Binding-Time Analysis : Subtyping versus Subeffecting

Guangyu Zhang

June 2008

Contents

Abstract

Binding time analysis is an important program analysis mainly used in partial evaluation. It determines which part in a program is static, that is its value is computable at compile time and which part is dynamic. Different techniques have been described for implementing the analysis. In this thesis, I will specify and implement different variants of binding time analysis. Among these, the focus will be on the comparison of subtyping and subeffecting as extensions to a polyvariant analysis on a polymorphic language.

Chapter 1

Introduction

Static program analysis predicts safe and computable approximations to the set of variable values or behaviors of a program; this may be used to prove program correctness or in program optimization. The expressive power of an analysis is demonstrated by the accuracy of the predicted information. With different specifications, performing the same analysis may result in different expressive power. Typically, more expressive power demands a more complex implementation and higher resource consumption. Resource consumption mainly concerns two aspects : time consumption of performing the analysis and memory consumption. It will be helpful to discover the relations between the expressive power and resource consumption of an analysis when we want to develop a new program analysis. My research focus on an investigation of different implementation skills and the comparison of expressiveness and cost for binding-time analysis. Binding-time analysis is a typical type-based analysis and it is our hope that what we discover applies to other similar analysis as well.

Some work have been done in the area of binding-time analysis. In [4], Heldal and Hughes developed a polyvariant analysis for polymorphic language. Dussart, Henglein and Mossin specified a polyvariant analysis with subtyping for monomorphic language in [3]. Glynn, Stuckey, Sulzmann and Sondergaard designed a polyvariant analysis with subtyping for polymorphic language in [5]. The state of the art is to combine polyvariance with subtyping. However such a binding-time analysis is difficult to implement and has high resource consumption. Instead of pursuing more complicated techniques, we simplify it to see if we could gain similar expressive power while reducing the resource consumption. In this thesis, we focus on comparing subtyping and subeffecting [9] with their expressive power and resource consumption.

The thesis is organized into the following chapters:

- We introduce several general concepts that will be used in later chapters (Chapter 2).
- We define binding-time analysis and specify a monovariant analysis, referred as MMX, for a monomophic language. Upon this specification, we develop an inference algorithm similar to algorithm $\mathcal{W}$(Chapter 3).
- We extend the analysis specified in previous chapter with polyvariance referred as PPX, and extend the target language to a polymorphic language (Chapter 4).
- Based on the analysis in Chapter 4, we add the subsumption feature in forms of subeffecting and subtyping, referred as PPE and PPT (Chapter 5).

- We compare two pairs of specifications of binding time analysis. We explain in which way these analysis differ from each other in terms of expressive power and resource consumption (Chapter 6).

Chapter 2

Context/Preliminaries

2.1 Program Analysis

Program analysis is the task of deriving information from programs. It uses static compile-time techniques to predict values and behaviors of a program at runtime. One application of program analysis is program optimization, e.g., we use strictness analysis[11] in lazy evaluation, we use shape analysis[1] to avoid unnecessary garbage collection and we also use binding-time analysis in partial evaluation to decrease the computation of expressions at runtime. Another application of program analysis is program correctness which uses analysis information to verify functionalities of programs.

One common nature of all approaches of program analysis is that they only give a approximate analysis result. Since program analysis predicts dynamic behaviors for all executions of a program, the results cannot be optimal. As an example, we design an analysis that finds possible values of variables and consider a simple program[10]:

read (x); if x>0 then y:=1 else (y:=2; S); z := y

where S are some statements that do not contain assignment to y. Naturally, we predict that the possible values of y that can reach z := y are 1 and 2. However, if we know statements S will never terminate, the value of y will only be 1 which is a more precise analysis result than y being both 1 and 2. Unfortunately, this is undecidable because of the halting problem[12], that is, there does not exist a general procedure to decide correctly for all programs P and all data whether P(data) terminates. So in general, we expect our analysis to give a possible larger set of results than what will happen precisely during one execution of a program. In this example, we accept the analysis result of 1 and 2 rather than the result of 1 because {1,2} is the correct result for all executions of the program.

2.2 Type System

A type system defines how a programming language classifies values and expressions into types, how it can manipulate those types and how they interact.

2.2.1 Types

A type identifies a value or set of values as having a particular meaning or purpose. Considering a lambda calculus, we can have types defined as:

$$
\begin{array}{lll}
\tau \in \textbf{Type} & & \\
\quad ::= & \beta & \text{type variable} \\
\quad \mid & T & \text{type constant} \\
\quad \mid & \tau_1 \rightarrow \tau_2 & \text{composite type}
\end{array}
$$

Type variables represent arbitrary types to support polymorphic types[7]. They are also used in a type inference algorithm to denote an unknown type. Type constants usually include an integer type and a boolean type, referred to Int and Bool respectively. In this type definition, we only have one kind of composite type, the function type $\tau_1 \rightarrow \tau_2$ which denotes a function with arguments of type τ_1 and results of type τ_2.

2.2.2 Type inference

Type inference, or implicit typing, refers to the technique that deducts types of expressions automatically for a programming language. A type inference system usually includes two parts: a specification of type rules that can be used to construct a proof for the correctness of assigning a type to an expression and an inference algorithms whose inferred types conform to those deduction rules. There are various type rules and inference algorithm, but for modern functional languages such as Haskell and ML, their type systems are based on Hindley-Milner's type discipline.

Hindley-Milner rules are constructed by type judgments which have the form of:

$$\Gamma \vdash e : \tau$$

which is read as expression e has type τ under type environment Γ. Here, Γ is a type assumption or type environment. It provides type information for the free variables in expression e. Using the type definitions from the previous section, we have type rules such as:

[True] $\quad \emptyset \vdash \mathit{true} : \mathit{Bool}$

[App] $\quad \dfrac{\Gamma \vdash e_1 : \tau_2 \mapsto \tau_0 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_0}$

The type rule [True] states that the constant value *true* has the type *Bool*. For function application, we could infer the type of the application if the type of the second expression matches the argument type of the first expression.

One typical feature of Hindley-Milner type discipline is polymorphism: a term is assigned a polymorphic type in general; only when the types of its arguments and result can be uniquely determined from the context is it monomorphic[7]. Algorithm $\mathcal{W}$, developed by Damas and Milner, is used to infer the principle type scheme[2] of a term for a polymorphic type system.

2.3 Type-based Program Analysis

A type-based analysis is a program analysis which makes use of its type system to express the analysis properties of interest[10]. The idea is to annotate the types with these properties. Let us take totality analysis as an example. Totality analysis determines whether a function is total, i.e. that the application of the function to any argument results in a terminating computation[6]. Suppose we have a type system: $\tau ::= \mathit{Int} \mid \tau \rightarrow \tau$ and

we define the annotation of totality analysis as: $\varphi = \mathbf{T} \mid \mathbf{N} \mid \alpha$ where $\mathbf{T}$ and $\mathbf{N}$ represent for total and not known to be total respectively. We have the annotated types : $\hat{\tau} ::= Int \mid (\hat{\tau} \to \hat{\tau})^{\varphi}$ and a function with type $(Int \to Int)^{T}$ denotes that it is a total function that has underlying type $Int \to Int$.

2.4 Complexity of Type-based Program Analysis

A type-based analysis is specified by allowing annotations on top of the type inference system of the programming language. The annotated types are used to express both the analysis information and type information; the original type is often called the underlying type. Several techniques can be applied to specify type-based program analyses, which usually result in different complexity of the analyses. We can distinguish such techniques in the following three aspects.

Monomorphic and Polymorphic. Type system is a syntax method to make sure that funtions or precedures are only applied on a correct set of values so that we could prevent certain kind of erroneous behaviors such as applying addition operator on two strings. However, the flexibility is lost in a certain degree. For example, a function like identity function: $\lambda x.x$ is only allowed to be applied on one type of values and we are forced to define different identity functions if we want to apply it on integers, booleans and etc. In order to increase the flexibility and meanwhile to remain the correctness we gain from a type system, polymorphism are developed.

A polymorphic type system introduces type variables to express arbitrary types. In general, a inference algorithm tries to infer the most general type, also known as principle type, for a term. For example in Haskell, an identity(id) function usually has a polymorphic type: $\forall a.a \to a$. The quantified type variable a could be then instantiated to any type as required. Suppose we have an expression (id id) 1, the left most id will have type: $(Int \to Int) \to (Int \to Int)$ where a is instantiated to $Int \to Int$; the second id will have type: $Int \to Int$ where a is instantiated to Int.

A polymorphic analysis allows polymorphic types in the underlying type system.

Monovariant and Polyvariant. Similarly, polyvariant analysis allows polymorphism in annotations whereas monovariant analysis does not. A polyvariant annotation system gives an analysis the ability to vary the analysis properties of an expression depending on the context in which the expression is used.

Subsumption. Subsumption refers to certain kinds of type inference rules in a type system. For a subsumption rule in an annotated type system:

$$[\text{sub}] \quad \frac{\Gamma \vdash t : \hat{\tau}_1}{\Gamma \vdash t : \hat{\tau}_2} \qquad \text{if } \hat{\tau}_1 \leq \hat{\tau}_2$$

where $\leq$ is a partial ordering defined on types, it generally says that if a term has type $\hat{\tau}_1$ and $\hat{\tau}_1$ is more precise than $\hat{\tau}_2$, then the term can be said to be of type $\hat{\tau}_2$ safely. There are various ways of defining the partial ordering on annotated types. Two of these are subtyping and subeffecting. They differ on how to define the partial ordering for compound type constructors like function types. But firstly, we give the definition of a partial ordering for simple types, which is shared by both subtyping and subeffecting:

$$\frac{\varphi \sqsubseteq \varphi'}{\tau^{\varphi} \leq \tau^{\varphi'}}$$

where the definition of $\sqsubseteq$, partial ordering of annotations, depends on the analysis and τ represents any kinds of basic types supported by the underlying type system. Then in subtyping, the ordering of compound types is defined as:

$$\frac{\hat{\tau}_1' \leq \hat{\tau}_1 \quad \varphi \sqsubseteq \varphi' \quad \hat{\tau}_2 \leq \hat{\tau}_2'}{\hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2 \leq \hat{\tau}_1' \xrightarrow{\varphi'} \hat{\tau}_2'}$$

In this definition, function types have the same ordering as their result types but the reversed ordering as their argument types. we say that the arrow or the function type constructor which takes an argument type and a result type and produce a function type is covariant with result types because it preserves the ordering of result types and is contravariant with argument types because it reverses the ordering of argument types.

The reason that we define the ordering for function type this way is that a function produces a value of its result type but consumes a value of its argument type. Let us consider a type system which contains *Real* for real number, Int for integers, Nat for natural number and function types. The ordering of types is defined as subset relation which roughtly represents the meaning "is more specific than". For example, we have $Nat \leq Int$ and $Int \leq Real$ because a natural number is more specific than an integer and an integer is more specific than a real number. For a function type $Int \to Int$, it denotes that this function produces a integer value and we can also say it produces a real number because it is always safe to generalise what is produced. On the other hand, it is also safe to make it more specific of what is consumed, so it is possible to give a natural number as the arugment of this function. If we have a type system which support user defined data type, dealing with covariance and contravariance is relatively complicate because these types may contain the function-space constructor, and, hense can be used to construct new contravariant and covariant constructors. So a simplified form of subsumption, subeffecting is introduced. Subeffecting only makes use of top level annotations to determine the ordering of the annotated types:

$$\frac{\hat{\tau}_1 = \hat{\tau}_1' \quad \varphi \sqsubseteq \varphi' \quad \hat{\tau}_2 = \hat{\tau}_2'}{\hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2 \leq \hat{\tau}_1' \xrightarrow{\varphi'} \hat{\tau}_2'}$$

By combining different technologies in these three aspects, we have twelve methods of implementing a type-based analysis. It is not necessary to explore all these methods because our focus is on comparing subtyping and subeffecting. Four of them will be implemented in later chapters. First, we give a prototype for binding-time analysis by using the MMX method, where the first and second M represent monomophic and monovariant respectively and the X stands for "no subsumption". Although the analysis is the least powerful one, it builds the framework of the analysis and can serve to illustrate binding time analysis. Then we extend this prototype to PPT which stands for polyvariant analysis with polymorphic underlying type system and subtyping subsumption. PPT is expected to give the most precise analysis, but also be the most complex one, both in terms of implementation and execution. At last we simplify PPT a little to PPE with subeffecting subsumption and compare the two to discover the differences in terms of expressive power and cost.

2.5 Partial Evaluator

Partial evaluator is an automatic tool that transforms a program into a residual program which have the same dynamic behavior as the original one but hopefully with better performance. For example, we have a haskell program:

$$power\ m\ n = if\ n == 0\ then\ 1\ else\ m * power\ m\ (n-1)$$
$$square\ m = power\ m\ 2$$

We except a typical partial evaluator generates a residual program like:

$$power\ m\ n = if\ n == 0\ then\ 1\ else\ m * power\ m\ (n-1)$$
$$square\ m = m * m$$

In order to do that, a partial evaluator needs to know which part of the program can be evaluated at compile time, which motivates the developing of binding-time analysis. The partial evaluator uses the results of binding-time analysis to compute values for static terms and generate code for dynamic terms.

Chapter 3

Binding-time Analysis

In this chapter, we first give a formal definition of binding-time analysis(Section 3.1) and the target language on which our analysis applied(Section 3.2). In Section 3.3 we specify binding-time analysis in the form of annotated type inference system. Then we explain our implementation algorithm of the specification in detail(Section 3.4), and at last we define the metrics that describe the expressive power and resource consumption of the analysis.

3.1 Binding-Time Analysis

Binding-Time Analysis(BTA) is a kind of type-based program analysis that determines what expressions of a program can be safely evaluated at specialization-time(compile-time) and those cannot. Values that are known at specialization-time are called *Static*. A *Dynamic* binding time indicates that we defer the computation of the value to the place of runtime.

We simply use **S** to refer to static binding time and **D** for dynamic binding time. Consider a program:

$$\textbf{let } x = 1 \textbf{ in } x + y$$

where x is a let-binding variable that has value of 1 and y is a free variable in this program. The binding-time analysis for this program produces the typing for x and y as : $x :: Nat^{\mathbf{S}}$ and $y :: Nat^{\mathbf{D}}$. Here, x is a static natural number because its value is known to be 1 at specialization-time while y is a dynamic value as it is free in the program. In type-based analysis, annotations such as **S** and **D** are called *effect*[9], and we shall call them binding-time effects or binding-time annotations in our context of binding-time analysis.

Function types are also annotated with binding-time effects. For instance, we have function *succ*

$$succ = \lambda x \ . \ x + 1$$

and its type

$$succ :: Nat^{\mathbf{D}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{D}} \ .$$

The annotation **D** in the argument type of function *succ* indicates that the function does not require its argument value to be known at specialization-time. The annotation **D** in the result type indicates that *succ* produces a natural number that may not be evaluated at specialization-time. The annotation **S** on the function arrow shows that the function itself can be evaluated at specialization-time.

3.2 Syntax of Target Language

First, we define the syntax of the language that we will apply our analysis on. It is a simple functional language with basic control structure and lambda abstraction:

$p \in \mathbf{Prog}$	$::= t$		
$t \in \mathbf{Term}$			terms:
	$::=$	n^l	numeral
	$\mid$	b^l	boolean
	$\mid$	x^l	identifier
	$\mid$	$(\mathbf{fun}\ x => t)^l$	abstraction
	$\mid$	$(\mathbf{fun\ rec}\ x => t)^l$	recursive function
	$\mid$	$(t\ t)^l$	application
	$\mid$	$(\mathbf{let\ val}\ x = t\ \mathbf{in}\ t\ \mathbf{end}\ )^l$	local definition
	$\mid$	$(\mathbf{let\ dyn\ val}\ x :: tp\ \mathbf{in}\ t\ \mathbf{end})^l$	dynamic value
	$\mid$	$(t\ op\ t)^l$	binary operation
	$\mid$	$(\mathbf{if}\ t\ \mathbf{then}\ t\ \mathbf{else}\ t)^l$	branch.

In our language, each program is represented by an element of the set **Prog** which simply contains a term that defines the real content of the program. Adding this top level wrapper for a term enable us to perform program specific behaviors of program analysis. We explain this in more details in later sections. In the definition of **Term**, n , x and b range over natural numbers, identifiers and boolean respectively:

n	$\in \mathbb{N}$	natural numbers
x	$\in \mathbf{Ident}$	identifiers
b	$\in \mathbf{Bool}$	boolean
	$::= true$	
	$\mid\ false$	

Furthermore, l ranges over labels, which are used to uniquely identify subterms:

$$l \in \mathbf{Lab} \quad \text{labels.}$$

Finally, *op* ranges over binary operations, containing both arithmetic and relation operations:

$$op \in \mathbf{Op} \quad \text{binary operations.}$$

The language is in great degree an extension of a typed lambda calculus except that we have defined a clause of terms to introduce dynamic value explicitly. For a pure functional language, such as a lambda calculus or many of its extensions, all of the closed expressions of a program can be evaluated at specialization-time. Without side effect, there is no dynamic factors that influence the result of executing the program, thus the values of all expressions of a program can be determined before actually running it. Then a binding-time analysiser will always annotate all the expressions with a static binding time for a program written in such a language, which makes the analysis meaningless. In order to avoid this situation, we allow defining dynamic variables whose values can only

be known at runtime. In a term **let dyn val** $x :: tp$ **in** t **end**, tp denote the type of x, type is defined as

$$tp \in \mathbf{Type} \ ::= \ Nat \mid Bool \mid tp \rightarrow tp.$$

3.3 Specification of BTA-MMX

For a monovariant binding-time analysis with a monomorphic underlying type system and no separate rules for subsumption, we use the following annotations to express the analysis:

$\varphi \in \mathbf{Ann}$		annotations:
::=	$\mathbf{S}$	static binding time
|	$\mathbf{D}$	dynamic binding time.

It is easy to define an ordering $\sqsubseteq$: $\mathbf{Ann} \times \mathbf{Ann}$ for the set of annotation:

$$\varphi \sqsubseteq \varphi \qquad\qquad \mathbf{S} \sqsubseteq \mathbf{D}$$

and we shall write $\varphi_2 \sqsupseteq \varphi_1$ for $\varphi_1 \sqsubseteq \varphi_2$. A term being static is considered to be a stronger condition because a static term could be also treated as a dynamic term whereas a dynamic term can never be used as static one safely without further information. So we sometimes use the word "stronger" or "smaller" for the relation $\sqsubseteq$ and "weaker" or "bigger" for $\sqsupseteq$. Combining the annotation with the underlying type system, we have annotated types defined as:

$\hat{\tau} \in \mathbf{BTType}$		annotated types:
::=	Nat^{φ}	natural numbers
|	$Bool^{\varphi}$	booleans
|	$\hat{\tau} \xrightarrow{\varphi} \hat{\tau}$	functions

Then we can use notations $|\hat{\tau}|$ to denote the top level annotation of a binding time type:

$$\begin{aligned} |Nat^{\varphi}| &= \varphi \\ |Bool^{\varphi}| &= \varphi \\ |\hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2| &= \varphi \end{aligned}$$

So the result of binding-time analysis are summarized as a function mapping each label that appears in the program to an annotation:

$$\begin{gathered} \hat{\mathtt{B}} : \mathbf{Lab}_{\star} \rightarrow \mathbf{Ann} \\ \hat{\mathtt{B}}(l) = |\hat{\tau}_l| \end{gathered}$$

where $\hat{\tau}_l$ denotes the annotated type that assigned or assignable to the subterm with label l. Binding-time analysis is motivated by partial evaluation and this suggests imposing a well-formedness condition on types so that types like $Nat^{\mathbf{S}} \xrightarrow{\mathbf{D}} Nat^{\mathbf{S}}$ are regarded as being meaningless because such a type denotes a function whose definition itself is unknown but produces known results. We use $wft(\hat{\tau})$ to denote the well-formedness which is defined as:

- $wft(Nat^{\varphi})$ and $wft(Bool^{\varphi})$: a type constant is always well-formed.
- For function type $wft(\hat{\tau}_1 \xrightarrow{\alpha} \hat{\tau}_2)$ if $\alpha \sqsubseteq |\hat{\tau}_1|, \alpha \sqsubseteq |\hat{\tau}_2|$ and $wft(\hat{\tau}_1)$ and $wft(\hat{\tau}_2)$.

Now, the deduction rules for binding-time analysis can be described in a list of judgments which have the form

$$\hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ t : \hat{\tau}$$

where $\hat{\Gamma}$ is an annotated type environment that maps identifiers to annotated types:

$\hat{\Gamma} \in \mathbf{TEnv}$	annotated type environments
$\hat{\Gamma} ::=$	$[\,] \mid \hat{\Gamma}[x \mapsto \hat{\tau}]$

[num] $\quad \hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ n : Nat^{\varphi}$

[bool] $\quad \hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ b : Bool^{\varphi}$

[id] $\quad \dfrac{\hat{\Gamma}(x) = \hat{\tau}}{\hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ x : \hat{\tau}}$

[fun] $\quad \dfrac{\hat{\Gamma}[x \mapsto \hat{\tau}_x] \ \vdash^{t}_{\mathtt{BTA}} \ t_0 : \hat{\tau}_0 \qquad \varphi \sqsubseteq |\hat{\tau}_0| \qquad \varphi \sqsubseteq |\hat{\tau}_x|}{\hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ \mathbf{fun}\ x \Rightarrow t_0 : \hat{\tau}_x \xrightarrow{\varphi} \hat{\tau}_0}$

[rec] $\quad \dfrac{\hat{\Gamma}[self \mapsto \hat{\tau}_x \xrightarrow{\varphi} \hat{\tau}_0][x \mapsto \hat{\tau}_x] \ \vdash^{t}_{\mathtt{BTA}} \ t_0 : \hat{\tau}_0 \qquad \varphi \sqsubseteq \hat{\tau}_0 \qquad \varphi \sqsubseteq \hat{\tau}_x}{\hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ \mathbf{fun\ rec}\ x \Rightarrow t_0 : \hat{\tau}_x \xrightarrow{\varphi} \hat{\tau}_0}$

[app] $\quad \dfrac{\hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ t_1 : \hat{\tau}_2 \xrightarrow{\varphi} \hat{\tau}_0 \qquad \hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ t_2 : \hat{\tau}_2 \qquad \varphi \sqsubseteq \hat{\tau}_2 \qquad \varphi \sqsubseteq \hat{\tau}_0}{\hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ t_1 \ t_2 : \hat{\tau}_0}$

[let] $\quad \dfrac{\hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ t_0 : \hat{\tau}_0 \qquad \hat{\Gamma}[x \mapsto \hat{\tau}_0] \ \vdash^{t}_{\mathtt{BTA}} \ t : \hat{\tau}}{\hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ \mathbf{let\ val}\ x = t_0\ \mathbf{in}\ t\ \mathbf{end} : \hat{\tau}}$

[dyn] $\quad \dfrac{\hat{\Gamma}[x \mapsto dyn(tp)] \ \vdash^{t}_{\mathtt{BTA}} \ t : \hat{\tau}}{\hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ \mathbf{let\ dyn\ val}\ x :: tp\ \mathbf{in}\ t\ \mathbf{end} : \hat{\tau}}$

[op] $\quad \dfrac{\hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ t_1 : \tau^{\varphi}_{op1} \qquad \hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ t_2 : \tau^{\varphi}_{op2}}{\hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ t_1 \ op \ t_2 : \tau^{\varphi}_{op}}$

[if] $\quad \dfrac{\hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ t_1 : Bool^{\varphi} \qquad \hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ t_2 : \hat{\tau} \qquad \hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ t_3 : \hat{\tau} \qquad \varphi \sqsubseteq \hat{\tau}}{\hat{\Gamma} \ \vdash^{t}_{\mathtt{BTA}} \ \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 : \hat{\tau}}$

Table 1 specifies all the atomic rules for binding-time analysis. The [num] rules describe the types for integer constants. Its definition:

$$[\text{num}] \qquad \hat{\Gamma} \ \vdash^{t}_{\texttt{BTA}} \ n : Nat^{\varphi}$$

is an abbreviation of

$$[\text{num-static}] \qquad \hat{\Gamma} \ \vdash^{t}_{\texttt{BTA}} \ n : Nat^{\mathbf{S}}, \ [\text{num-dynamic}] \qquad \hat{\Gamma} \ \vdash^{t}_{\texttt{BTA}} \ n : Nat^{\mathbf{D}},$$

and we employ a similar abbreviation in some of the other rules. The rule for dynamic value indicates that a dynamic variable will always be assigned a dynamic binding-time type produced by $dyn(tp)$ which is defined by

$$\begin{aligned} dyn(Nat) &= Nat^{\mathbf{D}} \\ dyn(Bool) &= Bool^{\mathbf{D}} \\ dyn(tp_1 \to tp_2) &= dyn(tp_1) \xrightarrow{\mathbf{D}} dyn(tp_2) \end{aligned}$$

Rule [fun] defines how to deduce types for a lambda abstraction. The auxiliary predicates $\varphi \sqsubseteq |\hat{\tau}_0|$ and $\varphi \sqsubseteq |\hat{\tau}_x|$ ensure that the infered type is well-formed. For recursive functions, we use a special identifier *self* to denote the recursive function. In the [if] rule, we also require the annotation of the condition to be smaller than the type of the whole term. Without this requirement, a static *then* branch and a static *else* branch will always imply the whole term to be static, which could be wrong. Because if the condition is dynamic, we still can not determine the value of the whole term.

The superscript t in the type judgment notation $\vdash^{t}_{\texttt{BTA}}$ denotes that it infers the binding time type for terms. As mentioned in last section, we separate programs from terms because a program will have the type of its term only if the type is dynamic at top level. Here we require programs to be dynamic because we want the specializer to generate a program for further processing rather than its value. The rule that infers the type of a program is then defined as:

$$[\text{Prog}] \quad \frac{\hat{\Gamma} \ \vdash^{t}_{\texttt{BTA}} \ t : \hat{\tau} \qquad |\hat{\tau}| = \mathbf{D}}{\hat{\Gamma} \ \vdash^{p}_{\texttt{BTA}} \ t : \hat{\tau}}$$

3.4 Inference Algorithm

In this section we will devise an algorithm for binding-time analysis, whose results will be consistent with the deduction rules defined in the previous section. The algorithm is based on the type reconstruction algorithm, algorithm $\mathcal{W}$. In the algorithm, we first traverse the abstract syntax tree and find out the relations between augmented types and annotations, which are described in forms of unifications and constraints. Unifications are resolved immediately during the traversal; this result in a substitution. Then we try to solve the constraints.

3.4.1 Augmented types and annotations

First, we extend our annotated types and annotations with type variables and annotation variables to augmented types and augmented annotations:

$\hat{\tau} \in \widehat{\mathbf{BTType}}$	augmented types
$\beta \in \mathbf{TVar}$	type variables
$\varphi \in \widehat{\mathbf{Ann}}$	augmented annotations
$\alpha \in \mathbf{AVar}$	annotation variables

$$
\begin{array}{rll}
\hat{\tau} ::= & Nat^{\varphi} \mid bool^{\varphi} \mid \hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau} \mid \beta \\
\beta ::= & '1 \mid '2 \mid '3 \mid ... \\
\varphi ::= & \mathbf{S} \mid \mathbf{D} \mid \alpha \\
\alpha ::= & 1' \mid 2' \mid 3' \mid ...
\end{array}
$$

3.4.2 Substitution

Augmented types and annotations are only used inside the algorithm. Each time we introduce a new type variable or an annotation variable, it is because we could not determine the types of a term and its subterms based on the current information. However, in the end we must resolve all the type and annotation variables with a concrete or a set of concrete annotated types and annotations. So we define a substitution $\theta : (\mathbf{TVar} \rightarrow_{\mathbf{fin}} \widehat{\mathbf{BTType}}) \times (\mathbf{AVar} \rightarrow_{\mathbf{fin}} \widehat{\mathbf{Ann}})$ to be a finite, partial mapping that maps type variables to augmented types and maps annotation variables to augmented annotations. A substitution $\theta = (\theta', \theta'')$ is applied to an augmented type, as follows:

$$
\begin{array}{rcl}
\theta\ \hat{\tau} & = & \theta''\ (\theta'\ \hat{\tau}) \\
\\
\theta'\ Nat^{\varphi} & = & Nat^{\varphi} \\
\theta'\ bool^{\varphi} & = & bool^{\varphi} \\
\theta'\ \hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2 & = & (\theta'\ \hat{\tau}_1) \xrightarrow{\varphi} (\theta'\ \hat{\tau}_2 \\
\theta'\ \beta & = & \hat{\tau} \quad \textit{if} \quad \theta'\ \beta = \hat{\tau} \\
\\
\theta''\ Nat^{\varphi} & = & Nat^{\theta''\varphi} \\
\theta''\ bool^{\varphi} & = & bool^{\theta''\varphi} \\
\theta''\ \hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2 & = & (\theta''\ \hat{\tau}_1) \xrightarrow{\theta''\varphi} (\theta''\ \hat{\tau}_2 \\
\theta''\ \beta & = & \beta
\end{array}
$$

Table 2: Application of Substitution.

We write $\circ$ for substitution composition: $(\theta_1 \circ \theta_0)\hat{\tau} \ = \ \theta_1(\theta_0\ \hat{\tau})$.

3.4.3 Unification

Unification describes that two types or two annotations are equal. It computes a substitution from two augmented types. If we have $\theta = \mathcal{U}_{\mathtt{BTA}}(\hat{\tau}_1, \hat{\tau}_2)$, then $\theta\ \hat{\tau}_1 = \theta\ \hat{\tau}_2$ holds. Here we use $\mathcal{U}_{\mathtt{BTA}}$ for unification of types and $\mathcal{U}'_{\mathtt{BTA}}$ for unification of annotations. Table 3 shows the calculation of substitution.

$$
\begin{array}{rcl}
\mathcal{U}_{\mathtt{BTA}}(Nat^{\varphi_1}, Nat^{\varphi_2}) & = & \mathcal{U}'_{\mathtt{BTA}}(\varphi_1, \varphi_2) \\
\mathcal{U}_{\mathtt{BTA}}(bool^{\varphi_1}, bool^{\varphi_2}) & = & \mathcal{U}'_{\mathtt{BTA}}(\varphi_1, \varphi_2) \\
\mathcal{U}_{\mathtt{BTA}}(\hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2, \hat{\tau}'_1 \xrightarrow{\varphi'} \hat{\tau}'_2) & = & \begin{array}{ll} \text{let} & \theta_0 = \mathcal{U}'_{\mathtt{BTA}}(\varphi, \varphi') \\ & \theta_1 = \mathcal{U}_{\mathtt{BTA}}(\theta_0\ \hat{\tau}_1, \theta_0\ \hat{\tau}'_1) \\ & \theta_2 = \mathcal{U}_{\mathtt{BTA}}(\theta_1(\theta_0\ \hat{\tau}_2), \theta_1(\theta_0\ \hat{\tau}'_2)) \\ \text{in} & \theta_2 \circ \theta_1 \circ \theta_0 \end{array} \\
\mathcal{U}_{\mathtt{BTA}}(\hat{\tau}, \beta) & = & \begin{cases} [\beta \mapsto \hat{\tau}] & \texttt{if } \beta \texttt{ does no occur in } \hat{\tau} \\ & \texttt{or if } \beta \texttt{ equals } \hat{\tau} \\ \texttt{fail} & \texttt{otherwise} \end{cases} \\
\mathcal{U}_{\mathtt{BTA}}(\beta, \hat{\tau}) & = & \mathcal{U}_{\mathtt{BTA}}(\hat{\tau}, \beta) \\
\mathcal{U}_{\mathtt{BTA}}(\hat{\tau}_1, \hat{\tau}_2) & = & \texttt{fail} \qquad \texttt{in all other cases} \\
\mathcal{U}'_{\mathtt{BTA}}(\mathbf{S}, \mathbf{S}) & = & id \\
\mathcal{U}'_{\mathtt{BTA}}(\mathbf{D}, \mathbf{D}) & = & id \\
\mathcal{U}'_{\mathtt{BTA}}(\varphi, \alpha) & = & [\alpha \mapsto \varphi] \\
\mathcal{U}'_{\mathtt{BTA}}(\alpha, \varphi) & = & [\alpha \mapsto \varphi] \\
\mathcal{U}'_{\mathtt{BTA}}(\varphi_1, \varphi_2) & = & \texttt{fail} \qquad \texttt{in all other cases}
\end{array}
$$

Table 3: Unification of augmented types and annotations

3.4.4 Constraints

Similar to unification, constraints are used to describe relations between types and annotations, except that unification will always result in a substitution whereas constrains may be left unresolved. In the MMX specification, only one form of inequality is present: $\varphi \sqsubseteq \hat{\tau}$ and in our constraints system that is represented by notation $\leq_h$ and have the form $\varphi \leq_h \hat{\tau}$. So a constraint $\mathbf{D} \leq_h Nat^{\alpha}$ can be resolved and a substitution $[\alpha \mapsto D]$ is its result. For a constraint $\mathbf{D} \leq_{\mathbf{h}} \beta$, we cannot compute a substitution that maps β to a type because the shape of β is undetermined.
In the algorithm, we collect a list of constraints which is defined as :

$$
\begin{array}{rcll}
c \in \mathbf{Con} & ::= & \varphi \leq_h \hat{\tau} & \text{constraint.} \\
cs \in \mathbf{Cons} & ::= & [\,] \mid c : cs & \text{constraints.}
\end{array}
$$

We use + for the operation of list concatenation and $[c]$ to refer to $c : [\,]$. Since constraints will not necessarily resolve to substitutions, unresolved constraints may remain. So the algorithm that solves the constraints is described as :

$$\omega(cs) = (cs', cs'', \theta)$$

where ω takes a list of constraints to be resolved as its arguments and returns a list of constraints that could be solved, a list of constraints that could not, and a substitution.

$$
\begin{array}{lll}
\omega([\,]) = ([\,], [\,], id) & & \\
\omega([\mathbf{S} \leq_{\mathbf{h}} \hat{\tau}]) = ([\mathbf{S} \leq_{\mathbf{h}} \hat{\tau}], [\,], id) & & \\
\omega([\alpha \leq_h \hat{\tau}]) = \mathit{case}\ |\hat{\tau}|\ \mathit{of} & & \\
\qquad\qquad \mathbf{S} & \rightarrow ([\alpha \leq_h \hat{\tau}], [\,], [\alpha \mapsto \mathbf{S}]) & \\
\qquad\qquad \alpha' & \rightarrow ([\,], [\alpha \leq_h \hat{\tau}], id) & \\
\qquad\qquad \mathbf{D} & \rightarrow ([\alpha \leq_h \hat{\tau}], [\,], id) & \\
\qquad\qquad \text{otherwise} & \rightarrow ([\,], [\alpha \leq_h \hat{\tau}], id) & \\
\omega([\mathbf{D} \leq_{\mathbf{h}} \hat{\tau}]) = \mathit{case}\ |\hat{\tau}|\ \mathit{of} & & \\
\qquad\qquad \mathbf{S} & \text{fails} & \\
\qquad\qquad \alpha & \rightarrow ([\mathbf{D} \leq_h \hat{\tau}], [\,], [\alpha \mapsto \mathbf{D}]) & \\
\qquad\qquad \mathbf{D} & \rightarrow ([\mathbf{D} \leq_h \hat{\tau}], [\,], id) & \\
\qquad\qquad \text{otherwise} & \rightarrow ([\,], [\mathbf{D} \leq_h \hat{\tau}], id) & \\
\omega(c : cs) = & \mathit{let}\ (s_1, s_2, \theta_1) = \omega([c]) & \\
& \mathit{in}\ \ \mathit{case}\ s_2\ \mathit{of} & \\
\qquad\qquad [\,] & \rightarrow & \mathit{let}\ (s_3, s_4, \theta_2) = \omega(\theta_1\ cs) \\
& & \mathit{in}\ \ (s_1 + s_3, s_4, \theta_2 \circ \theta_1) \\
\qquad\qquad \text{otherwise} & \rightarrow & \mathit{let}\ \ (s_3, s_4, \theta_2) = \omega(cs) \\
& & \qquad (s_5, s_6, \theta_3) = \omega((\theta_2\ s_2) + s_4) \\
& & \mathit{in}\ \ (s_3 + s_5, s_6, \theta_3 \circ \theta_2)
\end{array}
$$

Table 4: Algorithm of solving constraints

3.4.5 Algorithm

To infer the annotated types of a program automatically, we develop an algorithm $\mathcal{W}_{\mathtt{BTA}}$ for binding-time analysis, which is an extension of the type reconstruction algorithm $\mathcal{W}$. It has the form:

$$\mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t) \ = \ (\hat{\tau}, \theta, C)$$

Given a type environment, the algorithm calculates the binding time type for a term together with a substitution and a set of constraints. The algorithm is described in Table 5. The set of constraints denotes that the typing we infer is only correct if all the constraints are valid. So after applying algorithm $\mathcal{W}_{\mathtt{BTA}}$, we use the algorithm described in previous section to resolve the constraints.

$$\mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, c) = \mathtt{let}\ \alpha\ \mathtt{be\ fresh\ in}\ (Nat^{\alpha}, id, \emptyset)$$
$$\mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, true) = \mathtt{let}\ \alpha\ \mathtt{be\ fresh\ in}\ (bool^{\alpha}, id, \emptyset)$$
$$\mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, false) = \mathtt{let}\ \alpha\ \mathtt{be\ fresh\ in}\ (bool^{\alpha}, id, \emptyset)$$
$$\mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, x) = (\hat{\Gamma}(x), id, \emptyset)$$
$$\begin{array}{lll}\mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, \textbf{fun}\ x => t_0) = & \text{let} & \beta_x, \alpha_f\ \mathtt{be\ fresh} \\ & & (\hat{\tau}_0, \theta_0, C_0) = \mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}[x \mapsto \beta_x], t_0) \\ & \text{in} & (\theta_0\ \beta_x \xrightarrow{\alpha_f} \hat{\tau}_0,\ \theta_0,\ C_0 \cup \alpha_f \leq_h \theta_0\ \beta_x \cup \alpha_f \leq_h \hat{\tau}_0)\end{array}$$
$$\begin{array}{l}\mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, \textbf{fun rec}\ x => t_0) = \\ \quad \begin{array}{lll}\text{let} & \beta_x, \beta_0, \alpha_f\ \mathtt{be\ fresh} \\ & (\hat{\tau}_0, \theta_0, C_0) = \mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}[self \mapsto \beta_x \xrightarrow{\alpha_f} \beta_0][x \mapsto \beta_x], t_0) \\ & \theta_1 = \mathcal{U}_{\mathtt{BTA}}(\hat{\tau}_0, \theta_0\ \beta_0) \\ \text{in} & (\theta_1(\theta_0\ \beta_x) \xrightarrow{\theta_1(\theta_0\ \alpha_f)} \theta_1\ \hat{\tau}_0,\ \theta_1 \circ \theta_0, \\ & \qquad \theta_1\ C_0 \cup \theta_1(\theta_0\ \alpha_f) \leq_h \theta_1(\theta_0\ \beta_x) \cup \theta_1(\theta_0\ \alpha_f) \leq_h \theta_1\ \hat{\tau}_0)\end{array}\end{array}$$
$$\begin{array}{lll}\mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_1\ t_2) = & \text{let} & \beta_0, \alpha_f\ \mathtt{be\ fresh} \\ & & (\hat{\tau}_1, \theta_1, C_1) = \mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_1) \\ & & (\hat{\tau}_2, \theta_2, C_2) = \mathcal{W}_{\mathtt{BTA}}(\theta_1\ \hat{\Gamma}, t_2) \\ & & \theta_3 = \mathcal{U}_{\mathtt{BTA}}(\theta_2\ \hat{\tau}_1, \hat{\tau}_2 \xrightarrow{\alpha_f} \beta_0) \\ & \text{in} & (\theta_3\ \beta_0, \theta_3 \circ \theta_2 \circ \theta_1,\ \theta_3(\theta_2\ C_1) \cup \theta_3\ C_2)\end{array}$$
$$\begin{array}{lll}\mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, \textbf{let val}\ x = t_0\ \textbf{in}\ t\ \textbf{end}) = & \text{let} & (\hat{\tau}_0, \theta_0, C_0) = \mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_0) \\ & & (\hat{\tau}, \theta, C) = \mathcal{W}_{\mathtt{BTA}}(\theta_0\ \hat{\Gamma}[x \mapsto \hat{\tau}_0], t) \\ & \text{in} & (\hat{\tau}, \theta \circ \theta_0,\ C \cup \theta\ C_0)\end{array}$$
$$\mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, \textbf{let dyn val}\ x :: tp\ \textbf{in}\ t\ \textbf{end}\ = \mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}[x \mapsto dyn(tp)], t)$$
$$\begin{array}{lll}\mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_1\ op\ t_2) = & \text{let} & \alpha\ \mathtt{be\ fresh} \\ & & (\hat{\tau}_1, \theta_1, C_1) = \mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_1) \\ & & (\hat{\tau}_2, \theta_2, C_2) = \mathcal{W}_{\mathtt{BTA}}(\theta_1\ \hat{\Gamma}, t_2) \\ & & \theta_3 = \mathcal{U}_{\mathtt{BTA}}(\theta_2\ \hat{\tau}_1, \tau_{op1}^{\alpha}) \\ & & \theta_4 = \mathcal{U}_{\mathtt{BTA}}(\theta_3\ \hat{\tau}_2, \tau_{op2}^{\theta_3\ \alpha}) \\ & \text{in} & (\tau_{op}^{\theta_4(\theta_3\ \alpha)},\ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1,\ \theta_4(\theta_3(\theta_2\ C_1)) \cup \theta_4(\theta_3\ C_2))\end{array}$$
$$\begin{array}{l}\mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3) = \\ \quad \begin{array}{lll}\text{let} & \alpha\ \mathtt{be\ fresh} \\ & (\hat{\tau}_1, \theta_1, C_1) = \mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_1) \\ & (\hat{\tau}_2, \theta_2, C_2) = \mathcal{W}_{\mathtt{BTA}}(\theta_1\ \hat{\Gamma}, t_2) \\ & (\hat{\tau}_3, \theta_3, C_3) = \mathcal{W}_{\mathtt{BTA}}(\theta_2(\theta_1\ \hat{\Gamma}), t_3) \\ & \theta_4 = \mathcal{U}_{\mathtt{BTA}}(\theta_3(\theta_2\ \hat{\tau}_1), bool^{\alpha}) \\ & \theta_5 = \mathcal{U}_{\mathtt{BTA}}(\theta_4\ \hat{\tau}_3, \theta_4(\theta_3\ \hat{\tau}_2)) \\ \text{in} & (\theta_5(\theta_4\ \hat{\tau}_3),\ \theta_5 \circ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1, \\ & \qquad \theta_5(\theta_4(\theta_3(\theta_2\ C_1))) \cup \theta_5(\theta_4(\theta_3\ C_2)) \\ & \qquad\quad \cup \theta_5(\theta_4\ C_3) \cup \theta_5(\theta_4\ \alpha) \leq_h \theta_5(\theta_4\ \hat{\tau}_3))\end{array}\end{array}$$

Table 5: Algorithm $\mathcal{W}_{\mathtt{BTA}}$

3.5 Metrics

Two aspects need to be measured for binding-time analysis. The expressive power of the analysis is described by the number of static subterms in a program:

$$power(p) = |\{\hat{\mathtt{B}}(l) | \forall l \in lab(p), \hat{\mathtt{B}}(l) = \mathbf{S}\}|,$$

where $lab(p)$ calculates the set of labels that appear in program p and function $\hat{\mathtt{B}}$ is defined in Section 3.2. The number of static terms is a very general metric to measure the preciseness of binding time analysis. Since the main usage of binding time analysis is in a partial evaluator, we also use another measurement: the number of static applications which a partial evaluator is more concerned with because that is the place where a partial evaluator reduces a term. A static application $(t_1\ t_2)$ means that the top level annotation of t_1 is static.

The cost of binding-time analysis is described by the time consumption for calculating the binding time types of each expression. In algorithm $\mathcal{W}_{\mathtt{BTA}}$, this is proportional to the computation of substitutions from unifications and constraints. So we use the total number of unifications and constraints to denote the cost of the analysis. The number of recursive calls to unification is defined as :

$$\begin{array}{rcll}
num(\mathcal{U}_{\mathtt{BTA}}(\hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2, \hat{\tau}_1' \xrightarrow{\varphi'} \hat{\tau}_2')) & = & \text{let} & n1 = num(\mathcal{U}_{\mathtt{BTA}}(\hat{\tau}_1, \hat{\tau}_1')) \\
 & & & n2 = num(\mathcal{U}_{\mathtt{BTA}}(\varphi, \varphi')) \\
 & & & n3 = num(\mathcal{U}_{\mathtt{BTA}}(\hat{\tau}_2, \hat{\tau}_2')) \\
 & & \text{in} & n1 + n2 + n3 \\
num(\mathcal{U}_{\mathtt{BTA}}(\hat{\tau}_1, \hat{\tau}_2)) & = 1 & & \textit{in other cases} \\
num(\mathcal{U}'_{\mathtt{BTA}}(\varphi_1, \varphi_2)) & = 1 & &
\end{array}$$

Another important aspect of measuring complexity of binding-time analysis is its implementation effort such as the number of modules we used, the total number of functions defined, the lines of the program and etc. Although this information does not show runtime performance of the program, it provides useful information of design and implementation of the analysis.

Chapter 4

Polymorphism and Polyvariance

In this chapter, we extend our monomorphic language to a polymorphic typed language and develop a polyvariant binding-time analysis for it. In the context of binding-time analysis, "binding-time polymorphism" is often referred to as *polyvariance*. As mentioned in previous chapter, we call this analysis PPX.

4.1 Binding-Time Type Schemes

Binding-time analysis can be specified as binding-time logics, a type inference system. A monomorphic binding-time logic only allow a single binding-time to be assigned to a variable. This is a severe weakness of expressive power, because each variable must be assigned with a conservative approximation of all uses of the variable. For example, we have a program:

$$
\begin{array}{l}
\textbf{let val } id = (\textbf{fun } x \Rightarrow x^1)^2 \\
\textbf{in } \textbf{ let val } y = (id^3\ 2^4)^5 \\
\quad \textbf{in } (id^6\ 3^7)^8 \\
\quad \textbf{end}^9 \\
\textbf{end}^{10}.
\end{array}
$$

Since the result of the program should be dynamic, the expression (id 3) will have type $Nat^{\mathbf{D}}$, which leads to the function id having type $Nat^{\mathbf{D}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{D}}$. In a monomorphic monovariant system, variables have monomorphic types, so (id 2) has type $Nat^{\mathbf{D}}$ and 2 has type $Nat^{\mathbf{D}}$. If we only consider expression (id 2), we could assign them with better types: $id : Nat^{\mathbf{S}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{S}}$ and $2 : Nat^{\mathbf{S}}$. However, we have to weaken the binding time condition to $id : Nat^{\mathbf{D}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{D}}$ and $2 : Nat^{\mathbf{D}}$ because the analysis should give a safe answer. Further more the variable y, which has nothing to do with the result of the program, will also be assigned a dynamic annotation. In a monovariant analysis, dynamic binding times spread easily, and pollute expressions that could have a static binding time in a more expressive system.

A solution to this problem is to copy a variable(function) in as many variants as there are different uses of it in a program, so that we can always give best approximation of each variable(function), because there will be at most one use for each variable(function) defined. For the example above, we transform the program to:

$$
\begin{array}{l}
\textbf{let val } id = (\textbf{fun } x \Rightarrow x^1)^2 \\
\textbf{in } \textbf{ let val } y = ((\textbf{fun } x \Rightarrow x)^3\ 2^4)^5 \\
\quad \textbf{in } ((\textbf{fun } x \Rightarrow x)^6\ 3^7)^8 \\
\quad \textbf{end}^9 \\
\textbf{end}^{10}.
\end{array}
$$

However the downside of this approach is obvious that the copying increases the complexity of a program dramatically. Furthermore, it imposes a burden on the programmer who ideally could stay completely unaware of binding time analysis and partial evaluation.

An alternative way is to use polyvariance. The idea is to assign each variable a polyvarint binding-time property which may have binding-time parameters that can be instantiated into various binding-times in line with the use context of the variable. Furthermore, we also extend the target language to a polymorphic language to increase the expressive power of the language itself. First we redefine annotations and binding-time types by introducing variables.

$\varphi \in \widehat{\textbf{Ann}}$	Annotations		
::=	$\textbf{S} \quad \mid \quad \textbf{D}$		
$\mid$	α		binding-time variable
$\hat{\tau} \in \widehat{\textbf{BTType}}$	Binding-time types		
::=	$Nat^{\varphi} \quad \mid \quad Bool^{\varphi} \quad \mid \quad \hat{\tau} \xrightarrow{\varphi} \hat{\tau}$		
$\mid$	β		binding-time type variable

Here, we use annotation and type variables to denote arbitrary annotations and types. This can be easily confused with their usage in an inference algorithm. In such algorithms, we often define an augment type system which allows type variables to represent unknown types in the process of type inference. To make it concise, we do not use new notations in the augment type systems of our polymorphic binding-time type system but differentiate the purpose and meaning of those variables by their use contexts. Then we define qualified types that is qualified with constraints and type schemes which have annotation and type variables being quantified over. Constraints are defined and explained in details in next section.

$\hat{\rho} \in \widehat{\textbf{QualTp}}$	Qualified types	
::=	$\hat{\tau}$	
$\mid$	$C \Rightarrow \hat{\rho}$	qulified types
$\hat{\sigma} \in \widehat{\textbf{BTSch}}$	Binding-time type schemes	
::=	$\hat{\rho}$	
$\mid$	$\forall\alpha.\ \hat{\sigma}$	polyvariant types
$\mid$	$\forall\beta.\ \hat{\sigma}$	polymorphic types

The toplevel operation for type shcemes is then defined as :

$$
\begin{array}{rll}
|Nat^{\varphi}| & = \varphi & \\
|Bool^{\varphi}| & = \varphi & \\
|\hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2| & = \varphi & \\
|\beta| & = undefined & \text{undefined value} \\
|C \Rightarrow \hat{\sigma}| & = \begin{cases} undefined & \texttt{if } |\hat{\sigma}| = undefined \\ |\hat{\sigma}| & \texttt{otherwise} \end{cases} & \\
|\forall \alpha.\ \hat{\sigma}| & = \begin{cases} undefined & \texttt{if } |\hat{\sigma}| = undefined \\ |\hat{\sigma}| & \texttt{otherwise} \end{cases} & \\
|\forall \beta.\ \hat{\sigma}| & = \begin{cases} undefined & \texttt{if } |\hat{\sigma}| = undefined \\ |\hat{\sigma}| & \texttt{otherwise} \end{cases} &
\end{array}
$$

4.2 Deduction Rules

Intuitively, $\beta \xrightarrow{\alpha} \beta$ could be a correct type of the *identity* function **fun** $x \Rightarrow x$. However, this cannot be determined by using the [fun] typing rule of the MMX deduction system, because we have introduced variables into our type system and it is not able to devise a rule to answer whether or not the well-formedness condition such as $\varphi \sqsubseteq |\hat{\tau}|$ holds when $\hat{\tau}$ and φ are type and annotation variables. To solve this problem, we capture this kind of inequality relations between annotations and types in a constraint and encode them into a type as a qualification to form a qulified type. Later on we will also find out that subsumption relations can be easily integrated into the framework of qulified types. Instead of assigning $\beta \xrightarrow{\alpha} \beta$ to **fun** $x \Rightarrow x$, we expect $\alpha \leq_h \beta \Rightarrow \beta \xrightarrow{\alpha} \beta$ to be the correct type of the function and $\forall \alpha. \forall\ \beta. \alpha \lhd \beta \Rightarrow \beta \xrightarrow{\alpha} \beta$ to be the most general type or principal type.

First, we refine the constraint system. We capture two kinds of inequality relations between annotations and types. We use $\varphi_1 \leq_a \varphi_2$ to denote that φ_1 is weaker than φ_2 and we use $\varphi \leq_h \hat{\tau}$ to denote that φ is weaker than the top level annotation of $\hat{\tau}$. So we have the binding-time constraint defined as:

$$
\begin{array}{rll}
c \in \mathbf{Con} & ::= \varphi_1 \leq_a \varphi_2 \mid \varphi \leq_h \hat{\tau} & \text{binding-time constraint.} \\
cs \in \mathbf{Cons} & = \mathcal{P}(\mathbf{Con}) & \text{binding-time constraints.}
\end{array}
$$

Here, notation $\mathcal{P}$ denotes power set. Thus constraints are defined as the set of constraint so they could be manipulated by set operations such as conjunction and set difference. Similarly to type rules, we define constraint rules in term of constraint judgment which has the form:

$$C \vdash c \quad .$$

This judgment indicates that a certain constraint c holds under the assumption of constraints C holds. The inference rules of the constraint system are specified in Table 6.

[C-Sta]	$C \vdash \mathbf{S} \leq_a \varphi$	$C \vdash \mathbf{S} \leq_h \hat{\tau}$
[C-Dyn]	$C \vdash \varphi \leq_a \mathbf{D}$	
[C-Elem]	$\dfrac{c \in C}{C \vdash c}$ $\quad$ if $c = \begin{cases} \alpha \leq_a \varphi \\ \varphi \leq_a \alpha \\ \varphi \leq_h \beta \end{cases}$	
[CA-Refl]	$C \vdash \varphi \leq_a \varphi$	
[CA-Trans]	$\dfrac{C \vdash \varphi_1 \leq_a \varphi_2 \quad C \vdash \varphi_2 \leq_a \varphi_3}{C \vdash \varphi_1 \leq_a \varphi_3}$	
[CH-Chan]	$\dfrac{C \vdash \varphi_1 \leq_a \varphi_2 \quad \lvert\hat{\tau}\rvert = \varphi_2}{C \vdash \varphi_1 \leq_h \hat{\tau}}$	
[C-Cons]	$\dfrac{\forall c \in C_2 \;.\; C_1 \vdash c}{C_1 \models C_2}$	

Table 6: Binding-time constraint rules

Most of the rules are quite straightforward, for example the transitivity rule shows that the partial ordering of annotations is transitive and the C-Refl rule shows the reflexivity of it. For convenience, we also define the judgment for constraints as in the rule [C-Cons] which is quite obvious that if any constraint in C_2 holds under constraints C_1 then C_2 holds in the context of C_1.

Using the constraint defined above we develop a new form of typing judgment:

$$C, \hat{\Gamma} \;\vdash^t_{\texttt{BTA}}\; t : \hat{\sigma}$$

Besides typing environment, we add a constraints environment and they together form called a typing context. Then we reformulate the MMX deduction rules in Table 7. Most rules remain the same as before with simple extending from typing environments to typing context except for lambda abstraction and branch(if...then...else...) terms. When we derive the type of a lambda abstraction, instead of requiring the condition that the annotation of the function must not be more dynamic than its argument type and result type, we make sure that the constraints that represent those inequality relations holds under a certain typing context. So for function $\mathbf{fun}x \Rightarrow x$ we come up with its typing $\{\alpha \leq_h \beta\}, [\,] \;\vdash^t_{\texttt{BTA}}\; \mathbf{fun}\; x \Rightarrow x : \beta \xrightarrow{\alpha} \beta$. The proof is shown below:

$$\dfrac{\dfrac{[x \mapsto \beta](x) = \beta}{\{\alpha \leq_h \beta\}, [x \mapsto \beta] \;\vdash^t_{\texttt{BTA}}\; x : \beta} \quad \dfrac{\alpha \leq_h \beta \in \{\alpha \leq_h \beta\}}{\{\alpha \leq_h \beta\} \vdash\; \alpha \leq_h \beta} \quad \dfrac{\alpha \leq_h \beta \in \{\alpha \leq_h \beta\}}{\{\alpha \leq_h \beta\} \vdash\; \alpha \leq_h \beta}}{\{\alpha \leq_h \beta\}, [\,] \;\vdash^t_{\texttt{BTA}}\; \mathbf{fun}\; x \Rightarrow x : \beta \xrightarrow{\alpha} \beta}$$

[num] $C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} n : Nat^{\varphi}$

[bool] $C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} b : Bool^{\varphi}$

[id] $$\frac{\hat{\Gamma}(x) = \hat{\sigma}}{C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} x : \hat{\sigma}}$$

[fun] $$\frac{C, \hat{\Gamma}[x \mapsto \hat{\tau}_x] \vdash^t_{\texttt{BTA}} t_0 : \hat{\tau}_0 \qquad C \vdash \varphi \leq_h \hat{\tau}_0 \qquad C \vdash \varphi \leq_h \hat{\tau}_x}{C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} \textbf{fun } x \Rightarrow t_0 : \hat{\tau}_x \xrightarrow{\varphi} \hat{\tau}_0}$$

[rec] $$\frac{C, \hat{\Gamma}[self \mapsto \hat{\tau}_x \xrightarrow{\varphi} \hat{\tau}_0][x \mapsto \hat{\tau}_x] \vdash^t_{\texttt{BTA}} t_0 : \hat{\tau}_0 \qquad C \vdash \varphi \leq_h \hat{\tau}_0 \qquad C \vdash \varphi \leq_h \hat{\tau}_x}{C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} \textbf{fun rec } x \Rightarrow t_0 : \hat{\tau}_x \xrightarrow{\varphi} \hat{\tau}_0}$$

[app] $$\frac{C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} t_1 : \hat{\tau}_2 \xrightarrow{\varphi} \hat{\tau}_0 \qquad C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} t_2 : \hat{\tau}_2 \qquad C \vdash \varphi \leq_h \hat{\tau}_1 \qquad C \vdash \varphi \leq_h \hat{\tau}_2}{C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} t_1\ t_2 : \hat{\tau}_0}$$

[let] $$\frac{C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} t_0 : \hat{\sigma}_0 \qquad C, \hat{\Gamma}[x \mapsto \hat{\sigma}_0] \vdash^t_{\texttt{BTA}} t : \hat{\tau}}{C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} \textbf{let val } x = t_0 \textbf{ in } t \textbf{ end} : \hat{\tau}}$$

[dyn] $$\frac{C, \hat{\Gamma}[x \mapsto dyn(tp)] \vdash^t_{\texttt{BTA}} t : \hat{\tau}}{C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} \textbf{let dyn val } x :: tp \textbf{ in } t \textbf{ end} : \hat{\tau}}$$

[op] $$\frac{C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} t_1 : \tau^{\varphi}_{op1} \qquad C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} t_2 : \tau^{\varphi}_{op2}}{C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} t_1\ op\ t_2 : \tau^{\varphi}_{op}}$$

[if] $$\frac{C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} t_1 : Bool^{\varphi} \qquad C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} t_2 : \hat{\tau} \qquad C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} t_3 : \hat{\tau} \qquad C \vdash \varphi \leq_h \hat{\tau}}{C, \hat{\Gamma} \vdash^t_{\texttt{BTA}} \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 : \hat{\tau}}$$

Table 7: Deduction rules for terms

So far, we have successfully infered the type judgement $\{\alpha \leq_h \beta\}, [\,] \vdash^t_{\texttt{BTA}} \textbf{fun } x \Rightarrow x : \beta \xrightarrow{\alpha} \beta$. But it is not precise enough to prove the correctness of assigning a certain type to an expression because such a typing is always made under certain constraints as long as its constraints environment is not empty. It is necessary to develop rules to eliminate constraints in the typing context by puting them into the type as qualifications, so that we are able to infer the typing with empty typing context $\emptyset, [\,] \vdash^t_{\texttt{BTA}} \textbf{fun } x \Rightarrow x : \{\alpha \leq_h \beta\} \Rightarrow \beta \xrightarrow{\alpha} \beta$ for the *identitiy* function. In addition, we also have to develop rules to introduce polyvariance and polymorphism.

[Qual]	$\dfrac{C_1 \cup C_2, \hat{\Gamma} \ \vdash^t_{\texttt{BTA}} \ t : \hat{\rho}}{C_1, \hat{\Gamma} \ \vdash^t_{\texttt{BTA}} \ t : C_2 \Rightarrow \hat{\rho}}$
[Res]	$\dfrac{C_1, \hat{\Gamma} \ \vdash^t_{\texttt{BTA}} \ t : C_2 \Rightarrow \hat{\rho} \qquad C \models C_1 \qquad C \models C_2}{C, \hat{\Gamma} \ \vdash^t_{\texttt{BTA}} \ t : \hat{\rho}}$
[TpGen]	$\dfrac{C, \hat{\Gamma} \ \vdash^t_{\texttt{BTA}} \ t : \hat{\sigma} \quad \beta \notin ftv(\hat{\Gamma}) \quad \beta \notin ftv(C)}{C, \hat{\Gamma} \ \vdash^t_{\texttt{BTA}} \ t : \forall \beta . \hat{\sigma}}$
[TpInst]	$\dfrac{C, \hat{\Gamma} \ \vdash^t_{\texttt{BTA}} \ t : \forall \beta . \hat{\sigma}}{C, \hat{\Gamma} \ \vdash^t_{\texttt{BTA}} \ t : [\beta \mapsto \hat{\tau}]\hat{\sigma}}$
[AnnGen]	$\dfrac{C, \hat{\Gamma} \ \vdash^t_{\texttt{BTA}} \ t : \hat{\sigma} \quad \alpha \notin fav(\hat{\Gamma}) \quad \alpha \notin fav(C)}{C, \hat{\Gamma} \ \vdash^t_{\texttt{BTA}} \ t : \forall \alpha . \hat{\sigma}}$
[AnnInst]	$\dfrac{C, \hat{\Gamma} \ \vdash^t_{\texttt{BTA}} \ t : \forall \alpha . \hat{\sigma}}{C, \hat{\Gamma} \ \vdash^t_{\texttt{BTA}} \ t : [\alpha \mapsto \varphi]\hat{\sigma}}$

Table 8:Non-syntax directed rules

Here, ftv and fav are functions that compute free type variables and free annotation varialbes for typing contexts. To make it concise, we treat these two functions as overloaded functions and give their definitions as:

Definition of free type variables:

$ftv(Nat^{\varphi}) = \emptyset$ $\quad$ $ftv(Bool^{\varphi}) = \emptyset$

$ftv(\beta) = \beta$ $\quad$ $ftv(\hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2) = ftv(\hat{\tau}_1) \cup ftv(\hat{\tau}_2)$

$ftv(C \Rightarrow \hat{\rho}) = ftv(\hat{\rho})$ $\quad$ $ftv(\forall \alpha .\ \hat{\sigma}) = ftv(\hat{\sigma})$

$ftv(\forall \beta .\ \hat{\sigma}) = ftv(\hat{\sigma}) - \{\beta\}$

$ftv([\,]) = \emptyset$ $\quad$ $ftv(\hat{\Gamma}[x \mapsto \hat{\sigma}]) = ftv(\hat{\Gamma}) \cup ftv(\hat{\sigma})$

$ftv(\varphi_1 \leq_a \varphi_2) = \emptyset$ $\quad$ $ftv(\varphi \leq_h \hat{\tau}) = ftv(\hat{\tau})$

$ftv(C) = \bigcup\{ftv(c) | \forall c \in C\}$

Definition of free annotation variables:

$fav(\mathbf{S}) = \emptyset$ $\quad$ $fav(\mathbf{D}) = \emptyset$

$fav(\alpha) = \{\alpha\}$

$fav(Nat^{\varphi}) = fav(\varphi)$ $\quad$ $fav(Bool^{\varphi}) = fav(\varphi)$

$fav(\beta) = \emptyset$ $\quad$ $fav(\hat{\tau}_1 \xrightarrow{\varphi} \hat{\tau}_2) = fav(\hat{\tau}_1) \cup fav(\hat{\tau}_2) \cup fav(\varphi)$

$fav(C \Rightarrow \hat{\rho}) = fav(\hat{\rho})$ $\quad$ $fav(\forall \alpha .\ \hat{\sigma}) = fav(\hat{\sigma}) - \{\alpha\}$

$fav(\forall \beta .\ \hat{\sigma}) = fav(\hat{\sigma})$

$fav([\,]) = \emptyset$ $\quad$ $fav(\hat{\Gamma}[x \mapsto \hat{\sigma}]) = fav(\hat{\Gamma}) \cup fav(\hat{\sigma})$

$fav(\varphi_1 \leq_a \varphi_2) = fav(\varphi_1) \cup fav(\varphi_2)$ $\quad$ $fav(\varphi \leq_h \hat{\tau}) = fav(\hat{\tau}) \cup fav(\varphi)$

$fav(C) = \bigcup\{fav(c) | \forall c \in C\}$

4.3 Generalization and Instantiation

With the specification, we develop an algorithm of a ML-style polymorphism. The idea is to infer the most generalized types which are usually type schemes for let-bound variables and then instantiate them into different types according to there use context.
Generalization. Generalization function takes a type and a typing context as arguments and generates a type scheme. It finds all type and effect variables that are free in the type but not free in the typing context and quantifies over these variables. The *gen* function is defined as

$$gen(\hat{\rho}, \hat{\Gamma}, C) = \forall\alpha_1...\forall\alpha_n.\ \forall\beta_1...\forall\beta_m.\ \hat{\rho} \quad \texttt{where} \begin{cases} \{\alpha_1...\alpha_n\} = fav(\hat{\rho}) - fav(C) - fav(\hat{\Gamma}) \\ \{\beta_1...\beta_m\} = ftv(\hat{\rho}) - ftv(C) - ftv(\hat{\Gamma}) \end{cases}$$

Instantiation. Opposite to generalization, instantiation function turns a type scheme into a binding-time type.

$$\begin{array}{rcll} inst(\hat{\rho}) & = & \hat{\rho} & \\ inst(\forall\alpha_i.\ \hat{\sigma}) & = & \theta_i\ inst(\hat{\sigma}) & \texttt{where}\ \theta_i = [\alpha_i \mapsto \alpha_i']\ \texttt{and}\ \alpha_i' \notin fav(\forall\alpha_i.\ \hat{\sigma}) \\ inst(\forall\beta_i.\ \hat{\sigma}) & = & \theta_i\ inst(\hat{\sigma}) & \texttt{where}\ \theta_i = [\beta_i \mapsto \beta_i']\ \texttt{and}\ \beta_i' \notin ftv(\forall\beta_i.\ \hat{\sigma}) \end{array}$$

4.4 Algorithm

When adapt our algorithm to the new deduction rules, we need to introduce polymorphism into our algorithm. We can dispense with explicit rules for polymorphism by integrating generalization and instantiation into all syntax-direct rules. Alternatively, we only incorporate polymorphism where it is really needed: generalization in let rule and instantiation in variable rule.

The idea of polymorphism is that we allow the same term to be assigned to different types in different use contexts. In our source language, one term is used in variant contexts through two kinds of variable binding: let-binding which binds local variables with their definitions and lambda-binding which binds formal arguments with actual arguments. So these two places are the places that need to incorporate generalization and we call them let polymorphism and lambda polymorphism. And integrating polymorphism in terms of other syntax constructors is not necessary because we have to instantiate the types immediately after we generalize them, which makes both generalization and instantiation meaningless. Since let-polymorphism and lambda-polymorphism are essentially the same, in this thesis, we focus on let-polymorphism. Luis Damas and Robin Milner had proved soundness and completeness of a let polymorphism algorithm, Algorithm $\mathcal{W}$, for a polymorphic type system[2], which can be then extended to annotated type system. Table 9 shows the refined inference algorithm for a polyvariant analysis on polymorphic language. It differs from the old one in two cases, the local definition and variables. The type we infer for the local definition is generalized into a type scheme and then the local variable is instantiated into different types in its use context.

$$\mathcal{W}_{\texttt{BTA}}(\hat\Gamma, c) = \texttt{let}\ \alpha\ \texttt{be fresh in}\ (Nat^{\alpha}, id, \emptyset)$$

$$\mathcal{W}_{\texttt{BTA}}(\hat\Gamma, true) = \texttt{let}\ \alpha\ \texttt{be fresh in}\ (bool^{\alpha}, id, \emptyset)$$

$$\mathcal{W}_{\texttt{BTA}}(\hat\Gamma, false) = \texttt{let}\ \alpha\ \texttt{be fresh in}\ (bool^{\alpha}, id, \emptyset)$$

$$\mathcal{W}_{\texttt{BTA}}(\hat\Gamma, x) = \begin{array}{ll} \text{let} & C \Rightarrow \hat\tau = inst(\hat\Gamma(x)) \\ \text{in} & (\hat\tau, id, C) \end{array}$$

$$\mathcal{W}_{\texttt{BTA}}(\hat\Gamma, \textbf{fun}\ x \Rightarrow t_0) = \begin{array}{ll} \text{let} & \beta_x, \alpha_f\ \texttt{be fresh} \\ & (\hat\tau_0, \theta_0, C_0) = \mathcal{W}_{\texttt{BTA}}(\hat\Gamma[x \mapsto \beta_x], t_0) \\ \text{in} & (\theta_0\ \beta_x \xrightarrow{\alpha_f} \hat\tau_0,\ \theta_0,\ C_0 \cup \alpha_f \leq_h \theta_0\ \beta_x \cup \alpha_f \leq_h \hat\tau_0) \end{array}$$

$$\begin{array}{l} \mathcal{W}_{\texttt{BTA}}(\hat\Gamma, \textbf{fun rec}\ x \Rightarrow t_0) = \\ \qquad \begin{array}{ll} \text{let} & \beta_x, \beta_0, \alpha_f\ \texttt{be fresh} \\ & (\hat\tau_0, \theta_0, C_0) = \mathcal{W}_{\texttt{BTA}}(\hat\Gamma[self \mapsto \beta_x \xrightarrow{\alpha_f} \beta_0][x \mapsto \beta_x], t_0) \\ & \theta_1 = \mathcal{U}_{\texttt{BTA}}(\hat\tau_0, \theta_0\ \beta_0) \\ \text{in} & (\theta_1(\theta_0\ \beta_x) \xrightarrow{\theta_1(\theta_0\ \alpha_f)} \theta_1\ \hat\tau_0,\ \theta_1 \circ \theta_0, \\ & \quad \theta_1\ C_0 \cup \theta_1(\theta_0\ \alpha_f) \leq_h \theta_1(\theta_0\ \beta_x) \cup \theta_1(\theta_0\ \alpha_f) \leq_h \theta_1\ \hat\tau_0) \end{array} \end{array}$$

$$\mathcal{W}_{\texttt{BTA}}(\hat\Gamma, t_1\ t_2) = \begin{array}{ll} \text{let} & \beta_0, \alpha_f\ \texttt{be fresh} \\ & (\hat\tau_1, \theta_1, C_1) = \mathcal{W}_{\texttt{BTA}}(\hat\Gamma, t_1) \\ & (\hat\tau_2, \theta_2, C_2) = \mathcal{W}_{\texttt{BTA}}(\theta_1\ \hat\Gamma, t_2) \\ & \theta_3 = \mathcal{U}_{\texttt{BTA}}(\theta_2\ \hat\tau_1, \hat\tau_2 \xrightarrow{\alpha_f} \beta_0) \\ \text{in} & (\theta_3\ \beta_0, \theta_3 \circ \theta_2 \circ \theta_1,\ \theta_3(\theta_2\ C_1) \cup \theta_3\ C_2) \end{array}$$

$$\mathcal{W}_{\texttt{BTA}}(\hat\Gamma, \textbf{let val}\ x = t_0\ \textbf{in}\ t\ \textbf{end}) = \begin{array}{ll} \text{let} & (\hat\tau_0, \theta_0, C_0) = \mathcal{W}_{\texttt{BTA}}(\hat\Gamma, t_0) \\ & \hat\sigma_0 = gen(C_0 \Rightarrow \hat\tau_0, \hat\Gamma, \emptyset) \\ & (\hat\tau, \theta, C) = \mathcal{W}_{\texttt{BTA}}(\theta_0\ \hat\Gamma[x \mapsto \hat\sigma_0], t) \\ \text{in} & (\hat\tau, \theta \circ \theta_0,\ C \cup \theta\ C_0) \end{array}$$

$$\mathcal{W}_{\texttt{BTA}}(\hat\Gamma, \textbf{let dyn val}\ x :: tp\ \textbf{in}\ t\ \textbf{end}\ = \mathcal{W}_{\texttt{BTA}}(\hat\Gamma[x \mapsto dyn(tp)], t)$$

$$\mathcal{W}_{\texttt{BTA}}(\hat\Gamma, t_1\ op\ t_2) = \begin{array}{ll} \text{let} & \alpha\ \texttt{be fresh} \\ & (\hat\tau_1, \theta_1, C_1) = \mathcal{W}_{\texttt{BTA}}(\hat\Gamma, t_1) \\ & (\hat\tau_2, \theta_2, C_2) = \mathcal{W}_{\texttt{BTA}}(\theta_1\ \hat\Gamma, t_2) \\ & \theta_3 = \mathcal{U}_{\texttt{BTA}}(\theta_2\ \hat\tau_1, \tau_{op1}^{\alpha}) \\ & \theta_4 = \mathcal{U}_{\texttt{BTA}}(\theta_3\ \hat\tau_2, \tau_{op2}^{\theta_3\ \alpha}) \\ \text{in} & (\tau_{op}^{\theta_4(\theta_3\ \alpha)},\ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1,\ \theta_4(\theta_3(\theta_2\ C_1)) \cup \theta_4(\theta_3\ C_2)) \end{array}$$

$$\begin{array}{l} \mathcal{W}_{\texttt{BTA}}(\hat\Gamma, \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3) = \\ \qquad \begin{array}{ll} \text{let} & \alpha\ \texttt{be fresh} \\ & (\hat\tau_1, \theta_1, C_1) = \mathcal{W}_{\texttt{BTA}}(\hat\Gamma, t_1) \\ & (\hat\tau_2, \theta_2, C_2) = \mathcal{W}_{\texttt{BTA}}(\theta_1\ \hat\Gamma, t_2) \\ & (\hat\tau_3, \theta_3, C_3) = \mathcal{W}_{\texttt{BTA}}(\theta_2(\theta_1\ \hat\Gamma), t_3) \\ & \theta_4 = \mathcal{U}_{\texttt{BTA}}(\theta_3(\theta_2\ \hat\tau_1), bool^{\alpha}) \\ & \theta_5 = \mathcal{U}_{\texttt{BTA}}(\theta_4\ \hat\tau_3, \theta_4(\theta_3\ \hat\tau_2)) \\ \text{in} & (\theta_5(\theta_4\ \hat\tau_3),\ \theta_5 \circ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1, \\ & \quad \theta_5(\theta_4(\theta_3(\theta_2\ C_1))) \cup \theta_5(\theta_4(\theta_3\ C_2)) \\ & \qquad \cup \theta_5(\theta_4\ C_3) \cup \theta_5(\theta_4\ \alpha) \leq_h \theta_5(\theta_4\ \hat\tau_3)) \end{array} \end{array}$$

Table 9: Algorithm $\mathcal{W}_{\texttt{BTA}}$for PPX

Chapter 5

Subeffecting and Subtyping

In this chapter, we first specify the subsumption rule as an extension to our existing type system(section 5.1). Then we explain two form of subsumption: subeffecting (section 5.2) and subtyping (section 5.3). At last we describe an algorithm (section 5.4) that handles subtyping and subeffecing.

5.1 Subsumption rule

Many mathematical concepts are useful in computer science, such as graphs, lists and sets. A type is essencial a set of values that have a certain kind of similarity. For example, a type *Real* can be seen as a set containing all the real numbers and a type Nat identifies natural numbers.

Let us consider a function *add* which takes two real numbers as arguments and returns the sum of these two number. Then it has type : $Real \to Real \to Real$. In mathematics, natural number is a subset of real number or a natural number can be coerced to a real number, so intuitively function *add* should also be able to apply on integer arguments. However, under the type system defined in previous chapter, such a function application will lead to a type error because the Nat type and *Real* type do not match. A subsumption typing rule solve this problem by allowing an expression of type A to have type B if A is a subset of B or values of A can be coerced to values of B.

Considering an annotated type system, subsumption enables the analysis to give a more precise result. Suppose our *add* function has binding-time type: $Nat^{\mathbf{D}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{D}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{D}}$. Then, for a expression $add\ 1\ (2+3)$, we will infer such typing:

$$\begin{array}{ll} \emptyset, [\,] \vdash^{t}_{\mathtt{BTA}} 1 : Nat^{\mathbf{D}} & \emptyset, [\,] \vdash^{t}_{\mathtt{BTA}} 2 : Nat^{\mathbf{D}} \\ \emptyset, [\,] \vdash^{t}_{\mathtt{BTA}} 3 : Nat^{\mathbf{D}} & \emptyset, [\,] \vdash^{t}_{\mathtt{BTA}} (2+3) : Nat^{\mathbf{D}} \end{array} .$$

If we have subsumption rule allowed, we will infer typings:

$$\begin{array}{ll} \emptyset, [\,] \vdash^{t}_{\mathtt{BTA}} 1 : Nat^{\mathbf{S}} & \emptyset, [\,] \vdash^{t}_{\mathtt{BTA}} 2 : Nat^{\mathbf{S}} \\ \emptyset, [\,] \vdash^{t}_{\mathtt{BTA}} 3 : Nat^{\mathbf{S}} & \emptyset, [\,] \vdash^{t}_{\mathtt{BTA}} (2+3) : Nat^{\mathbf{S}} \end{array}$$

because a static value can also be seen as a dynamic value. To increase the expressive power of the analysis, we extend our type system with a subsumption rule:

$$[\text{Sub}] \qquad \frac{C, \hat{\Gamma} \ \vdash^{t}_{\texttt{BTA}} \ t : \hat{\tau}_1 \qquad C \vdash \ \hat{\tau}_1 \leq_t \hat{\tau}_2}{C, \hat{\Gamma} \ \vdash^{t}_{\texttt{BTA}} \ t : \hat{\tau}_2}$$

It indicates that if a term has type $\hat{\tau}_1$ also has type $\hat{\tau}_2$ if between these two types, there exists an ordering relation which is abstracted by notation $\leq_t$. Such an ordering between binding-time types is also a kind of inequation so that can be formulated into our existing constraints system with a little refining.

$$c \in \mathbf{Con} \qquad ::= \varphi_1 \leq_a \varphi_2 \mid \varphi \leq_h \hat{\tau} \mid \hat{\tau}_1 \leq_t \hat{\tau}_2.$$

The rules for deriving the new constraints are defined in two ways, subeffecting and subtyping.

5.2 Subeffecting

Subeffecting is a kind of subsumption, where only the effects of terms are concerned when defining the partial order of types. In our binding-time analysis, the effect of a term is represented by the toplevel annotation of its binding-time type, so the ordering between two types is specified with the ordering of effects or annotations. Table 8 shows the constraint rules extended with subeffecting. Similar to the definition of effect ordering $\leq_a$, $\leq_t$ also has the properties of reflexivity and transitivity. The rules $[CT - Atom]$ and $[CT - Fun]$ indicate that the ordering of binding-time types is uniquely determined by their toplevel annotations. For example we could derive the following constraint from our constraints system.

$C \vdash Nat^{\mathbf{S}} \leq_t Nat^{\mathbf{D}}$ $\qquad$ $C \vdash Nat^{\mathbf{D}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{D}} \leq_t Nat^{\mathbf{D}} \xrightarrow{\mathbf{D}} Nat^{\mathbf{D}}$

[C-Sta]	$C \vdash \mathbf{S} \leq_a \varphi$	$C \vdash \mathbf{S} \leq_h \hat{\tau}$
[C-Dyn]	$C \vdash \varphi \leq_a \mathbf{D}$	
[C-Elem]	$\dfrac{c \in C}{C \vdash c}$ $\quad \texttt{if } c = \begin{cases} \alpha \leq_a \varphi \\ \varphi \leq_a \alpha \\ \varphi \leq_h \beta \\ \beta \leq_t \hat{\tau} \\ \hat{\tau} \leq_t \beta \end{cases}$	
[CA-Refl]	$C \vdash \varphi \leq_a \varphi$	
[CA-Trans]	$\dfrac{C \vdash \varphi_1 \leq_a \varphi_2 \quad C \vdash \varphi_2 \leq_a \varphi_3}{C \vdash \varphi_1 \leq_a \varphi_3}$	
[CH-Chan]	$\dfrac{C \vdash \varphi_1 \leq_a \varphi_2 \quad \lvert\hat{\tau}\rvert = \varphi_2}{C \vdash \varphi_1 \leq_h \hat{\tau}}$	
[CT-Atom]	$\dfrac{C \vdash \varphi_1 \leq_a \varphi_2}{C \vdash Nat^{\varphi_1} \leq_t Nat^{\varphi_2}}$	$\dfrac{C \vdash \varphi_1 \leq_a \varphi_2}{C \vdash Bool^{\varphi_1} \leq_t Bool^{\varphi_2}}$
[CT-Fun]	$\dfrac{C \vdash \varphi_1 \leq_a \varphi_2 \qquad C \vdash \varphi_2 \leq_h \hat{\tau}_1 \qquad C \vdash \varphi_2 \leq_h \hat{\tau}_2}{C \vdash \hat{\tau}_1 \xrightarrow{\varphi_1} \hat{\tau}_2 \leq_t \hat{\tau}_1 \xrightarrow{\varphi_2} \hat{\tau}_2}$	
[CT-Refl]	$C \vdash \hat{\tau} \leq_t \hat{\tau}$	
[CT-Trans]	$\dfrac{C \vdash \hat{\tau}_1 \leq_a \hat{\tau}_2 \quad C \vdash \hat{\tau}_2 \leq_a \hat{\tau}_3}{C \vdash \hat{\tau}_1 \leq_a \hat{\tau}_3}$	
[C-Cons]	$\dfrac{\forall c \in C_2 \;.\; C_1 \vdash c}{C_1 \models C_2}$	

Table 10: Binding-time constraint rules with subeffecting

5.3 Subtyping

The partial order of annotated types defined within subeffecting is simple because it only concerns toplevel annotations of types. A more complicate but more powerful form of subsumption is subtyping where the partial order of types is indicated by extending the ordering of effects to a shape-comformant ordering on types. The only difference between subeffecting and subtyping is the [CT-Fun] rules:

$$\frac{C \vdash \hat{\tau}_1' \leq_t \hat{\tau}_1 \quad C \vdash \varphi \leq_a \varphi' \quad C \vdash \hat{\tau}_2 \leq_t \hat{\tau}_2' \quad C \vdash \varphi \leq_h \hat{\tau}_2 \quad C \vdash \varphi' \leq_h \hat{\tau}_1' \quad C \vdash \varphi' \leq_h \hat{\tau}_2'}{C \vdash \hat{\tau}_1 \xrightarrow{\varphi_1} \hat{\tau}_2 \leq_t \hat{\tau}_1 \xrightarrow{\varphi_2} \hat{\tau}_2} \text{[CT-Fun]}$$

It shows that the ordering of effects is applied recursively into arguments types and result types. The ordering is called shape-comformant because the two types within a subtyping

or subeffecting relation have the same structure of their underlying types. In our case, specifically the two types have the same underlying types for both subtyping and subeffecting. So we do not capture such subsumption relation as $Nat^{\mathbf{S}} \leq_t Real^{\mathbf{S}}$, because our interests focus on the ordering on analysis properties.

From the definitions, it is easy to prove that a subeffecting relation is also a subtyping relation. Subtyping expresses richer relations on types than subeffecting. For example, we can infer the following ordering on types:

$$C \vdash Nat^{\mathbf{D}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{S}} \leq_t Nat^{\mathbf{S}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{S}} \text{ or } C \vdash Nat^{\mathbf{D}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{S}} \leq_t Nat^{\mathbf{D}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{D}},$$

whereas we cannot derive any types that is bigger than $Nat^{\mathbf{D}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{S}}$ within subeffecting.

5.4 Algorithm

To adape our algorithm to the subsumption rule, one method is to apply it to each syntax-directed case, which is safe but introduce unnecessary resource consumption. Similar to the situation of introducing polymorphism, we want to find out only part of the syntax cases where applying the subsumption rule has the same expressive power as applying subsumption rule to all syntax cases. The subsumption rule increases the expressive power of the analysis by allowing the type of a term to be static when it is unified with a dynamic term. So we only need to use subsumption rule where there are unifications. Three cases are included, application case, operation case and *if then else* case. The adaped algorithm is shown below.

$$
\begin{array}{lll}
\mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_1\ t_2) = & \text{let} & \beta_2, \beta_0, \alpha_f \ \mathtt{be\ fresh} \\
& & (\hat{\tau}_1, \theta_1, C_1) = \mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_1) \\
& & (\hat{\tau}_2, \theta_2, C_2) = \mathcal{W}_{\mathtt{BTA}}(\theta_1\ \hat{\Gamma}, t_2) \\
& & \theta_3 = \mathcal{U}_{\mathtt{BTA}}(\theta_2\ \hat{\tau}_1, \beta_2 \xrightarrow{\alpha_f} \beta_0) \\
& & C_3 = \hat{\tau}_2 \leq_t \beta_2 \\
& \text{in} & (\theta_3\ \beta_0, \theta_3 \circ \theta_2 \circ \theta_1,\ \theta_3(\theta_2\ C_1) \cup \theta_3\ C_2 \cup \theta_3\ C_3) \\
\mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_1\ op\ t_2) = & \text{let} & \alpha \ \mathtt{be\ fresh} \\
& & (\hat{\tau}_1, \theta_1, C_1) = \mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_1) \\
& & (\hat{\tau}_2, \theta_2, C_2) = \mathcal{W}_{\mathtt{BTA}}(\theta_1\ \hat{\Gamma}, t_2) \\
& & C_3 = \theta_2\ \hat{\tau}_1 \leq_t \tau_{op1}^{\alpha} \\
& & C_4 = \theta_3\ \hat{\tau}_2 \leq_t \tau_{op2}^{\theta_3\ \alpha} \\
& \text{in} & (\tau_{op}^{\alpha},\ \theta_2 \circ \theta_1,\ \theta_2\ C_1 \cup C_2 \cup C_3 \cup C_4) \\
\mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3) = & & \\
\quad \text{let} & \alpha, \beta \ \mathtt{be\ fresh} & \\
& (\hat{\tau}_1, \theta_1, C_1) = \mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_1) & \\
& (\hat{\tau}_2, \theta_2, C_2) = \mathcal{W}_{\mathtt{BTA}}(\theta_1\ \hat{\Gamma}, t_2) & \\
& (\hat{\tau}_3, \theta_3, C_3) = \mathcal{W}_{\mathtt{BTA}}(\theta_2(\theta_1\ \hat{\Gamma}), t_3) & \\
& \theta_4 = \mathcal{U}_{\mathtt{BTA}}(\theta_3(\theta_2\ \hat{\tau}_1), bool^{\alpha}) & \\
& C_4 = \theta_4(\theta_3\ \hat{\tau}_2) \leq_t \beta & \\
& C_5 = \theta_4\ \hat{\tau}_3 \leq_t \beta & \\
\quad \text{in} & (\theta_4\ \hat{\tau}_3,\ \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1, & \\
& \quad \theta_4(\theta_3(\theta_2\ C_1)) \cup \theta_4(\theta_3\ C_2) & \\
& \quad \cup \theta_4\ C_3 \cup \theta_4\ \alpha \leq_h \theta_4\ \hat{\tau}_3 \cup C_4 \cup C_5) &
\end{array}
$$

In [13], Geoffrey designed a similar type inference algorithm for a polymorphic language with subtyping and proved its syntax soundness and completeness. Although we do not give a formal prove for our algorithm, we have the faith that it is consistent with our specification.

5.5 Defaulting

When we generalise the type of local definition, we also generalise the constraints that are generated from the local definition. Then these constraints are instantiated as many times as the local variable is instantiated, which greatly increases the number of constraints that need to be solved. However not all the constraints we generalised relate to the inference of types. For example, we have a program:

$$\textbf{let val } y = (\textbf{fun } x \Rightarrow x)\ 10\ \textbf{in } y + y + y + y + y\ \textbf{end}$$

Normally, an algorithm assigns the type $Nat^{\alpha_1} \xrightarrow{\alpha_2} Nat^{\alpha_1}$ to term ($\textbf{fun } x \Rightarrow x$) and Nat^{α_3} to 10. It also generates constraints $\alpha_2 \leq_a \alpha_1$ and $\alpha_3 \leq_a \alpha_1$. After generalization, the type schema for y is $\forall\alpha_1.(\alpha_2 \leq_a \alpha_1, \alpha_3 \leq_a \alpha_1) \Rightarrow Nat^{\alpha_1}$. Since variable y are used five times in the body of let-term, the two constraints are also instantiated into ten different constraints. However, α_2 and α_3 do not appear in the type of the local variable, the values of them do not affect the type inference outside the scope of the local definition. So we are free to choose the value of those kind of annotation variables at the point of generalization to make the constraints generated by local definition as simple as possible as long as the values we select conform those constraints where they are related. In this case, we choose both α_2 and α_3 to be $\mathbf{S}$ and $\mathbf{S} \leq_a \alpha_1$ is obviously valid, so the type schema of y is a simple $\forall\alpha_1.Nat^{\alpha_1}$. This procedure is called defaulting.

For a typical local definition $\textbf{let val } x = t_0 \textbf{ in } t \textbf{ end}$, assume $(\hat{\tau}_0, \theta_0, C_0) = \mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_0)$. The algorithm of defaulting is described in the following steps,

- Find all free annotation in $\hat{\tau}_0$ and $\hat{(}\Gamma)$, regarded as $set_1 = \{\alpha_1, ..., \alpha_m\}$.
- Find all free annotation in C_0 but not in set_1, regarded as $set_2 = \{\alpha'_1, ..., \alpha'_n\}$.
- For each annotation α'_i in set_2, find all annotations that are smaller than α'_i in set_1, regarded as set'_i.
- For each set'_i, if there exists an upper bound α''_i, generate a subsititution $\alpha'_i \mapsto \alpha''_i$.
- Combine all subsititution and simplify C_0 to C'_0.

Chapter 6

Comparison

6.1 PPX and PPE

Expressive Power. A polyvariant analysis allows us to defer the time of assigning the binding time property to a variable from the point of its definition to the point when it is used. However, it does not capture the ordering between types. The power of subeffecting is shown by allowing a function to be applied to a term which has a different type from the argument type of the function as long as these two types fulfill a certain ordering relation.

Let us consider a simple program:

$$\begin{array}{l}\textbf{let val } id = \textbf{fun } x \Rightarrow x\\ \textbf{in } id\ 1\\ \textbf{end}\end{array}$$

In a polyvariant analysis, we are able to infer the type for the id function defined in the let term: $\forall\beta,\alpha.\alpha \leq_h \beta \Rightarrow \beta \xrightarrow{\alpha} \beta$. Then it is instantiated into type $Nat^{\mathbf{D}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{D}}$ because the result type of id must be dynamic. In PPX, the term 1 has type $Nat^{\mathbf{D}}$ because id requires a dynamic integer whereas in PPE, both $Nat^{\mathbf{S}}$ and $Nat^{\mathbf{D}}$ could be the type for term 1 because both static and dynamic term is acceptable for a function which expects a dynamic argument.

Now let us see a more complicate example with a higher order function whose second argument is also the argument of its first argument,

$$\begin{array}{l}\textbf{let val } apply = \textbf{fun } f \Rightarrow \textbf{fun } x \Rightarrow f\ x\\ \textbf{in let val } id = \textbf{fun } x \Rightarrow x\\ \quad\textbf{in } apply\ id\ 1\\ \quad\textbf{end}\\ \textbf{end}\end{array}$$

So in the type system without subsumptions, the type shcema of *apply* is
$\forall\beta_1,\beta_2,\alpha_1,\alpha_2,\alpha_3.$
$\alpha_2 \leq_a \alpha_1, \alpha_2 \leq_a \alpha_3, \alpha_1 \leq_h \beta_1, \alpha_1 \leq_h \beta_2, \alpha_3 \leq_h \beta_1, \alpha_3 \leq_h \beta_2$
$\Rightarrow \qquad (\beta_1 \xrightarrow{\alpha_1} \beta_2) \xrightarrow{\alpha_2} \beta_1 \xrightarrow{\alpha_3} \beta_2$
which will further refered as τ_x. In the type system with subeffecting, the type shcema is
$\forall\beta_1,\beta_2,\beta_3,\alpha_1,\alpha_2,\alpha_3.$
$\alpha_2 \leq_a \alpha_1, \alpha_2 \leq_a \alpha_3, \alpha_1 \leq_h \beta_1, \alpha_1 \leq_h \beta_2, \alpha_3 \leq_h \beta_3, \alpha_3 \leq_h \beta_2, \beta_3 \leq_t \beta_1$

$\Rightarrow \qquad (\beta_1 \xrightarrow{\alpha_1} \beta_2) \xrightarrow{\alpha_2} \beta_3 \xrightarrow{\alpha_3} \beta_2$

which is refered as τ_e. If we limit β_3 to be β_1 then τ_e becomes τ_x. So τ_e is a more general type than τ_x and can be instanciate into more types than τ_x.

Resouce Comsumption. Although subeffecting increases the expressive power of the analysis in a certain extent, it also complicates the algorithm by introducing more constraints. The main difference between the algorithm of PPX and PPE is in the application case. In PPX,

$$\begin{array}{rll} \mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_1\ t_2) = & \text{let} & \beta_0, \alpha_f \texttt{ be fresh} \\ & & (\hat{\tau}_1, \theta_1, C_1) = \mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_1) \\ & & (\hat{\tau}_2, \theta_2, C_2) = \mathcal{W}_{\mathtt{BTA}}(\theta_1\ \hat{\Gamma}, t_2) \\ & & \theta_3 = \mathcal{U}_{\mathtt{BTA}}(\theta_2\ \hat{\tau}_1, \hat{\tau}_2 \xrightarrow{\alpha_f} \beta_0) \\ & \text{in} & (\theta_3\ \beta_0, \theta_3 \circ \theta_2 \circ \theta_1,\ \theta_3(\theta_2\ C_1) \cup \theta_3\ C_2) \end{array}$$

The key step of infering types for $(t_1\ t_2)$ in PPX is to unify the type of t_2 with the expected argument type of t_1. In PPE we have,

$$\begin{array}{rll} \mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_1\ t_2) = & \text{let} & \beta_2, \beta_0, \alpha_f \texttt{ be fresh} \\ & & (\hat{\tau}_1, \theta_1, C_1) = \mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_1) \\ & & (\hat{\tau}_2, \theta_2, C_2) = \mathcal{W}_{\mathtt{BTA}}(\theta_1\ \hat{\Gamma}, t_2) \\ & & \theta_3 = \mathcal{U}_{\mathtt{BTA}}(\theta_2\ \hat{\tau}_1, \beta_2 \xrightarrow{\alpha_f} \beta_0) \\ & & C_3 = \hat{\tau}_2 \leq_t \beta_2 \\ & \text{in} & (\theta_3\ \beta_0, \theta_3 \circ \theta_2 \circ \theta_1,\ \theta_3(\theta_2\ C_1) \cup \theta_3\ C_2 \cup \theta_3\ C_3) \end{array}$$

With subsumption rule, it is not necessary the type of t_2 has to be the same as the type of t_1 argument. We first unify t_1's argument type with a newly introduced type variable β_2 which only serves as a representitive of it. Then we generate a constraint between the type of t_2 and β_2. To make the comparsion more clear, we change the algorithm of PPX a little as PPX',

$$\begin{array}{rll} \mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_1\ t_2) = & \text{let} & \beta_2, \beta_0, \alpha_f \texttt{ be fresh} \\ & & (\hat{\tau}_1, \theta_1, C_1) = \mathcal{W}_{\mathtt{BTA}}(\hat{\Gamma}, t_1) \\ & & (\hat{\tau}_2, \theta_2, C_2) = \mathcal{W}_{\mathtt{BTA}}(\theta_1\ \hat{\Gamma}, t_2) \\ & & \theta'_3 = \mathcal{U}_{\mathtt{BTA}}(\theta_2\ \hat{\tau}_1, \beta_2 \xrightarrow{\alpha_f} \beta_0) \\ & & \theta_3 = \mathcal{U}_{\mathtt{BTA}}(\theta'_3\ \beta_2, \hat{\tau}_2) \circ \theta'_3 \\ & \text{in} & (\theta_3\ \beta_0, \theta_3 \circ \theta_2 \circ \theta_1,\ \theta_3(\theta_2\ C_1) \cup \theta_3\ C_2) \end{array}$$

This increases one more unification by using the intermedia type variable β_2. The complexity difference between algorithm with and without subsumption lies on that between constraints and unifications. In PPE, it generates more constraints than in PPX but less unifications. In most cases, a constraint is more complicated to deal with than a unification. For example, the result of a unification $\mathcal{U}_{\mathtt{BTA}}(\beta, Nat^{\mathbf{D}} \xrightarrow{\alpha} Nat^{\mathbf{D}})$ is simply a subsititution mapping from β to that function type whereas a constraint $\beta \leq_t Nat^{\mathbf{D}} \xrightarrow{\alpha} Nat^{\mathbf{D}}$ will result in a subsititution $[\beta \mapsto Nat^{\mathbf{D}} \xrightarrow{\alpha'} Nat^{\mathbf{D}}]$ and a new constraint $\alpha' \leq_a \alpha$.

6.2 PPE and PPT

Expressive Power. The difference between subtyping and subeffecting is how they define the parital ordering, or subtype relation, between types. In our type system, two

type constants Nat and $Bool$ and one compound type, function type, are presented. In the cases of type constants, both subtyping and subeffecting share the same definition. For function types, subtyping applies the subtype relation recursively into the argument and result types. As shown in previous chapter, in our inference algorithm, the subsumption rule is used in the case of function application, where parameter's type can be smaller than the argument type of the function. So subtyping may be beneficial more than subeffecting the function is a higher function, that it takes another function as arguments.

Considering the following example:

$$
\begin{array}{l}
\textbf{let val } apply = \textbf{fun } f \Rightarrow \textbf{fun } x \Rightarrow f\ x \\
\textbf{in let dyn val } g :: (Nat \rightarrow Nat) \rightarrow Nat \\
\quad \textbf{in } apply\ g\ (\textbf{fun } x \Rightarrow 1) \\
\quad \textbf{end} \\
\textbf{end}
\end{array}
$$

Function *apply* is defined the same as in last example, so it has type:
$\forall \beta_1, \beta_2, \beta_3, \alpha_1, \alpha_2, \alpha_3.$
$\alpha_2 \leq_a \alpha_1, \alpha_2 \leq_a \alpha_3, \alpha_1 \leq_h \beta_1, \alpha_1 \leq_h \beta_2, \alpha_3 \leq_h \beta_3, \alpha_3 \leq_h \beta_2, \beta_3 \leq_t \beta_1$
$\Rightarrow \qquad (\beta_1 \xrightarrow{\alpha_1} \beta_2) \xrightarrow{\alpha_2} \beta_3 \xrightarrow{\alpha_3} \beta_2$
It takes a dynamic function as its first argument, where the dynamic function is also a higher order function that takes a function from natural number to natural number as its argument. So we can infer that β_1 is instantiated to $Nat^{\mathbf{D}} \xrightarrow{\mathbf{D}} Nat^{\mathbf{D}}$ and β_2 is instantiated to $Nat^{\mathbf{D}}$. Assume the second argument of *apply*, the term **fun** $x \Rightarrow 1$, has type β_4, then we have subtype relations: $\beta_4 \leq_t \beta_3$ *and* $\beta_3 \leq_t \beta_1$ which can be simplified to $\beta_4 \leq_t Nat^{\mathbf{D}} \xrightarrow{\mathbf{D}} Nat^{\mathbf{D}}$. With subeffecting system, β_4 can only be subsitituted to $Nat^{\mathbf{D}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{D}}$ so the term 1 is marked as a dynamic integer. However, in subtyping system, β_4 can be subsitituted to $Nat^{\mathbf{D}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{S}}$ according to the constraints rules and the term 1 in this way is a static integer.

Resouce Comsumption. Subtyping and Subeffecting have different algorithm for resolving constraints between compound types which are function types in our type system. These constraints can be categorized into two forms which is either $\beta \leq_t \hat{\tau}_1 \xrightarrow{\alpha} \hat{\tau}_2$ or $\hat{\tau}_1 \xrightarrow{\alpha} \hat{\tau}_2 \leq_t \hat{\tau}_1' \xrightarrow{\alpha'} \hat{\tau}_2'$. Let us first consider a simple case of the first form:

$$\beta \leq_t Nat^{\mathbf{S}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{D}}.$$

In subeffecting system, the algorithm generates a subsititution $[\beta \mapsto Nat^{\mathbf{S}} \xrightarrow{\alpha} Nat^{\mathbf{D}}]$ for β and a new constraint $\alpha \leq_a \mathbf{S}$. While in subtyping, it generates a subsititution $[\beta \mapsto \beta_1 \xrightarrow{\alpha} \beta_2]$ and three more constraints $\alpha \leq_a \mathbf{S}$, $Nat^{\mathbf{S}} \leq_t \beta_1$ and $\beta_2 \leq_t Nat^{\mathbf{D}}$ where the latter two further generate two new subsititution $\beta_1 \mapsto Nat^{\alpha_1}$ and $\beta_2 \mapsto Nat^{\alpha_2}$ and two new constraints $\mathbf{S} \leq_a \alpha_1$ and $\alpha_2 \leq_a \mathbf{D}$. In a general case,

$$\beta \leq_t \hat{\tau}_1 \xrightarrow{\alpha} \hat{\tau}_2,$$

applying subeffecting algorithm always results in one subsititution and one constraint. But the algorithm of subtyping will generate at least three subsititutions and five constraints when $\hat{\tau}_1$ and $\hat{\tau}_2$ are basic types and even more subsititutions and constraints if $\hat{\tau}_1$ or $\hat{\tau}_2$ is also function type. Roughly the number of constraints the algorithm resolves for a single constraint $\beta \leq_t \hat{\tau}_1 \xrightarrow{\alpha} \hat{\tau}_2$ is proportional to the complexity of the function type $\hat{\tau}_1 \xrightarrow{\alpha} \hat{\tau}_2$.

To resolve a constraint of form $\hat{\tau}_1 \xrightarrow{\alpha} \hat{\tau}_2 \leq_t \hat{\tau}_1' \xrightarrow{\alpha'} \hat{\tau}_2'$, both system generate the constraint $\alpha \leq_a \alpha'$ and after that subtyping system generates another two constraints $\hat{\tau}_1' \leq_t \hat{\tau}_1$ and $\hat{\tau}_2 \leq_t \hat{\tau}_2'$ whereas subeffecting system only perform two unifications $\mathcal{U}_{\texttt{BTA}}(\hat{\tau}_1, \hat{\tau}_1')$ and $\mathcal{U}_{\texttt{BTA}}(\hat{\tau}_2, \hat{\tau}_2')$. So it still takes more resource to resolve constraints in a subtyping system.

6.3 Examples

In this section, we use a list of examples which explore all the language features including lambda abstration, function application, local definition and *if then else* control structure. We show the analysis result of these examples with three techniques: PPX, PPE and PPT. For simple examples, we also show the result by annotating each subterm of the program with the toplevel annotation of the type of that subterm directily. For complete examples, we only give the statistic result.
Example 1 : Simple lambda abstraction

$$\textbf{fun}\ x \Rightarrow x$$

All the three analysis annotated the program to be $(\textbf{fun}\ x \Rightarrow x)^{\mathbf{D}}$ and the statistic data is:

	PPX	PPE	PPT
Number of terms	2	2	2
Number of static terms	0	0	0
Number of static application	0	0	0
Number of constraints	1	1	1
Number of unification	0	0	0

Example 2 : Simple function application

$$(\textbf{fun}\ x \Rightarrow x)\ (42 + 12)$$

The analysis result of PPX is $((\textbf{fun}\ x \Rightarrow x^{\mathbf{D}})^{\mathbf{S}}\ (42^{\mathbf{D}} + 12^{\mathbf{D}})^{\mathbf{D}})^{\mathbf{D}}$ while both PPE and PPT annotate the program to be $((\textbf{fun}\ x \Rightarrow x^{\mathbf{D}})^{\mathbf{S}}\ (42^{\mathbf{S}} + 12^{\mathbf{S}})^{\mathbf{S}})^{\mathbf{D}}$. The statistic data is:

	PPX	PPE	PPT
Number of terms	6	6	6
Number of static terms	1	4	4
Number of static application	1	1	1
Number of constraints	1	8	10
Number of unification	5	3	3

Example 3 : if then else structure

$$\textbf{if}\ \mathit{true}\ \textbf{then}\ 1\ \textbf{else}\ 2$$

The analysis result of PPX is $(\textbf{if}\ \mathit{true}^{\mathbf{S}}\ \textbf{then}\ 1^{\mathbf{D}}\ \textbf{else}\ 2^{\mathbf{D}})^{\mathbf{D}}$ while both PPE and PPT annotate the program to be $(\textbf{if}\ \mathit{true}^{\mathbf{S}}\ \textbf{then}\ 1^{\mathbf{S}}\ \textbf{else}\ 2^{\mathbf{S}})^{\mathbf{D}}$. The statistic data is:

	PPX	PPE	PPT
Number of terms	4	4	4
Number of static terms	1	3	3
Number of static application	0	0	0
Number of constraints	1	6	6
Number of unification	2	1	1

Example 4 : Higher order function

$$((\textbf{fun}\ f \Rightarrow (\textbf{fun}\ x \Rightarrow (f\ x)))\ (\textbf{fun}\ y \Rightarrow y))\ 42$$

The analysis result of PPX is:

$$(((\textbf{fun}\ f \Rightarrow (\textbf{fun}\ x \Rightarrow (f^{\mathbf{S}}\ x^{\mathbf{D}})^{\mathbf{D}})^{\mathbf{S}})^{\mathbf{S}}\ (\textbf{fun}\ y \Rightarrow y^{\mathbf{D}})^{\mathbf{S}})^{\mathbf{S}}\ 42^{\mathbf{D}})^{\mathbf{D}}$$

PPE shows the result of:

$$(((\textbf{fun}\ f \Rightarrow (\textbf{fun}\ x \Rightarrow (f^{\mathbf{S}}\ x^{\mathbf{S}})^{\mathbf{D}})^{\mathbf{S}})^{\mathbf{S}}\ (\textbf{fun}\ y \Rightarrow y^{\mathbf{D}})^{\mathbf{S}})^{\mathbf{S}}\ 42^{\mathbf{S}})^{\mathbf{D}}$$

PPT annotates the program to be:

$$(((\textbf{fun}\ f \Rightarrow (\textbf{fun}\ x \Rightarrow (f^{\mathbf{S}}\ x^{\mathbf{S}})^{\mathbf{D}})^{\mathbf{S}})^{\mathbf{S}}\ (\textbf{fun}\ y \Rightarrow y^{\mathbf{S}})^{\mathbf{S}})^{\mathbf{S}}\ 42^{\mathbf{S}})^{\mathbf{D}}$$

The statistic data is:

	PPX	PPE	PPT
Number of terms	10	10	10
Number of static terms	5	7	8
Number of static application	3	3	3
Number of constraints	5	12	27
Number of unification	9	7	7

This example shows that with higher order function involved, PPE and PPT may have different expressive power on the number of static terms.

Example 5 : nested branch structure

if $true == (\textbf{let dyn val}\ x :: Nat\ \textbf{in}\ (\textbf{if}\ 1 == 1\ \textbf{then}\ false\ \textbf{else}\ x == 2)\ \textbf{end})$
then 10
else $10 + (\textbf{if}\ 2 == 1\ \textbf{then}\ 1\ \textbf{else}\ 2)$

The statistic data is:

	PPX	PPE	PPT
Number of terms	21	21	21
Number of static terms	6	15	15
Number of static application	0	0	0
Number of constraints	1	36	45
Number of unification	16	3	3

Example 6 : nested local definition

```
let   val x = 1
 in     let   val id = fun z ⇒ z
         in     if       x == 10
                then     if       (fun x ⇒ x + 1) 9 == x
                         then     id x
                         else     id id x
                else     let val succ = fun x ⇒ x + 1 in succ x end
        end
end
```

The statistic data is:

	PPX	PPE	PPT
Number of terms	34	34	34
Number of static terms	16	25	25
Number of static application	5	5	5
Number of constraints	1	44	80
Number of unification	27	15	15

Example 7 : different number of static function application
The six examples above shows no difference in the number of static applications between PPE and PPT. That is because they all have very simple dynamic terms. In the next example, we define a complicated dynamic variable. As a consequence, we have different number of static applications between PPT and PPE.

let **dyn val** $f :: (Nat \to Nat \to Nat) \to Nat$
in **let** **val** $id = \textbf{fun}\ z \Rightarrow z$
in **let** **val** $add = \textbf{fun}\ x \Rightarrow \textbf{fun}\ y \Rightarrow x + y$
in $(\textbf{fun}\ x \Rightarrow \textbf{fun}\ y \Rightarrow \textbf{fun}\ z \Rightarrow f\ x + x\ y\ z)\ add\ 1\ 2$
end
end
end

The statistic data is:

	PPX	PPE	PPT
Number of terms	28	28	28
Number of static terms	5	12	13
Number of static application	3	4	5
Number of constraints	5	37	70
Number of unification	26	14	14

In this example, the dynamic variable we introduce is a higher order function. So when unifying function *add* with the argument of f, PPE and PPT have different results. Since subeffecting only allows weakening on top level annotation, the type for *add* is $Nat^{\mathbf{D}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{D}} \xrightarrow{\mathbf{D}} Nat^{\mathbf{D}}$ whereas in PPT, the type of *add* is infered as $Nat^{\mathbf{D}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{D}} \xrightarrow{\mathbf{S}} Nat^{\mathbf{D}}$.

Chapter 7

Conclusion

We conclude by summarise what we have done in this thesis and discuss some future work.

7.1 Summary

In this thesis, we first proposed a list of specifications of binding time analysis and the corresponding inference algorightms, which specifically are:

- A monovariant analysis for a monomorphic source language with no subsumption (MMX)
- A polyvariant analysis for a polymorphic source language with no subsumption (PPX)
- A polyvariant analysis for a polymorphic source language with subeffecting subsumption (PPE)
- A polyvariant analysis for a polymorphic source language with subtyping subsumption (PPT)

Then we compare the expressive power and resource consumption between two pairs of analyses:

- between PPX and PPE
- between PPE and PPT

We identify the expressive power that the subsumption rule adds to and the difference between the two kinds of subsumption, subeffecting and subtyping. We observe that when higher order functions are involved, PPE and PPT may differ in the number of static terms and only when dynamic values contain higher order functions, PPE and PPT may differ in the number of static applications.

7.2 Future Work

In this thesis, we focus on implementing a binding time analysiser and discussing the result of the analysiser. A futrue work could be implementing a partial evaluator[8]. Because in

Figure 7.1: A Partial Evaluation System

practice, a binding time analysiser is always combined with the use of a partial evaluator[8]. The binding time analysiser calculate the binding time for each term of a program, thus generates an annotated program. The partial evaluator then takes the annotated program as input and evaluates it, specialising static terms into intermediate values. At last the partial evaluator generate an residual program which has the same dynamic behavior as the original program but more efficience. Figure 1 illustrate the workflow of a analysiser and a partial evaluator.

Bibliography

[1] Neil Vachharajani Bolei Guo and David I. Shape analysis with inductive recursion synthesis. 2007.

[2] L. Damas and R. Milner. Principal type schemes for functional programs. 1982.

[3] Fritz Henglein Dirk Dussart and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. 1995.

[4] Rogardt Heldal and John Hughes. Binding-time analysis for polymorphic types. 2001.

[5] Martin Sulzmann Kevin Glynn, Peter J. Stuckey and Harald Soendergaard. Boolean constraints for binding-time analysis. 2001.

[6] Flemming Nielson Kirsten Lackner Solberg Gasser, Hanne Riis Nielson. Strictness and totality analysis. 1998.

[7] Robin Milner. A theory of type polymorphism in programming. 1978.

[8] Carsten K. Gomard Neil D. Jones and Peter Sestoft. Partial evaluation and automatic program generation. 1993.

[9] Flemming Nielson and Hanne Riis Nielson. Type and effect system. 1999.

[10] Flemming Nielson and Hanne Riis Nielson. Principles of program analysis. 2005.

[11] I. V. Ramakrishnan R. Sekar and P. Mishra. On the power and limitations of strictness analysis. 1997.

[12] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. 1953.

[13] Geoffrey Seward Smith. Polymorphic type inference for languages with overloading and subtyping.