or $v \in [\![T]\!]^{\complement}$. In the first case, $v \in [\![T]\!]$ and $v \in [\![U]\!]^{\complement}$, which witnesses $T \not<: U$. In the second case, $v \in [\![S]\!]$ and $v \in [\![T]\!]^{\complement}$, which witnesses $S \not<: T$. In either case, we have a decision procedure for $(S \not<: T) \vee (T \not<: U)$. □

2.4 Co- and Contra-variance of Subtyping for Functions

We now turn to co- and contra-variant subtyping of functions. These come in two flavors: when function types are introduced, and when function types are eliminated. When a function type is introduced, we we check that the arguments respect contravariant subtyping, and that the results respect covariant subtyping.

Lemma 2.7. *If $S' <: S$ and $T <: T'$ then $(S \rightarrow T) <: (S' \rightarrow T')$*

Proof. If $u \in [\![S \rightarrow T]\!]$ then from Fig 3, either

- $u = (a \mapsto (w))$ and $w \in [\![T]\!]$, so $T <: T'$ implies $w \in [\![T']\!]$, and so $(a \mapsto (w)) \in [\![S' \rightarrow T']\!]$,
- $u = ((v) \mapsto r)$ and $v \in [\![S]\!]^{\complement}$, so $S' <: S$ and Lemma 2.3 implies $v \in [\![S']\!]^{\complement}$, and so $((v) \mapsto r) \in [\![S' \rightarrow T']\!]$,
- $u = (a \mapsto \text{diverge})$, so $(a \mapsto \text{diverge}) \in [\![S' \rightarrow T']\!]$, or
- $u = () \mapsto \text{check}$, so $(() \mapsto \text{check}) \in [\![S' \rightarrow T']\!]$.

In any case, $u \in [\![S' \rightarrow T']\!]$. □

When a function type is eliminated, we we check that the arguments reflect contravariant subtyping, and that the results reflect covariant subtyping.

Lemma 2.8. *If $(S \rightarrow T) <: (S' \rightarrow T')$ then $S' <: S$ and $T <: T'$*

Proof. If $v \in [\![S]\!]^{\complement}$ then from Fig 3 $((v) \mapsto \text{check}) \in [\![S \to T]\!]$, so since $(S \to T) <: (S' \to T')$, we have $((v) \mapsto \text{check}) \in [\![S' \to T']\!]$ and so $v \in [\![S']\!]^{\complement}$. Thus, using Lemma 2.3, we have established $S' <: S$.

If $w \in [\![T]\!]$ then from Fig 3 $(() \mapsto (w)) \in [\![S \to T]\!]$, so since $(S \to T) <: (S' \to T')$, we have $(() \mapsto (w)) \in [\![S' \to T']\!]$ and so $w \in [\![T']\!]$. Thus we have established $T <: T'$. □

Note that these Lemmas rely on pragmatic semantic subtyping. Lemma 2.7 is true for set-theoretic semantic subtyping, but Lemma 2.8 is only true for set-theoretic models when S and T are inhabited types. In pragmatic semantic subtyping, we do not have special corner cases, in particular Lemma 2.8 is true for all types, and does not require special cases about inhabitance.

2.5 Distribution of Intersection and Union Over Functions

Finally, we turn to cases where intersection and union distribute through functions. Since Luau uses intersections of functions as types for overloaded functions, it is unsurprising that intersection does not in general distribute through functions. For example:

$$(\texttt{boolean} \to \texttt{boolean}) \cap (\texttt{number} \to \texttt{number})$$

does not distribute to:

$$(\texttt{boolean} \cup \texttt{number}) \to (\texttt{boolean} \cap \texttt{number})$$

For example:

$$\begin{aligned}(\texttt{false} \mapsto \texttt{true}) &\in [\![(\texttt{boolean} \to \texttt{boolean}) \cap (\texttt{number} \to \texttt{number})]\!]\\ (\texttt{false} \mapsto \texttt{true}) &\in [\![(\texttt{boolean} \cup \texttt{number}) \to (\texttt{boolean} \cap \texttt{number})]\!]^{\complement}\end{aligned}$$

is a witness for:

$$\begin{aligned}&((\texttt{boolean} \to \texttt{boolean}) \cap (\texttt{number} \to \texttt{number}))\\ &\not<: ((\texttt{boolean} \cup \texttt{number}) \to (\texttt{boolean} \cap \texttt{number}))\end{aligned}$$

Now in general we can not distribute intersection through functions, on both the left and right, but we can distribute on just left, and just right. This is similar to the situation with premonoids categories [?] which are functorial on both sides, but are not binary functorials.

Lemma 2.9.

(1) $[\![(S_1 \to T) \cap (S_2 \to T)]\!] = [\![(S_1 \cup S_2) \to T]\!]$
(2) $[\![(S \to T_1) \cap (S \to T_2)]\!] = [\![S \to (T_1 \cap T_2)]\!]$

Proof.

(1⇒) If $u \in [\![S_1 \to T]\!] \cap [\![S_2 \to T]\!]$, then from Fig 3, either
- $u = (a \mapsto (w))$ and $w \in [\![T]\!]$, and so $(a \mapsto (w)) \in [\![(S_1 \cup S_2) \to T]\!]$,
- $u = ((v) \mapsto r)$, $v \in [\![S_1]\!]^{\complement}$ and $v \in [\![S_2]\!]^{\complement}$, so $v \in [\![S_1 \cup S_2]\!]^{\complement}$, and so $((v) \mapsto r) \in [\![(S_1 \cup S_2) \to T]\!]$,
- $u = (a \mapsto \text{diverge})$, so $(a \mapsto \text{diverge}) \in [\![(S_1 \cup S_2) \to T]\!]$, or
- $u = () \mapsto \text{check}$, so $(() \mapsto \text{check}) \in [\![(S_1 \cup S_2) \to T]\!]$.

In any case, $u \in [\![(S_1 \cup S_2) \to T]\!]$.

(1⇐) If $u \in [\![(S_1 \cup S_2) \to T]\!]$, then from Fig 3, either
- $u = (a \mapsto (w))$ and $w \in [\![T]\!]$, and so $(a \mapsto (w)) \in [\![S_1 \to T]\!] \cap [\![S_2 \to T]\!]$,
- $u = ((v) \mapsto r)$, $v \in [\![S_1]\!]^{\complement} \cap [\![S_2]\!]^{\complement}$ and so $((v) \mapsto r) \in [\![S_1 \to T]\!] \cap [\![S_2 \to T]\!]$,,
- $u = (a \mapsto \text{diverge})$, so $(a \mapsto \text{diverge}) \in [\![S_1 \to T]\!] \cap [\![S_2 \to T]\!]$, or
- $u = () \mapsto \text{check}$, so $(() \mapsto \text{check}) \in [\![S_1 \to T]\!] \cap [\![S_2 \to T]\!]$.

In any case, $u \in [\![S_1 \to T]\!] \cap [\![S_2 \to T]\!]$.

(2⟹) If $u \in [\![S \to T_1]\!] \cap [\![S \to T_2]\!]$, then from Fig 3, either

- $u = (a \mapsto (w))$, $w \in [\![T_1]\!]$ and $w \in [\![T_2]\!]$, and so $(a \mapsto (w)) \in [\![S \to (T_1 \cap T_2)]\!]$,
- $u = ((v) \mapsto r)$ and $v \in [\![S]\!]^{\complement}$ so $v \in [\![S]\!]^{\complement}$, and so $((v) \mapsto r) \in [\![S \to (T_1 \cap T_2)]\!]$,
- $u = (a \mapsto \text{diverge})$, so $(a \mapsto \text{diverge}) \in [\![S \to (T_1 \cap T_2)]\!]$, or
- $u = () \mapsto \text{check}$, so $(() \mapsto \text{check}) \in [\![S \to (T_1 \cap T_2)]\!]$.

In any case, $u \in [\![(S_1 \cup S_2) \to T]\!]$.

(2⟹) If $u \in [\![(S_1 \cup S_2) \to T]\!]$, then from Fig 3, either

- $u = (a \mapsto (w))$, $w \in [\![T_1]\!]$ and $w \in [\![T_2]\!]$, and so $(a \mapsto (w)) \in [\![S \to T_1]\!] \cap [\![S \to T_2]\!]$,
- $u = ((v) \mapsto r)$ and $v \in [\![S]\!]^{\complement}$ so $v \in [\![S]\!]^{\complement}$, and so $((v) \mapsto r) \in [\![S \to T_1]\!] \cap [\![S \to T_2]\!]$,
- $u = (a \mapsto \text{diverge})$, so $(a \mapsto \text{diverge}) \in [\![S \to T_1]\!] \cap [\![S \to T_2]\!]$, or
- $u = () \mapsto \text{check}$, so $(() \mapsto \text{check}) \in [\![S \to T_1]\!] \cap [\![S \to T_2]\!]$.

In any case, $u \in [\![S \to T_1]\!] \cap [\![S \to T_2]\!]$.

This provides cases. □

In contract, union does distribute over functions.

Lemma 2.10. $[\![(S_1 \to T_1) \cup (S_2 \to T_2)]\!] = [\![(S_1 \cap S_2) \to (T_1 \cup T_2)]\!]$

Proof.

(⟹) If $u \in [\![S_1 \to T_1]\!] \cup [\![S_2 \to T_2]\!]$, then from Fig 3, either

- $u = (a \mapsto (w))$, $w \in [\![T_1]\!]$, and so $(a \mapsto (w)) \in [\![(S_1 \cap S_2) \to (T_1 \cup T_2)]\!]$,
- $u = (a \mapsto (w))$, $w \in [\![T_2]\!]$, and so $(a \mapsto (w)) \in [\![(S_1 \cap S_2) \to (T_1 \cup T_2)]\!]$,
- $u = ((v) \mapsto r)$, $v \in [\![S_1]\!]^{\complement}$ and so $((v) \mapsto r) \in [\![(S_1 \cap S_2) \to (T_1 \cup T_2)]\!]$,
- $u = ((v) \mapsto r)$, $v \in [\![S_2]\!]^{\complement}$ and so $((v) \mapsto r) \in [\![(S_1 \cap S_2) \to (T_1 \cup T_2)]\!]$,
- $u = (a \mapsto \text{diverge})$, so $(a \mapsto \text{diverge}) \in [\![(S_1 \cap S_2) \to (T_1 \cup T_2)]\!]$, or
- $u = () \mapsto \text{check}$, so $(() \mapsto \text{check}) \in [\![(S_1 \cap S_2) \to (T_1 \cup T_2)]\!]$.

In any case, $u \in [\![(S_1 \cap S_2) \to (T_1 \cup T_2)]\!]$.

(⟸) If $u \in [\![(S_1 \cap S_2) \to (T_1 \cup T_2)]\!]$, then from Fig 3, either

- $u = (a \mapsto (w))$, $w \in [\![T_1]\!]$, and so $(a \mapsto (w)) \in [\![S_1 \to T_1]\!] \cup [\![S_2 \to T_2]\!]$,
- $u = (a \mapsto (w))$, $w \in [\![T_2]\!]$, and so $(a \mapsto (w)) \in [\![S_1 \to T_1]\!] \cup [\![S_2 \to T_2]\!]$,
- $u = ((v) \mapsto r)$, $v \in [\![S_1]\!]^{\complement}$ and so $((v) \mapsto r) \in [\![S_1 \to T_1]\!] \cup [\![S_2 \to T_2]\!]$,
- $u = ((v) \mapsto r)$, $v \in [\![S_2]\!]^{\complement}$ and so $((v) \mapsto r) \in [\![S_1 \to T_1]\!] \cup [\![S_2 \to T_2]\!]$,
- $u = (a \mapsto \text{diverge})$, so $(a \mapsto \text{diverge}) \in [\![S_1 \to T_1]\!] \cup [\![S_2 \to T_2]\!]$, or
- $u = () \mapsto \text{check}$, so $(() \mapsto \text{check}) \in [\![S_1 \to T_1]\!] \cup [\![S_2 \to T_2]\!]$.

In any case, $u \in [\![S_1 \to T_1]\!] \cup [\![S_2 \to T_2]\!]$.

This provides cases. □

Now, we have that the Luau pragmatic semantic subtyping unions distinction, the type normal for functions is:

$$(S_1 \to T_1) \cap \cdots \cap (S_N \to T_N)$$

This the most important feature of type normalization.

2.6 Type Normalization

.

3 IMPLEMENTATION OF FULL LUAU

Stuff features

- Functions with arity.

- Functions with vararg.
- Tables.
- Tables with indexers.
- Tables with tagged unions.
- Singleton types.
- Genetics.
- FBI.

Stuff imp's

- How we implement type normalization.
- Aim for type inference in 16ms.
- Telemetry with very few `CodeTooComplex`.
- SAT solves.

4 FURTHER WORK

REFERENCES

[1] L. Brown, A. Friesen, and A. S. A. Jeffrey. 2023. Goals of the Luau Type System: Two Years Only. In *Proc. Human Aspects of Types and Reasoning Assistants*. https://asaj.org/papers/hatra23.pdf

[2] L. Brown and A. S. A. Jeffrey. 2023. Luau Prototype Typechecker. https://github.com/luau-lang/agda-typeck

[3] G. Castagna and A. Frisch. 2005. A Gentle Introduction to Semantic Subtyping. In *Proc. Principles and Practice of Declarative Programming*.

[4] A. Frisch, G. Castagna, and V. Benzaken. 2008. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM* 55, 19 (2008).

[5] A. S. A. Jeffrey. 2022. Semantic Subtyping in Luau. Roblox Technical Blog. https://blog.roblox.com/2022/11/semantic-subtyping-luau/

[6] A. M. Kent. 2021. Down and Dirty with Semantic Set-theoretic Types (a tutorial). https://pnwamk.github.io/sst-tutorial/

[7] Roblox. 2022. Roblox Coporation 2022 Annual Report. https://ir.roblox.com/financials/annual-reports/default.aspx