HOT Compilation (15-501/15-819) Fall 2005
Project 2: Type Inference

William Lovas (wlovas@cs)

Out: Thursday, October 6, 2005
Due: Thursday, October 20, 2005 (before midnight)

1 Introduction

Static typing offers the advantage of catching many programming errors at compile time rather than runtime, but writing down type annotations for complex programs can become quite tedious, especially in higher-order polymorphic code. Therefore, many higher-order polymorphically-typed languages, like ML and Haskell, allow programmers to write unannotated programs, relying on *type inference* to determine the best possible type to assign to those programs.

In this assignment, you will implement a type inference engine for a small but interesting subset of Standard ML. In implementing your typechecker, you will have to deal with several important issues that arise in the front-end compilation of ML-like languages, including how to communicate type constraints between disparate pieces of code and when and how to introduce and eliminate polymorphism. This handout will explain how to answer these questions.

2 Overview

Your task is to create a file `typecheck.sml` defining a structure `Typecheck` matching the signature `TYPECHECK` found in `typecheck-sig.sml`, reproduced here for your convenience.

```
signature TYPECHECK =
  sig
    exception Check of string (* for ordinary type errors *)
    exception Error of string (* for internal, unrecoverable errors *)

    datatype dec = DecExpTy of Syntax.variable * Syntax.polyty
                 | DecTyVar of Syntax.variable
    type context = dec list

    val unify : Syntax.ty -> Syntax.ty -> unit

    val check_program : context -> Syntax.program
                     -> (Syntax.variable * Syntax.polyty) list
  end
```

The key component of your typechecker is the `check_program` function. This function takes as its input a typing `context` and a `Syntax.program`, which is simply a list of bindings. It produces as its output a list of identifier typing declarations, one for each binding in the input program. Your inference engine will depend upon the `unify` function which implements type unification.

Judgment	Inputs	Outputs	Interpretation
$\Gamma;\Sigma \vdash \text{infer } e \Rightarrow \Sigma';\sigma;\tau$	Γ, Σ, e	Σ', σ, τ	Infer type τ for expression e
$\Gamma;\Sigma \vdash \text{infer } \mathit{bindings} \Rightarrow \Sigma';\sigma;\Gamma'$	Γ, Σ, *bindings*	Σ', σ, Γ'	*bindings* introduce context Γ'
$\Gamma;\Sigma \vdash \text{infer } \mathit{binding} \Rightarrow \Sigma';\sigma;(x{:}\tau)$	Γ, Σ, *binding*	Σ', σ, x, τ	*binding* introduces declaration $x{:}\tau$
$\Sigma \vdash \text{unify } \tau_1 \text{ with } \tau_2 \Rightarrow \Sigma';\sigma$	Σ, τ_1, τ_2	Σ', σ	Unify types τ_1 and τ_2
$\Sigma \vdash \text{oc}[E][S]\tau \Rightarrow \Sigma';\sigma$	Σ, E, S, τ	Σ', σ	Perform "occurs" check for $E[S]$ in τ

Table 1: Summary of type inference judgements for Poly-SML

For this assignment, a `context` can contain two sorts of declarations: ordinary expression variable typing declarations and type variable declarations. Since the language for this assignment has no kinds other than "type", type variables go into the context unadorned. (Furthermore, as you will see, the type variables really only serve as "scoping tokens" to determine how much polymorphism to introduce.)

You should implement `unify` and `check_program` using the type inference algorithm outlined in class and Section 3. Signal any type inference errors by raising `Check` with an appropriate error message; you may use the `Error` exception for internal consistency errors such as violated invariants or the occurrence of "impossible" conditions.

The file `syntax.sml` defines the `Syntax` structure, which contains all of the datatypes your checker will manipulate.

You may use the test harness interface in the `Top` structure (see `top-sig.sml`) to experiment with your implementation. Several interesting examples are included in the `examples.sml` file.

Submit your code via AFS by copying `typecheck.sml` to the directory

```
/afs/andrew/course/15/501-819/submit/<your andrew id>/proj2
```

Note: Your submission will be graded automatically; if you submit a `typecheck.sml` that fails to compile under SML/NJ 110.0.7 using this assignment's base distribution, we won't be able to grade it.

3 Details

In this assignment, we'll be working with a simple polymorphic subset of Standard ML, which we'll refer to as Poly-SML. The abstract syntax of Poly-SML is shown in Figure 1. This grammar corresponds closely to the datatypes defined in `syntax.sml`. The rules implementing type inference and its constituent parts are shown in Figures 3, 4, and 5. The algorithm is based on five judgments, summarized briefly in Table 1.

Type inference proceeds by making up fresh evars whenever it needs a type — like in the rule for **fn** $x \Rightarrow e$, for the type of the bound variable — and using unification to determine what those evars should be bound to whenever it encounters constraining information — like in the rule for $e_1\ e_2$, when it knows that e_1 must have an arrow type.

All of the judgments take an existential context Σ describing what evars are currently "in play", that is which evars have not yet been substituted away. All of the judgments also produce a new Σ, since they may have created new evars or substituted away old ones, as well as an evar substitution σ delineating what types old evars have been bound to. By design, the τ in $\Gamma;\Sigma \vdash \text{infer } e \Rightarrow \Sigma';\sigma;\tau$ has already been acted on by σ, and similarly the Γ' in $\Gamma;\Sigma \vdash \text{infer } \mathit{bindings} \Rightarrow \Sigma';\sigma;\Gamma'$ has already been acted on by σ. In your implementation, evars will be implemented by mutable `ref` cells, and instead of maintaining a substitution, you'll simply update the `ref` cells as appropriate.

The unification judgment, $\Sigma \vdash \text{unify } \tau_1 \text{ with } \tau_2 \Rightarrow \Sigma';\sigma$, takes the place of type equivalence; instead of simply checking that two types are equivalent, it tries to *make* the two types equivalent by filling in evars as appropriate. You can see this in the two rules for unifying a type τ with an evar E, where the existential context and evar substitution returned reflect that E has been bound to τ, provided that τ and E pass the "occurs" check.

The "occurs" check, $\Sigma \vdash \text{oc}[E][S]\tau \Rightarrow \Sigma';\sigma$, has three important effects:

$\tau ::=$	bool	booleans
	$\mid \mathsf{int}$	integers
	$\mid \tau_1 \times \tau_2$	pairs
	$\mid \tau\,\mathsf{list}$	lists
	$\mid \tau_1 \to \tau_2$	functions
	$\mid \alpha$	universal variables
	$\mid E[S]$	existential variables
$T ::=$	τ	monomorphic types
	$\mid \forall\alpha.T$	polymorphic types
$op ::=$	$+$	addition
	$\mid -$	subtraction
	$\mid *$	multiplication
	$\mid =$	integer equality
$e ::=$	x	variables
	$\mid \mathsf{true}$	boolean literals
	$\mid \mathsf{false}$	
	$\mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3$	boolean test
	$\mid \bar{n}$	integer literals
	$\mid e_1\ op\ e_2$	integer operations
	$\mid (e_1, e_2)$	pairing
	$\mid \pi_1\ e$	first projection
	$\mid \pi_2\ e$	second projection
	$\mid \mathsf{nil}$	empty list
	$\mid e_1 :: e_2$	non-empty list
	$\mid \textbf{case } e \textbf{ of } \mathsf{nil} \Rightarrow e_n \mid x :: xs \Rightarrow e_c$	list destructor
	$\mid \textbf{fn } x \Rightarrow e$	abstraction
	$\mid e_1\ e_2$	application
	$\mid \textbf{let } \mathit{bindings} \textbf{ in } e \textbf{ end}$	let-binding
	$\mid e : \tau$	constrained expressions
$\mathit{binding} ::=$	$\textbf{val } x = e$	value binding
	$\mid \textbf{fun } f\ x = e$	recursive function binding
$\mathit{bindings} ::=$	$\cdot \mid \mathit{binding}\ \mathit{bindings}$	
$\mathit{program} ::=$	$\mathit{bindings}$	program

Figure 1: Poly-SML, a subset of SML including polymorphism

$\Gamma ::= \cdot$	empty context
$\mid \Gamma, x{:}T$	expression variable declaration
$\mid \Gamma, \alpha$	universal variable declaration
$\Sigma ::= \cdot$	empty existential context
$\mid \Sigma, E[S]$	existential variable with dependency set S
$\sigma ::= \mathsf{id}$	identity substitution
$\mid [\tau_1, \ldots, \tau_n / E_1, \ldots, E_n]$	substitution for evars
$\mid \sigma_1 \circ \sigma_2$	composition of substitutions

Figure 2: Syntax of contexts and substitutions

1. It ensures that the evar E does not appear in the type τ, since the unification of E and τ would create a cyclic type,
2. It ensures that any uvars in τ are in E's dependency set S, and
3. It forces any other evars E' in τ to have smaller dependency sets.

The latter two properties are used to implement polymorphic generalization. When we come to a point where we want to introduce polymorphism, i.e. a **val** or **fun** binding, we make up a fresh type variable α. Then, after inferring a type for the body of the binding, we scan its type for evars that can depend on α. These are exactly the evars that were introduced but never eliminated while inferring a type for the body. Since they weren't substituted away, they represent unconstrained types that may be polymorphically generalized. (**Note:** for simplicity's sake, we haven't bothered with the value restriction — all bindings will have their types generalized.)

It is important to realize that the "type variables" we add to the context during type inference don't actually represent real types; rather they serve merely as "scoping tokens" to allow us to determine which evars represent unconstrained types as opposed to those that represent constraints which just haven't been determined yet. (If we were implementing the value restriction, such latent constraints could arise from non-generalized bindings, like **val** $l = ($**fn** $x \Rightarrow x)$ nil.)

Using unification, we only make choices that are forced upon us by real constraints due to the structure of a term. Therefore, after we finish typechecking a term, we can be sure that we have found the least-constrained type possible for that term. This corresponds to determining the "most general unifier" for a set of constraints in the literature on unification.

The inverse process to generalization is instantiation: when we encounter a variable with a polymorphic type, we instantiate its type variables with fresh evars — see the inference rule for variables in Figure 4. In this way, every use of a polymorphic variable can be given a different type, just like we'd expect.

$$\boxed{\Gamma;\Sigma \vdash \text{infer } e \Rightarrow \Sigma';\sigma;\tau}$$

$$\frac{}{\Gamma;\Sigma \vdash \text{infer } \mathsf{true} \Rightarrow \Sigma;\mathsf{id};\mathsf{bool}} \qquad \frac{}{\Gamma;\Sigma \vdash \text{infer } \mathsf{false} \Rightarrow \Sigma;\mathsf{id};\mathsf{bool}}$$

$$\frac{\begin{array}{c}\Gamma;\Sigma \vdash \text{infer } e_1 \Rightarrow \Sigma_1;\sigma_1;\tau_1 \\ \sigma_1(\Gamma);\Sigma_1 \vdash \text{infer } e_2 \Rightarrow \Sigma_2;\sigma_2;\tau_2 \\ (\sigma_2\circ\sigma_1)(\Gamma);\Sigma_2 \vdash \text{infer } e_3 \Rightarrow \Sigma_3;\sigma_3;\tau_3 \\ \Sigma_3 \vdash \text{unify } (\sigma_3\circ\sigma_2)(\tau_1) \text{ with } \mathsf{bool} \Rightarrow \Sigma_4;\sigma_4 \\ \Sigma_4 \vdash \text{unify } (\sigma_4\circ\sigma_3)(\tau_2) \text{ with } \sigma_4(\tau_3) \Rightarrow \Sigma_5;\sigma_5\end{array}}{\Gamma;\Sigma \vdash \text{infer } \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Rightarrow \Sigma_5;(\sigma_5\circ\sigma_4\circ\sigma_3\circ\sigma_2\circ\sigma_1);(\sigma_5\circ\sigma_4)(\tau_3)}$$

$$\frac{}{\Gamma;\Sigma \vdash \text{infer } \bar{n} \Rightarrow \Sigma;\mathsf{id};\mathsf{int}}$$

$$\frac{\begin{array}{c}(op \in \{+,-,*\}) \\ \Gamma;\Sigma \vdash \text{infer } e_1 \Rightarrow \Sigma_1;\sigma_1;\tau_1 \\ \sigma_1(\Gamma);\Sigma_1 \vdash \text{infer } e_2 \Rightarrow \Sigma_2;\sigma_2;\tau_2 \\ \Sigma_2 \vdash \text{unify } \sigma_2(\tau_1) \text{ with } \mathsf{int} \Rightarrow \Sigma_3;\sigma_3 \\ \Sigma_3 \vdash \text{unify } \sigma_3(\tau_2) \text{ with } \mathsf{int} \Rightarrow \Sigma_4;\sigma_4\end{array}}{\Gamma;\Sigma \vdash \text{infer } e_1\ op\ e_2 \Rightarrow \Sigma_4;(\sigma_4\circ\sigma_3\circ\sigma_2\circ\sigma_1);\mathsf{int}} \qquad \frac{\begin{array}{c}\Gamma;\Sigma \vdash \text{infer } e_1 \Rightarrow \Sigma_1;\sigma_1;\tau_1 \\ \sigma_1(\Gamma);\Sigma_1 \vdash \text{infer } e_2 \Rightarrow \Sigma_2;\sigma_2;\tau_2 \\ \Sigma_2 \vdash \text{unify } \sigma_2(\tau_1) \text{ with } \mathsf{int} \Rightarrow \Sigma_3;\sigma_3 \\ \Sigma_3 \vdash \text{unify } \sigma_3(\tau_2) \text{ with } \mathsf{int} \Rightarrow \Sigma_4;\sigma_4\end{array}}{\Gamma;\Sigma \vdash \text{infer } e_1 = e_2 \Rightarrow \Sigma_4;(\sigma_4\circ\sigma_3\circ\sigma_2\circ\sigma_1);\mathsf{bool}}$$

$$\frac{\Gamma;\Sigma \vdash \text{infer } e_1 \Rightarrow \Sigma_1;\sigma_1;\tau_1 \qquad \sigma_1(\Gamma);\Sigma_1 \vdash \text{infer } e_2 \Rightarrow \Sigma_2;\sigma_2;\tau_2}{\Gamma;\Sigma \vdash \text{infer } (e_1,e_2) \Rightarrow \Sigma_2;(\sigma_2\circ\sigma_1);\sigma_2(\tau_1)\times\tau_2}$$

$$\frac{\begin{array}{c}\Gamma;\Sigma \vdash \text{infer } e \Rightarrow \Sigma';\sigma';\tau' \\ (E_1,E_2 \notin \Sigma', S = BTV(\Gamma)) \\ \Sigma', E_1[S], E_2[S] \vdash \text{unify } \tau' \text{ with } E_1\times E_2 \Rightarrow \Sigma'';\sigma''\end{array}}{\Gamma;\Sigma \vdash \text{infer } \pi_1\ e \Rightarrow \Sigma'';(\sigma''\circ\sigma');\sigma''(E_1)} \qquad \frac{\begin{array}{c}\Gamma;\Sigma \vdash \text{infer } e \Rightarrow \Sigma';\sigma';\tau' \\ (E_1,E_2 \notin \Sigma', S = BTV(\Gamma)) \\ \Sigma', E_1[S], E_2[S] \vdash \text{unify } \tau' \text{ with } E_1\times E_2 \Rightarrow \Sigma'';\sigma''\end{array}}{\Gamma;\Sigma \vdash \text{infer } \pi_2\ e \Rightarrow \Sigma'';(\sigma''\circ\sigma');\sigma''(E_2)}$$

$$\frac{(E \notin \Sigma, S = BTV(\Gamma))}{\Gamma;\Sigma \vdash \text{infer } \mathsf{nil} \Rightarrow \Sigma, E[S];\mathsf{id};E\ \mathsf{list}} \qquad \frac{\begin{array}{c}\Gamma;\Sigma \vdash \text{infer } e_1 \Rightarrow \Sigma_1;\sigma_1;\tau_1 \\ \sigma_1(\Gamma);\Sigma_1 \vdash \text{infer } e_2 \Rightarrow \Sigma_2;\sigma_2;\tau_2 \\ \Sigma_2 \vdash \text{unify } \tau_2 \text{ with } \sigma_2(\tau_1)\ \mathsf{list} \Rightarrow \Sigma_3;\sigma_3\end{array}}{\Gamma;\Sigma \vdash \text{infer } e_1 :: e_2 \Rightarrow \Sigma_3;(\sigma_3\circ\sigma_2\circ\sigma_1);(\sigma_3\circ\sigma_2)(\tau_1)\ \mathsf{list}}$$

$$\frac{\begin{array}{c}\Gamma;\Sigma \vdash \text{infer } e \Rightarrow \Sigma_1;\sigma_1;\tau_1 \\ (E \notin \Sigma_1, S = BTV(\Gamma)) \\ \Sigma_1, E[S] \vdash \text{unify } \tau_1 \text{ with } E\ \mathsf{list} \Rightarrow \Sigma_2;\sigma_2 \\ (\sigma_2\circ\sigma_1)(\Gamma);\Sigma_2 \vdash \text{infer } e_n \Rightarrow \Sigma_3;\sigma_3;\tau_3 \\ (\sigma_3\circ\sigma_2\circ\sigma_1)(\Gamma, x{:}E, xs{:}E\ \mathsf{list});\Sigma_3 \vdash \text{infer } e_c \Rightarrow \Sigma_4;\sigma_4;\tau_4 \\ \Sigma_4 \vdash \text{unify } \sigma_4(\tau_3) \text{ with } \tau_4 \Rightarrow \Sigma_5;\sigma_5\end{array}}{\Gamma;\Sigma \vdash \text{infer } (\mathbf{case}\ e\ \mathbf{of}\ \mathsf{nil} \Rightarrow e_n \mid x :: xs \Rightarrow e_c) \Rightarrow \Sigma_5;(\sigma_5\circ\sigma_4\circ\sigma_3\circ\sigma_2\circ\sigma_1);\sigma_5(\tau_4)}$$

Figure 3: Type inference rules for Poly-SML

(continued) $\boxed{\Gamma; \Sigma \vdash \text{infer } e \Rightarrow \Sigma'; \sigma; \tau}$

$$\frac{(E \notin \Sigma, S = BTV(\Gamma)) \qquad (\Gamma, x{:}E); (\Sigma, E[S]) \vdash \text{infer } e \Rightarrow \Sigma'; \sigma'; \tau'}{\Gamma; \Sigma \vdash \text{infer } (\mathbf{fn}\ x \Rightarrow e) \Rightarrow \Sigma'; \sigma'; \sigma'(E) \rightarrow \tau'}$$

$$\frac{\begin{array}{c}\Gamma; \Sigma \vdash \text{infer } e_1 \Rightarrow \Sigma_1; \sigma_1; \tau_1 \qquad \sigma_1(\Gamma); \Sigma_1 \vdash \text{infer } e_2 \Rightarrow \Sigma_2; \sigma_2; \tau_2 \\ (E \notin \Sigma_2, S = BTV(\Gamma)) \qquad \Sigma_2, E[S] \vdash \text{unify } \sigma_2(\tau_1) \text{ with } \tau_2 \rightarrow E \Rightarrow \Sigma_3; \sigma_3\end{array}}{\Gamma; \Sigma \vdash \text{infer } e_1\ e_2 \Rightarrow \Sigma_3; (\sigma_3 \circ \sigma_2 \circ \sigma_1); \sigma_3(E)}$$

$$\frac{\Gamma; \Sigma \vdash \text{infer } \mathit{bindings} \Rightarrow \Sigma'; \sigma'; \Gamma' \qquad (\sigma'(\Gamma), \Gamma'); \Sigma' \vdash \text{infer } e \Rightarrow \Sigma''; \sigma''; \tau}{\Gamma; \Sigma \vdash \text{infer } \mathbf{let}\ \mathit{bindings}\ \mathbf{in}\ e\ \mathbf{end} \Rightarrow \Sigma''; (\sigma'' \circ \sigma'); \tau}$$

$$\frac{\Gamma(x) = \forall \alpha_1. \ldots \forall \alpha_n. \tau \qquad (E_1, \ldots, E_n \notin \Sigma, S = BTV(\Gamma))}{\Gamma; \Sigma \vdash \text{infer } x \Rightarrow (\Sigma, E_1[S], \ldots, E_n[S]); \mathsf{id}; [E_1, \ldots, E_n/\alpha_1, \ldots, \alpha_n]\tau}$$

$$\frac{\Gamma; \Sigma \vdash \text{infer } e \Rightarrow \Sigma'; \sigma'; \tau' \qquad \Sigma' \vdash \text{unify } \tau' \text{ with } \tau \Rightarrow \Sigma''; \sigma''}{\Gamma; \Sigma \vdash \text{infer } (e : \tau) \Rightarrow \Sigma''; (\sigma'' \circ \sigma'); \tau}$$

$\boxed{\Gamma; \Sigma \vdash \text{infer } \mathit{bindings} \Rightarrow \Sigma'; \sigma; \Gamma'}$

$$\frac{}{\Gamma; \Sigma \vdash \text{infer } \cdot \Rightarrow \Sigma; \mathsf{id}; \cdot}$$

$$\frac{\Gamma; \Sigma \vdash \text{infer } \mathit{binding} \Rightarrow \Sigma'; \sigma'; (x{:}T) \qquad (\sigma'(\Gamma), x{:}T); \Sigma' \vdash \text{infer } \mathit{bindings} \Rightarrow \Sigma''; \sigma''; \Gamma''}{\Gamma; \Sigma \vdash \text{infer } \mathit{binding}\ \mathit{bindings} \Rightarrow \Sigma''; (\sigma'' \circ \sigma'); (x{:}\sigma''(T), \Gamma'')}$$

$\boxed{\Gamma; \Sigma \vdash \text{infer } \mathit{binding} \Rightarrow \Sigma'; \sigma; (x{:}T)}$

$$\frac{\begin{array}{c}(\Gamma, \alpha); \Sigma \vdash \text{infer } e \Rightarrow \Sigma'; \sigma'; \tau \qquad (\{E_1, \ldots, E_n\} = \{ \text{ evars } E \text{ in } \tau \ \mid \alpha \in \Sigma'(E) \ \}) \\ (\alpha_1, \ldots, \alpha_n \notin \mathrm{dom}(\Gamma)) \qquad \sigma'' = [\alpha_1, \ldots, \alpha_n/E_1, \ldots, E_n]\end{array}}{\Gamma; \Sigma \vdash \text{infer } \mathbf{val}\ x = e \Rightarrow (\Sigma' \setminus \{E_1, \ldots, E_n\}); (\sigma'' \circ \sigma'); (x{:}\forall \alpha_1. \ldots \forall \alpha_n. \sigma''(\tau))}$$

$$\frac{\begin{array}{c}(\alpha \notin \mathrm{dom}(\Gamma)) \qquad (E_1, E_2 \notin \Sigma, S = BTV(\Gamma) \cup \{\alpha\}) \\ (\Gamma, \alpha, f{:}E_1 \rightarrow E_2, x{:}E_1); (\Sigma, E_1[S], E_2[S]) \vdash \text{infer } e \Rightarrow \Sigma'; \sigma'; \tau \\ \Sigma' \vdash \text{unify } \sigma'(E_2) \text{ with } \tau \Rightarrow \Sigma''; \sigma'' \\ (\{E_1, \ldots, E_n\} = \{ \text{ evars } E \text{ in } (\sigma'' \circ \sigma')(E_1 \rightarrow E_2) \ \mid \alpha \in \Sigma''(E) \ \}) \\ (\alpha_1, \ldots, \alpha_n \notin \mathrm{dom}(\Gamma)) \qquad \sigma = [\alpha_1, \ldots, \alpha_n/E_1, \ldots, E_n] \circ \sigma'' \circ \sigma'\end{array}}{\Gamma; \Sigma \vdash \text{infer } \mathbf{fun}\ f\ x = e \Rightarrow (\Sigma'' \setminus \{E_1, \ldots, E_n\}); \sigma; (f{:}\forall \alpha_1. \ldots \forall \alpha_n. \sigma(E_1 \rightarrow E_2))}$$

Figure 4: Type inference rules for Poly-SML, continued

$$\boxed{\Sigma \vdash \text{unify } \tau_1 \text{ with } \tau_2 \Rightarrow \Sigma'; \sigma}$$

$$\frac{(E \in \Sigma)}{\Sigma \vdash \text{unify } E \text{ with } E \Rightarrow \Sigma; \mathsf{id}}$$

$$\frac{(\Sigma(E) = S) \qquad \Sigma \vdash \text{oc}[E][S]\tau \Rightarrow \Sigma'; \sigma}{\Sigma \vdash \text{unify } E \text{ with } \tau \Rightarrow (\Sigma' \setminus E); ([\sigma(\tau)/E] \circ \sigma)} \qquad \frac{(\Sigma(E) = S) \qquad \Sigma \vdash \text{oc}[E][S]\tau \Rightarrow \Sigma'; \sigma}{\Sigma \vdash \text{unify } \tau \text{ with } E \Rightarrow (\Sigma' \setminus E); ([\sigma(\tau)/E] \circ \sigma)}$$

$$\frac{}{\Sigma \vdash \text{unify } \mathsf{bool} \text{ with } \mathsf{bool} \Rightarrow \Sigma; \mathsf{id}} \qquad \frac{}{\Sigma \vdash \text{unify } \mathsf{int} \text{ with } \mathsf{int} \Rightarrow \Sigma; \mathsf{id}} \qquad \frac{}{\Sigma \vdash \text{unify } \alpha \text{ with } \alpha \Rightarrow \Sigma; \mathsf{id}}$$

$$\frac{\Sigma \vdash \text{unify } \tau_1 \text{ with } \tau_2 \Rightarrow \Sigma'; \sigma' \qquad \Sigma' \vdash \text{unify } \sigma'(\tau_1') \text{ with } \sigma'(\tau_2') \Rightarrow \Sigma''; \sigma''}{\Sigma \vdash \text{unify } \tau_1 \to \tau_1' \text{ with } \tau_2 \to \tau_2' \Rightarrow \Sigma''; (\sigma'' \circ \sigma')}$$

$$\frac{\Sigma \vdash \text{unify } \tau_1 \text{ with } \tau_2 \Rightarrow \Sigma'; \sigma' \qquad \Sigma' \vdash \text{unify } \sigma'(\tau_1') \text{ with } \sigma'(\tau_2') \Rightarrow \Sigma''; \sigma''}{\Sigma \vdash \text{unify } \tau_1 \times \tau_1' \text{ with } \tau_2 \times \tau_2' \Rightarrow \Sigma''; (\sigma'' \circ \sigma')}$$

$$\frac{\Sigma \vdash \text{unify } \tau_1 \text{ with } \tau_2 \Rightarrow \Sigma'; \sigma'}{\Sigma \vdash \text{unify } \tau_1\, \mathsf{list} \text{ with } \tau_2\, \mathsf{list} \Rightarrow \Sigma'; \sigma'}$$

$$\boxed{\Sigma \vdash \text{oc}[E][S]\tau \Rightarrow \Sigma'; \sigma}$$

$$\frac{(E \neq E') \qquad (\Sigma(E') = S') \qquad (E'' \notin \Sigma)}{\Sigma \vdash \text{oc}[E][S]E' \Rightarrow ((\Sigma \setminus E'), E''[S \cap S']); [E''/E']}$$

$$\frac{}{\Sigma \vdash \text{oc}[E][S]\mathsf{bool} \Rightarrow \Sigma; \mathsf{id}} \qquad \frac{}{\Sigma \vdash \text{oc}[E][S]\mathsf{int} \Rightarrow \Sigma; \mathsf{id}} \qquad \frac{(\alpha \in S)}{\Sigma \vdash \text{oc}[E][S]\alpha \Rightarrow \Sigma; \mathsf{id}}$$

$$\frac{\Sigma \vdash \text{oc}[E][S]\tau_1 \Rightarrow \Sigma_1; \sigma_1 \qquad \Sigma_1 \vdash \text{oc}[E][S]\sigma_1(\tau_2) \Rightarrow \Sigma_2; \sigma_2}{\Sigma \vdash \text{oc}[E][S]\tau_1 \to \tau_2 \Rightarrow \Sigma_2; (\sigma_2 \circ \sigma_1)}$$

$$\frac{\Sigma \vdash \text{oc}[E][S]\tau_1 \Rightarrow \Sigma_1; \sigma_1 \qquad \Sigma_1 \vdash \text{oc}[E][S]\sigma_1(\tau_2) \Rightarrow \Sigma_2; \sigma_2}{\Sigma \vdash \text{oc}[E][S]\tau_1 \times \tau_2 \Rightarrow \Sigma_2; (\sigma_2 \circ \sigma_1)}$$

$$\frac{\Sigma \vdash \text{oc}[E][S]\tau \Rightarrow \Sigma'; \sigma'}{\Sigma \vdash \text{oc}[E][S]\tau\, \mathsf{list} \Rightarrow \Sigma'; \sigma'}$$

Figure 5: Unification and the “occurs” check