Specification of Source §1 Type Inference—2021 edition

Martin Henz, K Muruges, Raynold Ng, Daryl Tan, Tse Hiu Fung

National University of Singapore
School of Computing

July 14, 2021

1 Notation

1.1 The language Source §1

The set of expressions E is the least set that satisfies the following rules, where x ranges over a set of names V, n ranges over the positive integers, p_1 ranges over the set of unary primitive operations $P_1 = \{\texttt{!}\}$, and p_2 ranges over the set of binary primitive operations $P_2 = \{\texttt{||}, \texttt{\&\&}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}, \texttt{===}, \texttt{!==}, \texttt{>}, \texttt{<}, \texttt{<=}, \texttt{>=}\}$.

$$\frac{}{x} \qquad \frac{}{i} \qquad \frac{}{s} \qquad \frac{}{\texttt{true}} \qquad \frac{}{\texttt{false}} \qquad \frac{}{\texttt{undefined}}$$

$$\frac{E}{p_1[E]} \qquad \frac{E_1 \quad E_2}{p_2[E_1, E_2]} \qquad \frac{E \quad E_1 \quad E_2}{E \texttt{ ? } E_1 \texttt{ : } E_2} \qquad \frac{E \quad E_1 \quad \cdots \quad E_n}{E \texttt{ (} E_1 \texttt{,} \cdots \texttt{,} E_n \texttt{)}}$$

The letters x, i and s stand for names, numbers, strings, respectively.
The set of statements is the least set that satisfies the following seven rules.

$$\frac{S}{\texttt{function } f \texttt{ (} x_1, \cdots, x_n \texttt{) \{ } S \texttt{ \}}} \qquad \frac{E}{\texttt{let } x \texttt{ = } E \texttt{ ;}} \qquad \frac{E}{\texttt{return } E \texttt{ ;}}$$

The identifiers $x_1, \ldots, x_n$ must be pairwise distinct.

$$\frac{S_1 \quad S_2}{S_1\ S_2} \qquad \frac{E}{E \texttt{ ;}} \qquad \frac{S}{\texttt{\{ } S \texttt{ \}}} \qquad \frac{E \quad S_1 \quad S_2}{\texttt{if (} E \texttt{) \{ } S_1 \texttt{ \} else \{ } S_2 \texttt{ \}}}$$

We introduce the following additional rule for expressions, in order to define functions.

$$\frac{S}{\texttt{(} x_1, \cdots, x_n \texttt{) => \{ } S \texttt{ \}}}$$

We treat function declaration statements of the form

$$\texttt{function } f \texttt{ (} x_1, \cdots, x_n \texttt{) \{ } S \texttt{ \}}$$

as abbreviations for constant declaration statements as follows

$$\texttt{const } f \texttt{ = (} x_1, \cdots, x_n \texttt{) => \{ } S \texttt{ \};}$$

function definitions of the form

$$\texttt{(} x_1, \cdots, x_n \texttt{) => } E$$

as abbreviations for the following

$$\texttt{(} x_1, \cdots, x_n \texttt{) => \{ return } E \texttt{ ; \}}$$

Conditional statements of the form

```
if (x1) {
    const x = 1;
} else if (x2) {
    const y = 3;
} else if (x3) {
    const a = 3;
} else {
    const b = 3;
}
```

are treated as abbreviations for the following

```
if (x1) {
    const x = 1;
} else {
    if (x2) {
        const y = 3;
    } else {
        if (x3) {
            const a = 3;
        } else {
            const b = 3;
        }
    }
}
```

1.2 A Language of Types

We introduce the following language of types for type inference:

$$\frac{}{T_i} \qquad \frac{}{A_i}$$

$$\frac{}{\texttt{number}} \qquad \frac{}{\texttt{bool}} \qquad \frac{}{\texttt{string}} \qquad \frac{}{\texttt{undefined}}$$

$$\frac{t_1 \quad \cdots \quad t_n \quad t}{(t_1, \ldots t_n) \rightarrow t} \qquad \frac{t}{\forall(t)}$$

where $n \geq 1$, and T_i and A_i represent type variables. We will capitalize type variables, as in T_1, A_2. We will also refer to the types in the second row (i.e. `bool`, `undefined`, `number`, `string`) as *base types*. The symbols t_i in the rules above are meta-variables that stand for types and must not be confused with type variables that *are* types. As usual, parentheses can be used in practice for grouping. Examples of valid types are `number` and $(\texttt{number}, () \rightarrow \texttt{bool}, \texttt{undefined}, T_1) \rightarrow (\texttt{bool} \rightarrow A_2)$. Types of the form $\forall(t)$ are called *polymorphic types*, whereas all other are called *monomorphic types*.

We distinguish two kinds of type variables, T_i and A_i, to be able to handle the overloading of operators such as + (for numbers and strings). A type variable A_i can only represent "addable" types, i.e. `number` or `string`, and a type variable T_i can represent any type.

1.3 Type Environments

For Source, well-typedness of an statement depends on the context in which the statement appears. The expression `x + 3` within a statement may or may not be well-typed, depending on the type of x. Thus in order to formalize the notion of a context, we define a *type environment*, denoted by Γ, that keeps track of the type of names appearing in the statement. More formally, the partial function Γ from names to types expresses a context, in which a name x is associated with type $\Gamma(x)$.

We define a relation $\Gamma[x \leftarrow t]\Gamma'$ on type environments Γ, names x, types t, and type environments Γ', which constructs a type environment that behaves like the given one, except that the type of x is t. More formally, if $\Gamma[x \leftarrow t]\Gamma'$, then $\Gamma'(y)$ is t, if $y = x$ and $\Gamma(y)$ otherwise. Obviously, this uniquely identifies Γ' for a given Γ, x, and t, and thus the type environment extension relation is functional in its first three arguments.

The set of names, on which a type environment Γ is defined, is called the domain of Γ, denoted by *dom*(Γ).

For each non-overloaded primitive operator, we add a binding to our initial type environment Γ_0 as follows:

$$\begin{aligned}
&\emptyset[-_2 \leftarrow (\texttt{number}, \texttt{number}) \rightarrow \texttt{number}]\\
&\ [* \leftarrow (\texttt{number}, \texttt{number}) \rightarrow \texttt{number}]\\
&\ [/ \leftarrow (\texttt{number}, \texttt{number}) \rightarrow \texttt{number}]\\
&\ [\% \leftarrow (\texttt{number}, \texttt{number}) \rightarrow \texttt{number}]\\
&\ [\&\& \leftarrow \forall((\texttt{bool}, T) \rightarrow T)]\\
&\ [|| \leftarrow \forall((\texttt{bool}, T) \rightarrow T)]\\
&\ [! \leftarrow \texttt{bool} \rightarrow \texttt{bool}]\\
&\ [-_1 \leftarrow \texttt{number} \rightarrow \texttt{number}]\Gamma_{-2}
\end{aligned}$$

The overloaded binary primitive are handled as follows:

$$\begin{aligned}
&\Gamma_{-2}[+ \leftarrow \forall((A, A) \rightarrow A)]\\
&\quad [=== \leftarrow \forall((A, A) \rightarrow \texttt{bool})]\\
&\quad [!== \leftarrow \forall((A, A) \rightarrow \texttt{bool})]\\
&\quad [> \leftarrow \forall((A, A) \rightarrow \texttt{bool})]\\
&\quad [>= \leftarrow \forall((A, A) \rightarrow \texttt{bool})]\\
&\quad [< \leftarrow \forall((A, A) \rightarrow \texttt{bool})]\\
&\quad [<= \leftarrow \forall((A, A) \rightarrow \texttt{bool})]\Gamma_{-1}
\end{aligned}$$

$$
\begin{array}{lllllllll}
\Gamma_{-1} & [& \texttt{display} & \leftarrow & \forall(T) & & &] & \\
 & [& \texttt{error} & \leftarrow & \forall(T) & & &] & \\
 & [& \texttt{Infinity} & \leftarrow & \mathtt{number} & & &] & \\
 & [& \texttt{is_boolean} & \leftarrow & \forall(T & \rightarrow & \mathtt{bool}) &] & \\
 & [& \texttt{is_function} & \leftarrow & \forall(T & \rightarrow & \mathtt{bool}) &] & \\
 & [& \texttt{is_number} & \leftarrow & \forall(T & \rightarrow & \mathtt{bool}) &] & \\
 & [& \texttt{is_string} & \leftarrow & \forall(T & \rightarrow & \mathtt{bool}) &] & \\
 & [& \texttt{is_undefined} & \leftarrow & \forall(T & \rightarrow & \mathtt{bool}) &] & \\
 & [& \texttt{math_abs} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_acos} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_acosh} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_asin} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_asinh} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_atan} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_atan2} & \leftarrow & (\mathtt{number}, \mathtt{number}) & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_atanh} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_cbrt} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_ceil} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_clz32} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_cos} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_cosh} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_exp} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_expm1} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_floor} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_fround} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_hypot} & \leftarrow & \forall(T) & & &] & \\
 & [& \texttt{math_imul} & \leftarrow & (\mathtt{number}, \mathtt{number}) & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_LN2} & \leftarrow & \mathtt{number} & & &] & \\
 & [& \texttt{math_LN10} & \leftarrow & \mathtt{number} & & &] & \\
 & [& \texttt{math_log} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_log1p} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_log2} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_LOG2E} & \leftarrow & \mathtt{number} & & &] & \\
 & [& \texttt{math_log10} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_LOG10E} & \leftarrow & \mathtt{number} & & &] & \\
 & [& \texttt{math_max} & \leftarrow & \forall(T) & & &] & \\
 & [& \texttt{math_min} & \leftarrow & \forall(T) & & &] & \\
 & [& \texttt{math_PI} & \leftarrow & \mathtt{number} & & &] & \\
 & [& \texttt{math_pow} & \leftarrow & (\mathtt{number}, \mathtt{number}) & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_random} & \leftarrow & () & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_round} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_sign} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_sin} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_sinh} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_sqrt} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_SQRT1_2} & \leftarrow & \mathtt{number} & & &] & \\
 & [& \texttt{math_SQRT2} & \leftarrow & \mathtt{number} & & &] & \\
 & [& \texttt{math_tan} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_tanh} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{math_trunc} & \leftarrow & \mathtt{number} & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{NaN} & \leftarrow & \mathtt{number} & & &] & \\
 & [& \texttt{parse_int} & \leftarrow & (\mathtt{string}, \mathtt{number}) & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{prompt} & \leftarrow & \mathtt{string} & \rightarrow & \mathtt{string} &] & \\
 & [& \texttt{get_time} & \leftarrow & () & \rightarrow & \mathtt{number} &] & \\
 & [& \texttt{stringify} & \leftarrow & \forall(T & \rightarrow & \mathtt{string}) &] & \\
 & [& \texttt{undefined} & \leftarrow & \mathtt{undefined} & & &] & \Gamma_0
\end{array}
$$

1.4 Preparing Programs for Type Inference

To facilitate the process of type inference, we annotate each component of the given program with unique type variables and introduce a simple transformation at the toplevel.
A *toplevel transformation* clarifies the nature of the names declared outside of function definitions, and the type of the overall statement. The toplevel transformation wraps the given program into a block, and introduces **`return`** keywords in front of expression statements, when these are the last statements in a sequence to be evaluated, even when they occur within conditional statements.
Examples:

```
const x = 1;
x + 2;
```

becomes

```
{
    const x = 1;
    return x + 2;
}
```

and

```
if (true) {
    const x = 1;
    x + 2;
} else {
    const y = 3;
    y + 4;
}
```

becomes

```
{
    if (true) {
        const x = 1;
        return x + 2;
    } else {
        const y = 3;
        return y + 4;
    }
}
```

To facilitate the process of type inference, we annotate each component of the given program with unique type variables. We write the type variable as a superscript after the component, and use parentheses for clarification. For example, the Source §1 program

```
{ const x = 1; return x + 2; }
```

is represented by the annotated program

$$\texttt{const x}^{T_1} = \texttt{1}^{T_2}\texttt{; return }\ (\texttt{x}^{T_3} + \texttt{2}^{T_4})^{T_5}\ \texttt{;}$$

1.5 Type Constraints

We introduce type constraints Σ as conjunctions of type equations:

$$\frac{}{\top} \qquad\qquad \frac{}{t_1 = t_2} \qquad\qquad \frac{\Sigma_1 \qquad \Sigma_2}{\Sigma_1 \wedge \Sigma_2}$$

We require that constraints are kept in *solved form*:

$$t_1 = t'_1 \wedge \cdots \wedge t_i = t'_i \wedge \cdots \wedge t_n = t'_n$$

where:

- all t_i are type variables,
- for any type variable T_i, there is at most one equation $T_i = \cdots$,
- no variable t_i occurs in any equation $t_j = t'_j$ if $j > i$.

A constraint in solved form does not have any cycles $t^{(0)} = t^{(1)}, t^{(1)} = t^{(2)}, \ldots, t^{(k)} = t^{(0)}$. We *apply* a type constraint Σ in solved form to a type t as follows:

$$\frac{\text{if } t_i \text{ is a } \textit{base type} \text{ or } t_i = t'_i \text{ does not occur in } \Sigma}{\Sigma(t_i) = t_i} \qquad \frac{\text{if } t_i = t'_i \text{ occurs in } \Sigma}{\Sigma(t_i) = \Sigma(t'_i)}$$

$$\frac{t' = \Sigma(t) \qquad t'_1 = \Sigma(t_1) \qquad \cdots \qquad t'_n = \Sigma(t_n)}{\Sigma((t_1, \ldots, t_n) \to t) = (t'_1, \ldots, t'_n) \to t'}$$

Example: If $\Sigma = (T_1 = \texttt{number} \wedge T_2 = T_3 \to \texttt{bool} \wedge T_3 = \texttt{number} \to \texttt{bool})$, we have $\Sigma(\texttt{number} \to T_2) = \texttt{number} \to ((\texttt{number} \to \texttt{bool}) \to \texttt{bool})$.
Note that in our framework, type constraints never contain any polymorphic types. Thus you will never see "$\forall$" an a type constraint.
We add a constraint $t = t'$ to a solved form Σ by applying the following rules in the given order:

- If t is a *base type* and t' is also a *base type* of the same kind, do nothing.
- If t is not a type variable and t' is a type variable, then we now try to add $t' = t$ to Σ, following the same rules.
- If t is a type variable and $\Sigma(t')$ is a type variable with the same name as t, do nothing.
- If t is a type variable, $\Sigma(t')$ is a function type and t is contained in $\Sigma(t')$, then stop with a type error as we will have an infinite type. (e.g. A = B -> A)
- If t is A_i and $\Sigma(t')$ is not a type variable and not `number` or `string`, then stop with a type error.
- If t is a type variable and there is an equation $t = t''$ in Σ, then we now try to add the equation $t' = t''$ to Σ, following the same rules.
- If t is a type variable that does not occur on the left in any equation in Σ, then add $t = \Sigma(t')$ in the front of Σ. In addition, if $\Sigma(t)$ is an "addable" type variable A_i and $\Sigma(t')$ is a regular type variable T_j, we must convert $\Sigma(t')$ into an "addable" type A_j.
- If t is $(t_1, \ldots, t_n) \to t''$ and t' is $(t'_1, \ldots, t'_n) \to t'''$, then add n constraints $t_1 = t'_1, \cdots, t_n = t'_n, t'' = t'''$ to Σ, each time going through the above set of rules.
- Any other case (e.g. $\texttt{bool} = \texttt{string}$) stops with a type error.

This process is guaranteed to terminate either with a type error or with a new solved form.

2 Typing Relation

The set of well-typed programs is defined by the binary typing relation written $S : \Sigma$, where S is a toplevel-transformed, type-annotated program. The relation is defined using the quaternary typing relation $\Sigma, \Gamma \vdash S : \Sigma'$, as follows: $S : \Sigma$ holds if and only if $\top, \Gamma_0 \vdash S : \Sigma$ where Γ_0 is the intial type environment described above and $\top$ is the empty type constraint. The constraint Σ can be called the constraint *inferred from* S.
We define the typing relation for expressions and statements inductively with the following rules.

2.1 Typing Relation on Expressions

The type of a name needs to be provided by the type environment. The first rule applies when $\Gamma(x)$ is monomorphic, i.e. $\Gamma(x) \neq \forall t'$.

$$\frac{\Gamma(x) \neq \forall t' \qquad \Sigma' = (\Sigma \land t = \Gamma(x))}{\Sigma, \Gamma \vdash x^t : \Sigma'}$$

If $\Gamma(x)$ is polymorphic, i.e. $\Gamma(x) = \forall t'$, we replace all type variables in t' with fresh type variables:

$$\frac{\Gamma(x) = \forall t' \qquad \Sigma' = (\Sigma \land t = \mathit{fresh}(t'))}{\Sigma, \Gamma \vdash x^t : \Sigma'}$$

where $\mathit{fresh}(t')$ results from t' by replacing all type variables consistently with fresh type variables. Example: $\mathit{fresh}(\texttt{bool} \to (T_1 \to (T_2 \to T_2)))$ might return $\texttt{bool} \to (T_{77} \to (T_{88} \to T_{88})))$.
If $\Gamma(x)$ is not defined, then neither rule is applicable. In this case, we say that there is no type for x derivable from the type environment Γ.
Constants get the following types.

$$\frac{\Sigma' = (\Sigma \land t = \texttt{number})}{\Sigma, \Gamma \vdash n^t : \Sigma'} \qquad \frac{\Sigma' = (\Sigma \land t = \texttt{string})}{\Sigma, \Gamma \vdash s^t : \Sigma'}$$

where n denotes any literal number s denotes any literal string.

$$\frac{\Sigma' = (\Sigma \land t = \texttt{bool})}{\Sigma, \Gamma \vdash \mathbf{\texttt{true}}^t : \Sigma'} \qquad \frac{\Sigma' = (\Sigma \land t = \texttt{bool})}{\Sigma, \Gamma \vdash \mathbf{\texttt{false}}^t : \Sigma'}$$

Important for typing conditionals is that the consequent and alternative expressions get the same type.

$$\frac{(\Sigma_0 \land t_0 = \texttt{bool} \land t = t_1 \land t_1 = t_2), \Gamma \vdash E_0^{t_0} : \Sigma_1 \quad \Sigma_1, \Gamma \vdash E_1^{t_1} : \Sigma_2 \quad \Sigma_2, \Gamma \vdash E_2^{t_2} : \Sigma_3}{\Sigma_0, \Gamma \vdash (E_0^{t_0} \texttt{ ? } E_1^{t_1} \texttt{ : } E_2^{t_2})^t \ : \ \Sigma_3}$$

We have the following rule for function application.

$$\frac{\Sigma_0, \Gamma \vdash E_0^{t_0} : \Sigma_1 \quad \cdots \quad \Sigma_n, \Gamma \vdash E_n^{t_n} : \Sigma_{n+1} \quad (\Sigma_{n+1} \land t_0 = (t_1, \ldots, t_n) \to t) = \Sigma_{n+2}}{\Sigma_0, \Gamma \vdash (E_0^{t_0} \texttt{ (} E_1^{t_1}, \ldots, E_n^{t_n} \texttt{)})^t : \Sigma_{n+2}}$$

The type of the operator needs to be a function type with the right number of parameters, and the type of every argument needs to coincide with the corresponding parameter type of the function type. If all these conditions are met, the type of the function application is the same as the return type of the function type that is the type of the operator.
The typing of function definition statements is defined as follows.

$$\frac{\Sigma \land (t' = (t_1, \ldots, t_n) \to t), \Gamma[x_1 \leftarrow t_1] \cdots [x_n \leftarrow t_n] \vdash S^t : \Sigma'}{\Sigma, \Gamma \vdash (\texttt{(}x_1^{t_1}, \ldots, x_n^{t_n}\texttt{) => \{ } S^t \texttt{ \}})^{t'} : \Sigma'}$$

2.2 Typing Relation on Statements

The following rule deals with the typing of sequences. We assume that whenever there is a return statement or a conditional statement with a return statement within a sequence, it is the last statement in the sequence. (One could consider a "dead code" error otherwise.)

$$\frac{(\Sigma_1 \land t_3 = t_2), \Gamma \vdash S_1^{t_1} : \Sigma_2 \qquad \Sigma_2, \Gamma \vdash S_2^{t_2} : \Sigma_3}{\Sigma_1, \Gamma \vdash (S_1^{t_1}\ S_2^{t_2})^{t_3} : \Sigma_3}$$

Return statements are typed as follows.

$$\frac{(\Sigma \land t' = t), \Gamma \vdash E^t : \Sigma'}{\Sigma, \Gamma \vdash (\texttt{\textbf{return}}\ E^t\texttt{\textbf{;}})^{t'} : \Sigma'}$$

The type of conditional statements is similar to the type of conditional expressions.

$$\frac{(\Sigma_0 \land t_0 = \texttt{bool} \land t = t_1 \land t_1 = t_2), \Gamma \vdash E^{t_0} : \Sigma_1 \quad \Sigma_1, \Gamma \vdash \{\ S_1\ \}^{t_1} : \Sigma_2 \quad \Sigma_2, \Gamma \vdash \{\ S_2\ \}^{t_2} : \Sigma_3}{\Sigma_0, \Gamma \vdash (\texttt{\textbf{if}}\ \texttt{(}\ E^{t_0}\ \texttt{)}\ \ \texttt{\{}\ S_1\ \texttt{\}}^{t_1}\ \texttt{\textbf{else}}\ \texttt{\{}\ S_2\ \ \texttt{\}}^{t_2})^{t}\ :\ \Sigma_3}$$

The type of expression statements is `undefined`. Note that expression statements at toplevel get a return placed in front of them by the toplevel-transformation described above.

$$\frac{(\Sigma \land t' = \texttt{undefined}), \Gamma \vdash E^t : \Sigma'}{\Sigma, \Gamma \vdash (E^t\texttt{\textbf{;}})^{t'} : \Sigma'}$$

For blocks (including the bodies of function definitions), we discern whether the block contains constant declarations or not. If it does not contain constant declarations, the typing is easy:

$$\frac{S \text{ does not contain } \texttt{\textbf{const}} \qquad (\Sigma \land t' = t), \Gamma \vdash S^t : \Sigma'}{\Sigma, \Gamma \vdash \texttt{\{}\ S^t\ \texttt{\}}^{t'} : \Sigma_3}$$

Blocks (including the bodies of function definitions) that contain constant declarations introduce polymorphism. In the following rule we assume that S does not have any further constant declarations. The rule is a simplification of the general case, because statements other than constant declarations can appear before and between the constant declarations. The rule applies analogously in this case, without re-arranging the statements. This means that the body of a block has two parts:

- the part up to and including the last constant declaration, where all declared names are monomorphically typed, and
- the part after the last constant declaration, where all declared names are polymorphically typed.

$$\frac{\begin{array}{l}\Gamma[x_1 \leftarrow t_1] \cdots [x_n \leftarrow t_n]\Gamma' \\ \Sigma_1 = (\Sigma_0 \wedge t = t' \wedge t'_1 = undefined \wedge \cdots \wedge t'_n = undefined) \\ \Sigma_1, \Gamma' \vdash E_1^{t_1} : \Sigma_2 \cdots \Sigma_n, \Gamma' \vdash E_n^{t_n} : \Sigma_{n+1} \\ \Gamma'[x_1 \leftarrow \forall\Sigma_{n+1}(t_1)] \cdots [x_n \leftarrow \forall\Sigma_{n+1}(t_n)]\Gamma'' \\ \Sigma_{n+1}, \Gamma'' \vdash S^t : \Sigma_{n+2}\end{array}}{\Sigma_0, \Gamma \vdash \texttt{\{}\ (\textbf{\texttt{const}}\ x_1\ \texttt{=}\ E_1^{t_1}\texttt{;})^{t'_1}\ \cdots\ (\textbf{\texttt{const}}\ x_n\ \texttt{=}\ E_n^{t_n}\texttt{;})^{t'_n}\ \ S^t\,\texttt{\}}^{t'} : \Gamma_{n+2}}$$

3 Type Safety of Source

Now we can define what it means for a statement to be well-typed.

Definition 3.1 *A statement S is well-typed, if there is a consistent type constraint Σ such that $S : \Sigma$.*

Note that this definition of well-typedness requires that a well-typed statement has no free names.