Gradual Types as Error Suppression

A Constructive View of Type Warnings

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

This paper presents the view of gradual typing adopted by the Luau programming language. Prior work on gradual typing has been based on *type compatibility*, that is a relation on types $T \sim U$ given by contextually closing $T \sim \mathsf{any} \sim U$. Type systems based on type compatibility use $\sim$ rather than type equivalence (or a similar presentation for languages with subtyping). We take a different tack, which is to view type warnings *constructively* as a proof object $\mathsf{Warn}(\Gamma \vdash M : T)$ saying that the type derivation $\Gamma \vdash M : T$ should generate a warning. Viewing type warnings constructively allows us to talk about *error suppression*, for example type errors involving the the any type are suppressed, and so this type system is gradual in the sense that developers can explicitly annotate terms with the any type to switch off type warnings. This system has the usual "well-typed programs don't go wrong" result for program which do not have explicit type annotations with error suppressing types, except this property can now be stated as the presence of $\mathsf{Warn}(\Gamma \vdash M : T)$ rather than the absence of a run-time error. 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. 2025. Gradual Types as Error Suppression: A Constructive View of Type Warnings. *Proc. ACM Program. Lang.* 9, POPL, Article ?? (January 2025), 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 Gradual Typing

The aim of *gradual typing* [8, 9] is to allow a code base to migrate from being untyped to being typed. This is achieved by introducing a type any (also called ? or $*$) which is used as the type of a expression which is not subject to type checking. For example, in Luau, a variable can be declared as having type any, which is not subject to type checking:

```
local x : any = "hi"
print(math.abs(x))
```

This program generates a run-time error, but because x is declared has having type any, no type error is generated. Similarly, any expression can be cast to having type any, which is not subject to type checking:

```
print(math.abs("hi" :: any))
```

Again, this program generates a run-time error, but because "hi" is cast to having type any, no type error is generated.

Prior work on gradual typing has been based on *type compatibility*, that is a relation on types $T \sim U$ given by contextually closing $T \sim \mathsf{any} \sim U$. Type systems based on type compatibility use $\sim$ rather than type equivalence (or a similar presentation for languages with subtyping).

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

2475-1421/2025/1-ART?? $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

For example, the type rules for function application in [8] are:

$$\frac{\Gamma \vdash M : \mathsf{any} \quad \Gamma \vdash N : U}{\Gamma \vdash M(N) : \mathsf{any}}(\textsc{GApp1}) \qquad \frac{\Gamma \vdash M : (S \to T) \quad \Gamma \vdash N : U \quad S \sim U}{\Gamma \vdash M(N) : T}(\textsc{GApp2})$$

which requires that the argument type is *compatible* with (rather than equal to) to the function source type. For example:

$$\frac{\Gamma \vdash \mathtt{math.abs} : (\mathtt{number} \to \mathtt{number}) \quad \Gamma \vdash x : \mathtt{any} \quad \mathtt{number} \sim \mathtt{any}}{\Gamma \vdash \mathtt{math.abs}(x) : \mathtt{number}}$$

The problem we discovered in implementing gradual typing on top of type compatibility is that it is a source of subtle bugs, because the type system is very sensitive as to when type equality is used rather than type compatibility. For example comparing two type rules:

$$\frac{\Gamma \vdash M : F \quad \Gamma \vdash N : U \quad F = (S \to T) \quad S \sim U}{\Gamma \vdash M(N) : T}(\textsc{GApp2}') \qquad \frac{\Gamma \vdash M : F \quad \Gamma \vdash N : U \quad F \sim (S \to T) \quad S \sim U}{\Gamma \vdash M(N) : T}(\textsc{GApp2}'')$$

These rules only differ in whether they use type compatibility rater than equality, but they have very different semantics. Rule GApp2′ is the same as GApp2′ (and so is sound) but using GApp2″ we can derive:

$$\frac{\Gamma \vdash \mathtt{math.abs} : (\mathtt{number} \to \mathtt{number}) \quad \Gamma \vdash x : \mathtt{string} \quad (\mathtt{number} \to \mathtt{number}) \sim (\mathtt{any} \to \mathtt{number}) \quad \mathtt{any} \sim \mathtt{string}}{\Gamma \vdash \mathtt{math.abs}(x) : \mathtt{number}}$$

which is unsound. Problems like this come down eventually to the fact that ∼ is not transitive, and in the presence of other features such as unification and subtyping gave rise to subtle bugs (for example code that assumed that unification solved for type equality rather than compatibility).

1.2 Error Suppressing Types

There as been a long history of improving type errors reported to users, going back to the 1980s [5, 11]. One source of pain in type error reporting is *cascading* type errors, for example:

```
local x = "hi"
local y = math.abs(x)
local z = string.lower(y)
```

In this case `math.abs(x)` should generate a type error, since the type `string` is inferred for `x`, and the type of `math.abs` is `number` → `number`. It is not obvious whether a type error should be generated for `string.lower(y)`. If the type `number` is inferred for `y`, then an error should be reported, since the type of `string.lower` is `string` → `string`. But this will not be the best user experience, since this will give a common experience of multiple cascading type errors, of which only the first error is genuine.

One heuristic to eliminate cascading errors is to mark the type of any expression which causes a type error to be emitted as *error suppressing*. Error suppressing types are then used to percolate the information that a type error has already been generated, and so avoid cascading type errors.

For example, in Luau, a type `error` is introduced, and any type T which is a supertype of `error` is considered to be error suppressing. For instance, the types inferred for the above program are:

```
local x : string = "hi"
local y : number | error = math.abs(x)
local z : string | error = string.lower(y)
```

Since `number | error` is an error suppressing type, `string.lower(y)` will not report a type error.

Error suppressing types as a technique for minimizing cascading type errors appears to be folklore, for example it is implemented in Typed Racket [10], but does not appear to have been academically published.

1.3 Gradual Typing via Error Suppressing Types

At this point a reader might guess where this paper is going. Gradual types are well established research area which allows programmers to type expressions with type `any` to suppress type errors. Error suppressing types are well established folklore which allows type inference to type expressions with error suppressing types to suppress cascading type errors.

The connection between these two areas is pretty obvious, but has not been made before. If `any` is an error suppressing type, then gradual typing is very similar to error suppression, but does not require the non-transitive type compatibility relation.

In Luau, the use of error suppressing types explains why Luau, in common with TypeScript [7], has both an error suppressing type `any` and a non-error suppressing type `unknown`. `any` is the top type, and `unknown` is the top non-error-suppressing type. We consider `any` to be equivalent to `unknown | error`.

1.4 Constructive Type Errors

Error suppressing types is a folklore implementation technique for suppressing cascading type errors. In this paper we present a formalization as a constructive model of type errors. We do this by separating out type derivations from type errors.

For example, one traditional way to present the type rule for function application is:

$$\frac{\begin{array}{l}\Gamma \vdash M : T \\ \Gamma \vdash N : U \\ U <: \mathsf{src}(T)\end{array}}{\Gamma \vdash M(N) : \mathsf{apply}(T, U)}$$

using functions to calculate the domain of a function $\mathsf{src}(T)$, for example following [6, §5.2]:

$$\mathsf{src}(S \to T) = S \quad \mathsf{src}(S \cap T) = \mathsf{src}(S) \cup \mathsf{src}(T) \quad \cdots$$

and to calculate the result type of applying a function $\mathsf{apply}(T, U)$, for example following [6, §5.3]:

$$\mathsf{apply}(S \to T, U) = \begin{cases} T & \text{if } U <: T \\ \mathsf{any} & \text{otherwise} \end{cases} \qquad \mathsf{apply}(S \cap T, U) = \mathsf{apply}(S, U) \cap \mathsf{apply}(T, U) \quad \cdots$$

(The details of this function are spelled out in Section 2.)

This formulation is fine if is one is only interested in well typed programs, and in not in the behavior of badly typed programs. For example, the classic "well typed programs don't go wrong" can be stated as:

$$(M \to^* M') \to (\mathsf{RunTimeErr}(M')) \to \neg(\emptyset \vdash M : T)$$

(for an appropriate definition of $\mathsf{RunTimeErr}(M')$). Now, this does not have a constructive model of type warnings, since we are only considering badly typed programs as ones where $\neg(\emptyset \vdash M : T)$.

In this paper, we separate type derivations from their type errors. This is done by explicitly tracking the derivation tree for typing, for example:

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

which allows us to define which type derivations generate warnings, for example there are three ways a warning can be generated from a function application $M(N)$, bubbling up a warning from M or N, or by a failure of subtyping:

$$\frac{\mathsf{Warn}(D_1)}{\mathsf{Warn}(\mathsf{app}(D_1, D_2))} \qquad \frac{\mathsf{Warn}(D_2)}{\mathsf{Warn}(\mathsf{app}(D_1, D_2))} \qquad \frac{U \not<: \mathsf{src}(T)}{\mathsf{Warn}(\mathsf{app}(D_1, D_2))}$$

This is based on a constructive framing of failure of subtyping, fleshed out in Section 2:

$$\frac{\begin{array}{c}v \in [\![T]\!] \\ v \in [\![U]\!]^{\complement}\end{array}}{T \not<: U}$$

Now, there are two important results about this presentation of constructive type errors. The first is *infallible typing*, that every program can be type checked in every context:

$$\forall \Gamma, M \,.\, \exists T \,.\, (\Gamma \vdash M : T)$$

(We write $\mathsf{typeof}(\Gamma, M)$ for this derivation tree.) The second is that we can state "well typed programs don't go wrong" constructively:

$$(M \longrightarrow^* M') \longrightarrow (\mathsf{RunTimeErr}(M')) \longrightarrow \mathsf{Warn}(\mathsf{typeof}(\emptyset, M))$$

Since this is phased constructively, Curry Howard means we can think of this statement in two ways:

- as a proof that "well typed programs don't go wrong"
- as a time-travel debugger, that for any execution $(M \longrightarrow^* M')$ where M' has a run-time error, it can be run back in time to find a root cause type error for M'.

Constructive type errors can be easily adapted to error suppressing types, for example we only report a failure of subtyping when both the type of the function and the type of argument are not error suppressing:

$$\frac{\begin{array}{c}\mathtt{error} \not<: T \\ \mathtt{error} \not<: U \\ U \not<: \mathsf{src}(T)\end{array}}{\mathsf{Warn}(\mathsf{app}(D_1, D_2))}$$

Now, in general this breaks type soundness, because if every type is any then all type errors are suppressed. But we can show that in the case of a program in which no types are error suppressing, "well typed programs don't go wrong".

1.5 Contributions

In summary, this paper combines two well-explored areas:

- gradual types, and
- error suppressing types.

The new contributions of this paper are:

- combining gradual types and error suppressing types,
- formalizing error suppressing types as constructive type errors,
- presenting a new *pragmatic* model of semantic subtyping, and
- showing type soundness for programs which do not contain error suppressing types.

The results are mechanized in Agda [1]. The concepts in this paper are released as part of Roblox Studio, used by millions of creators.

2 PRAGMATIC SEMANTIC SUBTYPING

2.1 What is pragmatic semantic subtyping

In Luau, we use a variant of semantic subtyping [2, 4, 6]. The important properties of semantic subtyping are:

- there is a set $\mathcal{D}$ of semantic values,
- each type T has a semantics $[\![T]\!] \subseteq \mathcal{D}$,
- `any` 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 $[\![T]\!] \subseteq [\![U]\!]$.

In addition, the off-the-shelf presentation of semantic subtyping is *set theoretic* [2, §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 ($\texttt{never} \to T$) are identified, since:

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

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

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

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

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

  so we have to accept that in a set-theoretic model, the type rule for function application has corner cases for uninhabited types.
- 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 1D and 2D matrices:

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

Fig. 1. Semantic values

```
mul : (Matrix1D , Matrix1D) -> Matrix1D
    & (Matrix1D , Matrix2D) -> Matrix1D
    & (Matrix2D , Matrix1D) -> Matrix1D
    & (Matrix2D , Matrix2D) -> Matrix2D
```

Overloaded functions are a key part of the Roblox API, and we might expect that all function types can be presented as overloaded functions. We can do this if we can present unions of overloaded functions as overloaded functions. Now, union distributes through intersection, so all that is required is to 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:

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

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

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

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].

In addition, Luau does not support negation of all types, but only negation of *test types* [3], 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.

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.

2.2 Semantic values for Luau

In this presentation, 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:

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

The types we are considering are:

$$S, T ::= b \mid \texttt{error} \mid \texttt{any} \mid \texttt{never} \mid S \to T \mid S \cap T \mid S \cup T$$

which are:

- the *scalar types* b
- the *error-suppressing type* `error`,
- the *anything type* `any`,
- the *uninhabited type* `never`,
- a *function* type $S \to T$,
- 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 1. Semantic values are:

- *scalar values* s,
- *error values* `error`, witnessing a error-suppressing type, 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` ∪ `nil`,
- `true` and `false` and `error` are values in the error-suppressing type `boolean` ∪ `error`, and
- (`true`) ↦ (`false`) is a value in the function type `boolean` → `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:

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

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:

$$((5) \mapsto (37)) \in [\![\texttt{boolean} \to \texttt{boolean}]\!]$$

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

$$((\texttt{true}) \mapsto (37)) \notin [\![\texttt{boolean} \to \texttt{boolean}]\!]$$

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 $a \mapsto$ diverge)., for example:

$$((\texttt{true}) \mapsto \texttt{diverge}) \in [\![\texttt{boolean} \to \texttt{boolean}]\!]$$

and:

$$((5) \mapsto \texttt{diverge}) \in [\![\texttt{boolean} \to \texttt{boolean}]\!]$$

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

$$((5) \mapsto \texttt{check}) \in [\![\texttt{boolean} \to \texttt{boolean}]\!]$$

but:

$$((\texttt{true}) \mapsto \texttt{check}) \notin [\![\texttt{boolean} \to \texttt{boolean}]\!]$$

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

Fig. 2. 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:

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

and:

$$(() \mapsto \text{diverge}) \in [\![\texttt{boolean} \to \texttt{boolean}]\!]$$

and:

$$(() \mapsto \text{check}) \in [\![\texttt{boolean} \to \texttt{boolean}]\!]$$

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:

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

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.

2.3 Semantics of Luau type

The semantics of Luau types are given in Fig 2. This semantics is presented mechanically in Agda in [1], 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 3.

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* $(v \notin [\![T]\!])$.

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

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

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

Fig. 3. Complemented semantics of types as sets of values

.

2.4 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:

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

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`:

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

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

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

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 §2.5.

It is direct to show that $<:$ is transitive.

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

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 $\texttt{number?} \not<: \texttt{number}$ (witnessed by $\texttt{nil}$), so for a mid-point $\texttt{string}$ we have $\texttt{nil} \in [\![\texttt{string}]\!]^{\complement}$ which means we chose the constructive anti-subtype $\texttt{number?} \not<: \texttt{string}$ witnessed by $\texttt{nil} \in [\![\texttt{number?}]\!]$ and $\texttt{nil} \in [\![\texttt{string}]\!]^{\complement}$.

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

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 $(S \not<: T) \vee (T \not<: U)$. □

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* $(S \to T) <: (S' \to T')$

Proof. If $u \in [\![S \to T]\!]$ then from Fig 2, either

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

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

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* $(S \to T) <: (S' \to T')$ *then* $S' <: S$ *and* $T <: T'$

Proof. If $v \in [\![S]\!]^{\complement}$ then from Fig 2 $((v) \mapsto \text{check}) \in [\![S \to T]\!]$, 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 2 $(() \mapsto (w)) \in [\![S \to T]\!]$, 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 sematic subtyping, we do not have special corner cases, in particulr Lemma 2.8 is true for all types, and does not require special cases about inhabitance.

2.5 Type normalization

.

3 FURTHER WORK

REFERENCES

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

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

[3] G. Castagna, M. Laurent, K. Nguyễn, and M Lutze. 2022. On Type-Cases, Union Elimination, and Occurrence Typing. *Proc. ACM Program. Lang.* 6, POPL, Article 13 (2022), 31 pages. https://doi.org/10.1145/3498674

[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] G. F. Johnson and J. A. Walz. 1986. A Maximum Flow Approach to Anomaly Isolation. 44–57. https://doi.org/10.1145/512644.512649

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

[7] Microsoft. 2023. TypeScript. https://www.typescriptlang.org/

[8] J. G. Siek and W. Taha. 2006. Gradual Typing for Functional Languages. In *Proc. Scheme and Functional Programming Workshop*. 81–92.

[9] J. G. Siek and W. Taha. 2007. Gradual Typing for Objects. In *Proc. European Conf Object-Oriented Programming*. 2–27.

[10] S. Tobin-Hochstadt. 2008. New Error Handling for Type Parsing Errors. https://github.com/racket/racket/commit/3a9928474523b042f83a7a707346daa01ef63899 Commit to Typed Racket.

[11] M. Wand. 1986. Finding the Source of Type Errors. In *Proc. Principles of Programming Languages*. 38–43. https://doi.org/10.1145/512644.512648