Polymorphic Type Inference
(Polymorf Type Inferens)

Rasmusk Kock Thygesen, 201909745
Timur Bas, 201906748

Bachelor Report (15 ECTS) in Computer Science
Advisor: Lars Birkedal
Department of Computer Science, Aarhus University
June 2022

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

This thesis contributes to the understanding of how the Hindley-Milner Type system works based on a language that is an extended version of the lambda calculus with let-polymorphism. The proofs of the consistency between the *dynamic* semantics and *static* semantics of the language are given for the new expressions introduced in the language. In particular, the new expressions are *integers*, *booleans*, *strings*, *tuples*, the *fst operation*, and the *snd operation*.
Furthermore, two type inference algorithms are discussed, namely, $\mathcal{W}$ and $\mathcal{W}_{opt}$. The latter algorithm is an optimized version of the prior. Soundness proofs are given for $\mathcal{W}$. Implementations for both algorithms are given in OCaml.
Finally, the full implementation of both algorithms can be found on our GitHub page here.

Rasmusk Kock Thygesen and Timur Bas,
Aarhus, June 2022.

Contents

Chapter 1

Introduction

In many programming languages, the programmer has to make sure the types of the variables are coherent to avoid typing errors. One way to make it easier for the programmer to work with the types is to infer the types so the programmer can omit explicit type annotation. A compiler will then be able to deduce which variables have which types and an editor might even tell the programmer the type of different variables. This can give a better overview of the code and the programmer can receive on-the-fly error messages about types where the expected type can be mentioned. This can potentially make it easier to figure out how to fix errors. One problem with type inference algorithms is that it might be harder to infer the type in more complex languages. The Hindley-Milner type system is a type system based on lambda calculus and is complex enough to have polymorphic functions in the form of let polymorphism while being simple enough to have a practical type inference algorithm that can give the programmer hints in real-time while writing their code.

In Chapter 2 of this thesis, we will introduce the Hindley-Milner type system by defining the grammar of the language and present both the dynamic and static semantics. We will go over a few concepts that will help us build the algorithm such as substitutions, instantiation, generalization, and unification. After this, a proof of the consistency between the dynamic semantics and the static semantics will be presented. We will provide both pseudocode and a full working implementation of the type inference algorithm known as algorithm $\mathcal{W}$ as well as proof of the soundness of the algorithm in Chapter 3. In Chapter 4 we will then look into optimizing algorithm $\mathcal{W}$ by using a Union-Find data structure instead of substitutions to which we have also provided both pseudocode and a full working implementation. In the appendices of this thesis, we have summaries of some operations, examples of type inference, more detailed examples, additional proofs, and some extra practical code such as tests and pretty printers for both algorithms.

Chapter 2

Language Definition & Semantics

In order to reason about type systems and type inference, we will have to define a language and its semantics. The language we will consider in this thesis is a relatively small functional language. In this chapter, we will formally define the language by defining its grammar and semantics. Lastly, we will prove consistency between the two groups of semantics..

2.1 The Grammar of the Language

The grammar of the language defines exactly which expressions are allowed in the language. Before presenting the grammar, we have to introduce four sets. Let Var be the set of program variables and the rest is self-explanatory. Thus, we have

$$
\begin{aligned}
&x \in \text{Var} = \{\text{a, b, } \ldots\text{, x, y, } \ldots\} \\
&i \in \mathbb{Z} \\
&b \in \text{Boolean} = \{true,\ false\} \\
&s \in \text{String} = \{\text{"}myVariable\text{"},\text{"}myVariable2\text{"},\ \ldots\}
\end{aligned}
$$

The grammar of the language Exp ranged over by e is defined using a context-free grammar where the production rules are in Backus-Naur form

$$
\begin{array}{lll}
\langle e \rangle ::= & i & \text{Integer} \\
\mid & b & \text{Boolean} \\
\mid & s & \text{String} \\
\mid & x & \text{Variable} \\
\mid & \lambda x.e_1 & \text{Lambda Abstraction} \\
\mid & e_1\ e_2 & \text{Application} \\
\mid & \textbf{let } x = e_1 \textbf{ in } e_2 & \text{Let} \\
\mid & (e_1, e_2) & \text{Tuple} \\
\mid & \textbf{fst } (e_1, e_2) & \text{First Tuple} \\
\mid & \textbf{snd } (e_1, e_2) & \text{Second Tuple}
\end{array}
$$

note that this grammar is an extended version of the lambda calculus with let-polymorphism. In particular, we have extended the grammar with the first three production rules and the last three production rules. Parentheses are used to disambiguate expressions.

2.2 Semantics

In this section, we present the semantics of the language. The semantics of this language can be separated into two groups, namely the *dynamic semantics* and the *static semantics*. The former deals with the evaluation of programs while the latter deals with type checking programs.

2.2.1 Dynamic Semantics

In order to present the inference rules we have to introduce the semantic objects that the rules in this section will be built upon.

Figure 2.1: Semantic Objects

$$
\begin{aligned}
bv &\in \text{BasVal} = \mathbb{Z} \cup \text{Boolean} \cup \text{String} \\
v &\in \text{Val} = \text{BasVal} + \text{Clos} + \text{Val} \times \text{Val} \\
[x, e, E] &\in \text{Clos} = \text{Var} \times \text{Exp} \times \text{Env} \\
E &\in \text{Env} = \text{Var} \to \text{Val} \\
r &\in \text{Results} = \text{Val} + \{\mathit{wrong}\}
\end{aligned}
$$

Figure 2.1 presents the semantic objects being used in the inference rules. Note that the result *wrong* is not a value but merely an indicator of a nonsensical evaluation, for instance, trying to do the fst operation on a lambda abstraction.

Each inference rule has the general form

$$\frac{P_1, \ldots, P_n}{C} \quad n \geq 0$$

where each premise P_i for $0 \leq i \leq n$ all together allows us to *infer* the conclusion C. Each premise is either a sequent or a side condition written by standard mathematical concepts.

The inference rules for the dynamic semantics are the following

D-Lookup
$$\frac{x \in \text{Dom } E}{E \vdash x \to E(x)}$$

D-Lookup-Wrong
$$\frac{x \notin \text{Dom } E}{E \vdash x \to \mathit{wrong}}$$

D-Lambda
$$\frac{}{E \vdash \lambda x.e \to [x,\ e,\ E]}$$

D-App
$$\frac{\begin{array}{c} E \vdash e_1 \to [x_0, e_0, E_0] \\ E \vdash e_2 \to v_0 \\ E_0 \pm \{x_0 \mapsto v_0\} \vdash e_0 \to r \end{array}}{E \vdash e_1\ e_2 \to r}$$

D-App-Wrong1
$$\frac{E \vdash e_1 \to [x_0,\ e_0,\ E_0] \qquad E \vdash e_2 \to \mathit{wrong}}{E \vdash e_1\ e_2 \to \mathit{wrong}}$$

D-App-Wrong2
$$\frac{E \vdash e_1 \to w : w \in (\text{Val} \setminus \text{Clos}) \cup \{\mathit{wrong}\}}{E \vdash e_1\ e_2 \to \mathit{wrong}}$$

D-Let
$$\frac{E \vdash e_1 \to v_1 \quad E \pm \{x \mapsto v_1\} \vdash e_2 \to r}{E \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \to r}$$

D-Let-Wrong
$$\frac{E \vdash e_1 \to \mathit{wrong}}{E \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \to \mathit{wrong}}$$

D-Tuple
$$\frac{E \vdash e_1 \to v_1 \qquad E \vdash e_2 \to v_2}{E \vdash (e_1, e_2) \to (v_1, v_2)}$$

D-Tuple-Wrong1
$$\frac{E \vdash e_1 \to \mathit{wrong}}{E \vdash (e_1, e_2) \to \mathit{wrong}}$$

D-Tuple-Wrong2
$$\frac{E \vdash e_1 \to v_1 \quad E \vdash e_2 \to \mathit{wrong}}{E \vdash (e_1, e_2) \to \mathit{wrong}}$$

D-Fst
$$\frac{E \vdash e_1 \to v_1 \quad E \vdash e_2 \to v_2}{E \vdash \mathbf{fst}(e_1, e_2) \to v_1}$$

D-Fst-Wrong1
$$\frac{E \vdash e_1 \to \mathit{wrong}}{E \vdash \mathbf{fst}(e_1, e_2) \to \mathit{wrong}}$$

D-Fst-Wrong2
$$\frac{E \vdash e_1 \to v_1 \quad E \vdash e_2 \to \mathit{wrong}}{E \vdash \mathbf{fst}(e_1, e_2) \to \mathit{wrong}}$$

D-Snd
$$\frac{E \vdash e_1 \to v_1 \quad E \vdash e_2 \to v_2}{E \vdash \mathbf{snd}(e_1, e_2) \to v_2}$$

D-Snd-Wrong1
$$\frac{E \vdash e_1 \to \mathit{wrong}}{E \vdash \mathbf{snd}(e_1, e_2) \to \mathit{wrong}}$$

D-Snd-Wrong2
$$\frac{E \vdash e_1 \to v_1 \quad E \vdash e_2 \to \mathit{wrong}}{E \vdash \mathbf{snd}(e_1, e_2) \to \mathit{wrong}}$$

D-Fst
$$\frac{E \vdash (e_1, e_2) \to (v_1, v_2)}{E \vdash \mathbf{snd}(e_1, e_2) \to v_1}$$

D-Snd
$$\frac{E \vdash (e_1, e_2) \to (v_1, v_2)}{E \vdash \mathbf{snd}(e_1, e_2) \to v_2}$$

D-Int
$$\frac{}{E \vdash i \to i}$$

D-Bool
$$\frac{}{E \vdash b \to b}$$

D-String
$$\frac{}{E \vdash s \to s}$$

The notation $E \vdash e \to r$ is a ternary relation between elements of Env, Exp, and Results, respectively. One can read it as – *Under the Environment E when e is evaluated it results in r*. The domain and range of any map f are denoted Dom f and Range f respectively. For any f, g that is an infinite or finite map we define $f \pm g$ to be the map where f is modified by g. The resulting map will have domain Dom $f \pm g =$ Dom $f \cup$ Dom g and range Range $f \pm g =$ Range $g \cup \{f(x) : x \in \text{Dom } f \wedge x \notin \text{Dom } g\}$. We use $\pm$ because the $+$ resembles that the domain can possibly be larger than f and the $-$ because if f, g have the same element in their domains then the value in the range of g is used hence a value might disappear from f.

Example 1 (The $\pm$ operator)**.** Let $f = \{x \mapsto \alpha, y \mapsto \beta\}$ and $g = \{y \mapsto \mathit{int}, z \mapsto \gamma\}$ then $f \pm g = \{x \mapsto \alpha, y \mapsto \mathit{int}, z \mapsto \gamma\}$.

2.2.2 Static Semantics

The basic Hindley-Milner type system is defined by the following set of inference rules

S-Lookup
$$\frac{x \in \text{Dom } \Gamma \quad \Gamma(x) > \tau}{\Gamma \vdash x : \tau}$$

S-Lambda
$$\frac{\Gamma \pm \{x \mapsto \tau'\} \vdash e_1 \ : \ \tau}{\Gamma \vdash \lambda x.e_1 \ : \ \tau' \to \tau}$$

S-App
$$\frac{\Gamma \vdash e_1 \ : \ \tau' \to \tau \qquad \Gamma \vdash e_2 \ : \ \tau'}{\Gamma \vdash e_1 \ e_2 \ : \ \tau}$$

S-Let
$$\frac{\Gamma \vdash e_1 \ : \ \tau_1 \qquad \Gamma \pm \{x \mapsto \mathit{Clos}_\Gamma \tau_1\} \vdash e_2 \ : \ \tau}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \ : \ \tau}$$

However, due to the extension of the grammar, we have to define one rule per new expression

$$\text{S-Tuple}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \text{S-Fst}\ \frac{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{fst}\ (e_1, e_2) : \tau_1} \qquad \text{S-Snd}\ \frac{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{snd}\ (e_1, e_2) : \tau_2}$$

$$\text{S-Int}\ \frac{}{\Gamma \vdash i : int} \qquad \text{S-Bool}\ \frac{}{\Gamma \vdash b : bool} \qquad \text{S-String}\ \frac{}{\Gamma \vdash s : string}$$

Now that the type system is well-defined we can pick parts of the rules (e.g. the $Clos_\Gamma \tau_1$) in order to explain their meanings.

2.2.2.1 Types

Let **TyCon** be a finite set of nullary *type constructors* also known as the basic types. A nullary type constructor simply means that the type itself takes zero types as parameter. Furthermore, let **TyVar** be an infinite set of *type variables*.

$$\pi \in \mathbf{TyCon} = \{int, bool, string\}$$
$$\alpha \in \mathbf{TyVar} = \{\beta, \gamma, \delta, \ldots\}$$

A *type* τ and a *type scheme* σ are defined as follows

$$\begin{aligned} \tau ::= &\quad \pi \mid \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \\ \sigma ::= &\quad \tau \mid \forall \alpha . \sigma_1 \end{aligned}$$

We denote the set of τ's and σ's as **Type** and **TypeScheme** respectively.

A type τ can represent a nullary type constructor, a type variable, an abstraction, or a product. Note that for the case of abstraction and product the types τ_1, τ_2 that occur in them are defined recursively with respect to τ.

Example 2 (Type)**.** $\tau = (\beta \to bool) \times (string \to \gamma)$

Moreover, a type scheme σ is either a type τ or a type scheme that is quantified with type variables. Generally we write $\forall \alpha_1 \ \ldots \ \forall \alpha_n . \tau$ which we will abbreviate by writing $\forall \alpha_1, \ \ldots, \ \alpha_n . \tau$. If the set of type variables is $\emptyset$ then we will write τ but implicitly any type τ is quantified, written $\forall . \tau$ but we will not be bothered to write so.

Example 3 (Type schemes)**.** Let σ_1, σ_2 be type schemes and $\sigma_1 = \forall \alpha . \alpha \to \alpha$ and $\sigma_2 = int \to int$

Additionally, there is the concept of *bound* and *free* type variables in σ. If $\sigma = \forall \alpha_1, \ \ldots, \ \alpha_n . \tau$ then all $\{\alpha_1, \ \ldots, \ \alpha_n\}$ are said to be bound in σ. Contrarily, a type variable α is free if it is not bound and occurs in τ.

Example 4 (Free and bound type variables)**.** Let $\sigma = \forall \beta . (\beta \to bool) \times (string \to \gamma)$ then $\{\gamma\}$ is a free variable in σ and $\{\beta\}$ is a bound variable in σ.

2.2.2.2 Type Environment

A type environment Γ contains information about which program variables have which type schemes. Thus, it is a finite map from program variables to type schemes i.e. Γ : **Var** $\rightarrow$ **TypeScheme**. Lastly, a judgement of the form $\Gamma \vdash e : \tau$ is read as follows — *Under the type environment* Γ *the expression* e *is well-typed with* τ.

Example 5 (Type environment)**.** $\Gamma = \{x \mapsto \forall\alpha, \beta.\alpha \rightarrow \beta\}$

2.2.2.3 The Tyvar Map

The type variables map, tyvars, is a map from a type to the set of type variables occurring in the given type. In the following we will show the operations we can do with the tyvars map.

$\boxed{\text{tyvars}(\tau)}$

By case distinction of τ we get four cases

$\boxed{\tau = \pi}$

The result of the operation will be the empty set $\emptyset$.

$\boxed{\tau = \alpha}$

The result of the operation will be the set with α i.e. $\{\alpha\}$.

$\boxed{\begin{array}{l}\tau = \tau_1 \rightarrow \tau_2 \\ \tau = \tau_1 \times \tau_2\end{array}}$

The result of these two operation is defined as $\text{tyvars}(\tau_1) \cup \text{tyvars}(\tau_2)$. See Appendix A for a summary of the tyvars operations on types.

$\boxed{\text{tyvars}(\sigma)}$

If $\sigma = \forall\alpha_1 \ldots \alpha_n.\tau$ then we have that

$$\begin{aligned}\text{tyvars}(\sigma) &= \text{tyvars}(\forall\alpha_1 \ldots \alpha_n.\tau) \\ &= \text{tyvars}(\tau) \setminus \bigcup_{i=1}^{n} \alpha_i\end{aligned}$$

which is all type variables that occur in τ and are free in σ.

$\boxed{\text{tyvars}(\Gamma)}$

This operation is defined as $\bigcup_{\sigma \in \text{Range } \Gamma} \text{tyvars}(\sigma)$.

Example 6 (The tyvars map applied on a type environment)**.** Let $\Gamma = \{\text{x} \mapsto \forall\alpha, \gamma.(\alpha \times \beta) \times \gamma\}$ then

$$\begin{aligned}\text{tyvars}(\Gamma) &= \text{tyvars}(\forall\alpha, \gamma.(\alpha \times \beta) \times \gamma) \\ &= \text{tyvars}((\alpha \times \beta) \times \gamma) \setminus \{\alpha, \gamma\} \\ &= (\text{tyvars}(\alpha \times \beta) \cup \text{tyvars}(\gamma)) \setminus \{\alpha, \gamma\} \\ &= (\text{tyvars}(\alpha) \cup \text{tyvars}(\beta) \cup \text{tyvars}(\gamma)) \setminus \{\alpha, \gamma\} \\ &= \{\alpha, \beta, \gamma\} \setminus \{\alpha, \gamma\} \\ &= \{\beta\}\end{aligned}$$

Furthermore, whenever we write *fresh* type variables we mean type variables that do not exist in the *tyvars*(Γ) set nor in any of the bound variables in the range of Γ. Finally, a type τ is a monotype $\mu \in \mathbf{Type}$ if $\text{tyvars}(\tau) = \emptyset$.

2.2.2.4 Substitutions

A substitution S is a map from a type variable to a type, $S : \mathbf{TyVar} \to \mathbf{Type}$. We let the identity substitution be denoted *ID*. Moreover, every substitution is the identity substitution outside its domain and range. A substitution is *ground* if for all types τ in the range of any substitution S is a monotype. Similarly to the last section, we will define the substitution operations. Parentheses can be disregarded.

$\boxed{S(\tau)}$

By case distinction of τ we find out what the result of the operation would be.

$\boxed{\tau = \pi}$

The result of the substitution applied to the nullary type constructor π is π.

$\boxed{\tau = \alpha}$

If $\{\alpha \mapsto \tau_1\} \subseteq S$ then $S\alpha = \tau_1$ else $S\alpha = \alpha$.

Example Let $S = \{\beta \mapsto \gamma\}$. If $\alpha = \beta$ then $S\alpha = S\beta = \gamma$. However, if $\alpha = \gamma$ then $S\alpha = S\gamma = \gamma$.

$\boxed{\tau = \tau_1 \to \tau_2}$

The result of the substitution applied to an abstraction is defined as $S\tau_1 \to S\tau_2$. For the sake of proofs introduced later we explicitly write it as a definition

Definition 2.2.1. *For any substitution S and any type of the form $\tau_1 \to \tau_2$ then $S(\tau_1 \to \tau_2) = S\tau_1 \to S\tau_2$*

$\boxed{\tau = \tau_1 \times \tau_2}$

The result of the substitution applied to a tuple is defined as $S\tau_1 \times S\tau_2$. For the sake of proofs introduced later we explicitly write it as a definition

Definition 2.2.2. *For any substitution S and any type of the form $\tau_1 \times \tau_2$ then $S(\tau_1 \times \tau_2) = S\tau_1 \times S\tau_2$*

See Appendix A for a summary of the substitution operations on types.

$\boxed{S(\sigma)}$

For this operation, we will use the notation $\{\alpha_i \mapsto \beta_i\} = \{\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n\}$ where n is the number of bound variables in the respective σ. We would like to define $S\sigma$ such that the following definition holds for $\sigma_1 = \sigma$ and $\sigma_2 = S\sigma$.

Definition 2.2.3. *Let $\sigma_1 = \forall\alpha_1 \ldots \alpha_n.\tau_1$ and $\sigma_2 = \forall\beta_1 \ldots \beta_n.\tau_2$ be type schemes and S be a substitution. We will write $\sigma_1 \xrightarrow{S} \sigma_2$ if all of the following holds*

1. $m = n$
2. $\{\alpha_i \mapsto \beta_i\}$ is a bijection and $\beta_i \notin \bigcup_{\tau \in \text{Range } S} \text{tyvars}(\tau)$
3. $(S \pm \{\alpha_i \mapsto \beta_i\})\tau_1 = \tau_2$

Definition 2.2.4. *Given a substitution S, a typescheme $\sigma = \forall\alpha_1 \dots \alpha_n.\tau$, and n fresh variables $\beta_1, \dots, \beta_n$, $S\sigma = S(\forall\alpha_1 \dots \alpha_n.\tau) = \forall\beta_1 \dots \beta_n.(S \pm \{\alpha_i \mapsto \beta_i\})\tau$*

Lemma 2.2.1. *For any typescheme σ and any substitution S, $\sigma \xrightarrow{S} S\sigma$*

Proof of lemma 2.2.1. The first condition obviously holds since we are using $1, \dots, n$ for the numbering of the bound variables in both σ and $S\sigma$. The second condition holds since $\beta_i, \dots, \beta_n$ are fresh type variables and thus are not present anywhere else. We see that in our case $\tau_1 = \tau$ and $\tau_2 = (S \pm \{\alpha_i \mapsto \beta_i\})\tau_1$ which is exactly what the third condition requires. Thus, the third condition holds as well. □

Example 7 (Substitution on a type scheme)**.** Let $\sigma = \forall\alpha_1.\alpha \to \alpha_1$ and $S = \{\alpha \mapsto \mathit{bool}, \alpha_1 \mapsto \mathit{int}\}$ then

$$\begin{aligned} S\sigma &= S(\forall\alpha_1.\alpha \to \alpha_1) \\ &= \forall\beta_1.(S \pm \{\alpha_1 \mapsto \beta_1\})(\alpha \to \alpha_1) \\ &= \forall\beta_1.(\{\alpha \mapsto \mathit{bool}, \alpha_1 \mapsto \mathit{int}\} \pm \{\alpha_1 \mapsto \beta_1\})(\alpha \to \alpha_1) \\ &= \forall\beta_1.\{\alpha \mapsto \mathit{bool}, \alpha_1 \mapsto \beta_1\}(\alpha \to \alpha_1) \\ &= \forall\beta_1.\mathit{bool} \to \beta_1 \end{aligned}$$

Note that the $\pm$ operator prioritizes the second operand which is why we get $\{\alpha \mapsto \mathit{bool}, \alpha_1 \mapsto \mathit{int}\} \pm \{\alpha_1 \mapsto \beta_1\} = \{\alpha \mapsto \mathit{bool}, \alpha_1 \mapsto \beta_1\}$.

$\boxed{S(\Gamma)}$

Similarly to $S\sigma$, we would like to define $S\Gamma$ such that the following definition holds for $\Gamma_1 = \Gamma$ and $\Gamma_2 = S\Gamma$.

Definition 2.2.5. *Let Γ_1, Γ_2 be type environments. We write $\Gamma_1 \xrightarrow{S} \Gamma_2$ if the following holds*

1. $\text{Dom } \Gamma_1 = \text{Dom } \Gamma_2$
2. $\forall x \in \text{Dom } \Gamma_1, \Gamma_1(x) \xrightarrow{S} \Gamma_2(x)$

Definition 2.2.6. *Given a substitution S and a type environment Γ,*

$$S\Gamma = \bigcup_{x \in Dom\ \Gamma} \{x \mapsto S(\Gamma(x))\}$$

Lemma 2.2.2. *For any type environment Γ and any substitution S, $\Gamma \xrightarrow{S} S\Gamma$*

Proof of lemma 2.2.2. The lemma is a special case of 2.2.5. By definition 2.2.6 we have Dom $S\Gamma$ which should be equal to Dom Γ so the first condition holds. We see that the second condition in our case is $\Gamma(x) \xrightarrow{S} S\Gamma(x)$ which by our definition is the same as $\Gamma(x) \xrightarrow{S} S(\Gamma(x))$. Since Γ is a map to type schemes we get $\sigma \xrightarrow{S} S\sigma$ for some σ and therefore the second condition must hold by lemma 2.2.1.

Example 8 (Substitution on a type environment)**.** Let $\Gamma = \{\mathrm{x} \mapsto \forall\alpha_1.\alpha \to \alpha_1, y \mapsto \forall\alpha_1, \alpha_2.\alpha \to int\}$ and $S = \{\alpha \mapsto int\}$ then

$$\begin{aligned} S\Gamma &= \{\mathrm{x} \mapsto S(\Gamma(x))\} \cup \{\mathrm{y} \mapsto S(\Gamma(y))\} \\ &= \{\mathrm{x} \mapsto S(\forall\alpha_1.\alpha \to \alpha_1)\} \cup \{\mathrm{y} \mapsto S(\forall\alpha_1, \alpha_2.\alpha \to int)\} \\ &= \{\mathrm{x} \mapsto \forall\beta_1.int \to \beta_1\} \cup \{\mathrm{y} \mapsto \forall\beta_1, \beta_2.int \to int\} && \text{By definition 2.2.4} \\ &= \{\mathrm{x} \mapsto \forall\beta_1.int \to \beta_1, \mathrm{y} \mapsto \forall\beta_1, \beta_2.int \to int\} \end{aligned}$$

□

$\boxed{S_n \dots S_2 S_1}$

With the operations being defined we look at the composition of substitutions. It is is defined generally as follows

$$S_n \dots S_2 S_1 = \bigcup_{\substack{\alpha \in \text{Dom } S_1, \\ \tau = S_1\alpha}} \{\alpha \mapsto S_n \dots S_2 \tau\}$$

The order of evaluation is from right to left. For instance, for three substitutions S_1, S_2, S_3 then for any type τ then $S_3 S_2 S_1 \tau = S_3(S_2(S_1\tau))$.

Example 9 (Composite substitution)**.** Assume three substitutions S_1, S_2, S_3

$$\begin{aligned} S_1 &= \{\alpha \mapsto \beta\} \\ S_2 &= \{\beta \mapsto \gamma\} \\ S_3 &= \{\gamma \mapsto \delta\} \end{aligned}$$

and let the composite substitution be $S_3 S_2 S_1$ then

$$\begin{aligned} S_3 S_2 S_1 &= \{\alpha \mapsto S_3 S_2 \beta\} \\ &= \{\alpha \mapsto S_3 \gamma\} \\ &= \{\alpha \mapsto \delta\} \end{aligned}$$

2.2.2.5 Instantiation

The $>$ operator in the S-Lookup rule indicates that a type τ_1 is an instantiation of a type scheme $\sigma = \forall\alpha_1 \dots \alpha_n.\tau_2$ written as $\sigma > \tau_1$. More specifically, τ_1 is an instantiation of σ if there exists a substitution S with domain $\{\alpha_1, \dots, \alpha_n\}$ and range $\{\beta_1, \dots, \beta_n\}$ where each of the type variables in the range are fresh such that $\tau_1 = S\tau_2$.

Example 10 (Instantiation)**.** Let $\sigma = \forall\alpha.\alpha \to \alpha$ and $\Gamma = \{x \mapsto \forall\beta.\beta \to int, y \mapsto \forall\gamma.\gamma \times \gamma\}$ and $S = \{\alpha \to \delta\}$. Obviously, δ is a fresh type variable. We have that $\tau_1 = S(\alpha \to \alpha) = S\alpha \to S\alpha = \delta \to \delta$.

2.2.2.6 Generalization

In the S-Let rule, the $Clos_\Gamma$ operation on any type τ is defined as $Clos_\Gamma\tau = \forall\alpha_1, \dots, \alpha_n.\tau$ where $\{\alpha_1, \dots, \alpha_n\}$ is defined as $\text{tyvars}(\tau) \setminus \text{tyvars}(\Gamma)$. Thus, this operation tries to generalize over each type variable in τ but only if it does not already exist in the free type variables of Γ.

Example 11 (Simple generalization)**.** If $\tau = \alpha \to \alpha$ and $\Gamma = \emptyset$ then to compute $Clos_\Gamma \tau$ we first compute $\text{tyvars}(\tau) \setminus \text{tyvars}(\Gamma) = \{\alpha\} \setminus \emptyset = \{\alpha\}$. Thus, $Clos_\Gamma \tau = \forall\alpha.\alpha \to \alpha$.

Example 12 (Complicated generalization)**.** A more complicated version would be if $\tau = \alpha \to (int \times \beta) \to (\gamma \times string) \to \delta \to int$ and $\Gamma = \{x \mapsto \forall\eta.\gamma \to \eta, y \mapsto \forall\alpha.\alpha \to \alpha\}$ then

$$\begin{aligned}\text{tyvars}(\tau) \setminus \text{tyvars}(\Gamma) &= \text{tyvars}(\alpha \to (int \times \beta) \to (\gamma \times string) \to \delta \to int) \\ &\quad \setminus \text{tyvars}(\forall\eta.\gamma \to \eta) \cup \text{tyvars}(\forall\alpha.\alpha \to \alpha) \\ &= \{\alpha, \beta, \gamma, \delta\} \setminus \{\gamma\} \\ &= \{\alpha, \beta, \delta\}\end{aligned}$$

which means we can quantify τ with α, β, δ and thus $Clos_\Gamma \tau = \forall\alpha, \beta, \delta.\alpha \to (int \times \beta) \to (\gamma \times string) \to \delta \to int$. See Appendix B for a more detailed derivation.

2.2.2.7 Derivation Examples

In Appendix C, we show a monomorphic and polymorphic type inference derivation for two different expressions. In the examples, we make use of what we have covered so far. It is strongly encouraged to take a look at these examples.

2.3 Consistency Proof

Although the static semantics is independent of the dynamic semantics we would still like to have consistency between them. The proofs presented in this section imply that a well-typed program cannot evaluate to *wrong*.

Before going into the proofs, we need a relation between values and types

Definition 2.3.1. *We say that v has monotype μ written $\vDash v : \mu$ if one of the following holds*

- *$v = bv$ and bv has the correct type i.e. for instance for $bv =$ "myVariable" then $\mu = string$ etc.*
- *$v = [x, e, E]$ and $\mu = \mu_1 \to \mu_2$ for some monotypes μ_1, μ_2 and for all v_1, r if $\vDash v_1 : \mu_1$ and $E \pm \{x \mapsto v_1\} \vdash e \to r$ then $r \neq wrong$ and $\vDash r : \mu_2$*

This is extended as follows.

Definition 2.3.2. *Let Γ° be a type environment that ranges over all closed type environments then we define the following*

$$\begin{aligned}&\vDash v : \tau \text{ if for all total ground substitutions } S \text{ we have } \vDash v : S\tau \\ &\vDash v : \forall\alpha_1 \ldots \alpha_n.\tau \text{ if } \vDash v : \tau \\ &\vDash E : \Gamma^\circ \text{ if } Dom\ E = Dom\ \Gamma^\circ \text{ and } \vDash E(x) : \Gamma^\circ(x) \text{ for all } x \in Dom\ E \\ &\vDash E : \Gamma \text{ if } \vDash E : \Gamma^\circ \text{ and for a ground substitution } S \text{ we have } \Gamma \xrightarrow{S} \Gamma^\circ\end{aligned}$$

We present consistency proofs for tuples, fst, and snd. The proofs for the other expressions can be found in [4] and the base cases for integers, booleans, and strings are trivial and therefore omitted.

Theorem 2.3.1 (Consistency of Static and Dynamic Semantic)**.**
If $\vDash E : \Gamma$ and $\Gamma \vdash e : \tau$ and $E \vdash e \to r$ then $r \neq wrong$ and $\vDash r : \tau$

Proof of theorem 2.3.1. By structural induction on e.

For all cases, assume that

$$\vDash E : \Gamma \tag{2.1}$$

$\boxed{e = (e_1, e_2)}$

The type inference of the expression must have been of the form

$$\frac{\Gamma \vdash e_1 \ : \ \tau_1 \qquad \Gamma \vdash e_2 \ : \ \tau_2}{\Gamma \vdash (e_1, e_2) \ : \ \tau_1 \times \tau_2} \ \text{S-TUPLE} \tag{2.2}$$

The evaluation must have been one of the following

$$\frac{E \vdash e_1 \to v_1 \qquad E \vdash e_2 \to v_2}{E \vdash (e_1, e_2) \to (v_1, v_2)} \ \text{D-TUPLE} \tag{2.3}$$

$$\frac{E \vdash e_1 \to \mathit{wrong} \qquad E \vdash e_2 \to v_2}{E \vdash (e_1, e_2) \to \mathit{wrong}} \ \text{D-TUPLE-WRONG1} \tag{2.4}$$

$$\frac{E \vdash e_1 \to v_1 \qquad E \vdash e_2 \to \mathit{wrong}}{E \vdash (e_1, e_2) \to \mathit{wrong}} \ \text{D-TUPLE-WRONG2} \tag{2.5}$$

We see that $e_1 \to r_1$ and $e_2 \to r_2$ for some $r_1, r_2 \in$ Results.

By our I.H. and 2.1, the first premise in 2.2, (2.3 / 2.4 / 2.5) we get

$$r_1 \neq \mathit{wrong} \tag{2.6}$$

$$\vDash r_1 \ : \ \tau_1 \tag{2.7}$$

This means the evaluation cannot have been 2.4.

Furthermore, by our I.H and 2.1, the second premise in 2.2, (2.3 / 2.5) we get

$$r_2 \neq \mathit{wrong} \tag{2.8}$$

$$\vDash r_2 \ : \ \tau_2 \tag{2.9}$$

This means the evaluation cannot have been 2.5. The evaluation must have been 2.3, meaning that $r_1 = v_1$, $r_2 = v_2$, and $r = (v_1, v_2) \in \text{Val} \times \text{Val} = \text{Val}$. Since a Val cannot be *wrong* then $r \neq \mathit{wrong}$.

From 2.7 and 2.9 we have that $\vDash r_1 : \tau_1$ and $\vDash r_2 : \tau_2$ respectively. Since $r = (v_1, v_2) = (r_1, r_2)$ and $\vDash (r_1, r_2) : \tau_1 \times \tau_2$, we must have that $\vDash r : \tau_1 \times \tau_2$.

$\boxed{e = \mathbf{fst}(e_1, e_2)}$

The type inference of the expression must have been of the form

$$\frac{\text{S-Fst}}{\Gamma \vdash (e_1, e_2) \ : \ \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{fst}(e_1, e_2) \ : \ \tau_1} \tag{2.10}$$

The evaluation must have been one of the following

$$\frac{\text{D-Fst}}{E \vdash e_1 \to v_1 \quad E \vdash e_2 \to v_2}{E \vdash \mathbf{fst}(e_1, e_2) \to v_1} \tag{2.11}$$

$$\frac{\text{D-Fst-Wrong1}}{E \vdash e_1 \to \mathit{wrong} \quad E \vdash e_2 \to v_2}{E \vdash \mathbf{fst}(e_1, e_2) \to \mathit{wrong}} \tag{2.12}$$

$$\frac{\text{D-Fst-Wrong2}}{E \vdash e_1 \to v_1 \quad E \vdash e_2 \to \mathit{wrong}}{E \vdash \mathbf{fst}(e_1, e_2) \to \mathit{wrong}} \tag{2.13}$$

We see that $e_1 \to r_1$ and $e_2 \to r_2$ for some $r_1, r_2 \in$ Results.

By our I.H. and 2.1, 2.10, (2.11 / 2.12 / 2.13) we get

$$r_1 \neq \mathit{wrong} \tag{2.14}$$
$$\vDash r_1 \ : \ \tau_1 \tag{2.15}$$

This means the evaluation cannot have been 2.12.

By our I.H. and 2.1, 2.10, (2.11 / 2.13) we get

$$r_2 \neq \mathit{wrong} \tag{2.16}$$
$$\vDash r_2 \ : \ \tau_2 \tag{2.17}$$

This means the evaluation cannot have been 2.13. The evaluation must have been 2.11, meaning that $r_1 = v_1$, $r_2 = v_2$, and $r = v_1 \in$ Val. Since a Val cannot be *wrong* then $r \neq \mathit{wrong}$.

From 2.15 we have that $\vDash r_1 : \tau_1$. Since $r = v_1 = r_1$ and $\vDash r_1 : \tau_1$, we must have that $\vDash r : \tau_1$.

$\boxed{e = \mathbf{snd}(e_1, e_2)}$

Similar to the $e = \mathbf{fst}(e_1, e_2)$ case and thus omitted here. However, the proof for this case can be found in Appendix D.

□

Chapter 3

Type Inference - Algorithm $\mathcal{W}$

With the language and its semantics being defined we will now dig into how an actual implementation of a type inference algorithm based on the language and its semantics from Chapter 2 works. In this chapter, we will present an inefficient type inference algorithm and soundness proofs for it. We will not provide completeness proof for the algorithm but in [1] they are presented for the lambda calculus with let-polymorphism.

3.1 The Algorithm

In this section, we present the first type inference algorithm denoted $\mathcal{W}$. It is heavily based on substitutions, but also instantiation, generalization, and an operation called unification. $\mathcal{W}$ is an inefficient algorithm due to the heavy usage of substitutions, however, in Chapter 4 we will investigate how to achieve a more efficient algorithm. The following two sections present the idea of unification and the pseudocode for $\mathcal{W}$.

3.1.1 Unification

The idea of unification is to *unify* two types τ_1, τ_2 i.e. making them equivalent to each other. In $\mathcal{W}$, the unification makes use of substitutions. Particularly, unification in the algorithm is defined as $\mathit{Unify} : (\mathbf{Type} \times \mathbf{Type}) \to \mathbf{Substitution}$ where the range is the set of all substitutions.

The possible operations for *Unify* are defined below

$\boxed{\mathit{Unify}(\pi_1, \pi_2)}$

If $\pi_1 = \pi_2$ then we return the empty substitution *ID*, otherwise we fail.

$\boxed{\mathit{Unify}(\alpha_1, \alpha_2)}$

If $\alpha_1 = \alpha_2$ then we return the empty substitution *ID* otherwise we make α_1 equal to α_2 and thus return $\{\alpha_1 \mapsto \alpha_2\}$.

$\boxed{\begin{array}{l}\mathit{Unify}(\alpha, \tau) \\ \mathit{Unify}(\tau, \alpha)\end{array}}$

If $\alpha \in \text{tyvars}(\tau)$ then we fail, otherwise we make α equal to τ and thus return $\{\alpha \mapsto \tau\}$.

$\boxed{\begin{array}{l}\mathit{Unify}(\tau_{11} \to \tau_{12}, \tau_{21} \to \tau_{22}) \\ \mathit{Unify}(\tau_{11} \times \tau_{12}, \tau_{21} \times \tau_{22})\end{array}}$

The two operations above do the following. First, we get $S_1 = \mathit{Unify}(\tau_{11}, \tau_{21})$ then $S_2 = \mathit{Unify}(S_1\tau_{21}, S_1\tau_{22})$. If both succeed then the substitution composition S_2S_1 is returned.

In all other possible cases the unification fails. The operations' semantics are taken from [3]. One thing to note is the criteria we have in the $\mathit{Unify}(\alpha, \tau)$ and $\mathit{Unify}(\tau, \alpha)$ operations. The criteria is needed to prevent creation of infinite types. For instance, consider $\tau = \alpha \times \alpha$ then $\mathit{Unify}(\alpha, \tau)$ must fail since there is no finite type solving the symbolic equation $\alpha = (\alpha \times \alpha)$.

Example 13 (Unification)**.** *Let the Unify function take the following two types as parameter* $\tau_1 = int \to \alpha$ *and* $\tau_2 = int \to (\beta \times \gamma)$ *then*

$$\begin{aligned}
\mathit{Unify}(\tau_1, \tau_2) &= \mathit{Unify}(int \to \alpha, int \to (\beta \times \gamma)) \\
&= \mathit{Unify}(\alpha, \beta \times \gamma) && \mathit{Since}\ \mathit{Unify}(int, int) = ID \\
&= \{\alpha \mapsto \beta \to \gamma\} && \mathit{Since}\ \alpha \notin \mathit{tyvars}(\beta \to \gamma)
\end{aligned}$$

Definition 3.1.1. *If* $S = Unify(\tau_1, \tau_2)$ *then* $S\tau_1 = S\tau_2$

This definition states that the two types τ_1, τ_2 we provide to the *Unify* function are equal under the returned substitution, i.e. $S\tau_1 = S\tau_2$.

3.1.2 Pseudocode

The pseudocode for $\mathcal{W}$ is presented below

$\mathcal{W}(\Gamma, e) =$ case e of
$i \implies (ID, int)$
$b \implies (ID, bool)$
$s \implies (ID, string)$
$x \implies$
  $if\ x \notin$ **Dom** Γ **then fail**
  else let $\forall\alpha_1 \ldots \alpha_n.\tau = \Gamma(x)$
  $\beta_1 \ldots \beta_n$ be new
  in $(ID, \{\alpha_i \mapsto \beta_i\}\tau)$
$\lambda x.e_1 \implies$
  let α be a new type variable
  $(S_1, \tau_1) = \mathcal{W}(\Gamma \pm \{x \mapsto \alpha\}, e_1)$
  in $(S_1, S_1(\alpha) \to \tau_1)$
$e_1\, e_2 \implies$
  $(S_1, \tau_1) = \mathcal{W}(\Gamma, e_1)$
  $(S_2, \tau_2) = \mathcal{W}(S_1\Gamma, e_2)$
  let α be a new type variable
  $S_3 = \mathit{Unify}(S_2(\tau_1), \tau_2 \to \alpha)$
  in $(S_3S_2S_1, S_3(\alpha))$

let $x = e_1$ **in** $e_2 \implies$
  $(S_1, \tau_1) = \mathcal{W}(\Gamma, e_1)$
  $(S_2, \tau_2) = \mathcal{W}(S_1\Gamma \pm \{x \mapsto \mathrm{Clos}_{S_1\Gamma\tau_1}\}, e_2)$
  in (S_2S_1, τ_2)
$(e_1, e_2) \implies$
  $(S_1, \tau_1) = \mathcal{W}(\Gamma, e_1)$
  $(S_2, \tau_2) = \mathcal{W}(S_1\Gamma, e_2)$
  $(S_2S_1, S_2\tau_1 \times \tau_2)$
fst $(e_1, e_2) \implies$
  let $(S_1, \tau_1 \times \tau_2) = \mathcal{W}(\Gamma, (e_1, e_2))$
  in (S_1, τ_1)
snd $(e_1, e_2) \implies$
  let $(S_1, \tau_1 \times \tau_2) = \mathcal{W}(\Gamma, (e_1, e_2))$
  in (S_1, τ_2)
$\_ \implies$ **fail**

Figure 3.1: Pseudocode for $\mathcal{W}$

where _ is the wildcard used to show all other cases, for instance, **fst** e_1 where e_1 is an expression and not a tuple.

3.2 Soundness Proof

In this section we prove the soundness of $\mathcal{W}$ for each expression in our grammar. $\mathcal{W}$ is sound in the following sense

Theorem 3.2.1 (Soundness of $\mathcal{W}$)**.**
If $(S, \tau) = \mathcal{W}(\Gamma, e)$ *succeeds and* $\Gamma \xrightarrow{S} \Gamma'$ *then* $\Gamma' \vdash e : \tau$

If $\mathcal{W}$ succeeds for some Γ, e and returns some S, τ and the relation $\Gamma \xrightarrow{S} \Gamma'$ is satisfied then e is well-typed with τ under Γ'. To prove an expression is sound we will also need the following lemma from [4]

Lemma 3.2.2. *If* $\Gamma \vdash e : \tau$ *and* $\Gamma \xrightarrow{S} \Gamma'$ *then* $\Gamma' \vdash e : S\,\tau$

We will not provide proofs for this lemma, however a proof for let expressions is provided in [4]. Furthermore, we will use lemma 2.2.2 which states that $\Gamma \xrightarrow{S} S\Gamma$ always holds.

Proof of theorem 3.2.1. By structural induction on e.

$\boxed{e = i}$

From the algorithm we have

$$(ID, int) = \mathcal{W}(\Gamma, i) \tag{3.1}$$

By lemma 2.2.2 where $\Gamma = \Gamma$ and $S = ID$

$$\Gamma \xrightarrow{ID} ID\Gamma \tag{3.2}$$

By filling in 3.1, 3.2 in 3.2.1 we get that we want to prove $ID\Gamma \vdash i : int$.

By our S-Int rule we have the conclusion since it has no premise(s)

$$\frac{}{\Gamma \vdash i \ : \ int}\ \text{S-Int} \tag{3.3}$$

By letting $\Gamma = ID\Gamma$[1] we have $ID\Gamma \vdash i : int$ which is exactly what we wanted to prove.

$\boxed{e = b}$

From the algorithm we have

$$(ID, bool) = \mathcal{W}(\Gamma, b) \tag{3.4}$$

By lemma 2.2.2 where $\Gamma = \Gamma$ and $S = ID$

$$\Gamma \xrightarrow{ID} ID\Gamma \tag{3.5}$$

By filling in 3.4, 3.5 in 3.2.1 we get that we want to prove $ID\Gamma \vdash b : bool$.

By our S-Bool rule we have $ID\Gamma \vdash b : bool$ which is what we wanted to show.

$\boxed{e = s}$

From the algorithm we have

$$(ID, string) = \mathcal{W}(\Gamma, s) \tag{3.6}$$

[1] This is allowed because the Γ in the S-Int rule is for all quantified i.e. it is for an arbitrary type environment Γ

By lemma 2.2.2 where $\Gamma = \Gamma$ and $S = ID$

$$\Gamma \xrightarrow{ID} ID\Gamma \tag{3.7}$$

By filling in 3.6, 3.7 in 3.2.1 we get that we want to prove $ID\Gamma \vdash s : string$

By our S-String rule we have $ID\Gamma \vdash s : string$ which is what we wanted to show.

$\boxed{e = x}$

From our algorithm we have

$$(ID, \{\alpha_i \mapsto \beta_i\}\tau) = \mathcal{W}(\Gamma, x) \tag{3.8}$$

By lemma 2.2.2 where $\Gamma = \Gamma$ and $S = ID$

$$\Gamma \xrightarrow{ID} ID\Gamma \tag{3.9}$$

By filling in 3.8, 3.9 in 3.2.1 we get that we want to prove

$$ID\,\Gamma \vdash x \ : \ \{\alpha_i \mapsto \beta_i\}\tau \tag{3.10}$$

By our S-Lookup rule

$$\text{S-Lookup}\quad \frac{x \in \text{Dom } \Gamma \quad \Gamma(x) > \tau}{\Gamma \vdash x : \tau} \tag{3.11}$$

we have two premises. From the algorithm we have that

$$if\ x \notin \textbf{Dom}\ \Gamma\ \textbf{then fail else let}\ \forall\alpha_1 \ldots \alpha_n.\tau = \Gamma(x) \tag{3.12}$$

Let us consider the two cases — $x \notin$ **Dom** Γ and $x \in$ **Dom** Γ. Assume $x \notin$ **Dom** Γ then by 3.12 $\mathcal{W}$ fails. However, by assumption $\mathcal{W}$ succeeds (by Theorem 3.2.1) and therefore we have a contradiction. Thus, $x \in$ **Dom** Γ. Furthermore, we have from the algorithm

$$\beta_1 \ldots \beta_n \text{ be new} \tag{3.13}$$

and the type from 3.10 corresponds to τ in $\Gamma(x) > \tau$.

Thus, we have that the two premises hold and by the conclusion from 3.11 we have $ID\Gamma \vdash x : \{\alpha_i \mapsto \beta_i\}\tau$ which is what we wanted to prove.

$\boxed{e = \lambda x.e_1}$

We want to show $S_1\Gamma \vdash \lambda x.e_1 : S_1\alpha \to \tau_1$.

From our algorithm we have

$$(S_1, \tau_1) = \mathcal{W}(\Gamma \pm \{\mathrm{x} \mapsto \alpha\}, e_1) \tag{3.14}$$

By lemma 2.2.2 where $\Gamma = \Gamma \pm \{\mathrm{x} \mapsto \alpha\}$ and $S = S_1$

$$\Gamma \pm \{x \mapsto \alpha\} \xrightarrow{S_1} S_1(\Gamma \pm \{\mathrm{x} \mapsto \alpha\}) \tag{3.15}$$

By our I.H. and 3.14, 3.15 we get

$$S_1(\Gamma \pm \{\mathrm{x} \mapsto \alpha\}) \vdash e_1 \ : \ \tau_1 \tag{3.16}$$

By applying the substitution S_1 on α and Γ we get

$$S_1\Gamma \pm \{\mathrm{x} \mapsto S_1\alpha\} \vdash e_1 \ : \ \tau_1 \tag{3.17}$$

Thus, by letting 3.17 be the premise of the S-Lambda rule

$$\frac{\text{S-Lambda}}{\dfrac{\Gamma \pm \{\mathrm{x} \mapsto \tau'\} \vdash e_1 \ : \ \tau}{\Gamma \vdash \lambda x.e_1 \ : \ \tau' \to \tau}} \tag{3.18}$$

we can conclude $S_1\Gamma \vdash \lambda x.e_1 : S_1\alpha \to \tau_1$ which is what we wanted to show.

$\boxed{e = e_1\ e_2}$

We want to show $S_3S_2S_1\Gamma \vdash e_1e_2 : S_3\alpha$.

From our algorithm we have

$$(S_1, \tau_1) = \mathcal{W}(\Gamma, e_1) \tag{3.19}$$

By lemma 2.2.2 where $\Gamma = \Gamma$ and $S = S_1$ we have

$$\Gamma \xrightarrow{S_1} S_1\Gamma \tag{3.20}$$

By our I.H. and 3.19, 3.20 we have

$$S_1\Gamma \vdash e_1 : \tau_1 \tag{3.21}$$

From our algorithm we have

$$(S_2, \tau_2) = \mathcal{W}(S_1\Gamma, e_2) \tag{3.22}$$

By lemma 2.2.2 where $\Gamma = S_1\Gamma$ and $S = S_2$ we have

$$S_1\Gamma \xrightarrow{S_2} S_2S_1\Gamma \tag{3.23}$$

By our I.H. and 3.22, 3.23 we have

$$S_2S_1\Gamma \vdash e_2 : \tau_2 \tag{3.24}$$

By lemma 3.2.2 and 3.21, 3.23 we have

$$S_2S_1\Gamma \vdash e_1 : S_2\tau_1 \tag{3.25}$$

From our algorithm we have

$$\begin{aligned} &\textbf{let } \alpha \text{ be a new type variable} \\ &S_3 = \mathit{Unify}(S_2(\tau_1), \tau_2 \to \alpha) \end{aligned} \tag{3.26}$$

By definition 3.1.1 and 3.26 we have

$$S_3S_2\tau_1 = S_3(\tau_2 \to \alpha) \tag{3.27}$$

By definition 2.2.1 and the RHS of the equality sign in 3.27 we have

$$S_3S_2\tau_1 = S_3(\tau_2 \to \alpha) = S_3\tau_2 \to S_3\alpha \tag{3.28}$$

By lemma 2.2.2 where $\Gamma = S_2S_1\Gamma$ and $S = S_3$ we have

$$S_2S_1\Gamma \xrightarrow{S_3} S_3S_2S_1\Gamma \tag{3.29}$$

By lemma 3.2.2 and 3.25, 3.29 we have

$$S_3S_2S_1\Gamma \vdash e_1 : S_3S_2\tau_1 \tag{3.30}$$

By the equality presented in 3.28 and by replacing the type in 3.30 we get

$$S_3S_2S_1\Gamma \vdash e_1 : S_3\tau_2 \to S_3\alpha \tag{3.31}$$

By lemma 3.2.2 and 3.24, 3.29 we have

$$S_3S_2S_1\Gamma \vdash e_2 : S_3\tau_2 \tag{3.32}$$

Thus, by letting 3.31 and 3.32 be the premises of the S-App rule

$$\text{S-App}\ \frac{\Gamma \vdash e_1\ :\ \tau' \to \tau \qquad \Gamma \vdash e_2\ :\ \tau'}{\Gamma \vdash e_1\ e_2\ :\ \tau}$$

we can conclude $S_3S_2S_1\Gamma \vdash e_1e_2 : S_3\alpha$ which is what we wanted to show.

$\boxed{e = (\textbf{let}\ x = e_1\ \textbf{in}\ e_2)}$

We want to show $S_2S_1\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2\ :\ \tau_2$.

From our algorithm we have

$$(S_1, \tau_1) = \mathcal{W}(\Gamma, e_1) \tag{3.33}$$

By lemma 2.2.2 where $\Gamma = \Gamma$ and $S = S_1$ we have

$$\Gamma \xrightarrow{S_1} S_1\Gamma \tag{3.34}$$

By our I.H. and 3.33, 3.34 we have

$$S_1\Gamma \vdash e_1 : \tau_1 \tag{3.35}$$

From our algorithm we have

$$(S_2, \tau_2) = \mathcal{W}(S_1\Gamma \pm \{x \mapsto \text{Clos}_{S_1\Gamma\tau_1}\}, e_2) \tag{3.36}$$

By definition of the $Clos\tau$ operation it can not make the algorithm stop and we can proceed without any issues. Thus, by lemma 2.2.2 where $\Gamma = S_1\Gamma \pm \{\text{x} \mapsto \text{Clos}_{S_1\Gamma\tau_1}\}$ and $S = S_2$ we have

$$S_1\Gamma \pm \{x \mapsto \text{Clos}_{S_1\Gamma\tau_1}\} \xrightarrow{S_2} S_2(S_1\Gamma \pm \{x \mapsto \text{Clos}_{S_1\Gamma\tau_1}\}) \tag{3.37}$$

By our I.H. and 3.36, 3.37 we have

$$S_2(S_1\Gamma \pm \{x \mapsto \text{Clos}_{S_1\Gamma\tau_1}\}) \vdash e_2\ :\ \tau_2 \tag{3.38}$$

Which is the same as

$$S_2S_1\Gamma \pm \{x \mapsto \text{Clos}_{S_2S_1\Gamma}S_2\tau_1\} \vdash e_2\ :\ \tau_2 \tag{3.39}$$

By lemma 2.2.2 where $\Gamma = S_1\Gamma$ and $S = S_2$ we have

$$S_1\Gamma \xrightarrow{S_2} S_2S_1\Gamma \tag{3.40}$$

By lemma 3.2.2 and 3.35, 3.40 we have

$$S_2 S_1 \Gamma \vdash e_1 \ : \ S_2 \tau_1 \tag{3.41}$$

Thus, by letting 3.41, 3.39 be the premises of the S-Let rule

$$\text{S-Let}\ \frac{\Gamma \vdash e_1 \ : \ \tau_1 \qquad \Gamma \pm \{x \mapsto Clos_\Gamma \tau_1\} \vdash e_2 \ : \ \tau}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \ : \ \tau}$$

we can conclude that $S_2 S_1 \Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2$ which is what we wanted to show.

$\boxed{e = (e_1, e_2)}$

We want to show $S_2 S_1 \Gamma \vdash (e_1, e_2) \ : \ S_2 \tau_1 \times \tau_2$.

From our algorithm we have

$$(S_1, \tau_1) = \mathcal{W}(\Gamma, e_1) \tag{3.42}$$

By lemma 2.2.2 where $\Gamma = \Gamma$ and $S = S_1$ we have

$$\Gamma \xrightarrow{S_1} S_1 \Gamma \tag{3.43}$$

By our I.H. and 3.42, 3.43 we have

$$S_1 \Gamma \vdash e_1 : \tau_1 \tag{3.44}$$

From our algorithm we have

$$(S_2, \tau_2) = \mathcal{W}(S_1 \Gamma, e_2) \tag{3.45}$$

By lemma 2.2.2 where $\Gamma = S_1 \Gamma$ and $S = S_2$ we have

$$S_1 \Gamma \xrightarrow{S_2} S_2 S_1 \Gamma \tag{3.46}$$

By lemma 3.2.2 and 3.44, 3.46 we have

$$S_2 S_1 \Gamma \vdash e_1 : S_2 \tau_1 \tag{3.47}$$

By our I.H. and 3.45, 3.46 we have

$$S_2 S_1 \Gamma \vdash e_2 : \tau_2 \tag{3.48}$$

Thus, by letting 3.47, 3.48 be the premises of the S-Tuple rule

$$\text{S-Tuple}\ \frac{\Gamma \vdash e_1 \ : \ \tau_1 \qquad \Gamma \vdash e_2 \ : \ \tau_2}{\Gamma \vdash (e_1, e_2) \ : \ \tau_1 \times \tau_2}$$

we can conclude $S_2 S_1 \Gamma \vdash (e_1, e_2) : S_2 \tau_1 \times \tau_2$

$\boxed{e = \textbf{fst}(e_1, e_2)}$

We want to show $S_1 \Gamma \vdash \textbf{fst}\ (e_1, e_2) : \tau_1$

From our algorithm we have

$$(S_1, \tau_1 \times \tau_2) = \mathcal{W}(\Gamma, (e_1, e_2)) \tag{3.49}$$

By lemma 2.2.2 where $\Gamma = \Gamma$ and $S = S_1$ we have

$$\Gamma \xrightarrow{S_1} S_1\Gamma \tag{3.50}$$

By our I.H. and 3.49, 3.50 we have

$$S_1\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \tag{3.51}$$

Thus, by letting 3.51 be the premise of the S-Fst rule

$$\frac{\text{S-Fst}}{\Gamma \vdash (e_1, e_2) \ : \ \tau_1 \times \tau_2}{\Gamma \vdash \textbf{fst} \ (e_1, e_2) \ : \ \tau_1}$$

we can conclude $S_1\Gamma \vdash \textbf{fst} \ (e_1, e_2) : \tau_1$

$\boxed{e = \textbf{snd}(e_1, e_2)}$

Similar to the $e = \textbf{fst}(e_1, e_2)$ case and therefore omitted. However, the proof for this case can be found in Appendix D. □

3.3 Implementation

In this section we will implement $\mathcal{W}$ for the language defined in Chapter 2. We will make use of the pseudocode for $\mathcal{W}$ defined in this chapter and the language and its semantics.

3.3.1 Code

Our chosen programming language for the implementation is OCaml. It is based on the Hindley-Milner type system which makes it an obvious candidate for another language based on the same type system. The relevant files for $\mathcal{W}$ are presented in the following

The types defined in the Section 2.2.2.1 are implemented as follows

types.ml

```
module SS = Set.Make(Int)

exception Fail of string

type tyvar = int

type tycon =
| Int
| Bool
| String

type typ =
| TyCon of tycon
| TyVar of tyvar
| TyFunApp of {t1: typ; t2: typ}
| TyTuple of {t1: typ; t2: typ}

type typescheme =
TypeScheme of {tyvars: SS.t; tau: typ}

type program_variable = string
```

The grammar defined in Section 2.1 is implemented as an AST (Abstract Syntax Tree)

ast.ml

```
open Types

type bas_val =
| Int of int
| Bool of bool
| String of string

type exp =
  | BasVal of bas_val
  | Var of program_variable
  | Lambda of { id : program_variable;
  e1 : exp }
  | App of { e1 : exp; e2 : exp }
  | Let of { id : program_variable;
  e1 : exp; e2 : exp }
  | Tuple of { e1 : exp; e2 : exp }
  | Fst of exp
  | Snd of exp
```

The type environment Γ defined in Section 2.2.2.2 is implemented as follows

typeEnv.ml

```
open Types

module Gamma =
  Map.Make (struct
    type t = program_variable
    let compare = compare
end)

type ts_map = typescheme Gamma.t

let wrap_monotype tau =
TypeScheme {tyvars=SS.empty; tau}

let empty = Gamma.empty

let add k v t = Gamma.add k v t

let look_up k t = Gamma.find_opt k t

let remove k t = Gamma.remove k t

let bindings t = Gamma.bindings t

let map m t = Gamma.map m t
let counter = ref 0
let get_next_tyvar () =
  counter := !counter + 1;
  !counter
let reset () = counter := 0
```

A small helper file

utils.ml

```
open Types

module TE = TypeEnv

let new_tyvar () = TyVar (TE.get_next_tyvar())
let ( +- ) gamma (id, ts) = TE.add id ts
gamma
let ( !& ) t = TE.wrap_monotype t
let ( => ) t1 t2 = TyFunApp { t1; t2 }
let ( ** ) t1 t2 = TyTuple { t1; t2 }

let combine_sets sets =
List.fold_left
(fun a b -> SS.union a b) SS.empty sets

let assoc_or_else bindings key ~default =
Option.value (List.assoc_opt key bindings)
~default:default
```

The substitution S and its operations as defined in Section 2.2.2.4 are implemented as follows

substitution.ml

```
open Types

module TE = TypeEnv

module Substitution =
  Map.Make (struct
    type t = tyvar
    let compare = compare
end)

type map_type = typ Substitution.t
let empty: map_type = Substitution.empty
let add k v t : map_type =
Substitution.add k v t
let look_up k t: typ option =
Substitution.find_opt k t
let remove k t: map_type =
Substitution.remove k t
let get_or_else k t ~default =
  match look_up k t with
    | Some v -> v
    | None -> default
let apply t typ: typ =
  let rec subst typ' =
    match typ' with
      | TyCon _ -> typ'
      | TyVar tv ->
      get_or_else tv t ~default:typ'
      | TyFunApp {t1; t2} ->
      TyFunApp {t1 = subst t1; t2 = subst t2}
      | TyTuple {t1; t2} ->
      TyTuple {t1 = subst t1; t2 = subst t2}
  in
  subst typ
let apply_to_typescheme t
(TypeScheme{tyvars; tau}) =
TypeScheme{tyvars; tau=apply t tau}
let apply_to_gamma t gamma =
TE.map (apply_to_typescheme t) gamma
let map m (t: map_type) = Substitution.map m t
let union a b c =
Substitution.union a b c
let compose (s2: map_type) (s1: map_type) =
union (fun _ _ v2 -> Some v2) s2
(map (fun v -> apply s2 v) s1)
let bindings t = Substitution.bindings t
let of_seq s = Substitution.of_seq s
let of_list l = of_seq (List.to_seq l)
```

algorithmW.ml

```
open Types
open Utils

module A = Ast
module S = Substitution

let rec find_tyvars tau = match tau with
  | TyCon _ -> SS.empty
  | TyVar alpha -> SS.singleton alpha
  | TyFunApp { t1; t2 } ->
  SS.union (find_tyvars t1) (find_tyvars t2)
  | TyTuple { t1; t2 } ->
  SS.union (find_tyvars t1) (find_tyvars t2)

let find_free_tyvars
(TypeScheme {tyvars; tau}) =
  SS.diff (find_tyvars tau) tyvars

let clos gamma tau =
  let free_tyvars_tau = find_tyvars tau in
  let free_tyvars_gamma =
  combine_sets (List.map (fun (_, v) ->
  find_free_tyvars v) (TE.bindings gamma)) in
  TypeScheme { tyvars = SS.diff
  free_tyvars_tau free_tyvars_gamma; tau }

let occurs_check tyvar tau =
  if SS.mem tyvar (find_tyvars tau) then
  raise (Fail "recursive unification")

let rec unify t1 t2 =
  match t1, t2 with
  | TyCon c1, TyCon c2 -> if c1 = c2 then
  S.empty else raise (Fail "cannot unify")
  | TyVar tv1, TyVar tv2 -> if tv1 = tv2 then
  S.empty else S.add tv1 t2 S.empty
  | TyVar tv, _ -> occurs_check tv t2;
  S.add tv t2 S.empty
  | _, TyVar tv -> occurs_check tv t1;
  S.add tv t1 S.empty
  | TyFunApp { t1 = t11; t2 = t12 },
  TyFunApp { t1 = t21; t2 = t22 }
  | TyTuple { t1 = t11; t2 = t12 },
  TyTuple { t1 = t21; t2 = t22 } ->
    let s1 = unify t11 t21 in
    let s2 =
    unify (S.apply s1 t12) (S.apply s1 t22) in
    S.compose s2 s1
  | _ -> raise (Fail "unify _ case")

let specialize (TypeScheme{tyvars; tau}) =
  let bindings =
  List.map (fun tv ->
  (tv, new_tyvar())) (SS.elements tyvars) in
  let subst = S.of_list bindings in
  (S.empty, S.apply subst tau)

let infer_type exp =
  let rec w gamma exp =
    match exp with
    | A.Var id -> (
      match TE.look_up id gamma with
      | None -> raise
      (Fail "id not in type environment")
      | Some ts -> specialize ts)
    | A.Lambda { id; e1 } ->
      let alpha = new_tyvar () in
      let s1, tau1 =
      w (gamma +- (id, !&alpha)) e1 in
      (s1, S.apply s1 alpha => tau1)
    | A.App { e1; e2 } ->
      let (s1, tau1) = w gamma e1 in
      let (s2, tau2) =
      w (S.apply_to_gamma s1 gamma) e2 in
      let alpha = new_tyvar () in
      let s3 =
      unify (S.apply s2 tau1) (tau2 => alpha)
      in (S.compose s3 (S.compose s2 s1),
      S.apply s3 alpha)
    | A.Let { id; e1; e2 } ->
      let (s1, tau1) = w gamma e1 in
      let s1_gamma =
      S.apply_to_gamma s1 gamma in
      let (s2, tau2) =
      w (s1_gamma +- (id, clos s1_gamma tau1)) e2
      in (S.compose s2 s1, tau2)
    | A.Tuple { e1; e2 } ->
      let (s1, tau1) = w gamma e1 in
      let (s2, tau2) =
      w (S.apply_to_gamma s1 gamma) e2 in
      (S.compose s2 s1,
      (S.apply s2 tau1) ** tau2)
    | A.Fst e1 -> (
      let (s1, tau1) = w gamma e1 in
      match tau1 with TyTuple { t1; _ } ->
      (s1, t1) | _ -> raise
      (Fail "expected tuple"))
    | A.Snd e1 -> (
      let (s1, tau1) = w gamma e1 in
      match tau1 with TyTuple { t2; _ } ->
      (s1, t2) | _ -> raise
      (Fail "expected tuple"))
    | A.BasVal b ->
      match b with
      | Int _ -> (S.empty, TyCon Int)
      | Bool _ -> (S.empty, TyCon Bool)
      | String _ -> (S.empty, TyCon String)
  in
  snd (w TE.empty exp)
```

The workhorse function is *infer_type* which contains the recursive function w corresponding to the pseudocode we presented in Section 3.1.2. The other functions are dedicated to unification as defined in Section 3.1.1, the Clos operation, the tyvars map (finding free type variables etc.), and instantiation as defined in Section 2.2.2.

The code and files presented so far are the ones that cover the implementation of $\mathcal{W}$, however, practicalities such as the examples we used to test our implementation are not crucial for our thesis, but can be found in Appendix E. The whole implementation i.e. all the files used for implementing $\mathcal{W}$ can be found on our GitHub page here.

Chapter 4

Optimizing Algorithm $\mathcal{W}$

4.1 The Algorithm

From Robin Milner himself —*$\mathcal{W}$ is hardly an efficient algorithm, substitutions are applied too often* [2]. In this Chapter, we present an alternative to substitutions which is a data structure, and we will present an optimized version of algorithm $\mathcal{W}$ which we will denote $\mathcal{W}_{opt}$.

4.1.1 Union-Find Data Structure

One way to optimize algorithm $\mathcal{W}$ is to implement the substitutions as a data structure. Particularly, the data structure we will use is called Union-Find. The idea behind Union-Find is to have disjoint sets of nodes. This means we have sets of nodes where all nodes in a set are connected. The data structure supports the creation of new sets, linking two nodes, and finding the representative node for the set containing a given node. In our case, a node is a type and we can *Link* a type τ_1 to a new type τ_2. Calling *Find* on any type τ in a set will now return the representative type in that set which is equivalent to performing the substitutions. This means we do not need to use explicit substitutions in our algorithm anymore.

4.1.2 Unification

When using substitutions, our *Unify* function from 3.1.1 would either fail or return a substitution that unifies the two types. With Union-Find, we do not have substitutions, so instead we link the types as needed. Let $Unify_{opt} : (\textbf{Type} \times \textbf{Type}) \rightarrow \textbf{Substitution}$ be the unification function for $\mathcal{W}_{opt}$ and let () be the unit type defined in OCaml. The possible operations for $Unify_{opt}$ are defined below

$\boxed{Unify_{opt}(\pi_1, \pi_2)}$

If $\pi_1 = \pi_2$ then () else **Fail**

$\boxed{Unify_{opt}(\alpha_1, \alpha_2)}$

If $\alpha_1 = \alpha_2$ then () else $Link(\alpha_1, \alpha_2)$

$\boxed{\begin{array}{l} Unify_{opt}(\alpha, \tau) \\ Unify_{opt}(\tau, \alpha) \end{array}}$

If $\alpha \in \text{tyvars}(\tau)$ then **Fail** else $Link(\alpha, \tau)$

$\boxed{\begin{array}{l} \mathit{Unify}_{opt}(\tau_{11} \to \tau_{12}, \tau_{21} \to \tau_{22}) \\ \mathit{Unify}_{opt}(\tau_{11} \times \tau_{12}, \tau_{21} \times \tau_{22}) \end{array}}$

We call $\mathit{Unify}_{opt}(\tau'_{11}, \tau'_{21})$ and $\mathit{Unify}_{opt}(\tau'_{12}, \tau'_{22})$ where the prime symbol is used to specify the find action on a type, i.e. τ' means $\mathit{Find}(\tau)$.

Example 14 (Unification)**.** Let the Unify_{opt} function take the following two types as parameter $\tau_1 = \mathit{int} \to \alpha$ and $\tau_2 = \mathit{int} \to (\beta \times \gamma)$ then $\mathit{Unify}_{opt}(\tau_1, \tau_2)$ will do the $\mathit{Link}(\alpha, \beta \times \gamma)$ operation i.e. putting α into the set with $\beta \times \gamma$ so we get $\{\alpha, \beta \times \gamma\}$ and whenever we do the $\mathit{Find}(\alpha)$ operation it returns $\beta \times \gamma$ since it is now the representative of that set.

4.1.3 Pseudocode

We will again use the prime symbol to specify the find action on a type, i.e. τ' means $\mathit{Find}(\tau)$.

$\mathcal{W}(\Gamma, e) =$ case e of

$i \implies (\mathit{ID}, \mathit{int})$

$b \implies (\mathit{ID}, \mathit{bool})$

$s \implies (\mathit{ID}, \mathit{string})$

$x \implies$
  $\mathit{if}\ x \notin$ **Dom** Γ **then fail**
  else let $\forall \alpha_1 \ldots \alpha_n . \tau = \Gamma(x)$
  $\beta_1 \ldots \beta_n$ be new
  in $\{\alpha_i \mapsto \beta_i\}\tau$

$\lambda x.e_1 \implies$
  let α be a new type variable
  $\tau_1 = \mathcal{W}(\Gamma \pm \{x \mapsto \alpha\}, e_1)$
  in $\alpha' \to \tau_1$

$e_1\ e_2 \implies$
  $\tau_1 = \mathcal{W}(\Gamma, e_1)$
  $\tau_2 = \mathcal{W}(\Gamma, e_2)$
  let α be a new type variable
  $\mathit{Unify}(\tau'_1, \tau_2 \to \alpha)$
  in α'

let $x = e_1$ **in** $e_2 \implies$
  $\tau_1 = \mathcal{W}(\Gamma, e_1)$
  $\tau_2 = \mathcal{W}(\Gamma \pm \{x \mapsto \mathrm{Clos}_\Gamma \tau_1\}, e_2)$
  in τ_2

$(e_1, e_2) \implies$
  $\tau_1 = \mathcal{W}(\Gamma, e_1)$
  $\tau_2 = \mathcal{W}(\Gamma, e_2)$
  $\tau_1 \times \tau_2$

fst $e_1 \implies$
  $\tau_1 = \mathcal{W}(\Gamma, e_1)$
  if **typeof**$(\tau_1) = \tau_2 \times \tau_3$
  then τ_2
  else fail

snd $e_1 \implies$
  $\tau_1 = \mathcal{W}(\Gamma, e_1)$
  if **typeof**$(\tau_1) = \tau_2 \times \tau_3$
  then τ_3
  else fail

Figure 4.1: Pseudocode for $\mathcal{W}_{opt}$

4.2 Implementation

The files **ast.ml** and **typeEnv.ml** have not been changed, but **substitution.ml** has obviously been deleted. In the following, we present the files that have been changed and we only write out the parts and/or functions that have been changed substantially.

We decided to implement the Union-Find data structure directly into the types instead of wrapping all types in nodes since we know that only tyvars can be linked to another type and this way we will not have to unwrap the types all the time.

Note that we have implemented *path compression* in the *Find* operation. Path compression makes sure that the pointer always points to the root instead of an intermediary.

types.ml

```
...
type typ =
...
| TyVar of tyvar ref
...

and tyvar = Int of int | Link of typ
...

(* Union-find *)
let rec find typ: typ = match typ with
| TyVar ({contents = Link t} as kind) ->
  let root = find t in
  kind := Link root;
  root
| _ -> typ
```

Small change since we changed our TyVar type

utils.ml

```
...
let new_tyvar () =
TyVar (ref (Int (TE.get_next_tyvar())))
...
```

A lot has changed since we use pointers, no substitution maps and a different approach to unification

algorithmW.ml

```
open Types
open Utils

module A = Ast

let rec find_tyvars tau = match ~$tau with
  | TyCon _ -> SS.empty
  | TyVar {contents = Int alpha} -> SS.singleton alpha
  | TyVar _ -> raise (Fail "find_tyvars link")
  | TyFunApp { t1; t2 } -> SS.union (find_tyvars t1) (find_tyvars t2)
  | TyTuple { t1; t2 } -> SS.union (find_tyvars t1) (find_tyvars t2)
```

```
...

let rec unify t1 t2 =
  match t1, t2 with
  | TyCon c1, TyCon c2 -> if c1 = c2 then () else raise (Fail "cannot unify")
  | TyVar tv1, TyVar tv2 -> if tv1 = tv2 then () else union tv1 t2
  | TyVar ({contents = Int i} as tv), _ -> occurs_check i t2; union tv t2
  | _, TyVar ({contents = Int i} as tv) -> occurs_check i t1; union tv t1
  | TyFunApp { t1 = t11; t2 = t12 }, TyFunApp { t1 = t21; t2 = t22 }
  | TyTuple { t1 = t11; t2 = t12 }, TyTuple { t1 = t21; t2 = t22 } ->
      unify ~$t11 ~$t21;
      unify ~$t12 ~$t22
  | _ -> raise (Fail "unify _ case")

let specialize (TypeScheme{tyvars; tau}) =
  let bindings = List.map (fun tv -> (tv, new_tyvar())) (SS.elements tyvars) in
  let rec subst_tyvars tau =
    let tau = ~$tau in
    match tau with
    | TyCon _ -> tau
    | TyVar {contents = Int i} -> assoc_or_else bindings i ~default:tau
    | TyVar _ -> raise (Fail "specialize link")
    | TyFunApp {t1; t2} -> subst_tyvars t1 => subst_tyvars t2
    | TyTuple {t1; t2} -> subst_tyvars t1 ** subst_tyvars t2
  in
  subst_tyvars tau

let infer_type exp =
  let rec w gamma exp =
    match exp with
    | A.Var id -> (
        match TE.look_up id gamma with
        | None -> raise (Fail "id not in type environment")
        | Some ts -> specialize ts)
    | A.Lambda { id; e1 } ->
        let alpha = new_tyvar () in
        let tau1 = w (gamma +- (id, !&alpha)) e1 in
        ~$alpha => tau1
    | A.App { e1; e2 } ->
        let tau1 = w gamma e1 in
        let tau2 = w gamma e2 in
        let alpha = new_tyvar () in
        unify ~$tau1 (tau2 => alpha);
        ~$alpha
    | A.Let { id; e1; e2 } ->
        let tau1 = w gamma e1 in
        let tau2 = w (gamma +- (id, clos gamma tau1)) e2 in
        tau2
    | A.Tuple { e1; e2 } ->
        let tau1 = w gamma e1 in
        let tau2 = w gamma e2 in
        tau1 ** tau2
    | A.Fst e1 -> (
        let tau1 = w gamma e1 in
```

```
        match tau1 with TyTuple { t1; _ } -> t1 | _ -> raise (Fail "expected tuple"))
    | A.Snd e1 -> (
        let tau1 = w gamma e1 in
        match tau1 with TyTuple { t2; _ } -> t2 | _ -> raise (Fail "expected tuple"))
    | A.BasVal b ->
        match b with
        | Int _ -> TyCon Int
        | Bool _ -> TyCon Bool
        | String _ -> TyCon String
in
w TE.empty exp
```

The examples used for testing are the same examples used in $\mathcal{W}$ and there is a small change in the Pretty Printer. The examples and the Pretty Printer can be found in Appendix E. The full implementation of $\mathcal{W}_{opt}$ can be found on our GitHub page here.

Chapter 5

Conclusion

In this thesis, we have described the grammar of the Hindley-Milner type system and extended this to include integers, booleans, strings, tuples, fst, and snd. We have defined dynamic and static semantics for the language and introduced concepts and operations such as type environments and substitutions to help us formulate and implement type inference for the language in practice. Formal definitions, as well as examples, have been presented for the described concepts and operations, and we have proven consistency between our dynamic and static semantics. Unification was introduced and its function in our algorithm, as well as its operations, were described. After this, we presented pseudocode for a non-optimized algorithm for type inference in our language which we call algorithm $\mathcal{W}$. The soundness of algorithm $\mathcal{W}$, was proven for all types, and we presented our working implementation of the algorithm with explanations as needed.

After having a working algorithm, we looked into optimizing it by using a Union-Find data structure instead of substitutions. We introduced the Union-Find data structure shortly along with its three operations, namely the creation of new sets, linking two nodes, and finding the representative node in a set. We then related this to types in our language and presented pseudocode for the new, optimized algorithm which we denoted $\mathcal{W}_{opt}$. After presenting the pseudocode for the optimized algorithm, we adapted our implementation to use a Union-Find data structure with path compression and presented the main differences in our new working implementation. Lastly, we added details, examples, and summaries to the appendices.

Bibliography

[1] Luis Manuel Martins Damas. Type assignment in programming languages. chapter 2, pages 74–82. 1984.

[2] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):369, 1978.

[3] Peter Sestoft. Programming language concepts for software developers. page 117, 2010.

[4] Mads Tofte. Operational semantics and polymorphic type inference. chapter 2. 1988.

Appendices

Appendix A

Operations

A.1 Tyvars Table

Parameter	Result
π	$\emptyset$
α	α
$\tau_1 \to \tau_2$ $\tau_1 \times \tau_2$	$tyvars(\tau_1) \cup tyvars(\tau_2)$

A.2 Substitution Table

Parameter	Result
π	π
α	if $\{\alpha \mapsto \tau\} \subseteq S$ then τ else α
$\tau_1 \to \tau_2$	$S(\tau_1) \to S(\tau_2)$
$\tau_1 \times \tau_2$	$S(\tau_1) \times S(\tau_2)$

A.3 Unification Table for $\mathcal{W}$

Parameter	Result
π_1, π_2	if $\pi_1 = \pi_2$ then ID else Fail
α_1, α_2	if $\alpha_1 = \alpha_2$ then ID else $\{\alpha_1 \mapsto \alpha_2\}$
α, τ τ, α	if $\alpha \in tyvars(\tau)$ then Fail else $\{\alpha \mapsto \tau\}$
$\tau_{11} \to \tau_{12}, \tau_{21} \to \tau_{22}$ $\tau_{11} \times \tau_{12}, \tau_{21} \times \tau_{22}$	let $S_1 = Unify(\tau_{11}, \tau_{21})$ in $Unify(S_1\tau_{12}, S_1\tau_{22})S_1$
otherwise	Fail

A.4 Unification Table for $\mathcal{W}_{opt}$

We will use the prime symbol to specify the find action on a type, i.e. τ' means $Find(\tau)$.

Parameter	Result
$\pi_1 = \pi_2$	if $\pi_1 = \pi_2$ then () else Fail
$\alpha_1 = \alpha_2$	if $\alpha_1 = \alpha_2$ then () else $Link(\alpha_1, \alpha_2)$
$\alpha = \tau$ $\tau = \alpha$	if $\alpha \in tyvars(\tau)$ then Fail else $Link(\alpha, \tau)$
$\tau_{11} \to \tau_{12} = \tau_{21} \to \tau_{22}$ $\tau_{11} \times \tau_{12} = \tau_{21} \times \tau_{22}$	$Unify(\tau'_{11}, \tau'_{21})$; $Unify(\tau'_{12}, \tau'_{22})$
otherwise	Fail

Appendix B

Generalization

B.1 Full Derivation of Example 12

$$
\begin{aligned}
\text{tyvars}(\tau) \setminus \text{tyvars}(\Gamma) &= \text{tyvars}(\alpha \to (\mathit{int} \times \beta) \to (\gamma \times \mathit{string}) \to \delta \to \mathit{int}) \setminus \text{tyvars}(\Gamma) \\
&= \text{tyvars}(\alpha) \cup \text{tyvars}(\mathit{int} \times \beta) \cup \text{tyvars}(\gamma \times \mathit{string}) \cup \\
&\quad \text{tyvars}(\delta) \cup \text{tyvars}(\mathit{int}) \setminus \text{tyvars}(\Gamma) \\
&= \{\alpha\} \cup \text{tyvars}(\mathit{int}) \cup \text{tyvars}(\beta) \cup \text{tyvars}(\gamma) \cup \\
&\quad \text{tyvars}(\mathit{string}) \cup \{\delta\} \setminus \text{tyvars}(\Gamma) \\
&= \{\alpha, \beta, \gamma, \delta\} \setminus \text{tyvars}(\Gamma) \\
&= \{\alpha, \beta, \gamma, \delta\} \setminus \bigcup_{\sigma \in \text{Range } \Gamma} \text{tyvars}(\sigma) \\
&= \{\alpha, \beta, \gamma, \delta\} \setminus (\text{tyvars}(\forall.\eta.\gamma \to \eta) \cup \text{tyvars}(\forall\alpha.\alpha \to \alpha)) \\
&= \{\alpha, \beta, \gamma, \delta\} \setminus \left(\left(\text{tyvars}(\gamma \to \eta) \setminus \bigcup_{i=1}^{n} \alpha_i \right) \cup \left(\text{tyvars}(\alpha \to \alpha) \setminus \bigcup_{i=1}^{n} \alpha_i \right) \right) \\
&= \{\alpha, \beta, \gamma, \delta\} \setminus \left(((\text{tyvars}(\gamma) \cup \text{tyvars}(\eta)) \setminus \{\eta\}) \cup \left(\text{tyvars}(\alpha \to \alpha) \setminus \bigcup_{i=1}^{n} \alpha_i \right) \right) \\
&= \{\alpha, \beta, \gamma, \delta\} \setminus \left((\{\gamma, \eta\} \setminus \{\eta\}) \cup \left(\text{tyvars}(\alpha \to \alpha) \setminus \bigcup_{i=1}^{n} \alpha_i \right) \right) \\
&= \{\alpha, \beta, \gamma, \delta\} \setminus \left(\{\gamma\} \cup \left(\text{tyvars}(\alpha \to \alpha) \setminus \bigcup_{i=1}^{n} \alpha_i \right) \right) \\
&= \{\alpha, \beta, \gamma, \delta\} \setminus (\{\gamma\} \cup (\text{tyvars}(\alpha) \cup \text{tyvars}(\alpha) \setminus \{\alpha\})) \\
&= \{\alpha, \beta, \gamma, \delta\} \setminus (\{\gamma\} \cup (\{\alpha\} \setminus \{\alpha\})) \\
&= \{\alpha, \beta, \gamma, \delta\} \setminus (\{\gamma\} \cup \emptyset) \\
&= \{\alpha, \beta, \gamma, \delta\} \setminus \{\gamma\} \\
&= \{\alpha, \beta, \delta\}
\end{aligned}
$$

Appendix C

Type Inference Examples

With all the details in place we can now present some type inference examples when given an expression e.

C.1 Monomorphic

$\boxed{e = \lambda \mathrm{x}.(\lambda \mathrm{y}.\mathrm{xy})1}$

We start off with the S-Lambda rule

$$\frac{\Gamma \pm \{\mathrm{x} \mapsto \alpha \to \beta\} \vdash (\lambda \mathrm{y}.\mathrm{xy})1 : \gamma}{\Gamma \vdash \lambda \mathrm{x}.(\lambda \mathrm{y}.\mathrm{xy})1 : (\alpha \to \beta) \to \gamma}$$

note that we have to guess the type of x for later use. Furthermore, by the S-App rule we have

$$\frac{\dfrac{\Gamma \vdash \lambda \mathrm{y}.\mathrm{xy} : \delta \to \gamma \qquad \Gamma \vdash 1 : \delta}{\Gamma \pm \{\mathrm{x} \mapsto \alpha \to \beta\} \vdash (\lambda \mathrm{y}.\mathrm{xy})1 : \gamma}}{\Gamma \vdash \lambda \mathrm{x}.(\lambda \mathrm{y}.x\mathrm{y})1 : (\alpha \to \beta) \to \gamma}$$

and by the S-Int rule we find out that 1 has type int and therefore we replace each occurence of δ with int, thus we have

$$\frac{\dfrac{\Gamma \vdash \lambda \mathrm{y}.\mathrm{xy} : int \to \gamma \qquad \dfrac{}{\Gamma \vdash 1 : int}}{\Gamma \pm \{\mathrm{x} \mapsto \alpha \to \beta\} \vdash (\lambda \mathrm{y}.\mathrm{xy})1 : \gamma}}{\Gamma \vdash \lambda \mathrm{x}.(\lambda \mathrm{y}.\mathrm{xy})1 : (\alpha \to \beta) \to \gamma}$$

by the S-Lambda rule we get

$$\frac{\dfrac{\dfrac{\Gamma \pm \{\mathrm{y} \mapsto int\} \vdash \mathrm{xy} : \gamma}{\Gamma \vdash \lambda \mathrm{y}.\mathrm{xy} : int \to \gamma} \qquad \dfrac{}{\Gamma \vdash 1 : int}}{\Gamma \pm \{\mathrm{x} \mapsto \alpha \to \beta\} \vdash (\lambda \mathrm{y}.\mathrm{xy})1 : \gamma}}{\Gamma \vdash \lambda \mathrm{x}.(\lambda \mathrm{y}.\mathrm{xy})1 : (\alpha \to \beta) \to \gamma}$$

by the S-App rule we get

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \mathrm{x} : \epsilon \to \gamma \qquad \Gamma \vdash \mathrm{y} : \epsilon}{\Gamma \pm \{\mathrm{y} \mapsto \mathit{int}\} \vdash \mathrm{xy} : \gamma}}{\Gamma \vdash \lambda \mathrm{y.xy} : \mathit{int} \to \gamma} \qquad \dfrac{}{\Gamma \vdash 1 : \mathit{int}}}{\Gamma \pm \{\mathrm{x} \mapsto \alpha \to \beta\} \vdash (\lambda \mathrm{y.xy})1 : \gamma}}{\Gamma \vdash \lambda \mathrm{x}.(\lambda \mathrm{y.xy})1 : (\alpha \to \beta) \to \gamma}$$

by the S-Lookup rule we get

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{x} \in \mathrm{Dom}\ \Gamma \quad \Gamma(\mathrm{x}) > \epsilon \to \gamma}{\Gamma \vdash \mathrm{x} : \epsilon \to \gamma} \qquad \dfrac{\mathrm{y} \in \mathrm{Dom}\ \Gamma \quad \Gamma(\mathrm{y}) > \epsilon}{\Gamma \vdash \mathrm{y} : \epsilon}}{\Gamma \pm \{\mathrm{y} \mapsto \mathit{int}\} \vdash \mathrm{xy} : \gamma}}{\Gamma \vdash \lambda \mathrm{y.xy} : \mathit{int} \to \gamma} \qquad \dfrac{}{\Gamma \vdash 1 : \mathit{int}}}{\Gamma \pm \{\mathrm{x} \mapsto \alpha \to \beta\} \vdash (\lambda \mathrm{y.xy})1 : \gamma}}{\Gamma \vdash \lambda \mathrm{x}.(\lambda \mathrm{y.xy})1 : (\alpha \to \beta) \to \gamma}$$

when evaluating $\Gamma(\mathrm{x}) > \epsilon \to \gamma$ and $\Gamma(\mathrm{y}) > \epsilon$ our $\Gamma = \{\mathrm{x} \mapsto \alpha \to \beta, \mathrm{y} \mapsto \mathit{int}\}$, thus those two operations gives

$$\begin{aligned} \Gamma(\mathrm{x}) > \epsilon \to \gamma \implies & \\ & \epsilon = \alpha \quad \gamma = \beta \\ \Gamma(\mathrm{y}) > \epsilon \implies & \epsilon = \mathit{int} \end{aligned}$$

By replacing the type variables accordingly, the resulting derivation tree looks as follows

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathrm{x} \in \mathrm{Dom}\ \Gamma \quad \Gamma(\mathrm{x}) > \mathit{int} \to \beta}{\Gamma \vdash \mathrm{x} : \mathit{int} \to \beta} \qquad \dfrac{\mathrm{y} \in \mathrm{Dom}\ \Gamma \quad \Gamma(\mathrm{y}) > \mathit{int}}{\Gamma \vdash \mathrm{y} : \mathit{int}}}{\Gamma \pm \{\mathrm{y} \mapsto \mathit{int}\} \vdash \mathrm{xy} : \beta}}{\Gamma \vdash \lambda \mathrm{y.xy} : \mathit{int} \to \beta} \qquad \dfrac{}{\Gamma \vdash 1 : \mathit{int}}}{\Gamma \pm \{\mathrm{x} \mapsto \mathit{int} \to \beta\} \vdash (\lambda \mathrm{y.xy})1 : \beta}}{\Gamma \vdash \lambda \mathrm{x}.(\lambda \mathrm{y.xy})1 : (\mathit{int} \to \beta) \to \beta}$$

and the inferred type of the expression is $(\mathit{int} \to \beta) \to \beta$.

C.2 Polymorphic

$\boxed{\mathrm{e} = \textbf{let}\ \mathrm{id} = \lambda \mathrm{x.x}\ \textbf{in}\ (\mathrm{id}\ 1, \mathrm{id}\ \text{"}\mathit{hello}\text{"})}$

By the S-Let rule we have

$$\dfrac{\Gamma \vdash \lambda \mathrm{x.x} : \beta \qquad \Gamma \pm \{\mathrm{id} \mapsto \mathrm{Clos}_\Gamma \beta\} \vdash (\mathrm{id}\ 1, \mathrm{id}\ \text{"}\mathit{hello}\text{"}) : \alpha}{\Gamma \vdash \textbf{let}\ \mathrm{id} = \lambda \mathrm{x.x}\ \textbf{in}\ (\mathrm{id}\ 1, \mathrm{id}\ \text{"}\mathit{hello}\text{"}) : \alpha}$$

By the S-Lambda rule we get

$$\frac{\dfrac{\Gamma \pm \{\mathrm{x} \mapsto \gamma\} \vdash x : \delta}{\Gamma \vdash \lambda \mathrm{x.x} : \beta} \qquad \Gamma \pm \{\mathrm{id} \mapsto \mathrm{Clos}_\Gamma \beta\} \vdash (\mathrm{id}\ 1, \mathrm{id}\ \text{"}hello\text{"}) : \alpha}{\Gamma \vdash \textbf{let}\ \mathrm{id} = \lambda \mathrm{x.x}\ \textbf{in}\ (\mathrm{id}\ 1, \mathrm{id}\ \text{"}hello\text{"}) : \alpha}$$

note that we get that $\beta = \gamma \to \delta$ so we replace each occurence of β with $\gamma \to \delta$

$$\frac{\dfrac{\Gamma \pm \{\mathrm{x} \mapsto \gamma\} \vdash x : \delta}{\Gamma \vdash \lambda \mathrm{x.x} : \gamma \to \delta} \qquad \Gamma \pm \{\mathrm{id} \mapsto \mathrm{Clos}_\Gamma \gamma \to \delta\} \vdash (\mathrm{id}\ 1, \mathrm{id}\ \text{"}hello\text{"}) : \alpha}{\Gamma \vdash \textbf{let}\ \mathrm{id} = \lambda \mathrm{x.x}\ \textbf{in}\ (\mathrm{id}\ 1, \mathrm{id}\ \text{"}hello\text{"}) : \alpha}$$

let $\Gamma_1 = \Gamma \pm \{\mathrm{x} \mapsto \gamma\}$ and by the S-Lookup rule we get

$$\frac{\dfrac{\dfrac{\mathrm{x} \in \mathrm{Dom}\ \Gamma_1 \quad \Gamma_1(\mathrm{x}) > \delta}{\Gamma \pm \{\mathrm{x} \mapsto \gamma\} \vdash x : \delta}}{\Gamma \vdash \lambda \mathrm{x.x} : \gamma \to \delta} \qquad \Gamma \pm \{\mathrm{id} \mapsto \mathrm{Clos}_\Gamma \gamma \to \delta\} \vdash (\mathrm{id}\ 1, \mathrm{id}\ \text{"}hello\text{"}) : \alpha}{\Gamma \vdash \textbf{let}\ \mathrm{id} = \lambda \mathrm{x.x}\ \textbf{in}\ (\mathrm{id}\ 1, \mathrm{id}\ \text{"}hello\text{"}) : \alpha}$$

where $\Gamma_1(\mathrm{x}) > \delta \implies \delta = \gamma$, thus we replace each occurence of δ with γ

$$\frac{\dfrac{\dfrac{\mathrm{x} \in \mathrm{Dom}\ \Gamma_1 \quad \Gamma_1(\mathrm{x}) > \gamma}{\Gamma \pm \{\mathrm{x} \mapsto \gamma\} \vdash x : \gamma}}{\Gamma \vdash \lambda \mathrm{x.x} : \gamma \to \gamma} \qquad \Gamma \pm \{\mathrm{id} \mapsto \mathrm{Clos}_\Gamma \gamma \to \gamma\} \vdash (\mathrm{id}\ 1, \mathrm{id}\ \text{"}hello\text{"}) : \alpha}{\Gamma \vdash \textbf{let}\ \mathrm{id} = \lambda \mathrm{x.x}\ \textbf{in}\ (\mathrm{id}\ 1, \mathrm{id}\ \text{"}hello\text{"}) : \alpha}$$

now, we have to calculate the $Clos_\Gamma \gamma \to \gamma$ operation to evaluate the tuple, thus we have

$$\begin{aligned}
\mathrm{tyvars}(\gamma \to \gamma) \setminus \mathrm{tyvars}(\Gamma) = &(\mathrm{tyvars}(\gamma) \cup \mathrm{tyvars}(\gamma)) \setminus \mathrm{tyvars}(\Gamma) = \\
&(\{\gamma\} \cup \{\gamma\}) \setminus \mathrm{tyvars}(\Gamma) = \\
&\{\gamma\} \setminus \mathrm{tyvars}(\Gamma) = \\
&\{\gamma\} \setminus \emptyset = \\
&\{\gamma\}
\end{aligned}$$

and therefore $Clos_\Gamma \gamma \to \gamma = \forall \gamma.\gamma \to \gamma$.

Let $\Gamma_2 = \Gamma \pm \{\mathrm{id}\ \mapsto \forall \gamma.\gamma \to \gamma\}$ and by the S-Tuple rule we get

$$\frac{\dfrac{\dfrac{\mathrm{x} \in \mathrm{Dom}\ \Gamma_1 \quad \Gamma_1(\mathrm{x}) > \gamma}{\Gamma \pm \{\mathrm{x} \mapsto \gamma\} \vdash x : \gamma}}{\Gamma \vdash \lambda \mathrm{x.x} : \gamma \to \gamma} \qquad \dfrac{\Gamma_2 \vdash \mathrm{id}\ 1 : \epsilon \qquad \Gamma_2 \vdash \mathrm{id}\ \text{"}hello\text{"} : \zeta}{\Gamma \pm \{\mathrm{id} \mapsto \forall \gamma.\gamma \to \gamma\} \vdash (\mathrm{id}\ 1, \mathrm{id}\ \text{"}hello\text{"}) : \alpha}}{\Gamma \vdash \textbf{let}\ \mathrm{id} = \lambda \mathrm{x.x}\ \textbf{in}\ (\mathrm{id}\ 1, \mathrm{id}\ \text{"}hello\text{"}) : \alpha}$$

where we replace all occurences of α with $\epsilon \times \zeta$

$$\dfrac{\dfrac{\dfrac{\mathrm{x} \in \mathrm{Dom}\ \Gamma_1 \quad \Gamma_1(\mathrm{x}) > \gamma}{\Gamma \pm \{\mathrm{x} \mapsto \gamma\} \vdash x : \gamma}}{\Gamma \vdash \lambda \mathrm{x.x} : \gamma \to \gamma} \qquad \dfrac{\Gamma_2 \vdash \mathrm{id}\ 1 : \epsilon \qquad \Gamma_2 \vdash \mathrm{id}\ \text{"}hello\text{"} : \zeta}{\Gamma \pm \{\mathrm{id} \mapsto \forall\gamma.\gamma \to \gamma\} \vdash (\mathrm{id}\ 1, \mathrm{id}\ \text{"}hello\text{"}) : \epsilon \times \zeta}}{\Gamma \vdash \textbf{let}\ \mathrm{id} = \lambda \mathrm{x.x}\ \textbf{in}\ (\mathrm{id}\ 1, \mathrm{id}\ \text{"}hello\text{"}) : \epsilon \times \zeta}$$

by the S-App rule on both expressions we get

$$\dfrac{\dfrac{\dfrac{\mathrm{x} \in \mathrm{Dom}\ \Gamma_1 \quad \Gamma_1(\mathrm{x}) > \gamma}{\Gamma \pm \{\mathrm{x} \mapsto \gamma\} \vdash x : \gamma}}{\Gamma \vdash \lambda \mathrm{x.x} : \gamma \to \gamma} \qquad \dfrac{\dfrac{\Gamma_2 \vdash \mathrm{id} : \eta \to \epsilon \qquad \Gamma_2 \vdash 1 : \eta}{\Gamma_2 \vdash \mathrm{id}\ 1 : \epsilon} \qquad \dfrac{\Gamma_2 \vdash \mathrm{id} : \theta \to \zeta \qquad \Gamma_2 \vdash \text{"}hello\text{"} : \theta}{\Gamma_2 \vdash \mathrm{id}\ \text{"}hello\text{"} : \zeta}}{\Gamma \pm \{\mathrm{id} \mapsto \forall\gamma.\gamma \to \gamma\} \vdash (\mathrm{id}\ 1, \mathrm{id}\ \text{"}hello\text{"}) : \epsilon \times \zeta}}{\Gamma \vdash \textbf{let}\ \mathrm{id} = \lambda \mathrm{x.x}\ \textbf{in}\ (\mathrm{id}\ 1, \mathrm{id}\ \text{"}hello\text{"}) : \epsilon \times \zeta}$$

by the S-Lookup, S-Int, and S-String rule and the replacement of $\eta = int$ and $\theta = string$ we get

$$\dfrac{\dfrac{\dfrac{\mathrm{x} \in \mathrm{Dom}\ \Gamma_1 \quad \Gamma_1(\mathrm{x}) > \gamma}{\Gamma \pm \{\mathrm{x} \mapsto \gamma\} \vdash x : \gamma}}{\Gamma \vdash \lambda \mathrm{x}.\mathrm{x} : \gamma \to \gamma} \qquad \dfrac{\dfrac{\dfrac{\mathrm{id} \in \mathrm{Dom}\ \Gamma_2 \quad \Gamma_2(\mathrm{id}) > \mathit{int} \to \epsilon}{\Gamma_2 \vdash \mathrm{id} : \mathit{int} \to \epsilon} \qquad \dfrac{}{\Gamma_2 \vdash 1 : \mathit{int}}}{\Gamma_2 \vdash \mathrm{id}\ 1 : \epsilon} \qquad \dfrac{\dfrac{\mathrm{id} \in \mathrm{Dom}\ \Gamma_2 \quad \Gamma_2(\mathrm{id}) > \mathit{string} \to \zeta}{\Gamma_2 \vdash \mathrm{id} : \mathit{string} \to \zeta} \qquad \dfrac{}{\Gamma_2 \vdash \text{"}\mathit{hello}\text{"} : \mathit{string}}}{\Gamma_2 \vdash \mathrm{id}\ \text{"}\mathit{hello}\text{"} : \zeta}}{\Gamma \pm \{\mathrm{id} \mapsto \forall\gamma.\gamma \to \gamma\} \vdash (\mathrm{id}\ 1, \mathrm{id}\ \text{"}\mathit{hello}\text{"}) : \epsilon \times \zeta}}{\Gamma \vdash \mathbf{let}\ \mathrm{id} = \lambda \mathrm{x}.\mathrm{x}\ \mathbf{in}\ (\mathrm{id}\ 1, \mathrm{id}\ \text{"}\mathit{hello}\text{"}) : \epsilon \times \zeta}$$

we now do the instantiations

$$\Gamma_2(\mathrm{id}) > int \to \epsilon \implies (S_1 = \{\alpha \mapsto \iota\})\alpha \to \alpha = int \to \epsilon = \iota \to \iota = int \to \epsilon \implies \iota = int \quad \iota = \epsilon$$

$$\Gamma_2(\mathrm{id}) > string \to \zeta \implies (S_2 = \{\alpha \mapsto \kappa\})\alpha \to \alpha = string \to \zeta = \kappa \to \kappa = string \to \zeta \implies \kappa = string \quad \kappa = \zeta$$

thus, $\epsilon = int$ and $\zeta = string$ and the resulting derivation tree looks like

$$\dfrac{\dfrac{\dfrac{\mathrm{x} \in \mathrm{Dom}\ \Gamma_1 \quad \Gamma_1(\mathrm{x}) > \gamma}{\Gamma \pm \{\mathrm{x} \mapsto \gamma\} \vdash x : \gamma}}{\Gamma \vdash \lambda \mathrm{x}.\mathrm{x} : \gamma \to \gamma} \qquad \dfrac{\dfrac{\dfrac{\mathrm{id} \in \mathrm{Dom}\ \Gamma_2 \quad \Gamma_2(\mathrm{id}) > \mathit{int} \to \mathit{int}}{\Gamma_2 \vdash \mathrm{id} : \mathit{int} \to \mathit{int}} \qquad \dfrac{}{\Gamma_2 \vdash 1 : \mathit{int}}}{\Gamma_2 \vdash \mathrm{id}\ 1 : \mathit{int}} \qquad \dfrac{\dfrac{\mathrm{id} \in \mathrm{Dom}\ \Gamma_2 \quad \Gamma_2(\mathrm{id}) > \mathit{string} \to \mathit{string}}{\Gamma_2 \vdash \mathrm{id} : \mathit{string} \to \mathit{string}} \qquad \dfrac{}{\Gamma_2 \vdash \text{"}\mathit{hello}\text{"} : \mathit{string}}}{\Gamma_2 \vdash \mathrm{id}\ \text{"}\mathit{hello}\text{"} : \mathit{string}}}{\Gamma \pm \{\mathrm{id} \mapsto \forall\gamma.\gamma \to \gamma\} \vdash (\mathrm{id}\ 1, \mathrm{id}\ \text{"}\mathit{hello}\text{"}) : \mathit{int} \times \mathit{string}}}{\Gamma \vdash \mathbf{let}\ \mathrm{id} = \lambda \mathrm{x}.\mathrm{x}\ \mathbf{in}\ (\mathrm{id}\ 1, \mathrm{id}\ \text{"}\mathit{hello}\text{"}) : \mathit{int} \times \mathit{string}}$$

The expression e thus have the type $int \times string$.

Appendix D

Additional Proofs

D.1 Consistency Proof for the $\mathbf{snd}(e_1, e_2)$

The type inference of the expression must have been of the form

$$\frac{\text{S-SND}}{\Gamma \vdash (e_1, e_2) \ : \ \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{snd}(e_1, e_2) \ : \ \tau_1} \tag{D.1}$$

and the evaluation must have been one of the following

$$\frac{\text{D-SND}}{E \vdash e_1 \to v_1 \quad E \vdash e_2 \to v_2}{E \vdash \mathbf{snd}(e_1, e_2) \to v_1} \tag{D.2}$$

$$\frac{\text{D-SND-WRONG1}}{E \vdash e_1 \to \mathit{wrong} \quad E \vdash e_2 \to v_2}{E \vdash \mathbf{snd}(e_1, e_2) \to \mathit{wrong}} \tag{D.3}$$

$$\frac{\text{D-SND-WRONG2}}{E \vdash e_1 \to v_1 \quad E \vdash e_2 \to \mathit{wrong}}{E \vdash \mathbf{snd}(e_1, e_2) \to \mathit{wrong}} \tag{D.4}$$

We see that $e_1 \to r_1$ and $e_2 \to r_2$ for some r_1, r_2.

By our I.H. and 2.1, D.1, (D.2 / D.3 / D.4) we get

$$r_1 \neq \mathit{wrong} \tag{D.5}$$

$$\vDash r_1 \ : \ \tau_1 \tag{D.6}$$

This means the evaluation cannot have been D.3

By our I.H. and 2.1, D.1, (D.2 / D.4) we get

$$r_2 \neq \mathit{wrong} \tag{D.7}$$

$$\vDash r_2 \ : \ \tau_2 \tag{D.8}$$

This means the evaluation cannot have been D.4. The evaluation must have been D.2, meaning that $r_1 = v_1$, $r_2 = v_2$, and $r = v_2 \in \text{Val}$. Since a Val cannot be wrong then $r \neq \mathit{wrong}$.

From D.8 we have that $\vDash r_2 : \tau_2$. Since $r = v_2 = r_2$ and $\vDash r_2 : \tau_2$, we must have that $\vDash r : \tau_2$.

D.2 Soundness Proof for $\mathbf{snd}(e_1, e_2)$

We want to show $S_1\Gamma \vdash \mathbf{snd}\ (e_1, e_2) : \tau_2$

From our algorithm we have

$$(S_1, \tau_1 \times \tau_2) = \mathcal{W}(\Gamma, (e_1, e_2)) \tag{D.9}$$

By lemma 2.2.2 where $\Gamma = \Gamma$ and $S = S_1$ we have

$$\Gamma \xrightarrow{S_1} S_1\Gamma \tag{D.10}$$

By our I.H. and D.9, D.10 we have

$$S_1\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \tag{D.11}$$

Thus, by letting D.11 be the premise of the S-Snd rule

$$\frac{\text{S-Snd}\quad \Gamma \vdash (e_1, e_2)\ :\ \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{snd}\ (e_1, e_2)\ :\ \tau_2}$$

we can conclude $S_1\Gamma \vdash \mathbf{snd}\ (e_1, e_2) : \tau_2$.

Appendix E

Practicalities of $\mathcal{W}$ and $\mathcal{W}_{opt}$

E.1 Examples Used For Test Correctness of $\mathcal{W}$ and $\mathcal{W}_{opt}$

The **examples.ml** file is 566 lines long and therefore it is not presented directly in this Appendix but click here to view the examples used for verifying our correctness for $\mathcal{W}$ and $\mathcal{W}_{opt}$.

E.2 Pretty Printer for $\mathcal{W}$

The **prettyPrinter.ml** is a file that contains functions that outputs a string containing useful information. For instance, it is very useful for visually seeing the result type τ when $\mathcal{W}$ returns it and therefore we would use the *string_of_tau* function.

```
open Types
open TypeEnv

let string_of_tau tau_node =
  let rec trav tau =
    match tau with
    | TyCon s -> (
        match s with Int -> "int" | Bool -> "bool" | String -> "string"), 0
    | TyVar i -> string_of_int i, 0
    | TyFunApp { t1; t2 } ->
        let a1, a2 = trav t1 in
        let b1, b2 = trav t2 in
        let string_a = if a2 > 0 then "(" ^ a1 ^ ")" else a1 in
        let string_b = if b2 > 1 then "(" ^ b1 ^ ")" else b1 in
        string_a ^ " -> " ^ string_b, 1
    | TyTuple { t1; t2 } ->
        let a1, a2 = trav t1 in
        let b1, b2 = trav t2 in
        let string_a = if a2 > 0 then "(" ^ a1 ^ ")" else a1 in
        let string_b = if b2 > 0 then "(" ^ b1 ^ ")" else b1 in
        string_a ^ " x " ^ string_b, 1
  in
  fst (trav tau_node)

let print_tau tau = print_string (string_of_tau tau ^ "\n")
```

```
let string_of_typescheme (TypeScheme { tyvars; tau }) =
  let tyvars = String.concat ", " (List.map (fun x -> string_of_int x) (SS.elements tyvars))
  in "forall " ^ tyvars ^ " . " ^ (string_of_tau tau)
let print_typescheme typescheme = print_string (string_of_typescheme typescheme ^ "\n")

let string_of_tyvars tyvars =
  let elems = String.concat ", " (List.map (fun x -> string_of_int x) tyvars) in
  "{ " ^ elems ^ " }"
let print_tyvars tyvars = print_string (string_of_tyvars tyvars ^ "\n")

let string_of_gamma gamma =
  let elems = String.concat ", " (List.map (fun (k, v) -> k ^ " -> " ^ string_of_typescheme v)
  (Gamma.bindings gamma)) in "{ " ^ elems ^ " }"

let print_gamma gamma = print_string (string_of_gamma gamma ^ "\n")
```

E.3 Pretty Printer for $\mathcal{W}_{opt}$

This Pretty Printer is almost identical to the one in E.2.

```
open Types
open TypeEnv
open Utils

let string_of_tau tau_node =
  let rec trav tau =
    match ~$tau with
    | TyCon s -> (
        match s with Int -> "int" | Bool -> "bool" | String -> "string"), 0
    | TyVar {contents = Int i} -> string_of_int i, 0
    | TyVar _ -> raise (Fail "string_of_tau tyvar link")
    | TyFunApp { t1; t2 } ->
        let a1, a2 = trav t1 in
        let b1, b2 = trav t2 in
        let string_a = if a2 > 0 then "(" ^ a1 ^ ")" else a1 in
        let string_b = if b2 > 1 then "(" ^ b1 ^ ")" else b1 in
        string_a ^ " -> " ^ string_b, 1
    | TyTuple { t1; t2 } ->
        let a1, a2 = trav t1 in
        let b1, b2 = trav t2 in
        let string_a = if a2 > 0 then "(" ^ a1 ^ ")" else a1 in
        let string_b = if b2 > 0 then "(" ^ b1 ^ ")" else b1 in
        string_a ^ " x " ^ string_b, 1
  in
  fst (trav tau_node)

let print_tau tau = print_string (string_of_tau tau ^ "\n")

let string_of_typescheme (TypeScheme { tyvars; tau }) =
  let tyvars = String.concat ", " (List.map (fun x -> string_of_int x) (SS.elements tyvars)) in
  "forall " ^ tyvars ^ " . " ^ (string_of_tau tau)
let print_typescheme typescheme = print_string (string_of_typescheme typescheme ^ "\n")

let string_of_tyvars tyvars =
```

```
  let elems = String.concat ", " (List.map (fun x -> string_of_int x) tyvars) in
  "{ " ^ elems ^ " }"
let print_tyvars tyvars = print_string (string_of_tyvars tyvars ^ "\n")

let string_of_gamma gamma =
  let elems = String.concat ", " (List.map (fun (k, v) -> k ^ " -> " ^ string_of_typescheme v)
  (Gamma.bindings gamma)) in "{ " ^ elems ^ " }"
let print_gamma gamma = print_string (string_of_gamma gamma ^ "\n")
```