A STATIC SEMANTICS FOR LABELS IN OCAML

BY

MICHAEL BAY

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Advisor:

Research Professor Elsa Gunter
Abstract

In this thesis, we are proposing a type system and a type inferencer for the label-feature of OCaml. Labels in OCaml allow the naming of arguments to functions and are intended to document the code. We provide a type system that describes whether a derived type for an OCaml expression containing labels is correct or not, a type inference algorithm, and show that OCaml’s function application does not strictly support currying.
To my parents, Christian and Petra Bay, for their love and support.
I am deeply indebted to my advisor Elsa Gunter, whose tireless support and guidance in the past 2 years has been of tremendous help to me. She facilitated me finding my research interest, provided clarity how my work fits into the bigger picture and devoted overtime work to support me. From teaching to advising, Elsa has significantly shaped my studies and research from my first day at graduate school. My graduation and this thesis would not have been possible without Elsa.

I would also like to thank Liyi Li, without whom I would not have discovered the label-feature of OCaml as an area of research. Our discussions about K have not only helped me understand K better but also improved my understanding of the semantics of OCaml. I hope that this project will aid further research for OCaml-K.

I would not like to miss research fellows and friends, Susannah, John, Everett and David, who provided me with valuable advice throughout this project.

Lastly, I would like to thank my family and friends. Their belief in me, whatever comes, pushes me forward.
# Table of Contents

List of Figures ....................................................... vi

Chapter 1 Introduction ............................................... 1

Chapter 2 Syntax ..................................................... 3
  2.1 Type syntax .................................................... 6

Chapter 3 Type System ................................................ 8
  3.1 Auxiliary functions ............................................ 9
  3.2 Type Rules ................................................... 14
    3.2.1 At-expressions ......................................... 14
    3.2.2 Patterns ............................................... 14
    3.2.3 Arguments ............................................ 15
    3.2.4 Parameters ........................................... 15
    3.2.5 Multiple-matchings ................................... 18
    3.2.6 Expressions ........................................... 18

Chapter 4 Type inference .......................................... 21
  4.1 Type unification ............................................. 21
  4.2 Type inference function(s) .................................. 22
    4.2.1 infer_at_expression: .................................. 23
    4.2.2 infer_pattern: ........................................ 23
    4.2.3 infer_argument: ...................................... 25
    4.2.4 infer_parameter: ..................................... 26
      4.2.4.1 no parentheses: .................................. 27
      4.2.4.2 no default expression: ......................... 27
      4.2.4.3 with expression e: ............................. 28
    4.2.5 infer_multiple-matching: ............................ 29
    4.2.6 infer_expression: ................................... 29

Chapter 5 Type system (continued) ................................ 35
  5.1 Function application special-case .......................... 35

Chapter 6 Testing .................................................. 37

Chapter 7 Conclusion ............................................... 38

References .......................................................... 39
List of Figures

3.1 Type rules for at-expressions ................................................. 14
3.2 Type rules for patterns .......................................................... 15
3.3 Type rules for arguments ....................................................... 16
3.4 Type rule for unlabeled parameters ........................................ 16
3.5 Type rules for non-optionally labeled parameters ........................ 16
3.6 Type rules for optionally labeled parameters ............................ 17
3.7 Type rule for multiple-matchings ............................................ 18
3.8 Type rule for expressions as at-expressions ............................... 18
3.9 Type rule for binary operator expressions .................................. 18
3.10 Type rule for tuple expressions ............................................. 19
3.11 Type rule for constructor expressions .................................... 19
3.12 Type rule for fun-abstractions ............................................. 19
3.13 Type rule for function application ........................................ 20

5.1 Type rule for function application-special case ........................... 35
Chapter 1

Introduction

OCaml is the predominant variant of the Caml programming language. The first implementation of the Caml programming language was developed in 1987 by Ascánder Suárez. In 1990, Xavier Leroy implemented a completely new version of Caml which was called Caml light and extended it to Caml Special Light in 1995. Didier Rémy and Jérôme Vouillon defined a type system, that could be integrated with Caml Special Light. As a result, Objective Caml was released in 1996[6] and renamed to OCaml in 2011. OCaml was used for development of the Coq system, the unikernel library operating system MirageOS[1] and the web development programming language Opa[5]. OCaml provides a native-code compiler(ocamlopt), an interpreter, a bytecode compiler (ocamlc), a reversible debugger (ocamldebug), a lex (ocamllex), and a parser (ocamlyacc)[2]. Among OCaml’s features are a static type system (using type inference and parametric polymorphism), user-definable data types, tail recursion, pattern matching, automatic memory management, functors (parametric modules), exception handling, object system and records[3]. OCaml is managed and maintained by the French Institute for Research in Computer Science and Automation (INRIA)[4]. In 2000, Jacques Garrigue extended OCaml with labeled and optional function arguments[6]. Labels in OCaml allow the naming of arguments to functions and are intended to document the code. The use of labels as keywords for parameter passing has also been employed in Common LISP, ADA, and LIFE. Besides parameter passing, symbolic labels are used as field designators in record structures in a variety of programming languages including OCaml.

There has never been proposed a complete formal semantics for OCaml. Like the authors of the formal semantics to C[8], Java[7] and PHP we believe that a programming language needs to have a formal semantics, which should be public and easily accessible for improved recognition and resolution of semantic gaps[7]. The most notable attempt in giving OCaml a formal semantics is Scott Owen’s semantics for OCaml light [12], which however is restricted to the core features of OCaml 3.09 and leaves out a number of other features, such as labels in OCaml. Labels present a behavior that is surprising and at times contrary to the expectation of the user. For example, one would expect that OCaml supports currying in any function application. However, it can be shown that, when passing optional labels, this is not necessarily the case. To bring clarity
to the semantics of labels, in this thesis, we propose a static semantics for OCaml label constructs, that is a type system and a type inferencer. There exists an alternate semantics by J. Garrigue for OCaml label constructs\[10\], that provides an efficient compilation method for a subset of the OCaml grammar for labels. There also exists the label-selective lambda calculus, the theoretical foundation that OCaml labels are based upon\[9\]. Our work differentiates itself from Garrigue’s work through providing a type inference algorithm, being constructive rather than purely declarative and supporting more complex OCaml constructs. It distinguishes itself from the label-selective lambda-calculus through providing a formal semantics, not to the label-selective lambda-calculus, but to OCaml, that is more directly computationally executable. We reengineer OCaml label constructs where they are insufficiently documented. For example, we provide an implementation of the \texttt{erase}-function. We also give the type inference function for the special-case rule of function application.
Chapter 2

Syntax

We are relying on the grammar of OCaml’s system release 4.02.3 from September 24, 2014 restricting our attention to label-constructs. We give here a slightly modified version of the grammar relevant to labels and refer the reader to [http://caml.inria.fr/pub/docs/manual-ocaml/expr.html](http://caml.inria.fr/pub/docs/manual-ocaml/expr.html) for the full language grammar of the more recent release 4.03[4]. The syntax of the language is given in BNF-like notation. Terminal symbols are set in typewriter font and surrounded with quotation marks (“like this”). Non-terminal symbols are set in italic font (like that). Angle brackets ⟨...⟩ denote zero or one of the enclosed components. Curly brackets with a trailing star sign {...}* denote zero, one or several repetitions of the enclosed components. Curly brackets with a trailing plus sign {...}+ denote one or several repetitions of the enclosed components.

\[
\begin{align*}
label-name & ::= \text{lowercase-ident} \\
value-name & ::= \text{lowercase-ident} \\
\text{...} \\
constant & ::= \text{integer-literal} \\
\text{...} \\
\text{...}
\end{align*}
\]
\[ binop ::= + | - | \ast | / | = | < | > | :: | \ldots \]

\[ constr ::= \text{Some} \]
\[ \quad | \text{None} \]
\[ \ldots \]

\[ at-expression ::= \text{value-name} \]
\[ \quad | \text{constant} \]
\[ \quad | "\{" \text{expression} ")" \]

\[ pattern ::= \text{value-name} \]
\[ \quad | \text{constant} \]
\[ \quad | ":" \]
\[ \quad | "(" \text{pattern} \"::\" \text{typexpr} ")" \]
\[ \quad | \text{constr pattern} \]
\[ \quad | \text{pattern} \"::\" \text{pattern} \]
\[ \quad | \text{pattern} \"\|\" \text{pattern} \]
\[ \quad | \text{pattern} \{"\",\" \text{pattern}\}^{+} \]
\[ \ldots \]

\[ argument ::= \text{at-expression} \]
\[ \quad | ":" \text{label-name} \]
\[ \quad | ":\" \text{label-name} ":\" \text{at-expression} \]
\[ \quad | ":\" \text{label-name} \]
\[ \quad | ":\" \text{label-name} ":\" \text{at-expression} \]
In the above grammar, label-names and value-names are sequences of letters, digits, _ (the underscore character), and ' (the single quote), starting with a lowercase letter or an underscore. We support integers, floats, characters, strings, booleans, constructors, and nil as constants, integer addition, subtraction, multiplication, division, equals, greater-than and less-than and the Cons-constructor as binary operators and Some and None as constructors. At-expressions can be value-names (variables), constants, or parenthesized expressions. A pattern can be a value-name, a constant, a wildcard, a parenthesized pattern, a parenthesized pattern typed with a type expression, a constructor applied to a pattern, a Cons-constructor applied to two patterns, an "Or"-pattern, or a tuple pattern. We differentiate two kinds of labels: non-optional labels denoted by tilde ~ and optional labels denoted by ?. Arguments to a function can be at-expressions or non-optionally or optionally labeled variable names possibly qualified by a pattern. A parameter can be a pattern, or a labeled parameter. Labeled parameters can be further qualified by a pattern or can be typed with a type expression. Additionally, optionally labeled parameters can be defaulted to an expression. Multiple-matchings take a non-empty juxtaposition of parameters and supply them to an expression.
Expressions can be at-expressions, binary operator-expressions, tuples, constructor applications, functions, or function applications. There are two syntactic forms to define functions, introduced respectively by the keyword `fun` and the keyword `function`. We restrict our attention to `fun` because it is the only one of the two that supports labels.

As a functional programming language, one would expect that OCaml’s function application is derived from the lambda calculus and that it supports currying, that is the evaluation of a function with multiple arguments can be translated into evaluating a sequence of functions, each with a single argument.

\[ e_1 e_2 e_3 e_4 = ((e_1 e_2) e_3) e_4 \]

However, as is revealed when applying a function to labeled arguments, in OCaml, currying is not always supported. Application is done as a function to a collection of arguments.

\[ e_1 e_2 e_3 e_4 = \text{function arguments} \]

As is shown in Chapter 6, currying may yield different results, that is:

\[ e_1 e_2 e_3 e_4 \neq ((e_1 e_2) e_3) e_4 \]

The INRIA documentation defines the OCaml grammar without at-expressions. Arguments to a function are allowed to be any expressions. This leads to an ambiguity in function application, since one function application could be parsed as multiple different applications:

\[ e_1 e_2 e_3 e_4 = \text{function arguments} \]

### 2.1 Type syntax

In the fashion of the course Programming Languages & Compilers at University of Illinois at Urbana-Champaign, we define our type syntax supporting both monomorphic as well as polymorphic types.
We support integers, floats, strings, booleans, and type-variables as atomic types and *, ->, option and list as type constructors. As we write in this grammar, the product-constructor * for tuples is left-associative. The function constructor -> is right-associative. It concatenates two types, where the left one can be labeled. Datatype constructors signified by tyconstr take at least one type argument. Polymorphic types are universally quantified monomorphic types $\tau$ in the form of

$$\forall \, \tau$$

The monomorphic type $\tau$ is viewed as the polymorphic type $\forall \tau$. A monomorphic type $\tau_1$ is an instance of the polymorphic type $\forall \tau_2$, in short $\tau_1 \prec \forall \tau_2$ if and only if there exists a substitution $\sigma$ such that $support(\sigma) \subseteq \{\tau_1, \tau_2\}$ and $\sigma(\tau_2) = \tau$. A substitution $\sigma$ is a function $\sigma: type-var \mapsto mon-type$ and $support(\sigma) = \{a \in type-var | \sigma(a) \neq a\}$. Through a standard abuse of notation, we will also lift $\sigma$ to apply to monomorphic and polymorphic types, expressions and typing environments by applying $\sigma$ to the type variables occurring therein (renaming bound variables as necessary to avoid free variable capture). The syntactic construct typeexpr (as it is introduced in Chapter 2: Syntax) is a monomorphic type mon-type. The grammar serves as an understanding of type syntax and not of a representation of correct precedence of operators without ambiguity.
Chapter 3

Type System

We define a type system (or static semantics) that states which type an expression can have in a given typing environment and substitution. A type judgment has the form

\[ \Gamma, \sigma \vdash \text{syntax} : \text{type-information} \]

The first component \( \Gamma \) is a typing environment that supplies polymorphic types of variables. It is of the form \( \{ (x \mapsto \theta), \ldots \} \). For any variable \( x \), there is at most one polymorphic type \( \theta \), such that \( (x \mapsto \theta) \in \Gamma \). The second component \( \sigma \) is the substitution for human provided type annotations, so that under the substitution \( \sigma \), the type judgment will have its provided type. The operator \( \vdash \) is pronounced “turnstile”, or “entails”.

The syntax ranges over the syntactic categories of at-expressions, patterns, arguments, parameters, multiple-matchings, and expressions. The third component type-information is dependent on the syntax. For all syntactic categories, it contains the monomorphic type \( \tau \) to be assigned to the syntactic category. Typing parameters yields a possibly updated environment \( \Gamma_2 \) while typing patterns returns an incremental typing environment \( \Delta \) that maps the variables contained in the pattern to their respective types.

\[ \Gamma, \sigma \vdash \text{at-expression} : \tau \]
\[ \Gamma, \sigma \vdash \text{pattern} : (\tau, \Delta) \]
\[ \Gamma, \sigma \vdash \text{argument} : \tau \]
\[ \Gamma_1, \sigma \vdash \text{parameter} : (\tau, \Gamma_2) \]
\[ \Gamma, \sigma \vdash \text{multiple-matching} : \tau \]
\[ \Gamma, \sigma \vdash \text{expression} : \tau \]

The first rule above would read as: ”In the typing environment \( \Gamma \), at-expression under the substitution \( \sigma \) has type \( \tau \).”
3.1 Auxiliary functions

To infer the type of constants and binary operators, we assume that there exists the function \texttt{type-signature} which returns the polymorphic type signature of a built-in binary operator or a constant, in the style of the course Programming Languages & Compilers at University of Illinois at Urbana-Champaign\[11\]. Further, we assume that there exists the function \(+\), that describes the composition of two typing environments. For any two typing environments \(\Gamma_1, \Gamma_2\):

\[
\Gamma_1 + \Gamma_2 = \{(x \mapsto \theta) | (x \mapsto \theta) \in \Gamma_2 \lor ((\exists \theta'. (x \mapsto \theta') \in \Gamma_2) \land (x \mapsto \theta) \in \Gamma_1)\}
\]

Given an arguments-term \(a_1a_2...a_n\), the function \texttt{unlabeled}(a_1a_2...a_n) returns \texttt{true} if there are only unlabeled arguments in \(a_1a_2...a_n\) and \texttt{false} if there is at least one labeled argument.

\[
\text{unlabeled}(a_1a_2...a_n) = \begin{cases} 
\text{false} & \text{if } a_1 \text{ is labeled} \\
\text{unlabeled}(a_2...a_n) & \text{else}
\end{cases}
\]

where \(a_1a_2...a_n\) is an arguments-term and \(a_1,a_2,...,a_n\) are arguments

\[
\text{unlabeled}(a) = \begin{cases} 
\text{false} & \text{if } a \text{ is unlabeled} \\
\text{true} & \text{else}
\end{cases}
\]

where \(a\) is an argument

Further, we give the function \texttt{has_known arity} which takes a monomorphic function type and returns \texttt{true} if the type has a known arity and \texttt{false} if it has not.

\[
\text{has_known arity}(\tau_1 \rightarrow \tau_2) = \tau_2
\]

\[
\text{has_known arity}(\tau_3) = \begin{cases} 
\text{false} & \text{if } \exists \tau_1, \tau_2. \tau_1 \rightarrow \tau_2 = \tau_3 \text{ then } (\text{if } \exists \tau_3 = 'a \text{ then false else true})
\end{cases}
\]

Optional arguments make use of the \('a option\) type. The type \('a option\) has two data constructors, \texttt{None}, which takes no argument and \texttt{Some}, which takes one argument. The \texttt{type-signature} of \texttt{None} is \(\forall 'a.a\) \texttt{ option}. The \texttt{type-signature} of \texttt{Some} is \(\forall 'a.a \rightarrow 'a option\). Optionally labeled parameters require their arguments to be of type \texttt{option}, while they themselves have not necessarily optional types. Informally, we firstly define \(A(\tau, \tau_3)\) that takes a function type \(\tau\) and the type of the argument that we refer to with \(\tau_3\). If \(\tau_3\) is labeled, then \(A(\tau, \tau_3)\) is \texttt{true} if the first parameter in the function type \(\tau\) that matches in the label-name with \(\tau_3\) has the same type \(\tau_3\). If \(\tau_3\) is unlabeled, \(A(\tau, \tau_3)\) is \texttt{true} if the type of the left-most unlabeled argument of \(\rightarrow\) in
function type $\tau$ is the same as the type $\tau_3$. The function $A(\tau, \tau_3)$ is *false* if either no parameter matches with $\tau_3$ or the parameter that matched in the label-name is different from $\tau_3$. The function $A'(\tau, \tau_3)$ takes the same inputs as $A(\tau, \tau_3)$. If $\tau_3$ is unlabeled, then $A'(\tau, \tau_3)$ is *true* if the first parameter in the function type $\tau$ that is non-optional (non-optionally labeled or unlabeled) has the same type $\tau_3$. If $\tau_3$ is labeled or no parameter matches, $A'(\tau, \tau_3)$ is *false*. We use the term *type suffix* to describe the type $\tau$ yielded from omitting labels preceding $\tau$. For example, the type suffix of $l : \tau$ is $\tau$. The function $AA(\tau, \tau_3, \alpha)$ takes a function type $\tau$, the type of the argument that we refer to with $\tau_3$ and a type variable $\alpha$. It returns the substitution resulting from the unification of the type suffix of the first argument that matches with $\tau_3$ and the type suffix of $\tau_3$. If no parameter matches with $\tau_3$, and the right-most type in $\tau$ in terms of $\rightarrow$ is a type variable $\beta$, then $\sigma$ contains the mapping of the type variable $\beta$ to $(\tau_3 \rightarrow \alpha)$. If no parameter matches with $\tau_3$ and the right-most type in $\tau$ in terms of $\rightarrow$ is not a type variable, then $AA(\tau, \tau_3, \alpha)$ is *undefined*. The function $AA'(\tau, \tau_3)$ returns the substitution resulting from the unification of the type suffix of the first argument that is either non-optionally labeled or unlabeled and the type suffix of $\tau_3$. If no parameter matches with $\tau_3$, then $AA'(\tau, \tau_3)$ is undefined. The functions $R(\tau, \tau_3)$ and $R'(\tau, \tau_3)$ return the function type $\tau$ after omitting $\tau_3$ from $\tau$. If no type in $\tau$ matches with $\tau_3$, then $R(\tau, \tau_3)$ and $R'(\tau, \tau_3)$ are *undefined*. More formally, we define:

```plaintext
label_of(l : \tau_1) = Some l
label_of(?l : \tau_1) = Some l
label_of(\tau) = None \text{ where } \nexists l. \tau_1. \tau = l : \tau_1 \lor \tau = ?l : \tau_1

\text{type_argument_match}_A(l : \tau_1, l : \tau_2) = (\tau_1 = \tau_2)
\text{type_argument_match}_A(l : \tau_1, ?l : \tau_2) = (\tau_1 = \tau_2)
\text{type_argument_match}_A(?l : \tau_1, ?l : \tau_2) = (\tau_1 \text{ option } = \tau_2)
\text{type_argument_match}_A(?l : \tau_1, l : \tau_2) = (\tau_1 = \tau_2)
\text{type_argument_match}_A(\tau_1, \tau_2) = (\tau_1 = \tau_2) \text{ where } \tau_1 \text{ and } \tau_2 \text{ are not labeled}
\text{type_argument_match}_A(\tau_1, \tau_2) = \text{undefined} \text{ where } \text{label_of}(\tau_1) \neq \text{label_of}(\tau_2)
```
type_argument_match_A'(l:\tau_1, \tau_2) = (\tau_1 = \tau_2) \text{ where } \tau_2 \text{ is not labeled}

type_argument_match_A'(\tau_1, \tau_2) = (\tau_1 = \tau_2) \text{ where } \tau_1 \text{ and } \tau_2 \text{ are not labeled}

\text{type_argument_match_A'}(\tau_1, \tau_2) = false \text{ where } \tau_2 \text{ is labeled or } \tau_1 \text{ is optionally labeled}

type_argument_match_AA(l:\tau_1, l:\tau_2) = \text{Unify}(\tau_1, \tau_2)

type_argument_match_AA(l:\tau_1, ?l:\tau_2) = \text{Unify}(\tau_1, \tau_2)

\text{type_argument_match_AA}(?l:\tau_1, ?l:\tau_2) = \text{Unify}(\tau_1 \text{ option}, \tau_2)

\text{type_argument_match_AA}(?l:\tau_1, l:\tau_2) = \text{Unify}(\tau_1, \tau_2)

\text{type_argument_match_AA}(\tau_1, \tau_2) = \text{Unify}(\tau_1, \tau_2) \text{ where } \tau_1 \text{ and } \tau_2 \text{ are not labeled}

\text{type_argument_match_AA}(\tau_1, \tau_2) = \text{undefined where } \text{label_of}(\tau_1) \neq \text{label_of}(\tau_2)

\text{type_argument_match_AA}'(l:\tau_1, \tau_2) = \text{Unify}(\tau_1, \tau_2) \text{ where } \tau_2 \text{ is not labeled}

\text{type_argument_match_AA}'(\tau_1, \tau_2) = \text{Unify}(\tau_1, \tau_2) \text{ where } \tau_1 \text{ and } \tau_2 \text{ are not labeled}

\text{type_argument_match_AA}'(\tau_1, \tau_2) = false \text{ where } \tau_1 \text{ is optionally labeled or } \tau_2 \text{ is labeled}

A(\tau_1 \rightarrow \tau_2, \tau_3) = \text{if } \text{label_of}(\tau_1) = \text{label_of}(\tau_3) \text{ then type_argument_match_A}(\tau_1, \tau_3) \text{ else } A(\tau_2, \tau_3)

A(\tau, \tau_3) = \text{undefined where } \not\exists \tau_1, \tau_2.(\tau_1 \rightarrow \tau_2) = \tau

A'(\tau_1 \rightarrow \tau_2, \tau_3) = \text{if } \tau_1 \text{ is optionally labeled then } A'(\tau_2, \tau_3) \text{ else type_argument_match_A'}(\tau_1, \tau_3)

A'(\tau, \tau_3) = false \text{ where } \not\exists \tau_1, \tau_2.(\tau_1 \rightarrow \tau_2) = \tau

AA(\tau_1 \rightarrow \tau_2, \tau_3, 'a) = \text{if } \text{label_of}(\tau_1) = \text{label_of}(\tau_3) \text{ then type_argument_match_AA}(\tau_1, \tau_3)

\text{else } AA(\tau_2, \tau_3, 'a)

AA(\tau, \tau_3, 'a) = \text{if } \tau \text{ is a type-variable then } \{ \tau \rightarrow ('a) \} \text{ else undefined}

\text{where } \not\exists \tau_1, \tau_2.(\tau_1 \rightarrow \tau_2) = \tau

AA'(\tau_1 \rightarrow \tau_2, \tau_3) = \text{if } \tau_1 \text{ is optionally labeled then } AA'(\tau_2, \tau_3) \text{ else type_argument_match_AA'}(\tau_1, \tau_3)

AA'(\tau, \tau_3) = false \text{ where } \not\exists \tau_1, \tau_2.(\tau_1 \rightarrow \tau_2) = \tau

R(\tau_1 \rightarrow \tau_2, \tau_3) = \text{if } \text{label_of}(\tau_1) = \text{label_of}(\tau_3) \text{ then } \tau_2 \text{ else } (\tau_1 \rightarrow R(\tau_2, \tau_3))

R(\tau, \tau_3) = \text{undefined where } \not\exists \tau_1, \tau_2, \tau_1 \rightarrow = \tau
\[ R'(\tau \rightarrow \tau_2, \tau_3) = \begin{cases} 
\text{if } \tau_1 \text{ is non-optional then } \tau_2 \text{ else } (\tau_1 \rightarrow R'(\tau_2, \tau_3)) 
\end{cases} \]

\[ R'(\tau, \tau_3) = \text{undefined} \quad \text{where } \#\tau_1, \tau_2, \tau_1 \rightarrow = \tau \]

Furthermore, we introduce the functions \texttt{erase} and \texttt{erase}'. The function \texttt{erase} is mentioned but not described as an implementable function in the label-selective lambda calculus\[9\]. Both functions \texttt{erase} and \texttt{erase}' take the type \(\tau_n\) of a function after removing the argument types from the function type and the type \(\tau\) of the function that was applied. They output the type where all optional arguments of \(\rightarrow\) that occurred on the left of an applied unlabeled argument have been erased. They call the auxiliary functions \texttt{erase aux} and \texttt{erase aux}', respectively, that in turn iterate through \(\tau_n\) and \(\tau\) along \(\rightarrow\). They create a stack of function calls that, at the end of the iteration, is applied and recursively removes optional arguments if an applied unlabeled argument occurred on the right. The first component of the return tuple is set to \texttt{true} if and only if an applied unlabeled argument has been met and it is set to \texttt{false} if no applied unlabeled argument has occurred yet. The function \texttt{erase} concerns the default case of function application and the function \texttt{erase}' concerns the special case (see Figures 3.11 & 3.12).

\[ \texttt{erase}(\tau_n, \tau) = \text{match } \texttt{erase aux}(\tau_n, \tau) \text{ with } \]
\[ (\_ , \tau') \rightarrow \tau' \]

\[ \texttt{erase aux}(\tau_n, \tau) = \text{match } (\tau_n, \tau) \text{ with } \]
\[ ((\tau_1 \rightarrow \tau_{\text{rest}}), (\tau'_1 \rightarrow \tau'_{\text{rest}})) \rightarrow \]
\[ \text{if } \tau_1 = \tau'_1 \text{ then } \text{(match } \tau_1 \text{ with } \]
\[ |(x:_) \rightarrow (\text{match } \texttt{erase aux}(\tau_{\text{rest}}, \tau'_{\text{rest}})) \text{ with } \]
\[ |(\text{true}, \tau_{\text{return}}) \rightarrow (\text{true}, \tau_{\text{return}}) \]
\[ |(\text{false}, \tau_{\text{return}}) \rightarrow (\text{false}, (\tau_1 \rightarrow \tau_{\text{return}}))) \]
\[ |\_ \rightarrow (\text{match } \texttt{erase aux}(\tau_{\text{rest}}, \tau'_{\text{rest}})) \text{ with } \]
\[ |(b, \tau_{\text{return}}) \rightarrow (b, (\tau_1 \rightarrow \tau_{\text{return}}))) \]
\[ \text{else if } \tau'_1 \text{ is not labeled then } \text{(match } \texttt{erase aux}(\tau_{\text{rest}}, \tau'_{\text{rest}})) \text{ with } \]
\[ |(\_ , \tau_{\text{return}}) \rightarrow (\text{true}, \tau_{\text{return}}) \]
\[ \text{else } \texttt{erase aux}(\tau_{\text{rest}}, \tau'_{\text{rest}}) \]
\[(\tau_1, (\tau_1' \rightarrow \tau'_\text{rest})) \rightarrow \text{if } \tau_1' \text{ is not labeled then } (\text{match } (\text{erase}_{\text{aux}}(\tau_{\text{rest}}, \tau'_\text{rest}))) \text{ with} \]
\[
\begin{aligned}
| (\_, \tau_{\text{return}}) & \rightarrow (\text{true}, \tau_{\text{return}}) \\
| (false, \tau_{\text{return}}) & \rightarrow (false, (\tau_1 \rightarrow \tau_{\text{return}}))
\end{aligned}
\]

\[\text{else erase}_{\text{aux}}(\tau_{\text{rest}}, \tau'_\text{rest})\]

\[(\tau_1, \tau'_1) \rightarrow (false, \tau_1)\]

\[\text{erase}'(\tau_n, \tau) = \text{match } \text{erase}'_{\text{aux}}(\tau_n, \tau) \text{ with} \]

\[
(\_, \tau') \rightarrow \tau'
\]

\[\text{erase}'_{\text{aux}}(\tau_n, \tau) = \text{match } (\tau_n, \tau) \text{ with} \]

\[
((\tau_1 \rightarrow \tau_{\text{rest}}), (\tau'_1 \rightarrow \tau'_{\text{rest}})) \rightarrow \\
\text{if } \tau_1 = \tau'_1 \text{ then } (\text{match } \tau_1 \text{ with} \]

\[
| (?x:\_) \rightarrow (\text{match } \text{erase}'_{\text{aux}}(\tau_{\text{rest}}, \tau'_1)) \text{ with} \\
| (true, \tau_{\text{return}}) \rightarrow (\text{true}, \tau_{\text{return}}) \\
| (false, \tau_{\text{return}}) \rightarrow (false, (\tau_1 \rightarrow \tau_{\text{return}}))
\]

\[\| (\_, \tau_{\text{return}}) \rightarrow (true, \tau_{\text{return}}) \]

\[\text{else if } \tau'_1 \text{ is non-optional then } (\text{match } \text{erase}'_{\text{aux}}(\tau_{\text{rest}}, \tau'_1)) \text{ with} \]

\[
| (false, \tau_{\text{return}}) \rightarrow (false, (\tau_1 \rightarrow \tau_{\text{return}}))
\]

\[\text{else erase}'_{\text{aux}}(\tau_{\text{rest}}, \tau'_1)\]

\[(\tau_1, (\tau'_1 \rightarrow \tau'_{\text{rest}})) \rightarrow \text{if } \tau'_1 \text{ is non-optional then } (\text{match } \text{erase}'_{\text{aux}}(\tau_{\text{rest}}, \tau'_1)) \text{ with} \]

\[
| (\_, \tau_{\text{return}}) \rightarrow (true, \tau_{\text{return}}) \\
| (false, \tau_{\text{return}}) \rightarrow (false, (\tau_1 \rightarrow \tau_{\text{return}}))
\]

\[\text{else erase}'_{\text{aux}}(\tau_{\text{rest}}, \tau'_1)\]

\[(\tau_1, \tau'_1) \rightarrow (false, \tau_1)\]
3.2 Type Rules

We introduce the type rules for all syntactic categories.

3.2.1 At-expressions

For variables and constants, the assumed type $\tau$ has to be an instance of the polymorphic type of $x$ or $c$, respectively. Parenthesized expressions ($e$) type as $\tau$, if the expression $e$ types as $\tau$.

\[
\begin{array}{c}
\tau \prec \theta \\
\Gamma + \{x \mapsto \theta\}, \sigma \vdash x : \tau
\end{array} \quad \begin{array}{c}
\tau \prec \text{type-signature}(c) \\
\Gamma, \sigma \vdash c : \tau
\end{array} \quad \begin{array}{c}
\Gamma, \sigma \vdash (e) : \tau
\end{array}
\]

Figure 3.1: Type rules for at-expressions

3.2.2 Patterns

Patterns can be value-names, constants, wildcards, parenthesized patterns, patterns annotated with a type expression, a constructor applied to a pattern, the Cons-constructor applied to two patterns, "Or"-patterns of two patterns, or tuple patterns. A value-name $x$ types with $\tau$ if its assumed incremental typing environment maps $x$ to $\forall.\tau$. A pattern constant $c$ types if the assumed type $\tau$ is an instance of the type-signature of $c$ and the assumed incremental typing environment is empty. A wildcard types with any assumed type if its assumed incremental typing environment is empty. A parenthesized pattern with any assumed type if $p$ types. If the pattern is annotated with a type expression $t$, $p : t$ will type with the assumed type $\tau$ if $p$ types with $\tau$ and the supplied substitution $\sigma$ lifts $t$ to $\tau$. The assumed type $\tau_2$ of a constructor applied to a pattern $p$ is combined with the type of the pattern $p$ to $(\tau, \Delta)$. The type $\tau_1 \rightarrow \tau_2$ has to be an instance of the type-signature of the constructor for the constructor applied to the pattern to type with $\tau_2$. If a Cons-constructor is applied to two patterns $p$ and $ps$, the pattern will only type with $(\tau, \Delta)$ if $p$ and $ps$ type with $(\tau, \Delta_p)$ and $(\tau \text{ list}, \Delta_{ps})$, respectively, $\Delta$ is the union of $\Delta_p$ and $\Delta_{ps}$, where the domains of $\Delta_p$ and $\Delta_{ps}$ do not overlap. Union is defined as $\Delta_p \cup \Delta_{ps} = \{ (x \mapsto \theta) | (x \mapsto \theta) \in \Delta_p \lor (x \mapsto \theta) \in \Delta_{ps} \}$. The mutual exclusion of the domains is given through $\text{Dom}(\Delta_p) \cap \text{Dom}(\Delta_{ps}) = \{ \}$. If the pattern $p$ is an "Or"-pattern of two patterns $p_1$ and $p_2$, $p$ types with $(\tau, \Delta)$ if both $p_1$ and $p_2$ type with the same assumed type and the same incremental typing environment. They are required to contain the same variables and their variables need to type in the same way. For example, the pattern $(19::\text{xs}|3::\text{xs})$ types with $(\text{int list}, \{\text{xs} \mapsto \forall.\text{int list}\})$, because both $19::\text{xs}$ and $3::\text{xs}$ type with $(\text{int list}, \{\text{xs} \mapsto \forall.\text{int list}\})$. The pattern $(x::[] | x::\text{xs})$ does not type because $x::[]$ does not type with the incremental typing environment $\{ x \mapsto \forall.\ 'a, \text{xs} \mapsto \forall.\ 'a \text{ list}\}$,
since \( \text{xs} \) is not contained in \( x : [] \) while it is in \( x : \text{xs} \). A tuple pattern types if each tuple component types

\[
\text{with its assumed type component and typing environment component and if the incremental environments}
\]

of the tuple components do not overlap, that is, there is no variable that occurs in two tuple components.

\[
\begin{align*}
\text{Figure 3.2: Type rules for patterns}
\end{align*}
\]

### 3.2.3 Arguments

Arguments can be unlabeled, non-optionally labeled, or optionally labeled. If they are unlabeled, they have to be at-expressions. The types of non-optionally and optionally labeled arguments have to be prefixed with \( \text{label-name} : \) and \( ?\text{label-name} : \), respectively. Both can be either qualified with an atomic expression \( e \) or defaulted to their \( \text{label-name} \). If they are qualified by an atomic expression \( e \), \( e \) has to type with the assumed type \( \tau \) for non-optionally labeled arguments and \( \tau \) for optionally labeled arguments. If they are defaulted to their \( \text{label-name} \), they can be thought of "as qualified by their \( \text{label-name} \)". The argument \( \sim x \) is the same as \( \sim x : x \) and the argument \( ?x \) is the same as \( ?x : x \). The arguments \( \sim x \) and \( ?x \) type with \( x : \tau \) and \( ?x : \tau \), respectively if \( x \) types with \( \tau \).

### 3.2.4 Parameters

Parameters can be non-labeled, non-optionally labeled and optionally labeled. If the parameter is non-labeled, it has to be a pattern. If it is non-optionally labeled, either it is just a \( \sim \) \( \text{label-name} \), the \( \text{label-name} \) typed with a type expression or described by a pattern. Regardless of the form of the non-optional
\[ \Gamma, \sigma \vdash a : \tau \quad \text{where } a \text{ is an at-expression} \]

\[ \Gamma, \sigma \vdash a : \tau \quad \text{where } a \text{ is an argument} \]

\[
\begin{align*}
\Gamma, \sigma & \vdash e : \tau \\
\Gamma, \sigma & \vdash \tilde{x} : e : (x : \tau)
\end{align*}
\]

\[
\begin{align*}
\Gamma, \sigma & \vdash e : \tau \\
\Gamma, \sigma & \vdash ?x : e : (\tau x : \tau)
\end{align*}
\]

\[
\begin{align*}
\Gamma, \sigma & \vdash x : \tau \\
\Gamma, \sigma & \vdash \tilde{x} : (x : \tau)
\end{align*}
\]

\[
\begin{align*}
\Gamma, \sigma & \vdash x : \tau \\
\Gamma, \sigma & \vdash ?x : (\tau x : \tau)
\end{align*}
\]

Figure 3.3: Type rules for arguments

\[
\begin{align*}
\Gamma, \sigma & \vdash p : (\tau, \Delta) \quad \text{where } p \text{ is a pattern} \\
\Gamma, \sigma & \vdash p : (\tau, \Gamma + \Delta) \quad \text{where } p \text{ is a parameter}
\end{align*}
\]

Figure 3.4: Type rule for unlabeled parameters

If a parameter is optionally labeled, it can be just the sole \(?\text{label-name}\) possibly parenthesized, qualified by a pattern, typed with a type expression, defaulted with an expression or a combination of the aforementioned.
Regardless of the form of the optional parameter, its assumed type has to be prefixed with \( ?\text{label-name} \). If it is the sole \( ?\text{label-name} \), the \( \text{label-name} \) typed with \( \tau \text{ option} \) has to be stored in the assumed typing environment. If the optional parameter is qualified by a pattern \( p \), then the parameter only types with \( (\text{?label-name}:\tau, \Gamma + \Delta) \) if \( p \) types with \( (\tau \text{ option}, \Delta) \). For example, \( \texttt{?x: (Some \ y)} \) types as \( (\texttt{?x: 'a, \Gamma + \{y \mapsto \forall. 'a}\}) \) if \( \texttt{Some \ y} \) types as \( (\forall. 'a \text{ option}, \{y \mapsto \forall. 'a}\}) \). If the \( \text{label-name} \) is annotated with a type expression \( t \), the assumed substitution \( \sigma \) has to allow for \( \sigma(t) = \tau \text{ option} \) and the \( \text{label-name} \) typed with \( \tau \text{ option} \) has to be stored in the assumed typing environment. Similarly to the type rule of non-optionally labeled parameters, the parameter \( ?x: 'a \) types with \( (\texttt{?x: int, \Gamma + \{x \mapsto \text{int option}\}}) \) for \( \sigma = \{ 'a \mapsto \forall. \text{int option}\} \). If an optionally labeled parameter is defaulted through an expression \( e \), then type expressions or patterns qualifying the parameter have to type with the same type \( \tau \) as \( e \). However, the type of the parameter \( ?x \), referred to with \( ?x: \tau \), and the type of the expression \( e \), referred to with \( \tau \), need not be optional.

\[
\begin{align*}
\Gamma, \sigma \vdash ?x: (\tau: \tau, \Gamma + \{x : \forall. \tau \text{ option}\}) \\
\Gamma, \sigma \vdash p: (\tau \text{ option}, \Delta) &\quad \Gamma, \sigma \vdash ?x: p: (\tau: \tau, \Gamma + \Delta) \\
\text{no parentheses} &\quad \Gamma, \sigma \vdash e: \tau \\
\Gamma, \sigma \vdash ?(x): (\tau: \tau, \Gamma + \{x : \forall. \tau \text{ option}\}) &\quad \Gamma, \sigma \vdash ?(x\equiv e): (\tau: \tau, \Gamma + \{x : \forall. \tau\}) \\
\Gamma, \sigma \vdash p: (\tau \text{ option}, \Delta) &\quad \Gamma, \sigma \vdash p: (\tau, \Delta) &\quad \Gamma, \sigma \vdash e: \tau \\
\Gamma, \sigma \vdash ?x: (p): (\tau: \tau, \Gamma + \Delta) &\quad \Gamma, \sigma \vdash ?x: (p\equiv e): (\tau: \tau, \Gamma + \Delta) \\
\sigma(t) = \tau \text{ option} &\quad \sigma(t) = \tau &\quad \Gamma, \sigma \vdash e: \tau \\
\Gamma, \sigma \vdash ?(x: t): (\tau: \tau, \Gamma + \{x : \forall. \tau \text{ option}\}) &\quad \Gamma, \sigma \vdash ?(x: t\equiv e): (\tau: \tau, \Gamma + \{x : \forall. \tau\})) \\
\Gamma, \sigma \vdash p: (\tau \text{ option}, \Delta) &\quad \sigma(t) = \tau \quad \Gamma, \sigma \vdash e: \tau \\
\Gamma, \sigma \vdash ?x: (p: t): (\tau: \tau, \Gamma + \Delta) &\quad \Gamma, \sigma \vdash ?x: (p: t\equiv e): (\tau: \tau, \Gamma + \Delta) \\
\text{no default expression} &\quad \text{with expression } e
\end{align*}
\]

Figure 3.6: Type rules for \textit{optionally labeled parameters}
3.2.5 Multiple-matchings

The type of a multiple-matching is derived by combining the types of its parameters with the type of its body. We update the environment $\Gamma$ as we type the parameters. Parameters occurring to the right of others can reuse the variables in parameters to their left. The multiple-matching $(y::ys) \ ?x:(z=ys) \rightarrow y::z$ does type with ‘a list -> ?x:'a list -> 'a list (the pattern $z$ gets bound to the type ‘a list of the expression $ys$), while the typing of ?x:(z=ys) (y::ys) -> y::z fails because $ys$ is unbound in the typing of ?x:(z=ys). On the other hand if a variable within a multiple-matching occurs twice, anything on the right of the second occurrence will use the variable type of the later occurrence. The multiple-matching $x$ x -> x types with ‘a -> ‘b -> ‘b, but does not with ‘a -> ‘b -> ‘a, because the $x$ in the body relies on the type of $x$ in the second parameter.

$$
\begin{align*}
&\frac{
\Gamma, \sigma \vdash p_1 : (\tau_1, \Gamma_1) \quad \Gamma_1, \sigma \vdash p_2 : (\tau_2, \Gamma_2) \quad \ldots \quad (\Gamma_{n-1}, \sigma) \vdash p_n : (\tau_n, \Gamma_n) \quad (\Gamma_n, \sigma) \vdash e : \tau
}{
\Gamma, \sigma \vdash p_1 \ p_2 \ \ldots \ p_n \rightarrow e : \tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau}
\end{align*}
$$

Figure 3.7: Type rule for multiple-matchings

3.2.6 Expressions

An expression $e$ has type $\tau$ if the at-expression $e$ has type $\tau$.

$$
\frac{
\Gamma, \sigma \vdash e : \tau \quad \text{where } e \text{ is an at-expression}
}{
\Gamma, \sigma \vdash e : \tau \quad \text{where } e \text{ is an expression}
\}
$$

Figure 3.8: Type rule for expressions as at-expressions

The type of a binary operator expression $\tau_3$ has to be the return type of an instance of the type-signature of the binary operator $\odot$. Consider the type rule for binary operator expressions applied to 1 :: [2]. We show

$$
\frac{
\Gamma, \sigma \vdash e_1 : \tau_1 \quad \Gamma, \sigma \vdash e_2 : \tau_2 \quad \text{type-signature}(\odot) : \theta \quad \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \leftarrow \theta
}{
\Gamma, \sigma \vdash e_1 \odot e_2 : \tau_3}
$$

Figure 3.9: Type rule for binary operator expressions

two examples of attempts at typing. In the first example, we conjecture that 1 :: [2] has type int list, in the second example, we conjecture that it has type ‘a list. Because in this instance, $e_1$ is 1 and $e_2$ is [2] and 1 is of type int while [2] is of type int list, the program only types if $\tau_1$ is int and $\tau_2$ is int list. Because :: has the type-signature $\forall \ 'a. 'a$ -> ‘a list -> ‘a list, $\tau_3$ has to be of type int list for
\[ \tau_1 \rightarrow \tau_2 \rightarrow \tau_3 (= \texttt{int} \rightarrow \texttt{int list} \rightarrow \tau_3) \] to be an instance of \( \forall 'a.'a \rightarrow 'a \texttt{list} \rightarrow 'a \texttt{list} \) with the substitution \{ 'a \mapsto \texttt{int} \}. The type \( \tau_3 \) cannot be \( 'a \texttt{list} \) because there is no substitution \( \sigma \) such that \( \texttt{int} \rightarrow \texttt{int list} \rightarrow 'a \texttt{list} \rightarrow 'a \texttt{list} \) is an instance of \( \forall 'a.'a \rightarrow 'a \texttt{list} \rightarrow 'a \texttt{list} \).

The types of tuple components are combined to form the type of the tuple. Each individual tuple component is typed against the combination. Consider the expression \( (8, [17]) \). Assume that it has the type \( \texttt{int} \ast \texttt{int} \). This would require \( 8 \) to be of type \( \texttt{int} \) and \( [17] \) to be of type \( \texttt{int} \) as well. Since \( [17] \) only types with type \( \texttt{int list} \), this leads to a contradiction. Only if we assume \( (8, [17]) \) to be of type \( \texttt{int} \ast \texttt{int list} \) will both \( 8 \) and \( [17] \) type. The environment \( \Gamma \) has to be the same for the type judgment of each tuple component.

\[
\frac{\Gamma, \sigma \vdash e_1 : \tau_1 \quad \Gamma, \sigma \vdash e_2 : \tau_2}{\Gamma, \sigma \vdash e_1, e_2 : (\tau_1 \ast \tau_2)}
\]

Figure 3.10: Type rule for tuple expressions

The assumed type \( \tau_2 \) of the constructor applied to the expression \( e \) is combined with the type of the expression \( e \) to \( (\tau_1 \rightarrow \tau_2) \). The type \( \tau_1 \rightarrow \tau_2 \) has to be an instance of the type-signature of the constructor.

\[
\frac{\Gamma, \sigma \vdash e : \tau_1 \quad (\tau_1 \rightarrow \tau_2) \prec \text{type-signature}(\text{constr})}{\Gamma, \sigma \vdash \text{constr } e : \tau_2}
\]

Figure 3.11: Type rule for constructor expressions

We treat the type of a \texttt{fun} abstraction as the same as the type of its multiple-matching.

\[
\frac{\Gamma, \sigma \vdash m : \tau}{\Gamma, \sigma \vdash \text{fun } m : \tau}
\]

Figure 3.12: Type rule for \texttt{fun}-abstractions

In function application, a function is applied to a number of arguments. First, the type of the function \( e \) is checked as \( \tau_0 \). Through the use of \( A \) and \( R \), we iterate through \( \tau_0 \) and the sequence of arguments. For each argument \( a_i \), we check that its type \( \tau'_i \) occurs in an argument position of the function type. If it does, we remove it from the function type through \( R \). The resulting type \( \tau_i \) can be thought of as the type of the function after applying it to \( a_i \). The type \( \tau_n \) is the type after applying the last argument \( a_n \). After applying the last argument, OCaml erases any optional parameters that have not been applied and that occurred to the left of an applied unlabeled parameter. We call the function \texttt{erase} with \( \tau_n \) and \( \tau_0 \) and iterate through \( \tau_n \), checking for each optionally labeled type whether it occurs before an unlabeled type and if it does, \texttt{erase} it.
Let us consider the function \( \text{fun } \sim x \rightarrow \text{fun } \sim y \rightarrow \text{fun } z \rightarrow (x, z) \) which we call \( f \). The function \( f \) types with \( x:\text{bool} \rightarrow ?y:\text{'a} \rightarrow \text{int} \rightarrow \text{bool} * \text{int} \) which we refer to with \( \tau_0 \). Let us consider the application of \( f \) to the arguments-term \( \sim 5 \sim x: \text{true} \). The first argument \( 5 \) types with \( \text{int} = \tau'_1 \) while the second argument \( \sim x: \text{true} \) types as \( x:\text{bool} = \tau'_2 \).

1. \( A(x:\text{bool} \rightarrow ?y:\text{'a} \rightarrow \text{int} \rightarrow \text{bool} * \text{int}, \text{int}) = \text{true} \)

2. \( \tau_1 = \text{R}(x:\text{bool} \rightarrow ?y:\text{'a} \rightarrow \text{int} \rightarrow \text{bool} * \text{int}, \text{int}) = x:\text{bool} \rightarrow ?y:\text{'a} \rightarrow \text{bool} * \text{int} \)
   
   (\text{int} \text{ was omitted through the application of } 5)

3. \( A(x:\text{bool} \rightarrow ?y:\text{'a} \rightarrow \text{bool} * \text{int}, x:\text{bool}) = \text{true} \)

4. \( \tau_2 = \text{R}(x:\text{bool} \rightarrow ?y:\text{'a} \rightarrow \text{bool} * \text{int}, x:\text{bool}) = ?y:\text{'a} \rightarrow \text{bool} * \text{int} \)

5. \( \text{erase}(?y:\text{'a} \rightarrow \text{bool} * \text{int}, x:\text{bool} \rightarrow ?y:\text{'a} \rightarrow \text{int} \rightarrow \text{bool} * \text{int}) = \text{bool} * \text{int} \)

Because \( 5 \) is an applied unlabeled argument that occurs after \( ?y \), the type of \( ?y \) is \textit{erased} from \( \tau_2 \), leaving us with \( \text{bool} * \text{int} \).

"As a special case, if the function has known arity, all the arguments are unlabeled and their number matches the number of non-optional parameters, then labels are ignored and non-optional parameters are matched in their definition order"[4]. One assumption of the special case type rule would be \( \forall \tau_0. (\Gamma, \sigma \vdash e : \tau_0) \implies \text{has_known arity}(\tau_0) \). The principle of inductive rule definition of a mutually recursive family of predicates \( p_i(x) \) by a family of rules does not allow assumptions of any rule to be of the form \( p_j(y) \implies q \). Hence, we cannot define the function application special-case type rule with the assumption \( \forall \tau_0. (\Gamma, \sigma \vdash e : \tau) \implies \text{has_known arity}(\tau_0) \). Instead, we will give the rule calling the type inference function in the assumption (see Section 3.4).
Chapter 4

Type inference

The type inferencer will investigate whether some syntax has a type that is an instance of a particular type using an instance of a given environment $\Gamma$ and will either return a substitution giving that instance or return failure. The idea of type inference is to first give each syntactic construct syntax a type variable $'a$ as its type. The function(s) compute a mapping of $'a$ to the correct type of the respective syntax. Throughout type inference we generate a number of type equations that represent constraints for successful inference. These equations have to be unified to check whether the syntax has a type. The resulting substitution is returned by the type inferencer.

4.1 Type unification

Unification is an algorithmic process of solving equations between symbolic expressions. In the style of the course Programming Languages & Compilers[11], we apply unification to type equations (represented as pairs). Given a set of type equations

$$\{(s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)\}$$

the unification problem is defined as: Does there exist a substitution $\sigma$ (the unification solution) of types for variables such that $\sigma(s_i) = \sigma(t_i)$ for all $i = 1, \ldots, n$? Let $S = \{(s_1, t_1), \ldots, (s_n, t_n)\}$ be a unification problem. If $S = \{\}$, then $\sigma = \{\}$ describes the identity function. If there exists an equation of types $\{(s, t)\}$ such that $S = \{(s, t)\} \cup S'$, then we differentiate ten cases:

1. If $s = t$, then $\text{Unify}(S) = \text{Unify}(S')$

2. If $x$ is a label-name and $s = (x : s_1) \land t = (x : t_1)$, then $\text{Unify}(S) = \text{Unify}(\{(s_1, t_1)\} \cup S')$

3. If $x$ is a label-name and $s = (?x : s_1) \land t = (?x : t_1)$, then $\text{Unify}(S) = \text{Unify}(\{(s_1, t_1)\} \cup S')$

4. If $s = (s_1 \to s_2) \land t = (t_1 \to t_2)$, then $\text{Unify}(S) = \text{Unify}(\{(s_1, t_1), (s_2, t_2)\} \cup S')$
5. If \( s = (s_1 \cdot s_2) \land t = (t_1 \cdot t_2) \), then \( \text{Unify}(S) = \text{Unify}(((s_1, t_1), (s_2, t_2)) \cup S') \)

6. If \( s = (s_1 \text{ list}) \land t = (t_1 \text{ list}) \), then \( \text{Unify}(S) = \text{Unify}(((s_1, t_1)) \cup S') \)

7. If \( s = (s_1 \text{ option}) \land t = (t_1 \text{ option}) \), then \( \text{Unify}(S) = \text{Unify}(((s_1, t_1)) \cup S') \)

8. If \( t \) is a type variable and \( s \) is not, \( \text{Unify}(S) = \text{Unify}((\{t, s\}) \cup S') \)

9. If \( s \) is a type variable, and \( s \) does not occur in \( t \), then let \( \phi = \{s \mapsto t\} \) and \( \rho = \text{Unify}(\phi(S')) \),
   \( \text{Unify}(S) = \{s \mapsto \rho(t)\} \circ \rho \)

10. If \( s \) is a type variable and \( s \) occurs in \( t \) but \( s \neq t \), or none of the above rules applies then \( \text{Unify}(S) \) returns failure

The first rule removes equations where the two types are equal. The second and third rule decompose unifying labeled types into unifying their type suffixes. The fourth, fifth, sixth and seventh rule decompose unifying function, tuple, list and option types into unifying their type components. If the equation contains a type variable, the eighth rule reorients the order and puts the type variable as the first component, the ninth rule Eliminate can transform a type equation into a substitution when the left part of the equation is a type variable. Due to its declarative nature, the order of application of unification rules does not matter. If there exists a unification, application of the rules will result in a substitution \( \sigma \) and if not, repeated application will return failure.

### 4.2 Type inference function(s)

For each syntactic category except value-names, label-names, constants, binops, and constructors, we differentiate a type inference function:

\[
\begin{align*}
\text{infer}_{\text{at expression}}(\Gamma, \text{at expression},'a) &= \sigma \\
\text{infer}_{\text{pattern}}(\Gamma, \text{pattern},'a) &= (\sigma, \Delta) \\
\text{infer}_{\text{argument}}(\Gamma, \text{argument},'a) &= \sigma \\
\text{infer}_{\text{parameter}}(\Gamma, \text{parameter},'a) &= (\sigma, \Gamma') \\
\text{infer}_{\text{multiple-matching}}(\Gamma, \text{multiple-matching},'a) &= \sigma \\
\text{infer}_{\text{expression}}(\Gamma, \text{expression},'a) &= \sigma
\end{align*}
\]
Each type-inference function takes a typing environment $\Gamma$ (a mapping from variable names to polymorphic types), some syntax and a type variable $'a$. It returns a substitution of types for type variables that are the constraints on type variables necessary and sufficient for $\Gamma \vdash \text{syntax} : 'a$. In addition, the function \text{infer\_pattern} returns an incremental typing environment $\Delta$ that contains all variables occurring in the pattern mapped to their types and the function \text{infer\_parameter} returns an updated environment $\Gamma'$. Both environments $\Gamma'$ and $\Delta$ are required for type inference of the body in multiple-matchings. The functions apply a match-construct to distinguish between different cases for each syntactic category. To instantiate quantified type variables with type variables that have not been used yet, we give the functions \text{freshInstance}(\theta) and \text{fresh}(). The function \text{freshInstance} takes a polymorphic type $\theta$ and replaces all quantified type variables in $\theta$ with fresh ones through calling the function fresh. The function fresh() generates a fresh type variable through instantiating a counter-variable that fresh() increments after every instantiation(in the style of the course Programming Languages & Compilers at University of Illinois at Urbana-Champaign[11]).

### 4.2.1 infer\_at\_expression:

The type of a variable $x$ is inferred as a fresh instance of the polymorphic type saved for $x$ in $\Gamma$. Constants are assigned a fresh instance of their polymorphic type signature. Parenthized expressions are assigned the same type as their expression.

- $\text{infer\_at\_expression}(\Gamma, x, 'a) = \text{Unify}\{ 'a, \text{freshInstance}(\Gamma(x))\}$
- $\text{infer\_at\_expression}(\Gamma, c, 'a) = \text{Unify}\{ 'a, \text{freshInstance}\left(\text{type-signature}(c)\right)\}$
- $\text{infer\_at\_expression}(\Gamma, (e), 'a) = \text{infer\_expression}(\Gamma, e, 'a)$

### 4.2.2 infer\_pattern:

Variable patterns and wildcards are inferred as the assumed type, constants as the unification of an instance of their \text{type-signature} and the assumed type $'a$. If the pattern is typed with a type expression $t$, $t$ is unified with $'a$. The type inference of a constructor applied to a pattern is split into four steps: First, we infer the type of the pattern. Second, we generate a fresh instance of the \text{type-signature} of the constructor. Third, we combine the type of the pattern and the assumed type $'a$ to a function type. Lastly, we unify the fresh instance of the \text{type-signature} of the constructor with the function type combination of the type of the pattern and the assumed type.

- $\text{infer\_pattern}(\Gamma, x, 'a) = (\emptyset, \{ x \mapsto \forall 'a \})$
\[ \text{infer\_pattern}(\Gamma, c, \cdot a) = (\text{Unify}\{(\cdot a, \text{freshInstance\(\{\text{type-signature\(c\}\}\)}), \{\}) \}) \]

\[ \text{infer\_pattern}(\Gamma, \cdot, \cdot a) = (\{\}, \{\}) \]

\[ \text{infer\_pattern}(\Gamma, (p), \cdot a) = \text{infer\_pattern}(\Gamma, p, \cdot a) \]

\[ \text{infer\_pattern}(\Gamma, (p : t), \cdot a) = (\sigma_2, \sigma_2(\Delta)) \text{ where} \]
\[ - (\sigma_1, \Delta) = \text{infer\_pattern}(\Gamma, p, t) \]
\[ - \sigma_2 = \text{Unify}(\cdot a, \sigma_1(t)) \circ \sigma_1 \]

\[ \text{infer\_pattern}(\Gamma, \text{constr\(p\)}, \cdot a) = (\text{Unify}\{(\text{freshInstance\(\{\text{type-signature\(\text{constr}\}\)}), \sigma(\cdot b \rightarrow \cdot a)}\}) \circ \sigma, \sigma(\Delta) \text{ where} \]
\[ - \cdot b \text{ is a fresh variable} \]
\[ - (\sigma, \Delta) = \text{infer\_pattern}(\Gamma, p, \cdot b) \]

If the pattern is a Cons-Constructor applied to two patterns \(p\) and \(ps\), the type of \(p\) is unified with the type of \(ps\) without the \text{list} addition. The type of \(ps\) in turn is unified with \(\cdot a\). When inferring the type of an "Or"-pattern, we firstly infer the types of both component patterns \(p_1\) and \(p_2\). Because both patterns must contain the same variables with the same types, we check that the domains of the incremental typing environments \(\Delta_1\) and \(\Delta_2\) are the same and then unify the types of every variable in \(\Delta_1\) and \(\Delta_2\). The substitution \(\sigma_{\text{result}}\) is the result of composing all substitutions generated through unifying individual variable types, inferring the patterns, and unifying the types of \(p_1\) and \(p_2\) with each other. The resulting type is unified with \(\cdot a\) to return \(\sigma'_{\text{result}}\). The substitution \(\sigma'_{\text{result}}\) and the incremental environment \(\sigma'_{\text{result}}(\Delta_1)\) are returned. It does not matter whether we return \(\sigma'_{\text{result}}(\Delta_1)\) or \(\sigma'_{\text{result}}(\Delta_2)\) because after comparing the domains, and updating each variable type, if \(\sigma'_{\text{result}}\) exists, \(\sigma'_{\text{result}}(\Delta_1) = \sigma'_{\text{result}}(\Delta_2)\). We infer tuple patterns through iterating through all tuple pattern components, checking that their incremental environments do not overlap in the domain and then unify their types.

\[ \text{infer\_pattern}(\Gamma, p : : ps, \cdot a) = (\sigma_4, \sigma_4(\Delta_p + \Delta_{ps})) \text{ where} \]
\[ - \cdot b, \cdot c \text{ are fresh type variables} \]
\[ - (\sigma_1, \Delta_p) = \text{infer\_pattern}(\Gamma, p, \cdot b) \]
\[ - (\sigma_2, \Delta_{ps}) = \text{infer\_pattern}(\sigma_1(\Gamma), \sigma_1(ps), \cdot c \text{ list}) \]
\[ - \text{requires } \text{Dom}(\Delta_p) \cap \text{Dom}(\Delta_{ps}) = \{\} \]
\[ - \sigma_3 = \text{Unify}\{(\sigma_2 \circ \sigma_1(\cdot b), \sigma_2 \circ \sigma_1(\cdot c))\} \circ \sigma_2 \circ \sigma_1 \]
- \( \sigma_4 = \text{Unify}\{('a, (\sigma_3('c list)))\} \circ \sigma_3 \)

- \text{infer\_pattern}(\Gamma, p_1|p_2, 'a) = (\sigma'_{\text{result}}, \sigma'_{\text{result}}(\Delta_1)) \) where

  - 'b, 'c are fresh type variables
  - \((\sigma_{p_1}, \Delta_1) = \text{infer\_pattern}(\Gamma, p_1, 'b)\)
  - \((\sigma_{p_2}, \Delta_2) = \text{infer\_pattern}(\sigma_{p_1}(\Gamma), \sigma_{p_1}(p_2), 'c)\)
  - requires \(\text{Dom}(\Delta_1) = \text{Dom}(\Delta_2)\)
  - \(\{x_1, x_2 \ldots, x_n\} = \text{Dom}(\Delta_1) = \text{Dom}(\Delta_2)\)
  - \(\sigma_1 = \text{Unify}\{(\Delta_1(x_1), \Delta_2(x_1))\} \circ \sigma_{p_1} \circ \sigma_{p_2}\)
  - for \(2 \leq i \leq n\)
    - \(\sigma_i = \text{Unify}\{(\Delta_1(x_i), \Delta_2(x_i))\} \circ \sigma_{i-1}\)
  - \(\sigma_{\text{result}} = \text{Unify}\{\sigma_n('b), \sigma_n('c)\} \circ \sigma_n\)
  - \(\sigma'_{\text{result}} = \text{Unify}\{('a, \sigma_{\text{result}}('b))\} \circ \sigma_{\text{result}}\)

- \text{infer\_pattern}(\Gamma, (p_1, p_2), 'a) = (\sigma_4, \sigma_4(\Delta_1 + \Delta_2)) \) where

  - 'b, 'c are fresh type variables
  - \((\sigma_1, \Delta_1) = \text{infer\_pattern}(\Gamma, p_1, 'b)\)
  - \((\sigma_2, \Delta_2) = \text{infer\_pattern}(\sigma_1(\Gamma), \sigma_1(p_2), 'c)\)
  - \(\sigma_3 = \sigma_2 \circ \sigma_1\)
  - \(\text{Dom}(\Delta_1) \cap \text{Dom}(\Delta_2) = \{\}\)
  - \(\sigma_4 = \text{Unify}\{('a, \sigma_3('b * 'c))\} \circ \sigma_3\)

**4.2.3 \text{infer\_argument}:**

For each argument, we differentiate between labeled and unlabeled arguments, optionally labeled and non-optionally labeled, modified with a pattern or not. We unify each argument type with its label-prefix with the assumed type 'a and return the resulting substitution \(\sigma_2\).

- \text{infer\_argument}(\Gamma, e, 'a) = \text{infer\_at\_expression}(\Gamma, e, 'a) if \(e\) is an at-expression

- \text{infer\_argument}(\Gamma, 'x : e, 'a) = \sigma_2 where

  - 'b is a fresh variable
\[
\sigma_1 = \text{infer} \text{expression}(\Gamma, e, 'b)
\]
\[
\sigma_2 = \text{Unify}\{('a, x: \sigma_1('b))\} \circ \sigma_1
\]

- \text{infer}\_\text{argument}(\Gamma, ?x:e, 'a) = \sigma_2 \text{ where }

- 'b is a fresh variable

- \sigma_1 = \text{infer} \text{expression}(\Gamma, e, 'b)

- \sigma_2 = \text{Unify}\{('a, \sigma_1(?x:'b))\} \circ \sigma_1

- \sigma = \text{Unify}\{('a, x: \text{freshInstance}(\Gamma(x)))\}

- \text{infer}\_\text{argument}(\Gamma, ?x,'a) = \sigma \text{ where }

- 'b is a fresh variable

- \sigma = \text{Unify}\{('a, ?x: \text{freshInstance}(\Gamma(x)))\}

4.2.4 \text{infer}\_\text{parameter}:

If no labeled parameter matches, we call \text{infer}\_\text{pattern}.

- \text{infer}\_\text{parameter}(\Gamma, p,'a) = (\sigma, \Gamma + \Delta) \text{ if } p \text{ is a pattern, where }

- (\sigma, \Delta) = \text{infer}\_\text{pattern}(\Gamma, p,'a)

Single non-optionally labeled parameters are type inferred through unification of the assumed type with an x:-prefixed fresh type variable. Non-optionally labeled parameters that are typed with a type expression are inferred through unifying the fresh type variable with the type expression. If the non-optionally labeled parameter is typed with a pattern \(p\), the type of the pattern is unified with the fresh type variable.

- \text{infer}\_\text{parameter}(\Gamma, \sim x,'a) = (\sigma, (\Gamma + \{x \mapsto \forall.'b\})) \text{ where }

- 'b is a fresh variable

- \sigma = \text{Unify}\{('a, x : 'b)\}

- \text{infer}\_\text{parameter}(\Gamma, \sim(x : t),'a) = (\sigma_2, \sigma_2(\Gamma + \{x \mapsto \forall.'b\})) \text{ where }

- 'b is a fresh variable

- \sigma_1 = \text{Unify}\{(t, 'b)\}
- $\sigma_2 = \text{Unify}\{\langle a, x : \sigma_1(b) \rangle \} \circ \sigma_1$

- \text{infer\_parameter}(\Gamma, \text{\textasciitilde}x : p, 'a) = \sigma_2, \sigma_2(\Gamma + \Delta) \quad \text{where}
  
  - 'b is a fresh variable
  
  - (\sigma_1, \Delta) = \text{infer\_pattern}(\Gamma, p, 'b)
  
  - $\sigma_2 = \text{Unify}\{\langle a, x : \sigma_1(b) \rangle \} \circ \sigma_1$

Optionally labeled parameters are inferred similarly to non-optionally labeled parameters with the subtle difference that they are given option-alized types if they are not defaulted with expressions. If they are not defaulted with expressions, the type of the expression is inferred and then unified with the assumed type, the type of the type expression and/or the type of the pattern.

4.2.4.1 no parentheses:

- \text{infer\_parameter}(\Gamma, ?x, 'a) = (\sigma, \sigma(\Gamma + \{x \mapsto \forall b.\text{option}\})) \quad \text{where}
  
  - 'b is a fresh variable
  
  - $\sigma = \text{Unify}\{\langle a, ?x : 'b \rangle \}$

- \text{infer\_parameter}(\Gamma, ?x : t, 'a) = (\sigma_2, \sigma_2(\Gamma + \{x \mapsto \forall b.\text{option}\})) \quad \text{where}
  
  - 'b is a fresh variable
  
  - $\sigma_1 = \text{Unify}\{\langle a, ?x : 'b \rangle \}$
  
  - $\sigma_2 = \text{Unify}\{\langle \sigma_1(t), \sigma_1(b.\text{option}) \rangle \} \circ \sigma_1$

4.2.4.2 no default expression:

- \text{infer\_parameter}(\Gamma, ?(x), 'a) = (\sigma, \sigma(\Gamma + \{x \mapsto \forall b.\text{option}\})) \quad \text{where}
  
  - 'b is a fresh variable
  
  - $\sigma = \text{Unify}\{\langle a, ?x : 'b \rangle \}$

- \text{infer\_parameter}(\Gamma, ?x : (p), 'a) = (\sigma_3, \sigma_3(\Gamma + \Delta)) \quad \text{where}
  
  - 'b, 'c are fresh variables
  
  - (\sigma_1, \Delta) = \text{infer\_pattern}(\Gamma, p, 'b)
  
  - $\sigma_2 = \text{Unify}\{\langle a, ?x : 'c \rangle \} \circ \sigma_1$
\[
\sigma_3 = \text{Unify}\{(\sigma_2('b), \sigma_2('c\text{ option})}\} \circ \sigma_2
\]

- infer\_parameter(\Gamma, ?x:p,'a) = (\sigma_3, \sigma_3(\Gamma + \Delta)) \text{ where }
  
  - 'b, 'c are fresh variables
  
  - (\sigma_1, \Delta) = \text{infer\_pattern}(\Gamma, p,'b)
  
  - \sigma_2 = \text{Unify}\{('a, ?x : 'c)\} \circ \sigma_1
  
  - \sigma_3 = \text{Unify}\{(\sigma_2('b), \sigma_2('c\text{ option})}\} \circ \sigma_2

- infer\_parameter(\Gamma, ?x:(p:t),'a) = (\sigma_3, \sigma_3(\Gamma + \Delta)) \text{ where }
  
  - 'b is a fresh variable
  
  - (\sigma_1, \Delta) = \text{infer\_pattern}(\Gamma, p,t)
  
  - \sigma_2 = \text{Unify}\{('a, ?x : 'b)\} \circ \sigma_1
  
  - \sigma_3 = \text{Unify}\{(\sigma_2(t), \sigma_2('b\text{ option})}\} \circ \sigma_2

4.2.4.3 with expression e:

- infer\_parameter(\Gamma, ?(x=e),'a) = (\sigma_2, \sigma_2(\Gamma + \{x \mapsto \forall.\sigma_2('c)\})) \text{ where }
  
  - 'b is a fresh variable
  
  - \sigma_1 = \text{infer\_expression}(\Gamma, e,'b)
  
  - \sigma_2 = \text{Unify}\{('a, ?x : 'b)\} \circ \sigma_1

- infer\_parameter(\Gamma, ?x:(p=e),'a) = (\sigma_3, \sigma_3(\Gamma + \Delta)) \text{ where }
  
  - 'b, 'c are fresh variables
  
  - \sigma_1 = \text{infer\_expression}(\Gamma, e,'b)
  
  - (\sigma_2, \Delta) = \text{infer\_pattern}(\sigma_1(\Gamma), \sigma_1(p), 'c)
  
  - \sigma'_2 = \sigma_2 \circ \sigma_1
  
  - \sigma_3 = \text{Unify}\{('a, ?x : \sigma'_2('b))\} \circ \sigma'_2
  
  - \sigma_4 = \text{Unify}\{(\sigma_3('b), \sigma_3('c))\} \circ \sigma_3

- infer\_parameter(\Gamma, ?(x:t=e),'a) = (\sigma_2, \sigma_2(\Gamma + \{x \mapsto \forall.t\})) \text{ where }
  
  - \sigma_1 = \text{infer\_expression}(\Gamma, e, t)
\[ \sigma_2 = \text{Unify}\{('a, ?x : t)\} \circ \sigma_1 \]

- \text{infer\_parameter}(\Gamma, ?x : (p : t = e), '\text{a}') = (\sigma_4, \sigma_4(\Gamma + \Delta)) \text{ where }

\[ '\text{b} \text{ are fresh variables} \]
\[ \sigma_1 = \text{infer\_expression}(\Gamma, e, '\text{b}') \]
\[ (\sigma_2, \Delta) = \text{infer\_pattern}(\sigma_1(\Gamma), \sigma_1(p), \sigma_1(t)) \]
\[ \sigma'_2 = \sigma_2 \circ \sigma_1 \]
\[ \sigma_3 = \text{Unify}\{\sigma'_2('b), \sigma'_2(t)\} \circ \sigma'_2 \]
\[ \sigma_4 = \text{Unify}\{('a, ?x : 'b)\} \circ \sigma_3 \]

4.2.5 \text{ infer\_multiple\_matching}:

We infer each parameter type of a multiple-matching, iteratively incrementing the environment \( \Gamma \) to infer the function body \( e \) with the updated \( \Gamma \). Lastly, we unify the combination of the types of all parameters and the function body \( e \) with the assumed type \('a\).

- \text{infer\_multiple\_matching}(\Gamma, p_1 p_2 \ldots p_n \rightarrow e, '\text{a}') = \text{Unify}\{('a, \sigma_{\text{result}}('b_1 \rightarrow 'b_2 \rightarrow \ldots \rightarrow 'b_n \rightarrow 'c))\} \circ \sigma_{\text{result}} \text{ where }

\[ '\text{b}_1, '\text{b}_2, \ldots, '\text{b}_n, 'c \text{ are fresh variables} \]
\[ (\sigma_1, \Gamma_1) = \text{infer\_parameter}(\Gamma, p_1, '\text{b}_1) \]
\[ \text{for } 2 \leq i \leq n \]
\[ * (\sigma'_i, \Gamma_i) = \text{infer\_parameter}(\sigma_{i-1}(\Gamma_{i-1}), \sigma_{i-1}(p_i), '\text{b}_i) \]
\[ * \sigma_i = \sigma'_i \circ \sigma_{i-1} \]
\[ \sigma_{\text{result}} = \text{infer\_expression}(\Gamma_n, \sigma_n(e), 'c) \circ \sigma_n \]

4.2.6 \text{ infer\_expression}:

- \text{infer\_expression}(\Gamma, e, '\text{a}') = \text{infer\_at\_expression}(\Gamma, e, '\text{a}') \text{ if } e \text{ is an atomic expression}

The type of the binary operator expression \( e_1 \otimes e_2 \) is inferred through inferring the types of its expressions \( e_1 \) and \( e_2 \) as well as the type-signature of its binary operator \( \otimes \). The substitution that is returned is the result of unifying a monomorphic instance of the type signature of \( \otimes \) with the combination of the types of \( e_1, e_2 \), and the assumed type variable \('a\).
• infer_expression(Γ, e₁ ⊗ e₂, 'a) = Unify{([σ₂('b → 'c → 'a), freshInstance(θ)]) ◦ σ₂} where

  - 'b, 'c are fresh variables
  - σ₁ = infer_expression(Γ, e₁, 'b)
  - σ₂ = infer_expression(σ₁(Γ), σ₁(e₂), 'c) ◦ σ₁
  - θ = type-signature(⊗)

Let us consider the type inference of the binary operator expression x + x in the environment Γ = {x ↦ ∀.'b}, that is infer_expression(Γ, x + x, 'a).

1. 'c, 'd are fresh variables

2. infer_expression({x ↦ ∀.'b}, x, 'c) = {'c → 'b}

3. infer_expression({'c ↦ 'b}({x ↦ ∀.'b}), x, 'd) ◦ {'c ↦ 'b} = {'d → 'b, 'c → 'b}

4. freshInstance(type-signature(+)) = int → int → int

5. Unify({'b → 'b → 'a}, int → int → int) ◦ {'d → 'b, 'c → 'b} = {'b → int, 'a → int, 'd → int, 'c → int}

The variables 'c and 'd are newly instantiated fresh type variables. The type of x is inferred as 'c and 'd with the substitutions {'c → 'b} and {'d → 'b}, respectively. The binary operator + is typed as int → int → int. To unify 'b → 'b → 'a with int → int → int, 'b and 'a have to be substituted with int.

The type of a tuple is inferred by separately inferring the types of its components and iteratively unifying them with their hypothesized types, generating σ₂ which is used to lift the individual tuple component types and unify their combination with the given type 'a. In the type inference of each tuple component, the environment Γ is updated with the substitution of previously inferred tuple component types to incorporate the substitution of type variables that are used in multiple tuple components. Let us consider the type inference of the tuple (x+x, x) in the environment Γ = {x ↦ ∀.'b}. The first component x+x will be inferred as int. If we inferred the second tuple component x in Γ, the resulting type would be 'b. Through updating Γ, the type of x in Γ will be updated to int and the type of the second tuple component x is inferred as int. If we do not update the environment Γ after inferring the type of x+x, the type of x will be inferred as 'b and the type of (x+x, x) will be inferred as int * 'b.

• infer_expression(Γ, (e₁, e₂), 'a) = Unify{('a, σ₂('b * 'c))} ◦ σ₂ where
The type inference of a constructor applied to an expression is split into four steps: First, we infer the type of the expression. Second, we generate a fresh instance of the type-signature of the constructor. Third, we combine the type of the expression and the assumed type to a function type. Lastly, we unify the fresh instance of the type-signature of the constructor with the function type combination of the type of the expression and the assumed type.

- infer_expression(\(\Gamma, e, 'a\)) = Unify\{\{\text{freshInstance(type-signature(constr))}, \sigma('b \to 'a)\}\} \circ \sigma \quad \text{where}
  
  - 'b is a fresh variable
  
  - \(\sigma = \text{infer_expression}(\Gamma, e, 'b)\)

The type of a fun-abstraction is inferred through inferring the type of its multiple-matching through the type inference function infer_multiple-matching.

- infer_expression(\(\Gamma, \text{fun } m, 'a\)) = infer_multiple-matching(\(\Gamma, m, 'a\))

To infer the type of function application, we firstly infer the type of the function, and then iteratively infer the type of each argument. Through calling AA, we check whether the argument type matches with a function argument type. If it does, we eliminate this type from the function type. The resulting substitution is the composition of all substitutions \(\sigma_n\) generated through unifying each argument type with its corresponding parameter type and lastly, erasing optional arguments where applicable as well as unifying the result type \(\tau\) with the assumed type variable 'a.

- infer_expression(\(\Gamma, e_0 a_1 a_2 \ldots a_n, 'a\)) = Unify\{\{'a, \sigma_n(\tau)\}\} \circ \sigma_n \quad \text{where}
  
  - 'b, 'c are fresh variables
  
  - \(\sigma_1 = \text{infer_expression}(\Gamma, e_1, 'b)\)
  
  - \(\sigma_2 = \text{infer_expression}(\sigma_1(\Gamma), \sigma_1(e_2), 'c) \circ \sigma_1\)

  if has_known arity(\(\tau_0\)), \#_\tau(\(\tau_0\)) = n and unlabeled(\(a_1 a_2 \ldots a_n\)) then (special case)

  * for \(1 \leq i \leq n\)
    
    - \(\sigma'_i = \text{infer_argument}(\sigma_{i-1}(\Gamma), \sigma_{i-1}(a_i), 'c_i) \circ \sigma_{i-1}\)
    
    - \(\sigma_i = \text{AA}'(\sigma'_i(\tau_{i-1}), \sigma'_i('c_i)) \circ \sigma'_i\)
\[ \tau_i = R'(\sigma_i(\tau_{i-1}), \sigma_i(c_i)) \]

* \( \tau = \text{erase}'(\sigma_n(\tau_n), \sigma_n(\tau_0)) \)

- else (default)

* for \( 1 \leq i \leq n \)

\[ \sigma'_i = \text{infer_argument}(\sigma_{i-1}(\Gamma), \sigma_{i-1}(a_i), c_i) \circ \sigma_{i-1} \]

\[ \sigma_i = AA(\sigma'_i(\tau_{i-1}), \sigma'_i(c_i), c_i) \circ \sigma'_i \]

\[ \tau_i = R(\sigma_i(\tau_{i-1}), \sigma_i(c_i)) \]

* \( \tau = \text{erase}(\sigma_n(\tau_n), \sigma_n(\tau_0)) \)

Let us consider the following function application type inference

\[
\text{infer_expression}([], (\text{fun } \sim x \rightarrow \text{fun } ?y \rightarrow x) \, 23, 'a)\
\]

1. Generate the fresh variables 'b\(_0\), 'c\(_1\), 'd\(_1\)

2. \( \sigma_0 = \text{infer_expression}([], (\text{fun } \sim x \rightarrow \text{fun } ?y \rightarrow x), 'b_0) = \{ 'b_0 \mapsto (x : 'b \rightarrow ?y : 'c \rightarrow 'b) \} \)

3. \( \tau_0 = (x : 'b \rightarrow ?y : 'c \rightarrow 'b) \)

4. has_known_arity\( (x : 'b \rightarrow ?y : 'c \rightarrow 'b) = \text{false} \)

5. default:

(a) \( \sigma'_1 = \text{infer_argument}([], 23, 'c_1) \circ \sigma_0 = \{ 'c_1 \mapsto \text{int}, 'b_0 \mapsto (x : 'b \rightarrow ?y : 'c \rightarrow 'b) \} \)

(b) \( \sigma_1 = AA(x : 'b \rightarrow ?y : 'c \rightarrow 'b, \text{int, 'd_1}) \circ \sigma'_1 = \{ 'b \mapsto (\text{int} \rightarrow 'd_1) \}
\]

\[
\circ \{ 'c_1 \mapsto \text{int}, 'b_0 \mapsto (x : 'b \rightarrow ?y : 'c \rightarrow 'b) \}
\]

\[
= \{ 'b \mapsto (\text{int} \rightarrow 'd_1), 'c_1 \mapsto \text{int}, 'b_0 \mapsto (x : (\text{int} \rightarrow 'd_1) \rightarrow ?y : 'c \rightarrow (\text{int} \rightarrow 'd_1)) \}
\]

(c) \( \tau_1 = R(\sigma_1(\tau_0), \sigma_1('c_1)) = R(\sigma_1(x : (\text{int} \rightarrow 'd_1) \rightarrow ?y : 'c \rightarrow (\text{int} \rightarrow 'd_1)), \text{int})
\]

\[
= x : (\text{int} \rightarrow 'd_1) \rightarrow ?y : 'c \rightarrow 'd_1
\]

(d) \( \tau = (\text{erase}(x : (\text{int} \rightarrow 'd_1) \rightarrow ?y : 'c \rightarrow 'd_1),
\]

\[
(x : (\text{int} \rightarrow 'd_1) \rightarrow ?y : 'c \rightarrow (\text{int} \rightarrow 'd_1))
\]

\[
= (x : (\text{int} \rightarrow 'd_1) \rightarrow 'd_1)
\]

6. Unify\( \{ (\sigma_1(\tau), \sigma_1('a)) \circ \sigma_1 = \{ 'a \mapsto (x : (\text{int} \rightarrow 'd_1) \rightarrow 'd_1) \} \circ \sigma_1 \)

\[
= \{ 'a \mapsto (x : (\text{int} \rightarrow 'd_1) \rightarrow 'd_1), 'b \mapsto (\text{int} \rightarrow 'd_1), 'c_1 \mapsto \text{int},
\]

\[
'b_0 \mapsto (x : (\text{int} \rightarrow 'd_1) \rightarrow ?y : 'c \rightarrow (\text{int} \rightarrow 'd_1)) \}
\]

32
Because x's type is 'b and there is no suitable parameter in \( \text{fun}\ x\rightarrow\text{fun}\ ?y\rightarrow x \) for 23 to match with, the return variable x's type 'b is expanded to int -> 'd_1 and the application is type inferred as (x:(int -> 'd_1) -> 'd_1). If the applied function does not have a variable return type, as in (fun ~x -> fun ?y -> x + 2), the special-case is applied.

\[
\text{infer}\ expression(\{\}, (\text{fun}\ ~x\rightarrow\text{fun}\ ?y\rightarrow x + 2)\ 23, 'a) \]

1. Generate the fresh variables 'b_0, 'c_1, 'd_1

2. \( \sigma_0 = \text{infer}\ expression(\{\}, (\text{fun}\ ~x\rightarrow\text{fun}\ ?y\rightarrow x + 2), 'b_0) = \{ 'b_0 \mapsto (x: \text{int} -> ?y: 'c \rightarrow \text{int}) \} \)

3. \( \tau_0 = x: \text{int} \rightarrow ?y: 'c \rightarrow \text{int} \)

4. \( \text{has\_known\_arity}(x: \text{int} \rightarrow ?y: 'c \rightarrow \text{int}) = \text{true} \)

5. \( \#\sim(x: \text{int} \rightarrow ?y: 'c \rightarrow \text{int}) = n = 1 \)

6. unlabeled(23)=true

7. special-case:
   
   (a) \( \sigma'_1 = \{ 'c_1 \mapsto \text{int}, 'b_0 \mapsto (x: \text{int} \rightarrow ?y: 'c \rightarrow \text{int}) \} \)
   
   (b) \( \sigma_1 = \text{AA}(x: \text{int} \rightarrow ?y: 'c \rightarrow \text{int}, \text{int}) \circ \sigma'_1 \)
       = \{ \} \circ \{ 'c_1 \mapsto \text{int}, 'b_0 \mapsto (x: \text{int} \rightarrow ?y: 'c \rightarrow \text{int}) \}
       = \{ 'c_1 \mapsto \text{int}, 'b_0 \mapsto (x: \text{int} \rightarrow ?y: 'c \rightarrow \text{int}) \}

   (c) \( \tau_1 = R'((\sigma_1(\tau_0), \sigma_1('c_1)) = R'(\sigma'_1(x: \text{int} \rightarrow ?y: 'c \rightarrow \text{int}), \text{int}) = ?y: 'c \rightarrow \text{int} \)

   (d) \( \tau = \text{erase}'((?y: 'c \rightarrow \text{int}), (x: \text{int} \rightarrow ?y: 'c \rightarrow \text{int})) = (?y: 'c \rightarrow \text{int}) \)

8. Unify((\sigma_1(\tau), \sigma_1('a))) \circ \sigma_1 = \{ 'a \mapsto (?y: 'c \rightarrow \text{int}) \} \circ \sigma_1 = \{ 'a \mapsto (?y: 'c \rightarrow \text{int}), 'c_1 \mapsto \text{int}, 'b_0 \mapsto (x: \text{int} \rightarrow ?y: 'c \rightarrow \text{int}) \}

The application (fun ~x -> fun ?y -> x + 2) 23 is type inferred as ?y: 'c -> int. If ~x does not type with int as in (fun ~x -> fun ?y -> x=true) 23, OCaml will reject the application.

\[
\text{infer}\ expression(\{\}, (\text{fun}\ ~x\rightarrow\text{fun}\ ?y\rightarrow x+2)\ 23, 'a) \]

1. Generate the fresh variables 'b_0, 'c_1, 'd_1

33
2. $\sigma_0 = \text{infer\_expression}(\{\}, (\text{fun } x \rightarrow \text{fun } y \rightarrow x=true), b_0) = \{b_0 \mapsto (x : \text{bool} \rightarrow y : c \rightarrow \text{bool})\}$

3. $\tau_0 = x : \text{bool} \rightarrow y : c \rightarrow \text{bool}$

4. $\text{has\_known\_arity}(x : \text{bool} \rightarrow y : c \rightarrow \text{bool}) = \text{true}$

5. $\#_\to(x : \text{bool} \rightarrow y : c \rightarrow \text{bool}) = n = 1$

6. $\text{unlabeled}(23) = \text{true}$

7. special-case:

   (a) $\sigma'_1 = \{c_1 \mapsto \text{int}, b_0 \mapsto (x : \text{bool} \rightarrow y : c \rightarrow \text{bool})\}$

   (b) $\sigma_1 = \text{AA}'(x : \text{bool} \rightarrow y : c \rightarrow \text{bool}, \text{int}) \circ \sigma'_1 = \text{failure}$

Given an expression $e$ of our language, a typing environment $\Gamma$, a substitution $\sigma$ for type variables in $e$, and a type variable $'a$ not occurring in $e$ or $\Gamma$, the type inferencer allows us, to infer a substitution $\sigma'$, such that $\Gamma, \sigma \vdash e : \sigma'('a)$. Moreover, if $\Gamma, \sigma \vdash e : \tau$, for any type $\tau$, then there exists an additional substitution $\sigma''$ such that $\tau = \sigma'' \circ \sigma('a)$. The algorithm given agrees with the observed behavior of OCaml.
Chapter 5

Type system (continued)

5.1 Function application special-case

Recall, "as a special case, if the function in a function application has known arity, all the arguments are unlabeled, and their number matches the number of non-optional parameters, then labels are ignored and non-optional parameters are matched in their definition order. Optional arguments are defaulted."[4] We assume that the type inference function correctly infers the most general type for an expression e. Given that the most general type \( \tau'_0 \) of the function has known arity and \( n \) non-optional arguments, any type \( \tau_0 \) that the function expressed through e types with will also have known arity and \( n \) non-optional arguments. Through the use of \( A' \) and \( R' \), we iterate through \( \tau_0 \) and the sequence of arguments. For each argument \( a_i \), we check that its type \( \tau'_i \) occurs in a non-optional argument position of the function type. If it does, we remove it from the function type through \( R' \). The resulting type \( \tau_i \) can be thought of as the type of the function after applying it to \( a_i \). The type \( \tau_n \) is the preliminary type after applying the last argument \( a_n \).

After applying the last argument, OCaml erases any optional parameters that occurred to the left of an applied non-optional parameter.

\[
\text{infer_expression}(\sigma(\Gamma), \sigma(e), 'a) = \sigma' \quad \tau'_0 = \sigma('a) \quad \text{has_known_arity}(\tau'_0) \quad \#\_\gamma(\tau'_0) = n
\]

where 'a is a fresh type variable not occurring in \( \Gamma \)

\[
\frac{\Gamma, \sigma \vdash e : \tau_0 \quad \Gamma, \sigma \vdash a_i : \tau'_i \quad A'(\tau_{i-1}, \tau'_i) \quad \tau_i = R'(\tau_{i-1}, \tau'_i) \quad \text{for } i = 1, \ldots, n}{\Gamma, \sigma \vdash (e \ a_1 \ldots a_n) : \text{erase}'(\tau_n, \tau_0)}
\]

Figure 5.1: Type rule for function application-special case

Let us consider the example of a function application \((\text{fun} ~ x \rightarrow \text{fun} ~ y \rightarrow x) ~ 24.5 ~ \text{false}\) in the environment \(\{\}\) and the substitution \(\{\}\).

1. \(\text{infer_expression}((\text{fun} ~ x \rightarrow \text{fun} ~ y \rightarrow x) ~ 24.5 ~ \text{false}, 'a) = \{ 'a \mapsto x : 'a \rightarrow y : 'b \rightarrow 'a \}\)

2. \(\tau'_0 = x : 'a \rightarrow y : 'b \rightarrow 'a\)

3. \(\text{has_known_arity}(x : 'a \rightarrow y : 'b \rightarrow 'a) = false\)
The application cannot be successfully typed with the special case rule. Consider the typing of
\((\text{fun } \sim x \to \text{fun } \sim y \to x + y) \ 1 \ 2\) in the environment \(\{\}\) and the substitution \(\{\}\):

1. \(\text{infer_expression}(\{\}, (\text{fun } \sim x \to \text{fun } \sim y \to x + y), 'a) = \{ 'a \mapsto x: \text{int} \to y: \text{int} \to \text{int}\}\)

2. \(\tau_0 = x: \text{int} \to y: \text{int} \to \text{int}\)

3. \(\text{has_known arity}(x: \text{int} \to y: \text{int} \to \text{int}) = \text{true}\)

4. \(#\sim\sim(x: \text{int} \to y: \text{int} \to \text{int}) = n = 2\)

5. \(\{\}, \{\} \vdash (\text{fun } \sim x \to \text{fun } \sim y \to x + y):(x: \text{int} \to y: \text{int} \to \text{int})\)

6. \(\{\}, \{\} \vdash 1:\text{int}\)

7. \(A'(x: \text{int} \to y: \text{int} \to \text{int}, \text{int}) = \text{true}\)

8. \(\tau_1 = R'(x: \text{int} \to y: \text{int} \to \text{int}, \text{int}) = y: \text{int} \to \text{int}\)

9. \(\{\}, \{\} \vdash 2:\text{int}\)

10. \(A'(y: \text{int} \to \text{int}, \text{int}) = \text{false}\)

11. \(\tau_2 = R'(y: \text{int} \to \text{int}, \text{int}) = \text{int}\)

12. \(\text{erase}'(\text{int}, x: \text{int} \to y: \text{int} \to \text{int}) = \text{int}\)

The program \((\text{fun } \sim x \to \text{fun } \sim y \to x + y) \sim x:1 \ 2\) does not type because \(\sim x:1\) is not an unlabeled argument. \(A'(x: \text{int} \to y: \text{int} \to \text{int}, x: \text{int}) = \text{false}\). The program \(((\text{fun } \sim x \to \text{fun } \sim y \to x + y) \sim x:1\) 2 does type because in the first application, the default function application type rule applies while for the second application the special case rule applies. Both the programs \((\text{fun } \sim x \to \text{fun } \sim y \to x + y) \sim x:1\) and \((\text{fun } \sim x \to \text{fun } \sim y \to x + y) \ 1 \ 2 \ 3\) do not type because the number of arguments is by one smaller and by one larger than the number of parameters of the function, respectively.

\(\#\sim\sim(x: \text{int} \to y: \text{int} \to \text{int}) = 2 \neq 1\) and \(\#\sim\sim(x: \text{int} \to y: \text{int} \to \text{int}) = 2 \neq 3\).
There exists a number of different interesting typing examples that demonstrate the subtleties of OCaml concerning currying and type expressions. As has already been introduced, OCaml does not support currying with optional parameters. This can be demonstrated by the following example. We consider a function with an optionally labeled parameter \(?x\) and an unlabeled parameter \(y\) that returns the optionally labeled parameter. In the first case, we apply this function to 1 and to \(?x: (\text{Some 2})\). The type of the application is \(\text{int option}\) since \(x\) evaluates to \(\text{int option}\). In the latter case, we apply the function firstly to 1, and then to \(?x: (\text{Some 2})\). If OCaml supported currying, it would type the program, but since the type inferencer erases \(?x\) after the first application, the type of the first application is no longer a function type but \('a option\). As a result, an error is thrown when \(?x: (\text{Some 2})\) is applied.

```ocaml
fun ?x y -> x;;
- : ?x: 'a -> 'b -> 'a option = <fun>
(fun ?x y -> x) 1 ?x: (Some 2);;
- : int option = Some 2
((fun ?x y -> x) 1) ?x: (Some 2);;
Error: This expression has type 'a option
This is not a function; it cannot be applied.
(fun ?x y -> x) 1;;
- : 'a option = None
```

The second example is more subtle. OCaml types programs where parameters are annotated with types that are more general than their actual type. For example, \(x\) is given the type annotation \('a\), but actually is inferred to have type \(\text{int}\).

```ocaml
fun (x: 'a) -> x+1;;
x: int -> int
```

Interestingly, the Standard ML equivalent does not type.

```ml
fn x: 'a => x+1
```

Unexpected exception (bug?) in SML
Chapter 7

Conclusion

The objective of this thesis was to give a type system and type inference algorithm for a small subset of the OCaml grammar containing all label constructs supported by OCaml. The function application special case could not be given by inductive rule definition without utilizing the type inference function in the type system definition, which is unusual in the definition of type systems. It has been shown that OCaml’s function application does not support currying when optional parameters and arguments are used, and there exist subtle differences in the strictness of checking type expressions. The type system and the type inferencer agree with the observed behavior of OCaml and differentiate itself from existing work through being constructive rather than purely declarative and supporting more complex OCaml constructs. The type inference algorithm may serve as a theoretical foundation for implementation of an OCaml type inferencer for labels and function application. Projects like OCaml-K may utilize it to specify OCaml in K.
References


