Flow-Sensitive Typing and Bounded Polymorphism

August 27, 2020

Abstract

abstract

1 Introduction

(TODO: concretize 'the main use case')

Flow-sensitive typing is a branch of type systems where the type of a variable can differ in multiple locations within the program. The Whiley language is typically considered the origin of flow-sensitive typing, and its impact is visible in other JVM languages such as Ceylon and Kotlin. This style of type system has recently also become popular when added on top of untyped languages such as TypeScript, Flow and Dart (TODO: more? cite?).

For the most part, the flow-sensitivity is a touted feature of the various languages employing them. However, there are still unsolved interactions with other features. A prominent and recent example is the interaction of flow-sensitivity with bounded polymorphism and indexed access types in TypeScript. Previously, it was possible to create unsoundness by combining these features. However, a recent update (TODO: ref 3.5 update, relevant issue: https://github.com/microsoft/TypeScript/pull/30769) which plugged an unsoundness hole related to indexed access types, also made a certain TypeScript pattern inexpressible without unsafe type assertions. Neither reverting to the old behaviour, nor keeping the current behaviour seems satisfactory. Therefore, we believe that it is time to create a thorough study on the interaction between these features.

In this paper, we discuss this problem by considering various alternatives and propose an argument for the preferred extension. In addition to this, we study this extension in traditional fashion by providing a minimal calculus and proving relevant properties about it.

Specifically, the contributions of this paper are:

- We present a novel calculus incorporating the various features related to the main use case: flow-sensitive if statements, union types, string literal

types and simple type-level functions (corresponding to indexed access types). The calculus is an extension to the base calculus system F-Sub.

- We contribute mechanized proofs, in the form of Agda code, of the relevant properties for this calculus.
- We propose a solution to the problem of interaction between these features as present in the main use case.

2 Motivation

In this section, we first develop some background on relevant type system features in TypeScript and then present the main use case.

(TODO: ref https://www.typescriptlang.org/docs/handbook/advanced-types.html)

2.1 String Literal Types

String literal types allow you to annotate the exact value of a string at the type-level. For example, if we want the variable `cat` to only hold the string `"cat"`, then we can state `let cat: "cat"`. Assigning any other string than `"cat"` to `cat`, for example `cat = "dog"`, is a type error.

2.2 Union Types

Union types represent an untagged sum type. They are used to represent a type which can be either one type or another type, but there is no runtime tag which specifies which of the two it is. For example, if we have a variable which can either be the string `"cat"` or `"dog"`, then we can state that `let catOrDog: "cat"| "dog"`. This uses the union type operator `|` combined with the string literal types `"cat"` and `"dog"`. Both `catOrDog = "cat"` and `catOrDog = "dog"` are well-typed, but `catOrDog = "bird"` is not.

2.3 Flow-Sensitive If Statement

A flow-sensitive if statement narrows the types in the scope of the then and else branch dependent on the tests in the statement. For example, if we start with `catOrDog`, which has type `"cat" | "dog"`, then we can test whether it is actually `"cat"`. Within the then branch, the type of `catOrDog` is narrowed to `"cat"`, while in the else branch it is narrowed to `"dog"`.

```
let catOrDog: "cat" | "dog";
if (catOrDog === "cat") {
  // catOrDog is narrowed to type "cat" here
} else {
  // catOrDog is narrowed to type "dog" here
}
```

2.4 Indexed Access Types

An indexed access type is a specific form of type-level function. It allows to retrieve the type of a record type at a specific index, at the type-level. For example, if we have a record type `type Cat = { lives: number, tag: "cat" }`, then we can query the type of the key `lives` as `type Lives = Cat["lives"]` which will be the type `number`.

2.5 Use Case

The main use case we want to consider is that of creating a function for which the output type depends on the value of the input. For example, in the `depFun` function below we have a function which takes a `"t"` or `"f"` string as input. The input value `str` does not have type `"t" | "f"` as might be expected, but instead has type `T` which is a type variable bounded by `"t" | "f"` (TODO: ref bounded quantification?). This is necessary since we want to refer to the type `T` in the result type. The result type, which is `F[T]`, encodes a type-level mapping for the output type dependent on the input type. In summary, the result type when the function is called as `depFun("t")` is `number`, and `boolean` when called as `depFun("f")`.

```
interface F {
  "t": number,
  "f": boolean,
}

function depFun<T extends "t" | "f">(str: T): F[T] {
  if (str === "t") {
    return 1;
  } else {
    return true;
  }
}

depFun("t"); // has type number
depFun("f"); // has type boolean
```

This style of pattern can be found in the official TypeScript bindings for certain browser APIs, for example the `querySelector` function[1]. The relevant code which utilizes this pattern is copied below.

```
interface HTMLElementTagNameMap {
  "a": HTMLAnchorElement;
  "abbr": HTMLElement;
  // ... and more
}

interface ParentNode {
  querySelector<K extends keyof HTMLElementTagNameMap>(
    selectors: K
  ): HTMLElementTagNameMap[K] | null;
}
```

[1] https://github.com/microsoft/TypeScript/blob/ca00b3248b1af2263d0223d68e792b7ca39abcab/lib/lib.dom.d.ts#L11050

The `HTMLElementTagNameMap` interface defines a record type containing the mapping of a HTML tag to the corresponding HTML element type resulting from the selection. The `keyof` operator is another type-level operator which returns all keys of a record type as a union of string literals. So in this case it corresponds to `"a" | "abbr" | ...`. With this pattern, callers of the `querySelector` function are given more precise type information on what the function will return.

3 Problems

This pattern is quite reminiscent of functions in dependently typed programming languages where output types can depend on the values of inputs. However, there are a few problems with it precisely because that is not exactly what the pattern expresses. In this section we illustrate what these problems are.

3.1 Problem in 3.4

In TypeScript 3.4, the `depFun` code given before would compile as is. However, it was actually because of allowed unsoundness that it was able to compile at all. The reason is that within the function, the result type of `depFun` is resolved to `string | number` throughout the whole function. Even within the flow-sensitive branches the return type stays unchanged.

```
// TypeScript 3.4
function depFun<T extends "t" | "f">(str: T): F[T] {
  if (str === "t") {
    // return type stays number | boolean
    return 1;
  } else {
    // return type stays number | boolean
    return true;
  }
}
```

This behaviour is clearly unsound: we can swap around the branches, which will still typecheck, but the caller of the function will observe wrong type information.

```
// TypeScript 3.4
function depFun<T extends "t" | "f">(str: T): F[T] {
  if (str === "t") {
    return true;
  } else {
    return 1;
  }
}

depFun("t"); // has type number, but is actually a boolean
depFun("f"); // has type boolean, but is actually a number
```

3.2 Problem in 3.5

In TypeScript 3.5 this behaviour changed, whenever an assignment is made to something with an indexed access type then it is resolved to an intersection type instead of a union type. A value of an intersection type `A & B` is considered to be both of type `A` and of type `B`. This intersection can resolve to a simplified version if applicable. For example, `"cat" & string` resolves to `"cat"`, but `number & boolean` is uninhabited and resolves to `never`. This fixes the soundness issue mentioned in the previous section.

However, the `depFun` example is impossible to implement now. The type `F[T]` is now considered an assignment to the result of the function. Thus, instead of the result type resolving to `number | boolean` it is instead resolved to `number & boolean` (and thus `never`), which makes the function unimplementable without resorting to unsafe type assertions.

```
// TypeScript 3.5
function depFun<T extends "t" | "f">(str: T): F[T] {
  if (str === "t") {
    return 1; // type error: 1 is not assignable to never
  } else {
    return true; // type error: true is not assignable to never
  }
}
```

As an aside, the browser API bindings are unaffected since they are merely TypeScript signatures on top of a JavaScript implementation. Meaning that the body of the function is never typechecked.

3.3 General Problem

What makes this problem somewhat tricky to solve is the fact that it is not quite a dependent function as is typical in dependently typed languages. The output type of the function is dependent on the *type* of one of its input variables, but *not the actual value of this input*. Of course, these are linked, but it is an important difference.

In fact, the intent of the function is that we query the record type `F` with the *value* of `str`. In other words, the return type we want to write is `F[str]`. In this case, it would be safe to narrow this return type in a branch where the actual value of `str` is checked.

Instead, however, we have to apply the trick as explained before, and the return type becomes `F[T]`, where `T` is the type of `str`. But, by doing this we have prevented the possibility of doing the narrowing we expected previously. Narrowing the type `T` to `"t"` in a branch where `str` has been checked is definitely unsound, illustrated in the following example.

In this example we have two inputs: `str` and `str2`. Checking the value of `str` does not tell us what the value of `str2` might be. So, within a branch where `str` is checked, it is unsafe to narrow `T` since `str2` also has type `T` but is possibly a different value.

```
function depFun2<T extends "t" | "f">(str: T, str2: T): F[T] {
```

```
  if (str === "t") {
    // if T is narrowed to "t", then str2 would have type "t" in
        this branch while it might actually be the value "f"
  } else {
    // ...
  }
}
```

4 Discussion of Solutions

In the previous section, we saw that a naive solution to the problem does not work. To solve the problem there are two main paths forward: add additional syntax to ensure that the narrowing of the type parameter becomes safe, or, do a more constrained narrowing which does *is* safe. We briefly discuss these in the next two sections

4.1 Uniform Generics

The first solution extends TypeScript with additional syntax for uniform generics. The programmer can write an exclamation mark `!` on a generic type, which can only be instantiated with (TODO: check pr `https://github.com/microsoft/TypeScript/pull/30284` in more detail to give a better description)
.

```
// with Uniform Generics
function depFun<T! extends "t" | "f">(str: T): F[T] {
  if (str === "t") {
    return 1;
  } else {
    return true;
  }
}
```

4.2 Safer Narrowing

The second solution does a smarter narrowing within the branches to ensure that everything stays sound. Instead of replacing the type variable `T` of a checked variable, we add a lower bound to the context. For instance in the `depFun` example, in the `str === "t"` branch we add the lower bound `"t" extends T`. Using some additional typing rules, we are able to derive `1 extends F[T]` and thus `return 1` typechecks in this branch. This enables the wanted behaviour, while still disallowing the previous unsoundness.

```
// with Safer Narrowing
function depFun<T extends "t" | "f">(str: T): F[T] {
  if (str === "t") {
    // type of str is narrowed to "t"
    // '"t" extends T' is added to the context
    // this allows '1 extends F[T]'
    return 1;
```

```
  } else {
    // type of str is narrowed to "f"
    // '"f" extends T' is added to the context
    // this allows 'true extends F[T]'
    return true;
  }
}
```

In the `depFun2` example, the unsoundness is avoided. The added constraint `"t" extends T` is still a valid claim if `str2` is the value `"f"`. Because in that case the type `T` can really only be `"t" | "f"`, and `"t" extends "t"| "f"` is valid. By the same reasoning as in the previous example, we are able to do `return 1`, but this behaviour is sound since the caller of the function will be expecting `number` or `number | boolean` depending on whether `T` is `"t"` or `"t" | "f"`.

```
function depFun2<T extends "t" | "f">(str: T, str2: T): F[T] {
  if (str === "t") {
    // type of str is narrowed to "t"
    // '"t" extends T' is added to the context
    // return 1 is safe, since caller will expect either number or
        number | boolean
    return 1;
  } else {
    // ...
  }
}

depFun("t", "t"); // T = "t" , return type is number
depFun("t", "f"); // T = "t" | "f" , return type is number |
    boolean
```

TODO: add note on difference between specification with lower bounds and actual implementation in compiler, or somewhere else?

4.3 Comparison

Pros/Cons of both approaches

Uniform Generics

- Pro: more general interaction with dependent-type-like features
- Con: additional syntax

Safer Narrowing

- Pro: no additional syntax
- Con: unsure on interaction with future features

5 Scenarios

5.1 Main Scenario

In typescript:

```
interface F {
  "true": "true",
```

```
  "false": "false",
}

function f<T extends "true" | "false">(x: T): F[T] {
  if (x === "true") {
    return "true";
  } else {
    return "false";
  }
}
```

6 Rules

6.1 Typing Relation

$$\frac{}{\Gamma \vdash \texttt{true} : True}\ \text{T-True} \qquad \frac{}{\Gamma \vdash \texttt{false} : False}\ \text{T-False}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T-Var} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\lambda(x : A).(t : B)) : (A \to B)}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash t_1 : A \to B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1\, t_2 : B}\ \text{T-App}$$

$$\frac{\Gamma, X <: U \vdash t : T}{\Gamma \vdash (\Lambda X <: U.t) : (\forall X <: U.T)}\ \text{T-TAbs}$$

$$\frac{\Gamma \vdash t : \forall X <: U.T \qquad \Gamma \vdash S <: U}{\Gamma \vdash t\,[S] : T[X \mapsto S]}\ \text{T-TApp}$$

$$\frac{\Gamma \vdash t : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash t : T}\ \text{T-Sub} \qquad \frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1 \texttt{==} t_2 : Bool}\ \text{T-Eq}$$

$$\frac{\Gamma \vdash t_1 : Bool \qquad \Gamma \vdash t_2 : A \qquad \Gamma \vdash t_3 : B}{\Gamma \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : A \vee B}\ \text{T-If}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : X \qquad \Gamma[x \mapsto \texttt{true}], True <: X \vdash t_2 : A \\ \Gamma[x \mapsto False], False <: X \vdash t_3 : B\end{array}}{\Gamma \vdash \texttt{if}\ x \texttt{==} \texttt{true}\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : A \vee B}\ \text{T-IfTrue}$$

6.2 Sub-Typing Relation 1: Sub-Typing Approach

$$\frac{}{\Gamma \vdash S <: S}\ \text{S-Refl} \qquad \frac{\Gamma \vdash S <: U \qquad \Gamma \vdash U <: T}{\Gamma \vdash S <: T}\ \text{S-Trans}$$

$$\frac{}{\Gamma \vdash S <: Top}\ \text{S-Top} \qquad \frac{X <: T \in \Gamma}{\Gamma \vdash X <: T}\ \text{S-TVarSub}$$

$$\frac{X :> T \in \Gamma}{\Gamma \vdash T <: X}\ \text{S-TVarSup} \qquad \frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash (S_1 \to S_2) <: (T_1 \to T_2)}\ \text{S-Arrow}$$

$$\frac{\Gamma, X <: U \vdash S <: T}{\Gamma \vdash (\forall X <: U.S) <: (\forall X <: U.T)}\ \text{S-All} \qquad \frac{\Gamma \vdash T <: L}{\Gamma \vdash T <: L \vee R}\ \text{S-UnionL}$$

$$\frac{\Gamma \vdash T <: R}{\Gamma \vdash T <: L \vee R}\ \text{S-UnionR} \qquad \frac{\Gamma \vdash L <: T \qquad \Gamma \vdash R <: T}{\Gamma \vdash L \vee R <: T}\ \text{S-UnionM}$$

$$\frac{}{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[True] <: T_t}\ \text{S-MapTrueL}$$

$$\frac{}{\Gamma \vdash T_t <: \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[True]}\ \text{S-MapTrueR}$$

$$\frac{}{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[False] <: T_f}\ \text{S-MapFalseL}$$

$$\frac{}{\Gamma \vdash T_f <: \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[False]}\ \text{S-MapFalseR}$$

$$\frac{}{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[Bool] <: T_t \vee T_f}\ \text{S-MapBoolL}$$

$$\frac{}{\Gamma \vdash T_t \wedge T_f <: \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[Bool]}\ (\text{S-MapBoolR})$$

$$\frac{\Gamma \vdash S <: T}{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[S] <: \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[T]}\ \text{S-Map}$$

Type *Bool* is defined as $True \vee False$.

6.3 Sub-Typing Relation 2: Type Evaluation Approach

$$\frac{}{\Gamma \vdash S <: S}\ \text{S-Refl} \qquad \frac{\Gamma \vdash S <: U \qquad \Gamma \vdash U <: T}{\Gamma \vdash S <: T}\ \text{S-Trans}$$

$$\frac{}{\Gamma \vdash S <: Top}\ \text{S-Top} \qquad \frac{X <: T \in \Gamma}{\Gamma \vdash X <: T}\ \text{S-TVarSub}$$

$$\frac{X :> T \in \Gamma}{\Gamma \vdash T <: X}\ \text{S-TVarSup} \qquad \frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash (S_1 \to S_2) <: (T_1 \to T_2)}\ \text{S-Arrow}$$

$$\frac{\Gamma, X <: U \vdash S <: T}{\Gamma \vdash (\forall X <: U.S) <: (\forall X <: U.T)}\ \text{S-All} \qquad \frac{\Gamma \vdash T <: L}{\Gamma \vdash T <: Bool}\ \text{S-UnionL}$$

$$\frac{\Gamma \vdash T <: R}{\Gamma \vdash T <: L \vee R}\ \text{S-UnionR} \qquad \frac{\Gamma \vdash L <: T \qquad \Gamma \vdash R <: T}{\Gamma \vdash L \vee R <: T}\ \text{S-UnionM}$$

$$\frac{\Gamma \vdash A \xrightarrow[r]{TE} A_2 \qquad \Gamma \vdash A_2 <: B}{\Gamma \vdash A <: B}\ \text{S-TEvalRead}$$

$$\frac{\Gamma \vdash B \xrightarrow[w]{TE} B_2 \qquad \Gamma \vdash A <: B_2}{\Gamma \vdash A <: B}\ \text{S-TEvalWrite}$$

6.4 Type Evaluation

$$\frac{}{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[True] \xrightarrow[\texttt{m}]{TE} T_t}\ \text{TE-MapTrue}$$

$$\frac{}{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[False] \xrightarrow[\texttt{m}]{TE} T_f}\ \text{TE-MapFalse}$$

$$\frac{}{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[Bool] \xrightarrow[r]{TE} T_t \vee T_f}\ \text{TE-MapBoolRead}$$

$$\frac{}{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[Bool] \xrightarrow[w]{TE} T_t \wedge T_f}\ \text{TE-MapBoolWrite}$$

$$\frac{\Gamma \vdash A <: B}{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[A] \xrightarrow[r]{TE} \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[B]}\ \text{TE-MapRead}$$

$$\frac{\Gamma \vdash A <: B}{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[B] \xrightarrow[w]{TE} \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[A]}\ \text{TE-MapWrite}$$

6.5 Grammar

- Types
$A, B, L, R, T_t, T_f, T, U ::= Top \mid A \to B \mid L \vee R \mid \forall X <: U.T \mid True \mid False \mid \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[T] \mid X$
- Terms
$t ::= \texttt{true} \mid \texttt{false} \mid \lambda x.t \mid t_1\, t_2 \mid \Lambda X <: U.t \mid t\,[T] \mid \texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \mid t_1 \texttt{==} t_2 \mid t_1 \texttt{\&\&}\, t_2 \mid (t\ :\ T)$
- Values
$v ::= \texttt{true} \mid \texttt{false} \mid \lambda x.t \mid \Lambda X <: U.t$
- Context
$\Gamma ::= \cdot \mid \Gamma, x : A \mid \Gamma, S < X <: T$
- Condition Context
$\Delta ::= \emptyset \mid \Delta, S < X$
Invariants context: each variabele has max 1 type, each type-variable has max 1 upper bound and max 1 lower bound (what in situation with more literals and false-branches ? formulate invariant more generic ?), if type variable has a lower bound it has an upper bound, no cycles allowed
- Type Evaluation Modes
$\texttt{m} ::= r \mid w$

6.6 Sub-Typing Relation 3: Type Evaluation Approach (Algorithmic)

$$\frac{}{\Gamma \vdash S <: Top}\ \text{SA-Top} \qquad \frac{}{\Gamma \vdash True <: True}\ \text{SA-ReflTrue}$$

$$\frac{}{\Gamma \vdash False <: False}\ \text{SA-ReflFalse} \qquad \frac{X <: T \in \Gamma}{\Gamma \vdash X <: X}\ \text{SA-ReflTVar}$$

$$\frac{}{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[T] <: \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[T]}\ \text{SA-ReflMap}$$

$$\frac{X <: S \in \Gamma \qquad \Gamma \vdash S <: T}{\Gamma \vdash X <: T}\ \text{SA-TransTVar}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash (S_1 \to S_2) <: (T_1 \to T_2)}\ \text{SA-Arrow}$$

$$\frac{\Gamma, X <: U \vdash S <: T}{\Gamma \vdash (\forall X <: U.S) <: (\forall X <: U.T)}\ \text{SA-All} \qquad \frac{\Gamma \vdash T <: L}{\Gamma \vdash T <: L \vee R}\ \text{SA-UnionL}$$

$$\frac{\Gamma \vdash T <: R}{\Gamma \vdash T <: L \vee R}\ \text{SA-UnionR} \qquad \frac{\Gamma \vdash L <: T \qquad \Gamma \vdash R <: T}{\Gamma \vdash L \vee R <: T}\ \text{SA-UnionM}$$

$$\frac{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[T] \xrightarrow[r]{TE}{}^{*} X \qquad \Gamma \vdash X <: A}{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[T] <: A}\ \text{SA-TEvalRead}$$

$$\frac{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[T] \xrightarrow[w]{TE}{}^{*} X \qquad \Gamma \vdash A <: X}{\Gamma \vdash A <: \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[T]}\ \text{SA-TEvalWrite}$$

6.7 Type Evaluation

$$\frac{}{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[True] \xrightarrow[\texttt{m}]{TE} T_t}\ \text{TE-MapTrue}$$

$$\frac{}{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[False] \xrightarrow[\texttt{m}]{TE} T_f}\ \text{TE-MapFalse}$$

$$\frac{}{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[L \vee R] \xrightarrow[r]{TE} \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[L] \vee \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[R]}\ \text{TE-}$$

$$\frac{S < X <: T \in \Gamma}{\Gamma \vdash X \xrightarrow[r]{TE} T}\ \text{TE-TVarRead} \qquad \frac{\{U\} < X <: T \in \Gamma}{\Gamma \vdash X \xrightarrow[w]{TE} U}\ \text{TE-TVarWrite}$$

$$\frac{\Gamma \vdash T \xrightarrow[\texttt{m}]{TE} T'}{\Gamma \vdash \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[T] \xrightarrow[\texttt{m}]{TE} \{\ \texttt{true}:\ T_t,\ \texttt{false}:\ T_f\ \}[T']}\ \text{TE-Map}$$

$$\frac{\Gamma \vdash L \xrightarrow[\texttt{m}]{TE} L'}{\Gamma \vdash L \vee R \xrightarrow[\texttt{m}]{TE} L' \vee R}\ \text{TE-UnionL} \qquad \frac{\Gamma \vdash R \xrightarrow[\texttt{m}]{TE} R'}{\Gamma \vdash L \vee R \xrightarrow[\texttt{m}]{TE} L \vee R'}\ \text{TE-UnionR}$$

The relation $\Gamma \vdash T_1 \xrightarrow[\texttt{m}]{TE}{}^{*} T_2$ is the transitive closure of $\Gamma \vdash T_1 \xrightarrow[\texttt{m}]{TE} T_2$.

6.8 Bidirectional Typing

$$\frac{\Gamma \vdash t \Leftarrow T}{\Gamma \vdash (t \ : \ T) \ \Rightarrow T} \text{ BT-Ann} \qquad \frac{}{\Gamma \vdash \texttt{true} \Rightarrow \mathit{True}} \text{ BT-True}$$

$$\frac{}{\Gamma \vdash \texttt{false} \Rightarrow \mathit{False}} \text{ BT-False} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x \Rightarrow T} \text{ BT-Var}$$

$$\frac{\Gamma \vdash t_1 \Rightarrow A \to B \qquad \Gamma \vdash t_2 \Leftarrow A}{\Gamma \vdash t_1 \, t_2 \Rightarrow B} \text{ BT-App}$$

$$\frac{\Gamma, X <: U \vdash t \Rightarrow T}{\Gamma \vdash (\Lambda X <: U.t) \Rightarrow (\forall X <: U.T)} \text{ BT-TAbs}$$

$$\frac{\Gamma \vdash t \Rightarrow \forall X <: U.T \qquad \Gamma \vdash S <: U}{\Gamma \vdash t \, [S] \Rightarrow T[X \mapsto S]} \text{ BT-TApp}$$

$$\frac{\Gamma \vdash t_1 \Rightarrow T_1 \qquad \Gamma \vdash t_2 \Rightarrow T_2}{\Gamma \vdash t_1 \texttt{==} t_2 \Rightarrow \mathit{Bool}} \text{ BT-Eq}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 \Rightarrow T_1 \\ \Gamma \vdash t_2 \Rightarrow T_2 \qquad \Gamma \vdash T_1 <: \mathit{Bool} \qquad \Gamma \vdash T_2 <: \mathit{Bool}\end{array}}{\Gamma \vdash t_1 \texttt{\&\&} t_2 \Rightarrow \mathit{Bool}} \text{ BT-And}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 \Leftarrow \mathit{Bool} \\ \Gamma \vdash t_1 \rightsquigarrow \Delta \qquad \Gamma \sqcap \Delta \vdash t_2 \Rightarrow A \qquad \Gamma \sqcap (\Gamma \vdash \neg\Delta) \vdash t_3 \Rightarrow B\end{array}}{\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Rightarrow A \vee B} \text{ BT-If}$$

$$\frac{\Gamma, x : A \vdash t \Leftarrow B}{\Gamma \vdash \lambda x.t \Leftarrow A \to B} \text{ BT-Abs} \qquad \frac{\Gamma \vdash t \Rightarrow S \qquad \Gamma \vdash S <: T}{\Gamma \vdash t \Leftarrow T} \text{ BT-Sub}$$

6.9 Condition Information

$$\frac{}{\Gamma \vdash \texttt{true} \rightsquigarrow \emptyset}\ \text{CI-True} \qquad \frac{}{\Gamma \vdash \texttt{false} \rightsquigarrow \emptyset}\ \text{CI-False}$$

$$\frac{x : X \in \Delta \qquad X <: T \in \Gamma \qquad \Gamma \vdash T <: \mathit{Bool}}{\Gamma \vdash x \texttt{ == true} \rightsquigarrow \emptyset, \{\mathit{True}\} < X}\ \text{CI-EqTrueR}$$

$$\frac{x : X \in \Gamma \qquad X <: T \in \Gamma \qquad \Gamma \vdash T <: \mathit{Bool}}{\Gamma \vdash \texttt{true == } x \rightsquigarrow \emptyset, \{\mathit{True}\} < X}\ \text{CI-EqTrueL}$$

$$\frac{x : X \in \Gamma \qquad X <: T \in \Gamma \qquad \Gamma \vdash T <: \mathit{Bool}}{\Gamma \vdash x \texttt{ == false} \rightsquigarrow \emptyset, \{\mathit{False}\} < X}\ \text{CI-EqFalseR}$$

$$\frac{x : X \in \Gamma \qquad X <: T \in \Gamma \qquad \Gamma \vdash T <: \mathit{Bool}}{\Gamma \vdash \texttt{false == } x \rightsquigarrow \emptyset, \{\mathit{False}\} < X}\ \text{CI-EqFalseL}$$

$$\frac{\Gamma \vdash t_1 \rightsquigarrow \Delta_1 \qquad \Gamma \vdash t_2 \rightsquigarrow \Delta_2}{\Gamma \vdash t_1 \texttt{ \&\& } t_2 \rightsquigarrow \Delta_1 \sqcap \Delta_2}\ \text{CI-EqAnd} \qquad \frac{\mathit{otherwise}}{\Delta \vdash t \rightsquigarrow \emptyset}\ \text{CI-EqOtherwise}$$

6.10 Context Operations

$$\frac{}{\Gamma \sqcap \emptyset \to \Gamma}\ \text{CO-IntersectEmpty}$$

$$\frac{S_1 < X <: T \in \Gamma}{\Gamma \sqcap (\Delta, S_2 < X) \to \Gamma[X \mapsto (S_1 \cap S_2) < X <: T] \sqcap \Delta}\ \text{CO-Intersect}$$

$$\frac{}{\Delta \sqcap \emptyset \to \Delta}\ \text{CO-SmallEmpty}$$

$$\frac{S_1 < X \in \Delta_1}{\Delta_1 \sqcap (\Delta_2, S_2 < X) \to \Delta_1[X \mapsto (S_1 \cap S_2) < X] \sqcap \Delta_2}\ \text{CO-SmallIntersect}$$

$$\frac{S_1 < X \notin \Delta_1}{\Delta_1 \sqcap (\Delta_2, S_2 < X) \to \Delta_1[X \mapsto S_2 < X] \sqcap \Delta_2}\ \text{CO-SmallAdd}$$

$$\frac{}{\Gamma \vdash \neg\emptyset \to \emptyset}\ \text{CO-InverseEmpty}$$

$$\frac{U < X <: T \in \Gamma \qquad U \setminus S_1 = S_2}{\Gamma \vdash \neg(\Delta, S < X) \to \neg\Delta, S_2 < X}\ \text{CO-InverseCons}$$

6.11 Term Evaluation

$$\frac{}{(\lambda x.t)\, v_2 \to t[x \mapsto v_2]}\ \text{E-AppAbs} \qquad \frac{t_1 \to t_1'}{t_1\, t_2 \to t_1'\, t_2}\ \text{E-App1}$$

$$\frac{t_2 \to t_2'}{v_1\, t_2 \to v_1\, t_2'}\ \text{E-App2} \qquad \frac{}{(\Lambda X <: U.t)\, [T] \to t[X \mapsto T]}\ \text{E-TappTabs}$$

$$\frac{t \to t'}{t\, [T] \to t'\, [T]}\ \text{E-Tapp} \qquad \frac{}{\texttt{if true then } t_1 \texttt{ else } t_2 \to t_1}\ \text{E-IfTrue}$$

$$\frac{}{\texttt{if false then } t_1 \texttt{ else } t_2 \to t_2}\ \text{E-IfFalse}$$

$$\frac{t \to t'}{\texttt{if } t \texttt{ then } t_1 \texttt{ else } t_2 \to \texttt{if } t' \texttt{ then } t_1 \texttt{ else } t_2}\ \text{E-If}$$

$$\frac{t_1 \to t_1'}{t_1 \texttt{ == } t_2 \to t_1' \texttt{ == } t_2}\ \text{E-EqL} \qquad \frac{t_2 \to t_2'}{v_1 \texttt{ == } t_2 \to v_1 \texttt{ == } t_2'}\ \text{E-EqR}$$

$$\frac{v_1 = v_2}{v_1 \texttt{ == } v_2 \to \texttt{true}}\ \text{E-EqTrue} \qquad \frac{v_1 \neq v_2}{v_1 \texttt{ == } v_2 \to \texttt{false}}\ \text{E-EqFalse}$$

$$\frac{t_1 \to t_1'}{t_1 \texttt{ \&\& } t_2 \to t_1' \texttt{ \&\& } t_2}\ \text{E-AndL} \qquad \frac{t_2 \to t_2'}{t_1 \texttt{ \&\& } t_2 \to t_1 \texttt{ \&\& } t_2'}\ \text{E-AndR}$$

$$\frac{}{\texttt{false \&\& } v_2 \to \texttt{false}}\ \text{E-AndFalseL} \qquad \frac{}{v_1 \texttt{ \&\& false} \to \texttt{false}}\ \text{E-AndFalseR}$$

$$\frac{}{\texttt{true \&\& true} \to \texttt{true}}\ \text{E-AndTrue}$$