QuB: A Resource Aware Functional Programming Language

By

Apoorv Ingle

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

Committee members

Dr. J. Garrett Morris, Chairperson

Dr. Perry Alexander

Dr. Andy Gill

Dr. Prasad Kulkarni

Date defended: July 21, 2018

The Thesis Committee for Apoorv Ingle certifies
that this is the approved version of the following thesis :

QuB: A Resource Aware Functional Programming Language

Dr. J. Garrett Morris, Chairperson

Date approved: July 21, 2018

Abstract

Modern programming languages treat resources as normal values. The static semantics of resources in such languages does not match with runtime semantics. Type systems based on linear logic (Girard, 1987) provide a modus operandi to help resource management. While these systems are more expressive than standard Hindley-Milner type system, there are various practical issues that prohibit leveraging it to its fullest degree in programming languages. In this thesis, we tackle the resource management problem by making resources first class citizens in the language, and concentrating on sharing or separation of resources.

We design and implement QuB which is a Curry-Howard interpretation of the logic based on the logic of bunched implications (***BI***) (O'Hearn, 2003). We introduce two kinds of program objects —restricted and unrestricted—and two kinds of functions—sharing and separating. The restricted objects model resources and the unrestricted values model program objects that do not contain any resources. Sharing functions—denoted by $\twoheadrightarrow$ connective—imply that functions share resources with its arguments, while separating arrow—denoted by $-\!\!*$ connective—imply that functions do not share resources with its arguments. We extended our work based on Quill (Morris, 2016), a functional language with linear qualifed types. We show how the use of monads with sharing and separating functions helps statically detecting resource errors while modeling exceptions that is difficult to capture in linear languages.

Acknowledgements

TODO TODO TODO DO DO DO DO.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Resources are treated as normal values in Hindley-Milner based type systems. Thus, statically typed modern programming languages such as Haskell, ML, etc. do little to track resource usage at compile time to detect errors, such as, not closing open file handles or closing the file handle more than once. Substructural logic systems such as linear logic, restrict the use of structural rules—contraction and weakening—to view values as resources. There have been several attempts to create a practical linear type system for programming languages but they have not been included in main stream languages due to practical limitations. We develop a type system based on the logic of bunched implications and implement a type inference algorithm $\mathcal{M}$ for the language. We then extend the system with kinds that allows users to define their own datatypes that expresses sharing and separation between fields.

The thesis is organized in the following manner: Chapter 3 illustrates how programming in QuB is different than programming in other languages like haskell by giving concrete examples. Chapter 4 and Chapter 5 describe the core language of type and terms and gives details about the type system, syntax directed type system with a type inference algorithm. Chapter 6 extends the language with kinds that enables users to define custom datatypes similar to GADTs. Finally, Chapter 7 gives a direction to future work.

The contributions made in the thesis are listed below:

- QuB, A type system based on logic of ***BI*** that treates resources as first class citizens in the programming language. We introduce two kinds of program objects—restricted and unrestriced—and two kinds of functions—the ones that share resrouces with their arguments

and the other that do not.

- Design a sound and complete syntax directed type system and implement a type inference algorithm based on modified Algorithm $\mathcal{M}$.
- Examples related to file handling and exceptions that show how QuB is more expressive than standard Hindley-Milner type system for by tracing resource use at compile time to detect anomalies.

Chapter 2

Background Work

2.1 Hindley-Milner Type System and Type Inference Algorithm

Hindley-Milner (**HM**) type system (1978) for lambda calculus extended with parametric polymorphism or restricted version of System F (Girard et al., 1989) forms the basis many modern functional programming languages such as Haskell, ML, etc. The type language contains type variables, primitive types (such as integers, floats), the type constructor ($\rightarrow$)— which constructs function types—and type scheme (σ) as shown in Fig. 2.1. The expression language contains variables, lambda expressions and applications extended with a polymorphic `let` construct. Type inference algorithm $\mathcal{W}$ (Damas & Milner, 1982) and its variant type checking algorithm $\mathcal{M}$ (Lee & Yi, 1998) is decidable in the sense, the algorithm always completes with a success or failure. The algorithms also guarantee a most general typing scheme or principal types for an expression.

$$
\begin{array}{rl}
 & t, u, \upsilon \in \text{Type Variables} \\
\text{Types} & \tau ::= t \mid \iota \mid \tau \rightarrow \tau \\
\text{Typing Scheme} & \sigma ::= \tau \mid \forall t.\tau \\
\text{Expressions} & M, N ::= x \mid \lambda x.M \mid MN \mid \texttt{let}\ x = M\ \texttt{in}\ N
\end{array}
$$

Figure 2.1: Hindley-Milner Type and Expression Language

Robinson's (1965) unification algorithm plays a key role in computation of well-formed types. Its purely syntactic approach in creating substitutions to unify types keeps the complete process elegant. Algorithm $\mathcal{M}$ works in an interesting way where the types of all well-typed terms can

be inferred automatically and if types are specified, the same algorithm can be used to verify that the specified type of expression term matches the actual type. The same can be obtained using algorithm $\mathcal{W}$ with an additional machinery of computing equality over types.

$$\frac{x:\sigma \in \Gamma}{\Gamma \vdash x:\sigma}\ [\text{VAR}] \qquad \frac{\Gamma \vdash M:\sigma \qquad \Gamma_x, x:\sigma \vdash N:\tau}{\Gamma \vdash (\texttt{let}\ x = M\ \texttt{in}\ N):\tau}\ [\text{LET}]$$

$$\frac{\Gamma \vdash M:\sigma \qquad t \notin \texttt{fvs}(\Gamma)}{\Gamma \vdash M:\forall t.\sigma}\ [\forall\ \text{I}] \qquad \frac{\Gamma \vdash M:\sigma \qquad (\sigma' \sqsubseteq \sigma)}{\Gamma \vdash M:\sigma'}\ [\forall\ \text{E}]$$

$$\frac{\Gamma_x, x:\tau \vdash M:\tau'}{\Gamma \vdash \lambda x.M:\tau \to \tau'}\ [\to \text{I}] \qquad \frac{\Gamma \vdash M:\tau \to \tau' \qquad \Gamma \vdash N:\tau}{\Gamma \vdash MN:\tau'}\ [\to \text{E}]$$

Figure 2.2: Typing Rules for **HM** Type System

The rules for type inference for **HM** type system are shown in Fig. 2.2. Γ is the collection of assumptions or context in which the expression is typed. The [VAR] rule is tautology or a simple lookup of the term variable x in the context Γ. [→I] and [→E] are rules for typing lambda terms and application respectively. We also include the rules for parametric polymorphism where [∀I] generalizes the type scheme by adding implicit universal quantifiers and [∀E] generates an instance of the type scheme by substituting free type variables. It is worthwhile to note that when we generalize over types using the rule [∀I] we need to ensure that the new type variable introduced should not be free in the existing typing context, hence, the side condition of $t \notin \texttt{fvs}(\Gamma)$ where $\texttt{fvs}(\Gamma)$ denotes free type variables in Γ. The [LET] allows implicit parametric polymorphism. For example, the expression, `g = λf. (f True, f 1)` will raise a type error as the type inference algorithm will not be able to unify the type of expression `f True : Bool` $\to \alpha$ with `f 1 : Int` $\to \beta$. However there indeed exists a polymorphic type for $\forall\alpha.\alpha \to \alpha$ that can be used to type check expression `g`. The `let` construct comes handy in such cases. The expression `g` can be defined as `g = let f = λ x. x in (f True, f 1)`. The type of the expression `f` is now computed as `f:` $\forall\alpha.\ \alpha \to \alpha$ and `g` is assigned a type `(Bool, Int)`.

The Curry-Howard correspondence of **HM** type system corresponds to the second order implication fragment of intuitionistic propositional logic. The grammar of intuitionistic logic is shown

$$A, B, C ::= X \mid A \to B \mid A \times B \mid A + B$$

Figure 2.3: Grammar for Intuitionistic Logic

$$\frac{}{\Phi \vdash \Phi}\ [\text{Ax}] \qquad \frac{\Gamma \vdash \Phi}{\Gamma, \Delta \vdash \Phi}\ [\text{WKN}] \qquad \frac{\Gamma, \Gamma \vdash \Phi}{\Gamma \vdash \Phi}\ [\text{CTR}]$$

$$\frac{\Psi \qquad x \notin \Psi}{\forall x.\Psi}\ [\forall \text{I}] \qquad \frac{\forall x.\Psi \qquad \Gamma}{\Psi[x/\Gamma]}\ [\forall \text{E}]$$

$$\frac{\Gamma \vdash \Phi \qquad \Phi \vdash \Psi}{\Gamma \vdash \Phi \supset \Psi}\ [\supset \text{I}] \qquad \frac{\Gamma \vdash \Phi \supset \Psi \qquad \Gamma \vdash \Phi}{\Gamma \vdash \Psi}\ [\supset \text{E}]$$

Figure 2.4: Logic rules: Implication fragment of second order Intuitionistic Propositional Logic

in Fig. 2.3 where $A \to B$ denotes implication, $A \times B$ denotes conjunction and $A + B$ denotes disjunction. The rules of the logic system are shown in Fig. 2.4 in Gentzen style natural deduction where Γ, Φ and Ψ are propositions. The [Ax] rule corresponds to [ID] rule while [→I] and [→E] correspond to [⊃I] and [⊃E] respectively. The structural rules of weakening [WKN] and contraction [CTR] are implicitly obeyed by **HM** type-system. The weakening rule states that we can add unrelated assumptions to derivations without affecting the proofs while contraction states that we can discard duplicate assumptions in our derivations and the proof will still hold. The [∀I] introduces the universal quantifier for a proposition and [∀E] instantiates the proposition. Both the rules correspond to parametric polymorphism introduction [∀I] and elimination [∀E] rules in Fig. 2.2

The algorithm $\mathcal{M}$ is given in figure Fig. 2.5. $\mathcal{U}$ is the Robinson's unification algorithm that computes the most general unifier required to unify two types and $\text{Gen}(\Gamma, \tau)$ generalizes a type to type scheme. All the type variables denoted by u are fresh and do not shadow the existing type variables in the context.

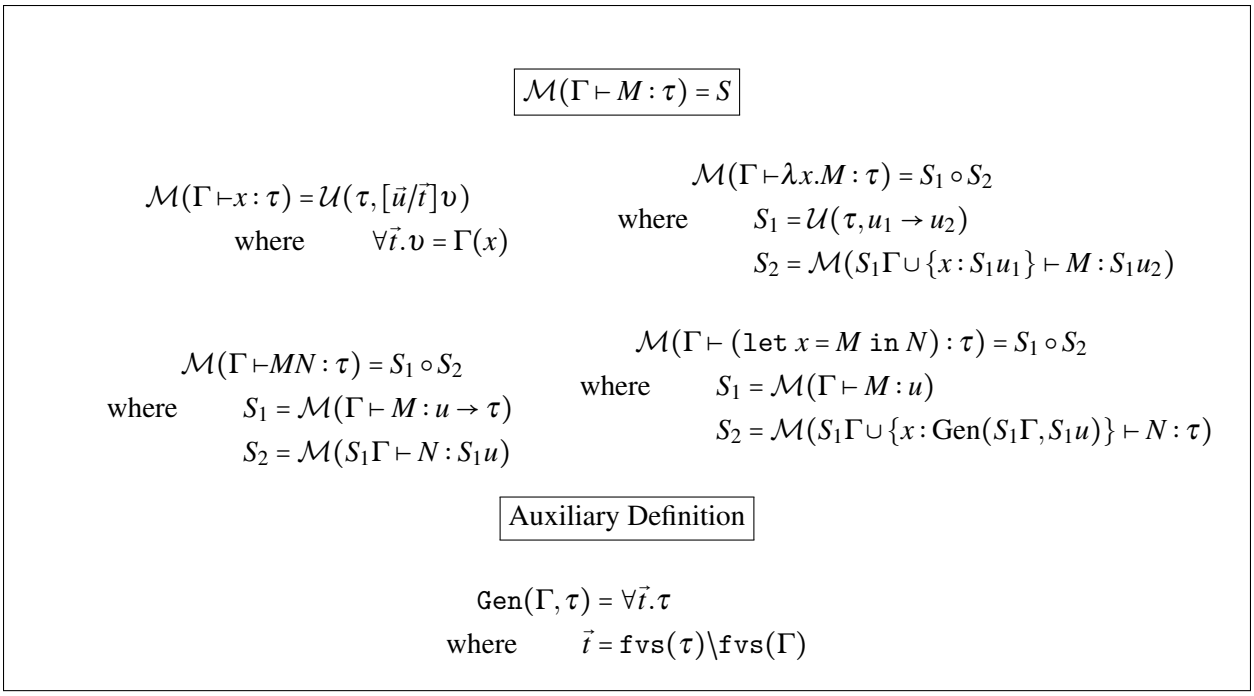


Figure 2.5: Algorithm $\mathcal{M}$ for **HM** type system

2.2 Linear Logic

While propositional logic deals with truth of propositions and their connectives, linear logic (Girard, 1987) deals with availability of resources. Linear logic promises to help cope with the resource and resource control problem. The core idea is that propositions cannot be freely duplicated or discarded as allowed in intuitionistic logic. In formal terms, the contraction and weakening logical rules are restricted. This instigates a view of propositions that behave like resources. In real world software applications, resources may not be freely copied or dropped from a program context. While handling entities like database connections, file handles or even in memory shared state can introduce bugs in industry grade software. Linear logic is proposed to be a remedy for these problems. If contraction and weakening is completely abandoned, the system gets overly restrictive. To work around, this modality operator ! is introduced which allows intuitionistic rules in fragments. Wadler describes a refinement of linear logic based on Girard's Logic of Unity (Wadler, 1993; Girard, 1993). It can be considered as a disjoint union of linear logic and intuitionistic logic. $[A]$ would mean that it is an intuitionistic assumption and the rules of weakening and contraction

are not restricted in this fragment, while $\langle A \rangle$ means that it is a linear assumption where weakening and contraction is prohibited.

In a linear logic setting, the assumptions can not be used more than once. This instigates a new view of implication which is different from intuitionistic logic. The implication obtains a meaning of consume which is represented as $A \multimap B$ where the assumption A cannot be used again after it has been used to obtain B. Its logical rules is given by [$\multimap$I] and [$\multimap$E]. Similarly there are two kinds of connectives, multiplicative and additive that arise in this logic system. More symbols are added in place of + and $\times$. Multiplicative conjunction is represented as $\otimes$, and additive conjunction and disjunction (& and $\oplus$). The structural and connective logical rules for the system is given in Fig. 2.7.

$$A, B, C ::= X \mid !A \mid A \multimap B \mid A \,\&\, B \mid A \otimes B \mid A \oplus B$$

Figure 2.6: Grammar for Intuitionistic Linear Logic

To relax linearity constraints, exponential modality ! is used, which signifies that an assumption can be duplicated or dropped without restriction. $!A$ can be thought of as *"as many A's as needed"*. The intuitionistic $A \to B$ can be encoded in linear logic by using the modality operator as $!A \multimap B$. Similarly $A + B$ would be represented as $!A \otimes !B$ and $A \times B$ would be represented as $A \,\&\, B$ (Wadler, 1993). We clearly see that this is a much powerful system in comparison to intuitionistic logic because of its enhanced expressivity with resources and propositions to have different treatment. However, there is an awkward asymmetry between the multiplicative and additive constructs. The multiplicative connectives follow the intuitionistic logic implication connectives but there is no notion of implication for additive connectives. In other words, $\multimap$ is a right adjoint for multiplicative $\otimes$ but there is no such right adjoint counter part for additive &.

There have been research efforts in the past to build prototype languages that have type systems based on linear logic. L^3(Ahmed et al., 2007) is an intermediate language that is built on a linear

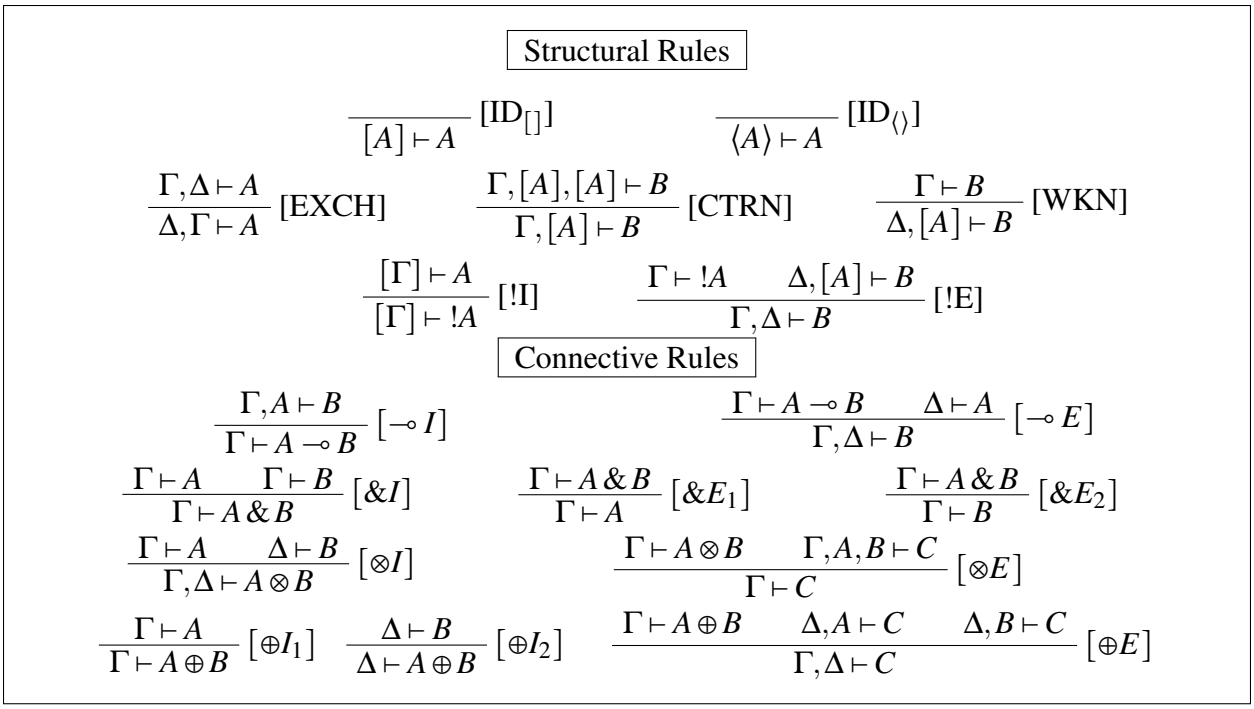


Figure 2.7: Intuitionistic Linear Logic Rules

type system and supports strong updates. Linear Haskell (Bernardy et al., 2017) is a surface level language that overloads function arrows to incorporate linearity.

2.3 Qualified Types

Jones (1994) proposed a general framework of incorporating predicates in the type language. Predicates are used to build constraints on the domain of the type of a term in the language expression. It introduces additional layer between polymorphic and monomorphic typing of programs. A modification of Damas-Milner algorithm $\mathcal{W}$ to incorporate predicates ensures that type inference is sound and complete. The types that satisfy all the predicates are called qualified types for the term. Qualified types are powerful enough to expresses type classes with functional dependencies (Jones, 2000), record types, sub-typing (Jones, 1994) and first class polymorphism (Jones, 1997).

The type language is modified from Fig. 2.1 to incorporate qualified types shown in Fig. 2.8. π and ω range over predicates and P and Q range over finite set of predicates. The types of the form

$\pi \Rightarrow \sigma$ denote those instances of σ that satisfy the predicate π, in general $P \Rightarrow \sigma$ would mean the instances of σ that satisfy all the predicates $\pi \in P$. The predicate entailment relation $P \Vdash \pi$ asserts that the predicate π can be inferred from the predicates in P. The typing rules in **HM** type system are slightly modified and 2 new rules are added as shown in Fig. 2.9. [⇒I] and [⇒E] serve the purpose of for introduction and elimination of qualified types respectively. The changes to the **HM** type system are highlighted. The predicate set P is threaded throughout the other rules but are not used anywhere except [⇒I] and [⇒E].

$$
\begin{aligned}
t, u, \upsilon &\in \text{Type Variables} \\
\pi, \omega &\in \text{Predicates} \\
P, Q &\in \text{Finite Predicate Set} \\
\text{Types} \quad \tau &::= t \mid \iota \mid \tau \to \tau \\
\text{Qualified Types} \quad \rho &::= \tau \mid \pi \Rightarrow \rho \\
\text{Type Scheme} \quad \sigma &::= \rho \mid \forall t.\sigma \\
\text{Expressions} \quad M, N &::= x : \sigma \mid \lambda x.M \mid MN \mid \texttt{let}\ x = M\ \texttt{in}\ N
\end{aligned}
$$

Figure 2.8: Qualified Types and Expression Language

$$
\frac{x : \sigma \in \Gamma}{P \mid \Gamma \vdash x : \sigma}\ [\text{VAR}] \qquad \frac{P \mid \Gamma \vdash M : \sigma \qquad Q \mid \Gamma_x, x : \sigma \vdash N : \tau}{P, Q \mid \Gamma \vdash (\texttt{let}\ x = M\ \texttt{in}\ N) : \tau}\ [\text{LET}]
$$

$$
\frac{P \mid \Gamma \vdash M : \sigma \qquad t \notin \texttt{fvs}(\Gamma) \cup \texttt{fvs}(P)}{P \mid \Gamma \vdash M : \forall t.\sigma}\ [\forall\ \text{I}] \qquad \frac{P \mid \Gamma \vdash M : \forall t.\sigma}{P \mid \Gamma \vdash M : [\tau/t]\sigma}\ [\forall\ \text{E}]
$$

$$
\frac{P \mid \Gamma_x, x : \tau \vdash M : \tau'}{P \mid \Gamma \vdash \lambda x.M : \tau \to \tau'}\ [\to \text{I}] \qquad \frac{P \mid \Gamma \vdash M : \tau \to \tau' \qquad P \mid \Gamma \vdash N : \tau}{P \mid \Gamma \vdash MN : \tau'}\ [\to \text{E}]
$$

$$
\frac{P, \pi \mid \Gamma \vdash M : \rho}{P \mid \Gamma \vdash M : \pi \Rightarrow \rho}\ [\Rightarrow I] \qquad \frac{P \mid \Gamma \vdash M : \pi \Rightarrow \rho \qquad P \Vdash \pi}{P \mid \Gamma \vdash M : \rho}\ [\Rightarrow E]
$$

Figure 2.9: Typing Rules for Qualified Types

2.4 Linear logic with Qualified Types: Quill

Quill (Morris, 2016) implements a linear type system with a sound and complete type inference using qualified types. It uses a modified version of Algorithm $\mathcal{M}$ to compute principal types of the terms. The key idea of Morris is to introduce two predicates for types into the language: `Un` and `Fun` with a predicate for ordering types depending on their admittance to structural rules. The predicate $\tau \geq \tau'$ will hold only if τ admits more structural rules than τ' or, in other words, if τ' is more restricting than τ. The predicate `Un` τ implies that the type τ is unrestricted which means it does not contain any resources or the resources that it captures can be easily duplicated and dropped. In traditional sense of type classes in Haskell, we can think of the `Un` to be a type-class with methods supporting the operation of duplication and dropping shown in Fig. 2.10. In a proof theoretic setting, it would mean that it admits to weakening and contraction. The predicate `Fun`τ implies that the type τ is of a function type. The function depending on its use, may or may not capture resources in its closure and the functions themselves can be of restricted or unrestricted type.

```
1 class Un where
2     dup  :: t -!> (t ⊗ t)
3     drop :: t -!> 1
```

Figure 2.10: `Un` as a Typeclass

Simple types such as integers and booleans are all of unrestricted type as they can be duplicated or dropped freely. While program resources such as file handles, database connections should be treated as restricted or linear types as we cannot freely duplicate or drop them. Consider a lambda expression that represents function application $\lambda f.\lambda x.fx$ and it is applied to some function $\mathcal{F}$. The linearity of this function $\lambda y.\mathcal{F}x$ would depend on the linearity of $\mathcal{F}$. To generalize, we can say that the linearity of the lambda expression depends on its closure. The type of $\lambda f.\lambda x.fx$ can be written as $(\tau \xrightarrow{f} \upsilon) \to \tau \xrightarrow{g} \upsilon$ This function would be well typed only if f is more restricting i.e. admits

more structural rules than g, so to say $f \geq g$.

2.5 Logic of Bunched Implications and $\alpha\lambda$-Calculus

In intuitionistic logic, the context is considered as a list or a set. In the theory of ***BI***, the context is treated as a tree. Contexts are called bunches and are syntactically combined using 2 connectives comma (,) or a semicolon (;). The logic of ***BI*** glues together intuitionistic linear logic with intuitionistic logic by permitting contexts connected with semicolon to undergo contraction and weakening while the context connected with comma are prohibited to undergo contraction and weakening. Comma and semicolon do not distribute over each other. Thus $A,(B;C) \neq A,B;A,C$ and $A;(B,C) \neq A;B,A;C$ where A, B and C are assumptions or propositions. There are two flavors of implication—additive and multiplicative—which is closely related to the idea of conjunction.

$$\frac{\Gamma,A \vdash B}{\Gamma \vdash A \Diamond B}$$

In the logic of ***BI*** the question then faced is choosing what kind of implication should be used in place of $\Diamond$—the additive kind or the multiplicative kind. O'Hearn & Pym (1999) introduce 2 kinds of arrows and use them depending on the connectives used in the context. A multiplicative implication ($-\!\!*$) is used when the context is connected with a comma and an additive implication ($\twoheadrightarrow$) is used when the context is connected using semicolon. This gives rise to two rules

$$\frac{\Gamma,A \vdash B}{\Gamma \vdash A -\!\!* B}\,[-\!\!* I] \qquad\qquad \frac{\Gamma;A \vdash B}{\Gamma \vdash A \twoheadrightarrow B}\,[\twoheadrightarrow I]$$

Γ,A cannot under go weakening or contraction to duplicate or get rid of either A or Γ. This hints to a notion that multiplicative implication ($-\!\!*$) exhibits property of the linear implication ($\multimap$). The linear implication cannot however be directly converted to a multiplicative implication as the latter does not exhibit properties of counting the number of uses of its arguments. Also,

in contrast to linear logic, the multiplicative implication cannot be converted into an intuitionistic implication as there is there is no modality introduced in the system.. The logic of ***BI*** combines the additive logic i.e. intuitionistic logic with the multiplicative side i.e. intuitionistic linear logic. The promise of this logic system is that the multiplicative side can be used to model the behavior of resources in the programming language while the additive side would help the programmers fall back to the non-resource intuitionistic parts. This patches up the awkward asymmetry experienced in linear logic. The multiplicative conjunction $\otimes$ has a right adjoint counter part as $\multimap$ while additive conjunction & has a right adjoint counterpart of $\twoheadrightarrow$. The logic of ***BI*** argues that instead of looking at the number of times an argument is used within the function, it should be viewed in terms of *sharing*. $\alpha\lambda$-calculus (O'Hearn, 1999; Pym, 2002) is the Curry-Howard interpretation of the logic of ***BI***. It introduces 2 kinds of arrows by modifying the the syntax of lambda calculus:

1. $\multimap$: Functions do not share resources with their arguments

2. $\twoheadrightarrow$: Functions share resources with their arguments

The types and terms of $\alpha\lambda$-calculus are summarized in Fig. 2.11. The contexts $\{\}_m$ represents a multiplicative empty context while the $\{\}_a$ represents an additive empty context. The structural and connective rules for $\alpha\lambda$-calculus are summarized in Fig. 2.12.

$$
\begin{array}{c}
t, u, \upsilon \in \text{Type Variables} \\
\text{Context} \quad \Gamma, \Delta ::= \{\}_m \mid \{\}_a \mid x : \tau \mid \Gamma, \Delta \mid \Gamma; \Delta \\
\text{Types} \quad \tau ::= t \mid \iota \mid \tau \twoheadrightarrow \tau \mid \tau \multimap \tau \\
\text{Expressions} \quad M, N ::= x \mid \lambda x.M \mid \alpha x.M \mid MN
\end{array}
$$

Figure 2.11: $\alpha\lambda$-Calculus Types and Terms

Due to the rules of $\alpha\lambda$-calculus $f : \tau \multimap \tau'; x : \tau \nvdash fx : \tau'$, as f needs an argument that does not share any resources with its context. The term $\lambda x.\alpha f.fxx : \tau \multimap (\tau \twoheadrightarrow \tau \twoheadrightarrow \tau') \twoheadrightarrow \tau'$ is a typable term in $\alpha\lambda$-calculus as shown in Fig. 2.13. This illustrates the difference between logic of ***BI*** and

$$\frac{x:\tau \in \Gamma}{\Gamma \vdash x:\tau}\ [\text{VAR}] \qquad \frac{\Gamma;\Gamma \vdash M:\tau}{\Gamma \vdash M:\tau}\ [\text{CTRN}] \qquad \frac{\Gamma \vdash M:\tau}{\Gamma;\Delta \vdash M:\tau}\ [\text{WKN}]$$

$$\frac{\Gamma_x, x:\tau \vdash M:\tau'}{\Gamma \vdash \lambda x.M:\tau \multimap \tau'}\ [\multimap \text{I}] \qquad \frac{\Gamma \vdash M:\tau \multimap \tau' \quad \Delta \vdash N:\tau}{\Gamma,\Delta \vdash MN:\tau'}\ [\multimap \text{E}]$$

$$\frac{\Gamma_x; x:\tau \vdash M:\tau'}{\Gamma \vdash \alpha x.M:\tau \twoheadrightarrow \tau'}\ [\twoheadrightarrow \text{I}] \qquad \frac{\Gamma \vdash M:\tau \twoheadrightarrow \tau' \quad \Delta \vdash N:\tau}{\Gamma;\Delta \vdash MN:\tau'}\ [\twoheadrightarrow \text{E}]$$

Figure 2.12: Typing Rules for $\alpha\lambda$-Calculus

linear logic as even tough the argument is separate from the function, it may be used twice. Linear logic would prohibit such use of arguments to function.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{f:\tau \twoheadrightarrow \tau \twoheadrightarrow \tau' \vdash f:\tau \twoheadrightarrow \tau \twoheadrightarrow \tau'}\ [\text{VAR}] \quad \dfrac{}{x:\tau \vdash x:\tau}\ [\text{VAR}]}{f:\tau \twoheadrightarrow \tau \twoheadrightarrow \tau';x:\tau \vdash fx:\tau \twoheadrightarrow \tau'}\ [\twoheadrightarrow\text{E}] \quad \dfrac{}{x:\tau \vdash x:\tau}\ [\text{VAR}]}{x:\tau; f:\tau \twoheadrightarrow \tau \twoheadrightarrow \tau';x:\tau \vdash fxx:\tau'}\ [\twoheadrightarrow\text{E}]}{x:\tau; f:\tau \twoheadrightarrow \tau \twoheadrightarrow \tau' \vdash fxx:\tau'}\ [\text{CTRN}]}{x:\tau \vdash \lambda x.fxx:(\tau \twoheadrightarrow \tau \twoheadrightarrow \tau') \twoheadrightarrow \tau'}\ [\twoheadrightarrow\text{I}]}{\vdash \lambda x.\alpha f.fxx:\tau \multimap (\tau \twoheadrightarrow \tau \twoheadrightarrow \tau') \twoheadrightarrow \tau'}\ [\multimap\text{I}]$$

Figure 2.13: Multiplicative Argument used Twice in $\alpha\lambda$-calculus

The use of logic of ***BI*** as a type inference system is currently an unexplored area of research in functional programming language implementation. There has been research on building proof theoretic and semantic models of the logic system (Pym, 2002). The use of bunches instead of lists as typing environment makes it difficult to have a direct implementation of the type inference algorithm. Atkey (2004) designs λ_{sep} calculus which is based on the affine variant of $\alpha\lambda$-calculus focusing on separation of resources used by objects. Collinson et al. (2005) designs a polymorphic variant of $\alpha\lambda$-calculus.

Chapter 3

Programming in QuB

In this chapter, we illustrate using examples how QuB is different from other functional languages and how a powerful type system based on logic of ***BI*** would be used to track resources. The examples show how the resources use can be tracked at compile time and resource leaks can be avoided.

3.1 File Handles

In modern programming languages resources such as files are treated as normal variables. It is the programmer's responsibility to check that that files are not closed twice and that there are no files that remain open when the program exits. This seemingly trivial responsibility becomes tedious and error prone as soon as the programming logic gets complex. Modern functional languages such as Haskell enforces the file input and output to be wrapped in a IO Monad. This is more declarative than imperative languages, but the type system is not powerful enough to detect whether a file handle is closed twice or is not closed at all. Consider the type signatures for functions for file handling shown in Fig. 3.1. We explicitly return the file resource i.e. `f`. A simple program in Haskell that opens a file and reads a line from it and then closes the file handle using this flavor of file handling functions is shown in Fig. 3.2.

Consider an incorrect version of a program where the file handle is closed twice after reading a line from it as shown in Fig. 3.3. It may not cause problems in a single threaded environment, but in a multi-threaded environment the second close may accidently close the file handle that may have been reused in the background by another thread. When another thread tries to write on this

```haskell
openFile :: FilePath -> IO FileHandle
closeFile :: FileHandle -> IO ()
readLine :: FileHandle -> IO (String, FileHandle)
writeFile :: String -> FileHandle -> IO ((), FileHandle)
```

Figure 3.1: File Handling Functions

```haskell
do f  <- openFile "sample.txt"
   (s, f)  <- readLine f
   () <- closeFile f
```

Figure 3.2: Reading from a file in Haskell

closed file handle, it would throw an exception. Haskell's type system would happily accept this program but it might generate a runtime exception.

```haskell
do f  <- openFile "sample.txt"
   (s, f)  <- readLine f
   () <- closeFile f
   () <- closeFile f
```

Figure 3.3: Reading from a File and Closing it Twice

Apple's goto fail bug that appeared in iOS 7.0 and caused a security vulnerability in 2012 is a similar example of closing the file twice. The code snippet that caused the SSL/TLS handshake to be completely skipped looked like in Fig. 3.4. The cases in which the condition is false, the second `goto fail;` on line 6 would force the protocol to skip the steps to be taken after the if-block. This made the system vulnerable to a man-in-middle attack.

Another example of incorrect way of using file handle is by not closing the file handle after using it shown in Fig. 3.5. In a short lived process, when the program exits, file handles that are not closed are freed by the operating system. But if it is a long running process it would run out of file handles and the whole process would crash with an error that it cannot open any more file

```
...
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
...
```

Figure 3.4: Goto Fail bug

handles. Abnormal exit from the process would interfere in the write process and the operating system would close the file handle without waiting for the buffer to be completely written on the file system.

```
do f <- openFile "sample.txt"
   (s, f)  <- readLine f
   return s
```

Figure 3.5: Reading from a File and Not Closing File Handle

We take a deeper dive into this problem by inspecting the desugared version of the "do" notation. Both the programs would be translated into bind (>>=) operations . Recall that type signature of bind function is given as `(>>=) :: (Monad m) => m t -> (t -> m u) -> m u`. Desugared version of Fig. 3.3 is shown in Fig. 3.7 and the desugared version of Fig. 3.5 is shown in Fig. 3.6.

In both the cases described above, the well typed looking program should be red flagged by the compiler as it would cause problems at runtime. To solve this problem, we introduce the concept of sharing and separation of resources from the logic of ***BI*** in QuB. The type $\rightarrow$ now has to be specified as either shared ($\twoheadrightarrow$) or separated ($-\!\!*$). In QuB program code, we will use `-*>` to mean

```
(>>=) (openFile "sample.txt" ReadMode) (\f ->
       >>= (closeFile f) (\(s, f) -> return s)
```

Figure 3.6: Reading from a File in Haskell and Not Closing It

```
(>>=1) (openFile "sample.txt") (\f ->
            (>>=2) (readFile f) (\(s, f) ->
                        (>>=3) (closeFile f) (\_ -> closeFile f)))
```

Figure 3.7: Reading from a File in Haskell and Closing Twice

$\multimap$ and -&> to mean $\twoheadrightarrow$. The file handling functions would have slightly different type signatures in QuB as shown in Fig. 3.8 to accommodate the new function implication flavors. Similarly, the bind operation will also be typed differently as shown below as the resources of the function are separate from its arguments.

```
openFile :: FilePath -*> IO FileHandle
closeFile :: FileHandle -*> IO ()
readLine :: FileHandle -*> IO (String, FileHandle)
writeFile :: String -*> FileHandle -*> IO ((), FileHandle)

(>>=) :: (Monad m) => m t -*> (t -*> m u) -*> m u
```

Figure 3.8: File Handling and Bind Function Type Signatures in QuB

We now consider the types for the two faulty programs previously described with respect to QuB. In Fig. 3.9 we see that the first (>>=1) and second bind (>>=2) functions would have appropriate types where each argument is separate from the the function. The file handing functions return a new binding for the file resource f. However, we notice that the third bind operation (>>=3) would have a problem. It would be typed as IO () -*> (() -*> IO ()) -&> IO () as both the arguments share the file variable f which would be a type error as the bind operation should have a type signature of IO () -*> (() -*> IO ()) -*> IO (). This mismatch in the

types would be caught statically during the type checking phase of compilation.

```
(>>=1) (openFile "sample.txt") (\f ->
            (>>=2) (readFile f) (\(s, f) ->
                        (>>=3) (close f) (\_ -> closeFile f)))


(>>=1):: IO FileHandle -*> (FileHandle -*> IO ()) -*> IO ()
(>>=1) (openFile "sample.txt" ReadMode) (\f -> ...)

(>>=2) :: IO FileHandle
            -*> (FileHandle -*> IO (String, Filehandle))
            -*> IO (String, FileHandle)}
(>>=2) (readLine f) (\(s,f) -> ... )

(>>=3) :: IO () -*> (() -*> IO ()) -&> IO ()
(>>=3) (closeFile f) (\_ -> closeFile f)}
```

Figure 3.9: Closing file twice in QuB

We also have the concept of unrestricted types similar to Quill. A type that contains no resources, or which is implicitly copied or dropped in the program is tagged as an unrestriced type. In the second faulty program, shown in Fig. 3.6 the file handle `f` is not closed. It is declared but not used in its scope. This would force the file handle to be be tagged as an unrestricted type by the QuB type checker. This is a violation of our assumption that resources cannot be of the unrestricted type. Thus the program would not type check due to mismatch of the file handle type to be inferred as unrestricted.

3.2 Exception handling

We expand on the file handling scenario and consider the code that can throw runtime exceptions. The motivation to do so lies in the fact that memory leaks are caused because of runtime exceptions where the part of code that is responsible to clean up resources or in this case closing the file is skipped due to an alternate execution path. In Haskell, error handing is done using `MonadError`.

The type class definition is shown in Fig. 3.10. `throwError` is used inside a monadic context to initiate exception processing and `catchError` is used to handle a previously thrown error and return to a normal execution.

```
class Monad m => MonadError e m | m -> e where
    throwError :: e -> m a
    catchError :: m a -> (e -> m a) -> m a
```

Figure 3.10: Haskell's MonadError Typeclass

A common way of handling error in Haskell is shown in Fig. 3.11. We use the same file handling API as defined previously in Fig. 3.1 In normal code execution path, the first line of the file will be returned after closing the file. In case of a runtime error, say `readLine f` throws an error, the `catchError` function will invoke the handler function and return an appropriate error message. We notice that in case of an error the file handle `f` is never closed and will cause a resource leak. The Haskell type system has no way to detect this memory leak.

```
do f <- openFile "sample.txt"
   ((s, f)  <- readLine f
    let c = caps s
    () <- closeFile f
    return $ Right c) `catchError` (\_ ->
                            return $ Left "Error in reading file")
```

Figure 3.11: Handling Errors in Haskell

We enforce resource cleanup in a systematic way in QuB. For a concrete example, see Fig. 3.12. We will assume that `readLine` can throw an exception during runtime, where it might fail to read a line due to the filehandle being stale and the file no longer exists. For the sake of simplicity we will assume `closeFile` does not throw exceptions. `openFile` throwing an exception would not cause a resource leak hence will not be a problem. We have two kinds of IO operations, `IO` that does not fail or throw exceptions and is safe, and `IOF` that can fail and throw exceptions. `throw`

function can be used by a function to start the processing of the exception. `catch` function can potentially convert a code that can fail, to a code that does not fail. The `onExcept` function gives a way to clean up resources after the code throws an exception. It may re-throw the exception after cleaning up the resources or a `catch` function can be used to handle the failure case. The sharing arrow between the two arguments of `onExcept` enables the clean up block of code to be executed on the same resources that were used its first argument.

```
openFile :: FilePath -*> IO FileHandle
closeFile :: FileHandle -*> IO ()
readFile :: FileHandle -*> IOF (String, FileHandle)
writeFile :: String -*> FileHandle -*> IOF ((), FileHandle)

throw :: Exception -*> IO a
catch :: IOF a -*> (Exception -> IO a) -&> IO a

onExcept :: IOF a -*> IOF () -&> IOF a
m `onExcept` n = m `catch` (\e -> n >> throw e)

readFromFile :: FilePath -*> IO (Either String String)
readFromFile fpath =
do fh  <- openFile fpath
   ((s, fh)  <- readLine fh
   let l = caps s
   () <- closeFile fh
   return $ Right l) `onExcept` (do () <- closeFile fh)
                     `catch` (\e ->
                                  return $ Left "Could not read file")
```

Figure 3.12: Exception Handling in Files

Chapter 4

Core Language Syntax and Types

In this chapter we give the formal description of the language syntax and types. We explain what it means for a type assignment to exist as binary trees. We then show how we generalize the tree into a sharing graph and represent it as a collection of three tuple sets in order to simplify our type inference algorithm.

$$
\begin{aligned}
t, u, \upsilon, \phi &\in \text{Type Variables} \\
P, Q &\in \text{Finite Predicate Set} \\
\text{Types} \quad \tau &::= t \mid \tau \to \tau \text{ where } \{\overset{!}{\twoheadrightarrow}, \twoheadrightarrow, \overset{!}{\multimap}, \multimap\} \subseteq \to \\
\text{Predicates} \quad \pi, \omega &::= \texttt{Un}\ \tau \mid \texttt{SeFun}\ \tau \mid \texttt{ShFun}\ \tau \mid \tau \geq \tau' \\
\text{Qualified Types} \quad \rho &::= \tau \mid \pi \Rightarrow \rho \\
\text{Type schemes} \quad \sigma &::= \rho \mid \forall t.\sigma
\end{aligned}
$$

Figure 4.1: Types QuB

The type language consists of type variables and four binary type constructors the sharing arrow ($\twoheadrightarrow$) and the separating arrow ($\multimap$) and unrestricted version of both the type constructors ($\overset{!}{\multimap}, \overset{!}{\twoheadrightarrow}$). The sharing arrow would mean that the function shares resources with its argument and the separating arrow would mean that the function does not share resources with its arguments. We would write both the arrows in an infix notation.

The predicate system enhances the expressivity of the type system. Following the same route taken in Quill (Morris, 2016) we use the predicate $\texttt{Un}\ \tau$ to denote that the type τ is unrestricted. Unrestricted types are the ones that do not have any resources or whose resources can be duplicated

or deleted easily. Built-in lightweight types such as `Int`, `Char` are considered unrestricted types. We write `ShFun` ϕ to describe that type ϕ may share resources with its argument types and `SeFun` ϕ to describe that type ϕ does not share any resources from its argument types. Notice that function types can also be unrestricted i.e.e it does not have any resources. If a type τ is unrestricted i.e. it qualifies with predicate `Un` and it also qualifies one of the function predicates—`SeFun` or `ShFun`—we write them as $\overset{\downarrow}{\multimap}$ and $\overset{\downarrow}{\twoheadrightarrow}$ respectively. We also define an ordering on types by using the predicate $\geq$. The predicate $\tau \geq \tau'$ holds if the type τ' is less restricting than τ or to say τ has admits more structural rules than τ' as previously explained in §2.4. The predicate entailment relations $P \Rightarrow Q$ are given in Fig. 4.2.

To keep the current system simple we have not included kinds. They are added into this system as a language extension to enable users to define custom types using type constructors. We describe this extension in Chapter 6.

$$\frac{\pi \in P}{P \Rightarrow \pi} \qquad \frac{\bigwedge_{\pi \in Q} P \Rightarrow \pi}{P \Rightarrow Q} \qquad \frac{}{P \Rightarrow \texttt{Un}\ (\tau \overset{\downarrow}{\multimap} \tau')} \qquad \frac{}{P \Rightarrow \texttt{Un}\ (\tau \overset{\downarrow}{\multimap} \tau')}$$

$$\frac{}{P \Rightarrow \tau \geq (v \multimap v')} \qquad \frac{}{P \Rightarrow \tau \geq (v \twoheadrightarrow v')} \qquad \frac{P \Rightarrow \texttt{Un}\ \tau}{P \Rightarrow \tau \geq (v \overset{\downarrow}{\multimap} v')} \qquad \frac{P \Rightarrow \texttt{Un}\ \tau}{P \Rightarrow \tau \geq (v \overset{\downarrow}{\twoheadrightarrow} v')}$$

$$\frac{\tau = \multimap \vee \tau = \overset{\downarrow}{\multimap}}{P \Rightarrow \texttt{SeFun}\ \tau} \qquad \frac{\tau = \twoheadrightarrow \vee \tau = \overset{\downarrow}{\twoheadrightarrow}}{P \Rightarrow \texttt{ShFun}\ \tau} \qquad \frac{P \Rightarrow \tau \geq \phi t \qquad t \text{ fresh}}{P \Rightarrow \tau \geq \phi}$$

Figure 4.2: Entailment Rules

In normal type systems, the contexts are represented as sets or lists. In logic of ***BI*** they are represented as binary trees and are called bunches. The leaf nodes contain the pair of term and its associated type. Internal nodes of the context tree are connectives which can either be a semicolon (;) or a comma (,). If a bunch Δ is a subtree of Γ, then we denote a subtree relation by $\Gamma(\Delta)$. Two bunches are equivalent ($\Gamma \equiv \Delta$) if they can be transformed into another by renaming identifiers. The bunches have a restriction that no identifier appears more than once. We restrict certain structural rules on the context depending on the connectives being used. If contexts are combined using a comma (,), contraction and weakening is not admissible, but if the contexts are combined using

a semicolon (;) then it can undergo contraction and weakening. Exchange rule is admissible in both the connectives. This distinction enables us to have a special treatment for resources in our language. By associating a resource with a comma constructor, our type system will not disposed it off by using the contraction rule. While, non-resourceful objects (or normal propositions) can be combined using the semi-colon constructor. An example bunch is shown in Fig. 4.3. a and b have a shared context while c is separate from the bunch a and b. If Γ represents the complete bunch of Fig. 4.3, $\Delta \equiv (a:A;b:B)$ and $\Delta' \equiv (c:C)$ then $\Gamma \equiv \Delta,\Delta'$ and $\Gamma(\Delta)$.

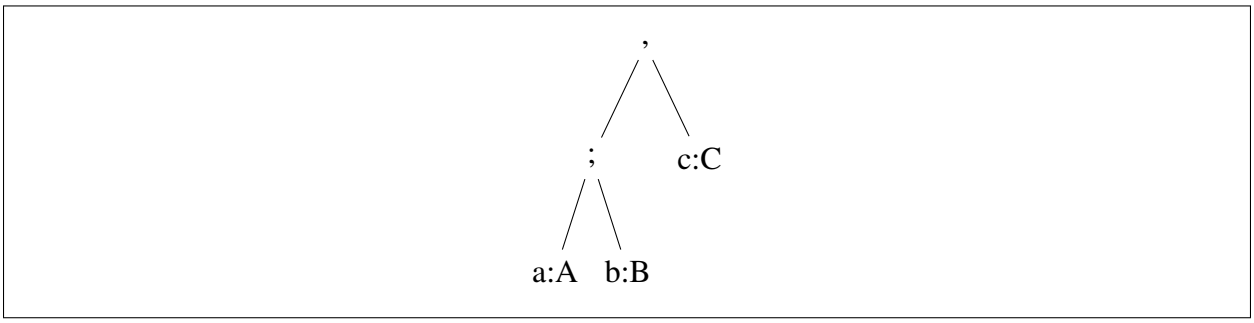


Figure 4.3: Bunches in ***BI***

In our type system, we generalize the tree approach into a graph where each node represents variables or resources and the edges between the nodes represent sharing between them. The example in Fig. 4.3 can be represented as what we would call a sharing graph shown in Fig. 4.4. A graph structure, in general, can represent a binary tree structure and its associated operations. They represent more complex structures than trees, thus will provide more flexibility in accepting well typed terms in our language making it more expressive.

We define sharing relation (Ψ) between variables to the collection of variables it is in sharing with. The relation $\Psi(x,\{y_1,y_2,y_3\})$ holds if x is in sharing with $\{y_1,y_2,y_3\}$. Domain of Ψ will be defined as $\texttt{dom}(\Psi) = \{x \mid (x,\vec{y}) \in \Psi\}$, where $\vec{y}$ is a shorthand for the denoting collection of variables shared with x. We can think of Ψ to be similar to Γ, but it contains the sharing information instead of the type of the variable. Extending the sharing for a variable will be denoted by $\Psi(x)+y$, which would mean the variable y is in sharing with x. We axiomatize the sharing operation to be reflexive,

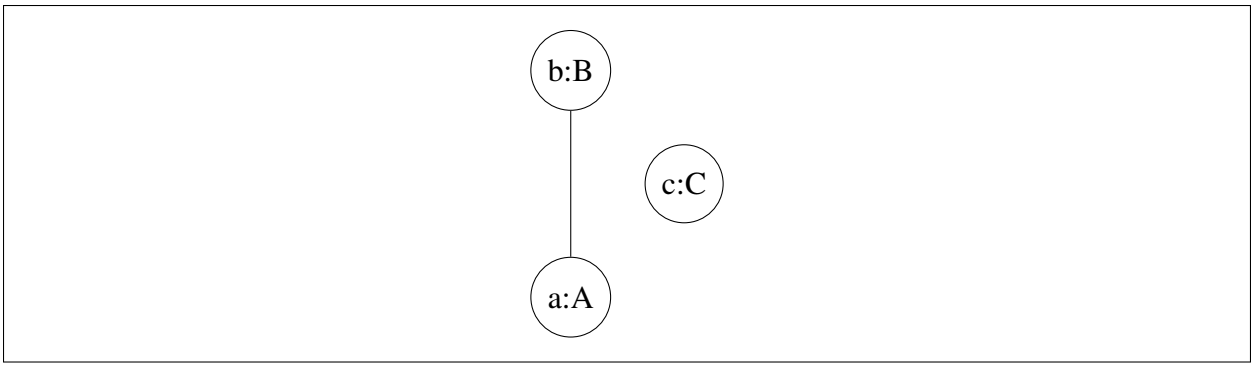


Figure 4.4: Sharing Graph

symmetric and non-transitive. So to say,

$$\forall_{x \in \mathtt{dom}(\Psi)}\ x \in \Psi(x) \qquad \text{(reflexive)}$$

$$\forall_{x,y \in \mathtt{dom}(\Psi)}\ \text{if } y \in \Psi(x) \text{ then } x \in \Psi(y) \qquad \text{(symmetric)}$$

Our final goal is to design a simple type inferencing algorithm for a term language. Using sharing graphs in implementing typing judgments would make the process considerably complex. We simplify the sharing graph by flattening it into an adjacency list or a collection of 3 tuple containing the variable identifier, its type and a collection of variables it shares with. Manipulating lists is much easier than manipulating graphs. For example, if a resource x has type τ and it shares with variables $\{y_1, y_2, y_3\}$ we would represent it as $x^{\{y_1,y_2,y_3\}} : \tau$ or just $x^{\bar{y}} : \tau$ for short. We would write $\Gamma, x^{\vec{y}} : \tau$ to mean $\Gamma \sqcup \{x^{\vec{y}} : \tau\}$. We can now formally define the typing context or environment as shown in Fig. 4.5.

$$\text{Typing Context} \quad \Gamma, \Delta ::= \varepsilon \mid x^{\bar{y}} : \sigma \mid \Gamma \oplus \Delta \mid \Gamma \circledast \Delta$$

Figure 4.5: Typing Context

We define a few auxiliary functions on the type assignments. $\mathtt{Vars}(\Gamma)$ is the set of all the term variables in Γ. $\mathtt{Shared}(\Gamma)$ computes the set of all the term variables that are in sharing with each

other. $\texttt{Used}(\Gamma)$ computes the union of all the term variables in the type assignment and the term variables shared by each of those. We define two partial operators on type assignments as shown in Fig. 4.7. The mapping function ($\Gamma^{[\vec{a}\mapsto\vec{b}]}$) extends the sharing relation between the terms. In the sharing graph perspective it would mean adding edges between the nodes.

$$\begin{aligned}
\texttt{Vars}(\Gamma, x^{\vec{y}} : \tau) &= \texttt{Vars}(\Gamma) \cup \{x\} \\
\texttt{Shared}(\Gamma, x^{\vec{y}} : \tau) &= \texttt{Shared}(\Gamma) \cup \{\vec{y}\} \\
\texttt{Used}(\Gamma) &= \texttt{Vars}(\Gamma) \cup \texttt{Shared}(\Gamma)
\end{aligned}
\qquad
(\Gamma, x^{\vec{y}} : \tau)^{[a\mapsto\vec{b}]} = \begin{cases} x \notin \vec{y} & (\Gamma^{[a\mapsto\vec{b}]}, x^{\vec{y}} : \tau) \\ x \in \vec{y} & (\Gamma^{[a\mapsto\vec{b}]}, x^{(\vec{y}\setminus a)\cup\vec{b}} : \tau) \end{cases}$$

$$\Gamma^{[\vec{a}\mapsto\vec{b}]} = (\ldots((\Gamma^{[a_1\mapsto\vec{b}]})^{[a_2\mapsto\vec{b}]})^{\cdots})^{[a_n\mapsto\vec{b}]}$$

Figure 4.6: Auxiliary Functions on Type Assignments

Two type assignments are said to be in disjoint union ($\circledast$) if either of the type assignments used terms are not in common with other type assignment's shared term. If the type assignments have an exact overlapping of terms being used, it is said to be in a sharing union ($\oplus$). The (#) in ($\circledast$) represents disjoint check and we use the standard notion of set equality for checking sharing union.

$$\begin{aligned}
\Gamma \circledast \Gamma' = \Gamma \sqcup \Gamma' &\Rightarrow \texttt{if } \texttt{Vars}(\Gamma)\#\texttt{Used}(\Gamma') \wedge \texttt{Vars}(\Gamma')\#\texttt{Used}(\Gamma) \\
\Gamma \oplus \Gamma' = \Gamma \sqcup \Gamma' &\Rightarrow \texttt{if } \texttt{Used}(\Gamma) = \texttt{Used}(\Gamma')
\end{aligned}$$

Figure 4.7: Type Assignment Operations

$$\begin{aligned}
\text{Term Variables} \quad & x, y, z \in \text{Var} \\
\text{Expressions} \quad & M, N ::= x \mid \lambda^{\multimap} x.M \mid \lambda^{\to} x.M \mid MN \mid \texttt{let } x = M \texttt{ in } N
\end{aligned}$$

Figure 4.8: Term Language

Our term language is similar to that of simply typed lambda calculus involving variables and

application but we have two types of lambda expressions, the alpha lambda ($\lambda^{\twoheadrightarrow}$) denotes sharing of the argument term with the expression M and the separating lambda term (λ^{*}) that implies the argument term has a separating context with the expression M. We also have `let` construct to enable implicit parametric polymorphism.

Chapter 5

Type System and Type Inference

In this chapter we describe the type system using the types and terms defined in previous chapter. We describe in QuB's type system §5.2 and then describe a syntax directed type system in §5.3 and give a type inference algorithm $\mathcal{M}$ in §5.4. To begin with, we give some conventions and notations with priliminary definitions that will be used throughout the sections that follow.

5.1 Conventions and Notations

The vector $\vec{t}$ is a shorthand for a finite set of variables $\{t_1, t_2, \ldots, t_n\}$ and $\forall \vec{t}.P \Rightarrow \tau$ abbreviates $\forall t_1 \ldots \forall t_n.P_1 \Rightarrow \ldots \Rightarrow P_m \Rightarrow \tau$. Γ denotes the type assignment. It is a list of three-tuples containg the variable, its type scheme and its sharing information. Γ_x denotes the type assignment excluding the type variable x. We write $\sigma = \Gamma(x)$ for the type scheme assigned to the term x in Γ. $\mathtt{dom}(\Gamma)$ is the set of identifiers in the type assignment i.e. $\mathtt{dom}(\Gamma) = \{x \mid (x^{\vec{y}} : \sigma) \in \Gamma\}$. $\Gamma \sqcup \Delta$ is a multiset union of two type assignments Γ and Δ.

Definition 1 (Free Type Variables)**.** $\mathtt{fvs}(\tau)$ is the set of free type variables in the type τ

$\mathtt{fvs}(\sigma)$ is the set of free type variables in a typing scheme $\sigma = \forall \vec{t} Q. \Rightarrow \tau$.

$\mathtt{fvs}(\sigma) = (\mathtt{fvs}(\tau) \cup \mathtt{fvs}(Q)) \backslash \vec{t})$

$\mathtt{fvs}(\Gamma)$ is the set of free type variables in the type assignment Γ.

$\mathtt{fvs}(\Gamma) = \bigcup_{x \in \mathtt{dom}(\Gamma)} \{\mathtt{fvs}(\Gamma(x))\}$

Definition 2 (Typing Judgment)**.** The expression $P \mid \Gamma \vdash M : \sigma$ denotes the assertion that the term M has a typing scheme σ when the predicates in P are satisfied and the free type variables in M are specified in type assignment Γ.

5.2 Type System

We split our type system into two parts. The first part includes structural rules shown in Fig. 5.1 and the second part includes connectives with introduction and elimination rules shown in Fig. 5.2.

$$\frac{}{P \mid x^{\vec{y}} : \sigma \vdash x : \sigma}\ [\text{ID}]$$

$$\frac{P \mid \Gamma \circledast \Delta \circledast \Delta \vdash M : \sigma \qquad P \vdash \Delta\ \mathtt{un}}{P \mid \Gamma \circledast \Delta \vdash M : \sigma}\ [\text{CTR-UN}] \qquad \frac{P \mid \Gamma \oplus \Delta \oplus \Delta \vdash M : \sigma}{P \mid \Gamma \oplus \Delta \vdash M : \sigma}\ [\text{CTR-SH}]$$

$$\frac{P \mid \Gamma \vdash M : \sigma \qquad P \vdash \Delta\ \mathtt{un}}{P \mid \Gamma \circledast \Delta \vdash M : \sigma}\ [\text{WKN-UN}] \qquad \frac{P \mid \Gamma \vdash M : \sigma}{P \mid \Gamma \oplus \Delta \vdash M : \sigma}\ [\text{WKN-SH}]$$

Figure 5.1: Structural Typing Rules

The tautology rule [ID] is a simple type assignment lookup for checking the type of the term. The weakening and contraction rules are made explicit in contrast to standard Hindley-Milner type system. The contraction sharing rule [CTR-SH] and weakening sharing rule [WKN-SH] convey that we can duplicate or drop certain pairs of type assignments as we know they are in sharing with other terms that remain in the context. The contraction separation rule [CTR-UN] and weakening separation rule [WKN-SH] can be applied to terms only if we can prove that they are of unrestricted type which is captured by the antecedent predicate $\Delta\ \mathtt{un}$ on the type that is dropped or duplicated.

The [LET] rule is used to enable implicit parametric polymorphism as usual. The rules of [$\twoheadrightarrow$I] and [$\multimap$I] describes the abstraction over shared and separating resources respectively, while [$\twoheadrightarrow$E] and [$\multimap$E] is the application rule for shared and separating resources respectively. [$\Rightarrow$I] and [$\Rightarrow$E] are the rules for qualified types that would add constraints on the type being computed. [$\forall$I] introduces polymorphism and [$\forall$E] eliminates it. The λ abstractions $\lambda^{\multimap} x.M$ and $\lambda^{\twoheadrightarrow}.M$ have a function type $\mathtt{ShFun}\ \phi$ or $\mathtt{SeFun}\ \phi$ only if it is more restricting that its environment. This is specified in the judgments $\cdot \geq \cdot$. To avoid name shadowing, we would assume that the binders introduce fresh names. In case of a $\lambda^{\twoheadrightarrow}$, the binder variable is added to the sharing information of all terms present in the type assignment and in case of $\lambda^{\multimap}$ the binder variable is skipped from

$$\frac{P \mid \Gamma \vdash M : \sigma \qquad P' \mid \Gamma'_x \sqcup x : \sigma \vdash N : \tau}{P \cup P' \mid \Gamma \sqcup \Gamma' \vdash (\texttt{let}\ x = M\ \texttt{in}\ N) : \tau}\ [\text{LET}]$$

$$\frac{P \mid \Gamma \vdash M : \sigma \qquad t \notin \texttt{fvs}(\Gamma) \cup \texttt{fvs}(P)}{P \mid \Gamma \vdash M : \forall t.\sigma}\ [\forall\ \text{I}] \qquad \frac{P \mid \Gamma \vdash M : \forall t.\sigma}{P \mid \Gamma \vdash M : [\tau \backslash t]\sigma}\ [\forall\ \text{E}]$$

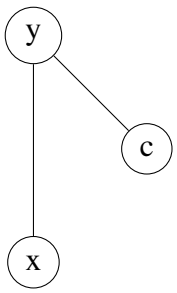
$$\frac{P, \pi \mid \Gamma \vdash M : \rho}{P \mid \Gamma \vdash M : \pi \Rightarrow \rho}\ [\Rightarrow \text{I}] \qquad \frac{P \mid \Gamma \vdash M : \pi \Rightarrow \rho \qquad P \vdash \pi}{P \mid \Gamma \vdash M : \rho}\ [\Rightarrow \text{E}]$$

$$\frac{\begin{array}{c} P \Rightarrow \texttt{ShFun}\ \phi \qquad P \vdash \Gamma \geq \phi \\ P \mid \Gamma^{[\varnothing \mapsto \{x\}]}, x^{\text{Vars}(\Gamma)} : \tau \vdash M : \tau' \end{array}}{P \mid \Gamma \vdash \lambda^{\twoheadrightarrow} x.M : \phi\,\tau\,\tau'}\ [\twoheadrightarrow \text{I}] \qquad \frac{\begin{array}{c} P \Rightarrow \texttt{ShFun}\ \phi \\ P \mid \Gamma \vdash M : \phi\,\tau\,\tau' \qquad P \mid \Gamma' \vdash N : \tau \end{array}}{P \mid \Gamma \oplus \Gamma' \vdash MN : \tau'}\ [\twoheadrightarrow \text{E}]$$

$$\frac{\begin{array}{c} P \Rightarrow \texttt{SeFun}\ \phi \qquad P \vdash \Gamma \geq \phi \\ P \mid \Gamma, x^{\varnothing} : \tau \vdash M : \tau' \end{array}}{P \mid \Gamma \vdash \lambda^{\multimap} x.M : \phi\,\tau\,\tau'}\ [\multimap \text{I}] \qquad \frac{\begin{array}{c} P \Rightarrow \texttt{SeFun}\ \phi \\ P \mid \Gamma \vdash M : \phi\,\tau\,\tau' \qquad P \mid \Gamma' \vdash N : \tau \end{array}}{P \mid \Gamma \circledast \Gamma' \vdash MN : \tau'}\ [\multimap \text{E}]$$

Figure 5.2: Connective Typing Rules

adding it to the sharing information to imply separation of resources. For example, consider the term $\lambda^{\multimap} c.\lambda^{\multimap} x.\lambda^{\twoheadrightarrow} y.cx$. The sharing graph and type assignment is shown in Fig. 5.3.



$$\Gamma = \{c^y : \tau \multimap \upsilon, x^y : \tau, y^{\{c,x\}} : \tau'\}$$

Figure 5.3: Sharing graph and typing context for $\lambda^{\multimap} c.\lambda^{\multimap} x.\lambda^{\twoheadrightarrow} y.cx$

The rules given in Fig. 5.4 are convenience rules for base cases that compute predicate constraints for types within a context.

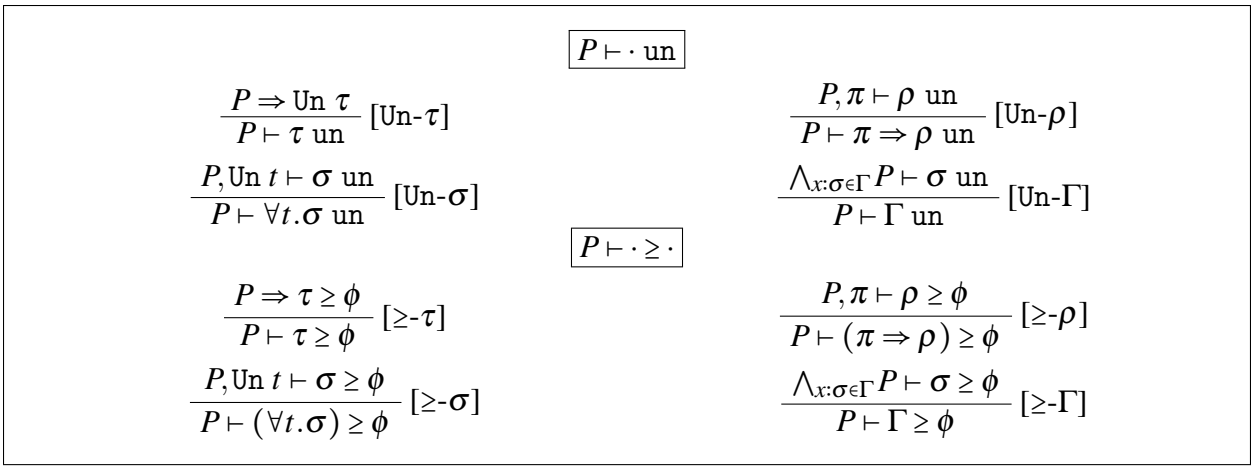


Figure 5.4: Entailment Rules for Base cases

5.3 Syntax Directed Typing rules

Ideally the typing rules and syntactic forms should have one-to-one correspondence. The type system explained in the §5.2 is not syntax directed and will not be fit to develop a type inference algorithm. In this section we will define syntax directed typing rules that will simplify our type inference system shown in Fig. 5.5

We define generalization and instantiation to express introduction and elimination of polymorphism in our syntax direct typing rules as follows:

Definition 3 (Instantiation)**.** For a type scheme $\sigma := \forall \vec{t}.P \Rightarrow \tau'$, we say $(Q \Rightarrow \tau)$ is an instance of σ and write it as $(Q \Rightarrow \tau) \sqsubseteq \sigma$, if there exists a $\vec{v}$ such that $\tau = [\vec{v}/\vec{t}]\tau'$ and $Q = [\vec{v}/\vec{t}]P$.

Definition 4 (Generalization)**.** For a type assignment Γ and qualified type ρ, we define type scheme $\texttt{Gen}(\Gamma, \rho) = \forall(\texttt{fvs}(\rho) \backslash \texttt{fvs}(\Gamma)).\rho$.

Definition 5 (Qualified Type Scheme)**.** A qualified type scheme is a pair of type scheme with a set of predicates written as $(P \mid \sigma)$, where $\sigma = \forall \vec{t}.Q \Rightarrow \tau$.

The elimination of polymorphism [$\forall$E] and qualified types[$\Rightarrow$E] is always done in the [Var^s], introduction of polymorphism [$\forall$I] and qualified types[$\Rightarrow$I] is done at `let` bindings [Let^s]. This

collapses the rules [∀E], [⇒E] and [ID] in one rule [VAR^s] where we use instantiation or specialization of type variables, and [∀I], [⇒I] and [Let] in one rule [Let^s] where we use generalization of type variables. [$\multimap$I^s] is used in occurence of $\lambda^{\multimap}$, and [$\twoheadrightarrow$I^s] is used in occurence of $\lambda^{\twoheadrightarrow}$. We would add the introduced abstraction variable, x, into the sharing context in case of [$\twoheadrightarrow$I^s]. We collapse the application rules [$\multimap$E] and [$\twoheadrightarrow$E] into one rule [App^s] where we check for sharing of the used variables in both the expressions and then assign a predicate of `ShFun` or `SeFun` depending on whether the variables are shared or not. The `un` predicates are added to the types of the terms that are not used directly in the expression or which are not in sharing with the terms used.

$$\frac{P \vdash \Gamma_{\vec{y}}\ \mathtt{un} \qquad (P \Rightarrow \tau) \sqsubseteq \sigma}{P \mid \Gamma, x^{\vec{y}} : \sigma \vdash^s x : \tau}\ [\mathrm{VAR}^s]$$

$$\frac{\begin{array}{c} Q \mid (\Gamma'_x \oplus \Gamma''_x) \circledast \Delta \vdash^s M : \upsilon \qquad P \vdash \Delta\ \mathtt{un} \\ P \mid (\Gamma_x \sqcup x^{\varnothing} : \sigma) \oplus \Gamma''_x \circledast \Delta \vdash^s N : \tau \qquad \sigma = \mathtt{Gen}(\{\Gamma' \oplus \Gamma''_x \circledast \Delta\}, Q \Rightarrow \upsilon) \end{array}}{P \mid (\Gamma \circledast \Gamma') \oplus \Gamma'' \circledast \Delta \vdash^s (\mathtt{let}\ x = M\ \mathtt{in}\ N) : \tau}\ [\mathrm{Let}^s]$$

$$\frac{\begin{array}{c} P \Rightarrow \mathtt{SeFun}\ \phi \qquad P \vdash \Gamma \geq \phi \\ P \mid \Gamma \circledast x^{\varnothing} : \tau \vdash^s M : \upsilon \end{array}}{P \mid \Gamma \vdash^s \lambda^{\multimap} x.M : \phi\,\tau\,\upsilon}\ [\multimap \mathrm{I}^s] \qquad \frac{\begin{array}{c} P \Rightarrow \mathtt{ShFun}\ \phi \qquad P \vdash \Gamma \geq \phi \\ P \mid \Gamma^{[\varnothing \mapsto \{x\}]} \oplus x^{\mathtt{Vars}(\Gamma)} : \tau \vdash^s M : \upsilon \end{array}}{P \mid \Gamma \vdash^s \lambda^{\twoheadrightarrow} x.M : \phi\,\tau\,\upsilon}\ [\twoheadrightarrow \mathrm{I}^s]$$

$$\frac{\begin{array}{c} P \mid \Gamma \circledast \Delta \vdash^s M : \phi\,\upsilon\,\tau \qquad P \mid \Gamma' \circledast \Delta \vdash^s N : \upsilon \qquad P \vdash \Delta\ \mathtt{un} \\ (\Gamma \mathbin{\widetilde{\oplus}} \Gamma' \wedge (P \Rightarrow \mathtt{ShFun}\ \phi)) \vee (\Gamma \mathbin{\widetilde{\circledast}} \Gamma' \wedge (P \Rightarrow \mathtt{SeFun}\ \phi)) \end{array}}{P \mid \Gamma \sqcup \Gamma' \circledast \Delta \vdash^s MN : \tau}\ [\mathrm{App}^s]$$

Figure 5.5: Syntax Directed Typing Rules

The [Let^s] rule defines an expression within another expression locally i.e. x would not be in scope other than its use in N. We partition the typing context into multiple parts. Γ contains the variables that exists exclusively in M and Γ' which are exclusively in N. Γ'' is common to both M and N while Δ is not used in either expressions M or N. Thus Γ and Γ' will be completely separate from each other while Γ'' would be in sharing with both Γ and Γ'. Δ would have to be unrestricted as it is not being used either in M or in N. The sharing of x with Γ would depend on whether Γ and Γ' are completely disjoint or empty. For the application rule [App^s] the type

assignment Γ would contain variables for M and Γ' for N. If they are completely separate, it would be a separating function application and M would be assigned a type $\tau' \twoheadrightarrow \tau$ else, they would have to be completely sharing and M would be assigned a type $\tau' \to \tau$. The conditions outlined for [App^s] do not clearly look as if they are syntax directed and in the cases where the type checker cannot directly infer if the resources used are separate or shared, the user would be expected to provide it using type annotations. $\Gamma \tilde{\circledast} \Delta$ is an assertion that there exists a proof either possible to find by inspecting the type assignment, or provided by the user that $\Gamma \circledast \Delta$ is defined. Similarly, $\Gamma \tilde{\oplus} \Delta$ means that $\Gamma \oplus \Delta$ is well defined.

We now state two important theorems regarding the type system and the syntax directed type system. The proofs of both the theorems are given in Appendix B.

Theorem 5.1 (Soundness of $\vdash^s$)**.** *If* $P \mid \Gamma \vdash^s M : \tau$ *then* $P \mid \Gamma \vdash M : \tau$

The soundness property captures the essence that derivations in the syntax directed type system follow the original type system.

Theorem 5.2 (Completeness of $\vdash^s$)**.** *If* $P \mid \Gamma \vdash M : \sigma$ *then* $\exists Q, \tau$ *such that* $Q \mid \Gamma \vdash^s M : \tau$ *and* $(P \mid \sigma) \sqsubseteq \mathit{Gen}(\Gamma, Q \Rightarrow \tau)$

The completeness theorem states that we can always find predicates Q and a suitable type τ for the term M under the assumptions Γ using syntax directed type system if the original type system asserts that the typing judgement indeed exists.

5.4 Type Inference and Algorithm $\mathcal{M}$

We now describe the type inference algorithm based on the previously defined syntax directed type system. We use a variation of algorithm $\mathcal{M}$ (Lee & Yi, 1998) for type inference. We address three independent concerns in the type inference algorithm. The first being treatment of polymorphism to be same as Hindley-Milner style. The second, we introduce `Un` predicates for types that are unrestricted. We track this with the help of carrying a collection of used variables throughout the

algorithm which detects whether a variable is discarded or used multiple types. The third being accounting for sharing of the variables. The complete algorithm is outlined in Fig. 5.7. The input to the algorithm includes the term M whose type has to be inferred, τ is the expected type of the term, S is the current substition and the sharing information Ψ. The output includes the set of new predicates P that are generated, the new set of substitutions S', the used variables Σ, and the new sharing information Ψ'. The type variables u with a subscript denote fresh variables.

We define some auxilary functions in Fig. 5.6 to lift predicates into the type system. $Leq(\phi, \Gamma)$ adds the predicate $\phi \geq \sigma$ for all σ that are in Γ. $\texttt{Weaken}(x, \sigma, \Sigma)$ adds the unrestricted predicate to the type σ if x does not belong to Σ. $\texttt{Un}(\Gamma)$ adds an unrestricted predicate to all the types of the variables that are in the domain of Γ. $\texttt{GenI}(\Gamma, P \Rightarrow \tau)$ generalizes the qualified type to a type scheme. $\mathcal{C}(\Gamma, \Psi, \Sigma)$ computes the sharing information of variables in Γ restricted to the variables in Σ.

$$Leq(\phi, \Gamma) = \bigcup_{(x:\sigma)\in\Gamma} \{P \mid P \vdash \sigma \geq \phi\} \qquad \texttt{Un}(\Gamma) = \bigcup_{(y:\sigma)\in\Gamma} \{P \mid P \vdash \sigma \ \texttt{un}\}$$

$$\texttt{Weaken}(x, \sigma, \Sigma) = \begin{cases} P & \text{if } x \notin \Sigma, P \vdash \sigma \ \texttt{un} \\ \varnothing & \textit{otherwise} \end{cases} \qquad \texttt{GenI}(\Gamma, P \Rightarrow \tau) = \forall(\texttt{fvs}(P) \cup \texttt{fvs}(\tau) \backslash \texttt{fvs}(\Gamma))).P \Rightarrow \tau$$

$$\mathcal{C}(\Gamma, \Psi, \Sigma) = \bigcup_{x\in\texttt{dom}(\Gamma)\wedge x\in\Sigma} \Psi(x)$$

Figure 5.6: Auxiliary Functions

The first case in algorithm $\mathcal{M}$ in Fig. 5.7 describes the variable case, where we are given the variable identifier and the expected type. We try to unify the expected type τ with the derived type scheme ν from the type assignment Γ. The return values of the algorithm are a new set of predicates which are nothing but instantiated version of the predicates obtained from the typing scheme, the variable x being used and the new substitution which is combination of the unification algorithms output and the original substitution. There is no change in the sharing information and

Ψ is returned as is.

In next case of sharing function introduction rule $\lambda^{\twoheadrightarrow}x.M$, as per the [$\twoheadrightarrow$I] rules, the entitites returned are union of four predicate categories. The first is assigning a the predicate of the function to be a sharing function $\texttt{ShFun}\ u_1$, where u_1 is a new type variable for the type of function argument x. The second assigning the function to be less restricting than the other variables in the typing assignment Γ. The third, assigning an unrestricted predicate to the binding variable x if it has not been used anywhere in the lambda body M, which is done by the `Weaken` function and the fourth being the predicates generated by recursively type checking the body of the lambda expression M. We create sharing links for x with all the variables within the typing context Γ in updated Ψ''. The case of separating function $\lambda^{*}x.M$ is very similar to the previous case of $\lambda^{\twoheadrightarrow}x.M$ except the that there is no sharing information to be updated as the argument to the function is separate from its body and the function predicate assigned is $\texttt{SeFun}\ u_1$ to denote this very separating relation instead of $\texttt{ShFun}\ u_1$

In the application case the algorithm typechecks the subexpressions M as a function type having an input of the type of N. The additional check done here is to identify whether M has a sharing application or a separating application. If all the variables used in M are also used in N then it is a sharing application i.e. M is assigned a sharing function predicate `ShFun` else if the used variables in M are disjoint to the variables that are shared by N or if the variables shared by variables in M are disjoint to the variables that are used in N, M is assigned the predicate type `SeFun` . Incase of a separating function application, the variables that are used in both are marked as unrestricted. This is captured by $\texttt{Un}(\Gamma|_{\Sigma\cap\Sigma'})$ where $\Gamma|_{\Sigma}$ means the typing assignment Γ restricted to the variables in Σ. In the polymorphic `let` case we first check for the type of the expression M and ensure that the variable binding x is not usd in it to avoid recursive definition which would lead to infinite types. We then generalize the computed type to σ and then check the type of expression N by expanding the type assignment Γ with the variable x and type scheme σ.

We now state the soundness theorem for the algorithm $\mathcal{M}$.

Theorem 5.3 (Soundness of $\mathcal{M}$)**.** *if* $\mathcal{M}(S,\Psi,\Gamma\vdash M:\tau) = P,S',\Sigma,\Psi'$ *then* $S'P \mid S'(\Gamma|_{\Sigma}) \vdash M : S'\tau$

The soundness theorem for algorithm $\mathcal{M}$ ensures that the original type system rules are obeyed by the type inference algorithm. The proof is given in Appendix B.

$$\boxed{\mathcal{M}(S,\Psi,\Gamma \vdash M:\tau) = P,S',\Sigma,\Psi'}$$

$$\mathcal{M}(S,\Psi,\Gamma \vdash x:\tau) = ([\vec{u}/\vec{t}]P), S' \circ S, \{x\}, \Psi$$
$$\text{where} \quad (x:\forall \vec{t}.P \Rightarrow \upsilon) \in S\Gamma$$
$$S' = \mathcal{U}([\vec{u}/\vec{t}]\upsilon, S\tau)$$

$$\mathcal{M}(S,\Psi,\Gamma \vdash \lambda^{\rightarrow} x.M:\tau) = \{P \cup Q\}, S', \Sigma\backslash x, \Psi''$$
$$\text{where} \quad P;S';\Sigma;\Psi' = \mathcal{M}(\mathcal{U}(\tau, u_1u_2u_3) \circ S, \Psi, \Gamma, x:u_2 \vdash M:u_3)$$
$$Q = \{\texttt{ShFun}\ u_1\} \cup \texttt{Leq}(u_1, \Gamma|_\Sigma) \cup \texttt{Weaken}(x, u_2, \Sigma)$$
$$\Psi'' = \{\forall_{y \in \texttt{dom}(\Psi')}.\Psi'(y) + x\} \cup \{(x, \{y \mid y \in \texttt{dom}(\Gamma)\})\}$$

$$\mathcal{M}(S,\Psi,\Gamma \vdash \lambda^{*} x.M:\tau) = \{P \cup Q\}, S', \Sigma\backslash x, \Psi''$$
$$\text{where} \quad P;S';\Sigma;\Psi' = \mathcal{M}(\mathcal{U}(\tau, u_1u_2u_3) \circ S, X; \Gamma, x:u_2 \vdash M:u_3)$$
$$Q = \{\texttt{SeFun}\ u_1\} \cup \texttt{Leq}(u_1, \Gamma|_\Sigma) \cup \texttt{Weaken}(x, u_2, \Sigma)$$
$$\Psi'' = \Psi' \cup \{(x, \{x\})\}$$

$$\mathcal{M}(S,\Psi,\Gamma \vdash MN:\tau) = \{P \cup P' \cup Q\}, R', \Sigma \cup \Sigma', \Psi''$$
$$\text{where} \quad P;R;\Sigma;\Psi' = \mathcal{M}(S,\Psi,\Gamma \vdash M:u_1u_2\tau)$$
$$P';R';\Sigma';\Psi'' = \mathcal{M}(SR,\Psi',S\Gamma \vdash N:u_2)$$
$$\text{if } \mathcal{C}(\Gamma,\Psi'',\Sigma) = \mathcal{C}(\Gamma,\Psi'',\Sigma')$$
$$\text{then } Q = \{\texttt{ShFun}\ u_1\}$$
$$\text{else if } (\Sigma \# \mathcal{C}(R\Gamma,\Psi'',\Sigma') \text{ and } \Sigma' \# \mathcal{C}(R\Gamma,\Psi'',\Sigma))$$
$$\text{then } Q = \{\texttt{SeFun}\ u_1\}$$

$$\mathcal{M}(S,\Psi,\Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N:\tau) = (P \cup Q), R', \Sigma \cup \{\Sigma'\backslash x\}, \Psi''$$
$$\text{where} \quad P;R;\Sigma;\Psi' = \mathcal{M}(S,\Psi,\Gamma \vdash M:u_1)$$
$$\sigma = \texttt{GenI}(R\Gamma; R(P \Rightarrow u_1))$$
$$P';R';\Sigma';\Psi'' = \mathcal{M}(R,\Psi',\Gamma,x:\sigma \vdash N:\tau)$$
$$Q = \texttt{Un}(\Gamma|_{\Sigma \cap \Sigma'}) \cup \texttt{Weaken}(x,\sigma,\Sigma')$$

Figure 5.7: Type Inference Algorithm $\mathcal{M}$

Chapter 6

QuB Extention and Datatypes

In this chapter we discuss how QuB can be extended to have a kind system which makes the type language powerful enough to accept user defined datatypes in §6.1. We also discuss how we can encode sums and multiplicative and additive products in §6.2.

6.1 Kind System

In the original system, it would be tedious to extend the system with new types as we would have to introduce new syntax and associated typing rules for each of the new syntax in our core language. The kind system generalizes the concept of adding new types by abstracting them as type constructors. This generalization alleviates the burden of modifying the core language by treating all types to be instances of type constructors. The idea was introduced by Barendregt (1991) and is used by Jones (1993) for qualified types. We will follow Jones' approach to add the language of type constructors and kinds in our system. The modified type system is shown in Fig. 6.1. All types have kind $\star$ and the kind of the type constructors depends on its arity.

Addition of a kind system changes the treatement to the type inference algorithm in some detail. All types and type constructors have to be annotated with their kind. T^{κ} denotes type constructors and $\tau^{\kappa' \to \kappa} \tau^{\kappa'}$ denotes application of types. The type constructor application rule that computes kinds is given in Fig. 6.2 where τ is of kind $\kappa' \to \kappa$ and τ' is of kind κ'. The application of both the constructors would result in a kind κ. $\twoheadrightarrow$, $\overset{!}{\twoheadrightarrow}$, $\to$ and $\overset{!}{\to}$ are now treated as type constructors with an arity of two and would have a kind $\star \to \star \to \star$, The type constructor for List will have a kind $\star \to \star$ while types like Int and Float will have a kind $\star$.

$$
\begin{array}{rl}
\text{Type Variables} & t, u, v \in \text{Type Variables} \\
\text{Kinds} & \kappa ::= \kappa \mid \kappa' \to \kappa \\
\text{Types} & \tau^{\kappa} ::= t^{\kappa} \mid T^{\kappa} \mid \tau^{\kappa' \to \kappa} \tau^{\kappa'} \\
\text{Type Constructors} & T^{\kappa} \in \mathcal{T}^{\kappa} \text{ where } \{\otimes, \&, \oplus, \overset{!}{\multimap}, \multimap, \overset{!}{\to}, \to\} \subseteq \mathcal{T}^{\star \to \star \to \star} \\
\text{Predicates} & \pi ::= \texttt{Un}\ \tau \mid \texttt{SeFun}\ \tau \mid \texttt{ShFun}\ \tau \mid \tau \geq \tau' \\
\text{Qualified Types} & \rho ::= \tau \mid \pi \Rightarrow \rho \\
\text{Type schemes} & \sigma ::= \rho \mid \forall t.\sigma
\end{array}
$$

Figure 6.1: Extended QuB Types and Kinds

$$
\frac{\tau :: \kappa' \to \kappa \qquad \tau' :: \kappa'}{\tau \tau' :: \kappa}
$$

Figure 6.2: Constructor Application Rule

The unification of types is now done via a modified version of Robinson's algorithm (1965) is used in order to deduce the most general unifier for type constructors. Formally we define S to be the *most general unifier* for type constructors T and T' if:

1. S is a unifier for T and T'.
2. For every unifier S' of T and T' we can write T' in a form of RS where R some kind preserving substitution.

We write $T \overset{S}{\sim}_{\kappa} T'$ for assertion that S is the unifier of the constructor types $T, T' \in T^{\kappa}$. The rules in Fig. 6.3 describe the unification algorithm for type constructors. [KVar] and [KVar'] contain and additional constraint of the type variable t to not be free in the type constructor's T type variables to ensure the unification does not lead to infinite types. The [KApply] rule states that type constructors of the form TT' can be unified with HH' only if T and H can be unified which asserts that they have to have the same kind $\kappa' \to \kappa$.

$$t \overset{id}{\sim}_{\kappa} t \qquad \text{([ID-KVar])}$$

$$T \overset{id}{\sim}_{\kappa} T \qquad \text{([ID-KConst])}$$

$$t \overset{[T/t]}{\sim}_{\kappa} T, t \notin \texttt{fvs}(T) \qquad \text{([KVar])}$$

$$T \overset{[T/t]}{\sim}_{\kappa} t, t \notin \texttt{fvs}(T) \qquad \text{([KVar'])}$$

$$\frac{T \overset{S}{\sim}_{\kappa' \to \kappa} T' \qquad SH \overset{S'}{\sim}_{\kappa'} SH'}{TT' \overset{SS'}{\sim}_{\kappa} HH'} \qquad \text{([KApply])}$$

Figure 6.3: Kind Preserving Unification of Type Constructors

$$\begin{aligned}
\text{Term Variables} \quad & x, y, z \in \text{Var} \\
\text{Expressions} \quad & M, N ::= x \mid \lambda^{*} x.M \mid \lambda^{\alpha} x.M \mid MN \mid \texttt{let}\ x = M\ \texttt{in}\ N \\
& \mid \langle M, N \rangle \mid \texttt{let}\ \langle x, y \rangle = M\ \texttt{in}\ N \mid \langle M; N \rangle \mid \texttt{fst}\ M \mid \texttt{snd}\ M \\
& \mid \texttt{case}\ M\ \texttt{of}\ \{\texttt{inl}\ x \mapsto N; \texttt{inr}\ y \mapsto N'\} \mid \texttt{inl}\ x \mid \texttt{inr}\ y
\end{aligned}$$

Figure 6.4: Extended QuB Language Syntax

add more details of how this helps and proofs remain the same

6.2 Pairs and Sums in QuB

Introduction of two kinds of arrows in our type system leads to different flavors of pairs. This distinction cannot be made in intuitionistic logic as the structural rules allow re-use of propositions. But due to restrictions in weakening and contraction we obtain two kinds of pairs, additive and multiplicative. In this section we illustrate how the extented QuB can be used to introduce new types. We introduce syntax and type constructors for multiplicative pairs in §6.2.1 and the same for additive pairs in §6.2.2. We then introduce the syntax and type constructors for sum types in

§6.2.3 and illustrate that they indeed work as expected.

6.2.1 Multiplicative Pair Type

Lambda encoding of muliplicative pairs is given in Fig. 6.5. The typing rules are given in Fig. 6.6. We give the proofs of the typing rules in Appendix A.1.2. The meaning of a multiplicative pair can be thought of as having separate resouce entities together in the program environment context and they would have to be explicitly disposed off. Failure to do so, would raise a type error regarding the resources not being unrestricted.

$$\otimes \in \mathcal{T}^{\star\to\star\to\star}$$
$$\tau \otimes \tau' = \tau \multimap \tau' \multimap (\tau \multimap \tau' \multimap \upsilon) \multimap \upsilon$$
$$(,) = \lambda^* x.\lambda^* y.\lambda^* f.fxy$$

Figure 6.5: Multiplicative Pair

$$\frac{P \mid \Gamma \vdash M : \tau \qquad P \mid \Delta \vdash N : \tau'}{P \mid \Gamma \circledast \Delta \vdash \langle M, N \rangle : \tau \otimes \tau'} [\otimes I]$$

$$\frac{P \mid \Gamma \vdash M : \tau \otimes \tau' \qquad P \mid \{x^{\{\vec{z} \mid \vec{z} \subseteq \mathtt{Vars}(\Gamma')\}} : \tau\} \circledast \{y^{\{\vec{z'} \mid \vec{z'} \subseteq \mathtt{Vars}(\Gamma')\}} : \tau'\} \sqcup \Gamma'_{x,y} \vdash N : \upsilon}{P \mid \Gamma \sqcup \Gamma' \vdash (\mathtt{let}\ \langle x, y \rangle = M\ \mathtt{in}\ N) : \upsilon} [\otimes E]$$

Figure 6.6: Derivable Typing Rules for Multiplicative Pair

6.2.2 Additive Pair Type

The lambda encoding for additive pairs is given in **??**. The typing rules are given in Fig. 6.8. We give proof of derivations for the rules in Appendix A.1.1. Additive pairs can be thought of as program entities that have sharing resources. We use `fst` and `snd` as projection functions on the additive pair to obtain the constituent objects. It is worthwhile to note that we do not provide

these projection functions for multiplicative pairs so as to restrict the programmer from implicitly discarding one of the components of the multipicative pair.

$$
\begin{aligned}
\& &\in \mathcal{T}^{\star\to\star\to\star} \\
\tau \,\&\, \tau' &= \tau \multimap \tau' \to (\tau \multimap \tau' \to \upsilon) \to \upsilon \\
(;) &= \lambda^{\star} x.\lambda^{\alpha} b y.\lambda^{\alpha} f.fxy
\end{aligned}
$$

Figure 6.7: Additive Pair

$$
\frac{P \mid \Gamma \vdash M : \tau \qquad P \mid \Delta \vdash N : \tau'}{P \mid \Gamma \oplus \Delta \vdash \langle M;N \rangle : \tau \,\&\, \tau'}\ [\&\ \mathrm{I}]
$$

$$
\frac{P \mid \Gamma \vdash M : \tau \,\&\, \tau'}{P \mid \Gamma \vdash \mathtt{fst}\ M : \tau}\ [\&\ \mathrm{E}_1] \qquad\qquad \frac{P \mid \Gamma \vdash M : \tau \,\&\, \tau'}{P \mid \Gamma \vdash \mathtt{snd}\ M : \tau'}\ [\&\ \mathrm{E}_2]
$$

Figure 6.8: Derivable Typing Rules for Additive Pair

6.2.3 Sum Type

The lambda encoding of sum types is given in Fig. 6.9 and the derivable typing rules are given in Fig. 6.10. We give proof of derivations in Appendix A.2. Sum types can be thought of as a choice between two types. At a given point of time only one of the two types would exist. We provide two functions to be used has handlers for both the types. The `case` M `of` $\{f;g\}$ construct is the deconstructor that would decide which function to use f or g depending on the value evaluated for M. The return type of both the functions f and g would have to be the same for them to have correct typing. Encoding sum types within the language would be helpful in defining data structures such as lists, trees and option types.

$$
\begin{aligned}
&\oplus \in \mathcal{T}^{\star\to\star\to\star} \\
&\tau \oplus \tau' = (\tau \to \upsilon) \to (\tau' \to \upsilon) \to \upsilon \\
&\texttt{inl} : \tau \twoheadrightarrow (\tau \oplus \tau') \\
&\texttt{inl} = \lambda^{*} x.\lambda^{\alpha} f.\lambda^{\alpha} g.fx \\
&\texttt{inr} : \tau' \twoheadrightarrow (\tau \oplus \tau') \\
&\texttt{inr} = \lambda^{*} y.\lambda^{\alpha} f.\lambda^{\alpha} g.gy
\end{aligned}
$$

Figure 6.9: Sum Type

$$
\frac{P \,|\, \Gamma \vdash M : \tau}{P \,|\, \Delta \vdash \texttt{inl}\ M : \tau \oplus \tau'}\ [\oplus \mathrm{I}_l] \qquad \frac{P \,|\, \Gamma \vdash M : \tau'}{P \,|\, \Delta \vdash \texttt{inr}\ M : \tau \oplus \tau'}\ [\oplus \mathrm{I}_r]
$$

$$
\frac{P \,|\, \Gamma \vdash M : \tau \oplus \tau' \qquad P \,|\, \Gamma \circledast x : \tau \vdash N : \upsilon \qquad P \,|\, \Gamma \circledast y : \tau' \vdash N' : \upsilon}{P \,|\, \Gamma \vdash \texttt{case}\ M\ \texttt{of}\ \{\texttt{inl}\ x \mapsto N; \texttt{inr}\ y \mapsto N'\} : \upsilon}\ [\oplus \mathrm{E}]
$$

Figure 6.10: Derivable Typing Rules for Sum Type

6.3 Generic Type Constructors

To leverage the full power of introducing the sharing concept in a programming language, we would want to have users be able to define datatypes with sharing and separation of its constituent resources. We add two new typing rules as shown in Fig. 6.11 for handling generic datatypes using type constructors. In the current implimentation, we only consider a specific case of type constructors where all the type variables involved are either all shared or all separate.

How do we write about type constructors?

$$
\frac{P \,|\, \Gamma \oplus x : \tau^{\kappa} \vdash C : \tau^{\kappa} \to T}{P \,|\, \Gamma \vdash Cx : T}\ [\text{C-sh}] \qquad \frac{P \,|\, \Gamma \circledast x : \tau^{\kappa} \vdash C : \tau^{\kappa} \twoheadrightarrow T}{P \,|\, \Gamma \vdash Cx : T}\ [\text{C-se}]
$$

Figure 6.11: User Defined Datatypes

We illustrate using examples how the type system works. The sharing pair has shared resources

and using `fst` and `snd` works to get both the individual constituents as shown in Fig. 6.12 The (!!) denots that a and b are in sharing. The fstp and sndp functions typecheck as expected.

```
data Pair' a b = ShP !! a b

fstp::Pair' a b -> a
fstp (ShP x y) = x

sndp :: Pair' a b -> b
sndp (ShP x y) = y
```

Figure 6.12: Shared Pairs

Separating pair is defined in Fig. 6.13. The functions fst and snd do not type check as on of the resources is being dropped and we would need the resource to be unrestricted.

```
data Pair a b = SeP a b

fst :: Pair' a b -> a
fst (SeP x y) = x

snd :: Pair' a b -> b
snd (SeP x y) = y
```

Figure 6.13: Separating Pair

Chapter 7

Conclusion and Future Work

missing pieces here.

We have not given any sematic model for our language. The typing context can be thought of as to generate a sharing graph. It is similar to a separating graph in λ_{sep} (Atkey, 2004) automoatic detection of addtitive and multiplicative types using polymorphism we assume that polymorphism does not affect our case becuase the resources we use are monoic types. We may have to incorporate work on polymorphism Collinson et al. (2005) for more generalized setting.

References

Ahmed, A., Fluet, M., & Morrisett, G. (2007). L^3: A linear language with locations. 77(4), 397–449.

Atkey, R. (2004). A λ-calculus for resource separation. In J. Díaz, J. Karhumäki, A. Lepistö, & D. Sannella (Eds.), *Automata, Languages and Programming*, volume 3142 (pp. 158–170). Springer Berlin Heidelberg.

Barendregt, H. (1991). Introduction to generalized type systems. *Journal of Functional Programming*, 1(2), 125–154.

Bernardy, J.-P., Boespflug, M., Newton, R. R., Peyton Jones, S., & Spiwack, A. (2017). Linear haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2, 5:1–5:29.

Collinson, M., Pym, D., & Robinson, E. (2005). On bunched polymorphism. In L. Ong (Ed.), *Computer Science Logic*, Lecture Notes in Computer Science (pp. 36–50). Springer, Berlin, Heidelberg.

Damas, L. & Milner, R. (1982). Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82 (pp. 207–212).: ACM.

Girard, J.-Y. (1987). Linear logic. *Theoretical Computer Science*, 50(1), 1–101.

Girard, J.-Y. (1993). On the unity of logic. *Annals of Pure and Applied Logic*, 59(3), 201–217.

Girard, J.-Y., Taylor, P., & Lafont, Y. (1989). *Proofs and Types*. Cambridge University Press.

Jones, M. P. (1993). A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93 (pp. 52–61).: ACM.

Jones, M. P. (1994). A theory of qualified types. *Science of Computer Programming*, 22(3), 231 – 256.

Jones, M. P. (1997). First-class polymorphism with type inference. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97 (pp. 483–496).: ACM.

Jones, M. P. (2000). Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming*: Springer-Verlag LNCS 1782.

Lee, O. & Yi, K. (1998). Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4), 707–723.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 348–375.

Morris, J. G. (2016). The best of both worlds: Linear functional programming without compromise. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016 (pp. 448–461).: ACM.

O'Hearn, P. W. (1999). Resource interpretations, bunched implications and the alpha lambda-calculus. In *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications*, TLCA '99 (pp. 258–279).: Springer-Verlag.

O'Hearn, P. W. (2003). On bunched typing. *Journal of functional Programming*, 13(4), 747–796.

O'Hearn, P. W. & Pym, D. J. (1999). The logic of bunched implications. *The Bulletin of Symbolic Logic*, 5(2), 215–244.

Pym, D. J. (2002). *The Semantics and Proof Theory of the Logic of Bunched Implications*. Applied Logic Series. Springer Netherlands.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1), 23–41.

Wadler, P. (1993). A taste of linear logic. In A. M. Borzyszkowski & S. Sokolowski (Eds.), *Mathematical Foundations of Computer Science 1993: 18th International Symposium, MFCS'93 Gdańsk, Poland, August 30–September 3, 1993 Proceedings* (pp. 185–210). Springer Berlin Heidelberg.

Appendix A

Derivations for Products and Sums

We start this section by adding few auxilary defitions for terms and types from first principles. By convention, we denote an empty typing context by I and empty predicate context with $\varnothing$

A.1 Derivable Typing Rules For Product Types (Addiditive and Multiplicative Pairs)

Pairs now have two meanings. Either they have sharing resources or they have separating resources. We define each of them below.

A.1.1 Additive Pairs

If the resources are in sharing we say they are additive pairs.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \mid y^{xf} : \tau' \vdash y : \tau'}\,[\mathrm{ID}] \quad \dfrac{\dfrac{}{\varnothing \mid x^{fy} : \tau \vdash x : \tau}\quad \dfrac{}{\varnothing \mid f^{xy} : \tau \twoheadrightarrow \tau' \twoheadrightarrow \upsilon \vdash f : \tau \twoheadrightarrow \tau' \twoheadrightarrow \upsilon}\,[\mathrm{ID}]}{\varnothing \mid x^{xy} : \tau \oplus f^{xy} : \tau \twoheadrightarrow \tau' \twoheadrightarrow \upsilon \vdash fx : (\tau' \twoheadrightarrow \upsilon)}\,[\multimap E]}{\varnothing \mid y^{xf} : \tau' \oplus x^{yf} : \tau \oplus f^{xy} : \tau \twoheadrightarrow \tau' \twoheadrightarrow \upsilon \vdash fxy : \upsilon}\,[\twoheadrightarrow E]}{\varnothing \mid x^{yf} : \tau \oplus y^{xf} : \tau' \oplus f^{xy} : \tau \twoheadrightarrow \tau' \twoheadrightarrow \upsilon \vdash fxy : \upsilon}\,[EXCH]}{\varnothing \mid x^{y} : \tau \oplus y^{x} : \tau' \vdash \lambda^{\twoheadrightarrow} f.fxy : (\tau \twoheadrightarrow \tau' \twoheadrightarrow \upsilon) \twoheadrightarrow \upsilon}\,[\twoheadrightarrow I]}{\varnothing \mid x^{\varnothing} : \tau \vdash \lambda^{\twoheadrightarrow} y.\lambda^{\twoheadrightarrow} f.fxy : \tau' \twoheadrightarrow (\tau \twoheadrightarrow \tau' \twoheadrightarrow \upsilon) \twoheadrightarrow \upsilon}\,[\twoheadrightarrow I]}{\varnothing \mid I \vdash \lambda^{\multimap} x.\lambda^{\twoheadrightarrow} y.\lambda^{\twoheadrightarrow} f.fxy : \tau \multimap \tau' \twoheadrightarrow (\tau \twoheadrightarrow \tau' \twoheadrightarrow \upsilon) \twoheadrightarrow \upsilon}\,[\multimap \mathrm{I}]$$

We assign a new type symbol (&) to describe type of additive pair

$$\tau \,\&\, \tau' = (\tau \twoheadrightarrow \tau' \twoheadrightarrow \upsilon) \twoheadrightarrow \upsilon$$

We now define sharing (additive) pair constructor as:

$$;\,:\tau \multimap \tau' \twoheadrightarrow (\tau \,\&\, \tau')$$

$$; = \lambda^{\multimap} x.\lambda^{\twoheadrightarrow} y.\lambda^{\twoheadrightarrow} f.fxy$$

We now derive left and right projections or deconstructors for sharing pairs below:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \mid x^{y} : \tau \vdash x : \tau}\,[\text{ID}]}{\varnothing \mid x^{y} : \tau \oplus y^{x} : \tau' \vdash x : \tau}\,[\text{WKN-SH}]}{\varnothing \mid x^{\varnothing} : \tau \vdash \lambda^{\twoheadrightarrow} y.x : \tau' \twoheadrightarrow \tau}\,[\twoheadrightarrow I]}{\varnothing \mid I \vdash \lambda^{\twoheadrightarrow} x.\lambda^{\twoheadrightarrow} y.x : \tau \twoheadrightarrow \tau' \twoheadrightarrow \tau}\,[\twoheadrightarrow \text{I}]$$

$$\texttt{fst} : \tau \twoheadrightarrow \tau' \twoheadrightarrow \tau$$

$$\texttt{fst} = \lambda^{\twoheadrightarrow} x.\lambda^{\twoheadrightarrow} y.x$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \mid y^{x} : \tau' \vdash y : \tau'}\,[\text{ID}]}{\varnothing \mid x^{y} : \tau \oplus y^{x} : \tau' \vdash y : \tau'}\,[\text{WKN-SH}]}{\varnothing \mid x^{\varnothing} : \tau \vdash \lambda^{\twoheadrightarrow} y : \tau' \twoheadrightarrow \tau'}\,[\twoheadrightarrow I]}{\varnothing \mid I \vdash \lambda^{\twoheadrightarrow} x.\lambda^{\twoheadrightarrow} y.y : \tau \twoheadrightarrow \tau' \twoheadrightarrow \tau'}\,[\twoheadrightarrow I]$$

$$\texttt{snd} : \tau \twoheadrightarrow \tau' \twoheadrightarrow \tau'$$

$$\texttt{snd} = \lambda^{\twoheadrightarrow} x.\lambda^{\twoheadrightarrow} y.y$$

A.1.2 Multiplicative Pairs

If the resources are separate we say they are multiplicative or separating pairs.

$$\dfrac{\dfrac{}{\varnothing \mid y^{\varnothing} : \tau' \vdash y : \tau'}\,[\mathrm{ID}] \quad \dfrac{\dfrac{}{\varnothing \mid x^{\varnothing} : \tau \vdash x : \tau}\,[\mathrm{ID}] \quad \dfrac{}{\varnothing \mid f^{\varnothing} : \tau \multimap \tau' \multimap \upsilon \vdash f : \tau \multimap \tau' \multimap \upsilon}\,[\mathrm{ID}]}{\varnothing \mid x^{\varnothing} : \tau \otimes f^{\varnothing} : \tau \multimap \tau' \multimap \upsilon \vdash fx : (\tau' \multimap \upsilon)}\,[\multimap \mathrm{E}]}{\dfrac{\varnothing \mid y^{\varnothing} : \tau' \otimes x^{\varnothing} : \tau \otimes f^{\varnothing} : \tau \multimap \tau' \multimap \upsilon \vdash fxy : \upsilon}{\dfrac{\varnothing \mid x^{\varnothing} : \tau \otimes y^{\varnothing} : \tau' \otimes f^{\varnothing} : \tau \multimap \tau' \multimap \upsilon \vdash fxy : \upsilon}{\dfrac{\varnothing \mid x^{\varnothing} : \tau \otimes y^{\varnothing} : \tau' \vdash \lambda^{\multimap} f.fxy : (\tau \multimap \tau' \multimap \upsilon) \multimap \upsilon}{\dfrac{\varnothing \mid x^{\varnothing} : \tau \vdash \lambda^{\multimap} y.\lambda^{\multimap} f.fxy : \tau' \multimap (\tau \multimap \tau' \multimap \upsilon) \multimap \upsilon}{\varnothing \mid I \vdash \lambda^{\multimap} x.\lambda^{\multimap} y.\lambda^{\multimap} f.fxy : \tau \multimap \tau' \multimap (\tau \multimap \tau' \multimap \upsilon) \multimap \upsilon}\,[\equiv]}\,[\multimap \mathrm{I}]}\,[\multimap \mathrm{I}]}\,[EXCH]}$$

We assign a type symbol ($\otimes$) for the multiplicative pair

$$\tau \otimes \tau' = (\tau \multimap \tau' \multimap \upsilon) \multimap \upsilon$$

We can now define separating (multiplicative) pair as:

$$, : \tau \multimap \tau' \multimap (\tau \otimes \tau')$$

$$, = \lambda^{\multimap} x.\lambda^{\multimap} y.\lambda^{\multimap} f.fxy$$

We will abuse the notation of lambda calculus for ; and , as use them as infix operators for syntactic convinence

$$\langle x, y \rangle \equiv (,)xy$$

$$\langle x; y \rangle \equiv (;)xy$$

We are now in a position to write the proof derivations for Fig. 6.8 and Fig. 6.6 using the auxilary definitions from above.

$$\frac{P \mid \Gamma \vdash M : \tau \qquad P \mid \Delta \vdash N : \tau'}{P \mid \Gamma \oplus \Delta \vdash \langle M; N \rangle : \tau \,\&\, \tau'}\,[\&\ \mathrm{I}]$$

$$\frac{P \mid \Gamma \vdash M : \tau \,\&\, \tau'}{P \mid \Gamma \vdash \mathtt{fst}\ M : \tau}\,[\&\ \mathrm{E}_1] \qquad \frac{P \mid \Gamma \vdash M : \tau \,\&\, \tau'}{P \mid \Gamma \vdash \mathtt{snd}\ M : \tau'}\,[\&\ \mathrm{E}_1]$$

$$\frac{P \mid \Gamma \vdash M : \tau \qquad P \mid \Delta \vdash N : \tau'}{P \mid \Gamma \circledast \Delta \vdash \langle M, N \rangle : \tau \otimes \tau'}\,[\otimes I]$$

$$\frac{P \mid \Gamma \vdash M : \tau \otimes \tau' \qquad P \mid \{x^{\{\vec{z} \mid \vec{z} \subseteq \mathtt{Vars}(\Gamma')\}} : \tau\} \circledast \{y^{\{\vec{z'} \mid \vec{z'} \subseteq \mathtt{Vars}(\Gamma')\}} : \tau'\} \sqcup \Gamma'_{x,y} \vdash N : \upsilon}{P \mid \Gamma \sqcup \Gamma' \vdash (\mathtt{let}\ \langle x, y \rangle = M\ \mathtt{in}\ N) : \upsilon}\,[\otimes E]$$

A.2 Derivable Typing rules for Sum Types

Sum types can hold only one of the enclosing types at a given point of time.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \mid g^{cf} : (B \twoheadrightarrow E) \vdash g : (B \twoheadrightarrow E)}\,[\text{ID}] \quad \dfrac{\dfrac{}{\varnothing \mid f^{cg} : (A \twoheadrightarrow E) \vdash f : (A \twoheadrightarrow E)}\,[\text{ID}] \quad \dfrac{}{\varnothing \mid c^{fg} : ((A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E) \vdash c : ((A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E)}\,[\text{ID}]}{\varnothing \mid c^{fg} : ((A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E) \oplus f^{cg} : (A \twoheadrightarrow E) \vdash cf : (B \twoheadrightarrow E) \twoheadrightarrow E}\,[\twoheadrightarrow\text{E}]}{\varnothing \mid c^{fg} : ((A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E) \oplus f^{cg} : (A \twoheadrightarrow E) \oplus g^{cf} : (B \twoheadrightarrow E) \vdash cfg : E}\,[\twoheadrightarrow\text{E}]}{\varnothing \mid c^{f} : ((A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E) \oplus f^{c} : (A \twoheadrightarrow E) \vdash \lambda^{\twoheadrightarrow} g.cfg : (B \twoheadrightarrow E) \twoheadrightarrow E}\,[\twoheadrightarrow\text{I}]}{\varnothing \mid c^{\varnothing} : ((A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E) \vdash \lambda^{\twoheadrightarrow} f.\lambda^{\twoheadrightarrow} g.cfg : (A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E}\,[\twoheadrightarrow\text{I}]}{\varnothing \mid I \vdash \lambda^{\multimap} c.\lambda^{\twoheadrightarrow} f.\lambda^{\twoheadrightarrow} g.cfg : ((A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E) \multimap (A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E}\,[\multimap\text{I}]$$

We now define sum type to be

$$A \oplus B = (A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E$$

We now define left and right *injections* or constructors for the sum type.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \mid x^{fg} : A \vdash x : A}\,[\text{ID}] \quad \dfrac{}{\varnothing \mid f^{gx} : (A \twoheadrightarrow E) \vdash f : (A \twoheadrightarrow E)}\,[\text{ID}]}{\varnothing \mid x^{fg} : A \oplus f^{xg} : (A \twoheadrightarrow E) \vdash fx : E}\,[\twoheadrightarrow\text{E}]}{\varnothing \mid x^{fg} : A \oplus f^{xg} : (A \twoheadrightarrow E) \oplus g^{fx} : (B \twoheadrightarrow E) \vdash fx : E}\,[\text{WKN-UN}]}{\varnothing \mid x^{f} : A \oplus f^{x} : (A \twoheadrightarrow E) \vdash \lambda^{\twoheadrightarrow} g.fx : (B \twoheadrightarrow E) \twoheadrightarrow E}\,[\twoheadrightarrow\text{I}]}{\varnothing \mid x^{\varnothing} : A \vdash \lambda^{\twoheadrightarrow} f.\lambda^{\twoheadrightarrow} g.fx : (A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E}\,[\twoheadrightarrow\text{I}]}{\varnothing \mid I \vdash \lambda^{\multimap} x.\lambda^{\twoheadrightarrow} f.\lambda^{\twoheadrightarrow} g.fx : A \multimap (A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E}\,[\multimap\text{I}]$$

Left injection defined below as:

$$\texttt{inl} : A \multimap A \oplus B$$

$$\texttt{inl} = \lambda^{\multimap} x.\lambda^{\twoheadrightarrow} f.\lambda^{\twoheadrightarrow} g.fx$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \mid y^{fg} : B \vdash y : B}\,[\mathrm{ID}] \quad \dfrac{}{\varnothing \mid g^{yf} : (B \multimap E) \vdash g : (B \multimap E)}\,[\mathrm{ID}]}{\varnothing \mid y^{fg} : B \oplus g^{yf} : (B \twoheadrightarrow E) \vdash gy : E}\,[{\twoheadrightarrow}\mathrm{I}]}{\varnothing \mid y^{fg} : B \oplus f^{yg} : (A \twoheadrightarrow E) \oplus g^{yf} : (B \twoheadrightarrow E) \vdash gy : E}\,[\mathrm{WKN\text{-}UN]}}{\varnothing \mid y^{f} : B \oplus f^{y} : (A \twoheadrightarrow E) \vdash \lambda^{\twoheadrightarrow} g.gy : (B \twoheadrightarrow E) \twoheadrightarrow E}\,[{\twoheadrightarrow}\mathrm{I}]}{\varnothing \mid y^{\varnothing} : B \vdash \lambda^{\twoheadrightarrow} f.\lambda^{\twoheadrightarrow} g.gy : (A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E}\,[{\twoheadrightarrow}\mathrm{I}]}{\varnothing \mid I \vdash \lambda^{\multimap} y.\lambda^{\twoheadrightarrow} f.\lambda^{\twoheadrightarrow} g.gy : B \multimap (A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E}\,[{\multimap}\mathrm{I}]$$

Right injection defined below as:

$$\mathtt{inr} : B \multimap A \oplus B$$

$$\mathtt{inr} = \lambda^{\multimap} y.\lambda^{\twoheadrightarrow} f.\lambda^{\twoheadrightarrow} g.gy$$

We can now derive sum types in our language using the auxilary definitions given above and provide a new syntax for deconstructing the sum type by matching on its structure by a case statement.

$$\texttt{case}\ c\ \texttt{of}\ \{f;g\} = \lambda^{\multimap} c.\lambda^{\twoheadrightarrow} f.\lambda^{\twoheadrightarrow} g.cfg$$

$$\frac{\varnothing \mid \texttt{inl} : A \multimap A \oplus B \vdash x : A}{\varnothing \mid I \vdash \texttt{inl}\ x : A \oplus B}\ [\oplus \mathrm{I}_1] \qquad \frac{\varnothing \mid \texttt{inr} : B \multimap A \oplus B \vdash y : B}{\varnothing \mid I \vdash \texttt{inr}\ y : A \oplus B}\ [\oplus \mathrm{I}_2]$$

$$\frac{\varnothing \mid \Gamma \vdash M : A \oplus B \qquad \varnothing \mid \Gamma \oplus x : A \vdash N_1 : E \qquad \varnothing \mid \Gamma \oplus y : B \vdash N_2 : E}{\varnothing \mid \Gamma \vdash \texttt{case}\ M\ \texttt{of}\ \{\texttt{inl}\ x \mapsto N_1; \texttt{inr}\ y \mapsto N_2\} : E}\ [\oplus \mathrm{E}]$$

Appendix B

Proofs

Theorem B.1 (Soundness of $\vdash^s$)**.** *If* $P \mid \Gamma \vdash^s M : \tau$ *then* $P \mid \Gamma \vdash M : \tau$

Proof. Proof by induction on the derivation of $P \mid \Gamma \vdash^s M : \tau$.

- Case [VAR^s]. By induction hypothesis we have a derivation of $P \mid x : \sigma \vdash x : \sigma$ by [VAR] rule. We proceed by repeated application of [$\forall$E] as $(Q \Rightarrow \tau) \sqsubseteq \sigma$, and we construct a derivation of $P \mid x : \sigma \vdash x : Q \Rightarrow \tau$. Then as $P \Rightarrow Q$ we can repeatedly apply [$\Rightarrow$E] to construct a derivation of $P \mid x : \sigma \vdash x : \tau$. Finally, depending on whether the bindings are in sharing with or separate from x we repeatedly apply [WKN-SH] or [WKN-UN] respectively for all the bindings in Γ to construct a derivation of $P \mid \Gamma \sqcup \{x : \sigma\} \vdash x : \tau$.

- Case [$\twoheadrightarrow$I^s]. By induction hypothesis we have a derivation of $P \mid \Gamma \oplus x : \tau \vdash M : \tau'$. We apply [$\twoheadrightarrow$I] and reuse the derivations for $\texttt{ShFun}\ \phi$ and $\Gamma \geq \phi$ to construct a derivation of $P \mid \Gamma \vdash \lambda^{\twoheadrightarrow} x.M : \phi\,\tau\,\tau'$.

- Case [$\multimap$I^s]. By induction hypothesis we have a derivation of $P \mid \Gamma \circledast x : \tau \vdash M : \tau'$. Similar to previous case, we apply [$\multimap$I] and reuse the derivations for $\texttt{SeFun}\ \phi$ and $\Gamma \geq \phi$ to construct a derivation of $P \mid \Gamma \vdash \lambda^{\multimap} x.M : \phi\,\tau\,\tau'$.

- Case [App^s]. By induction hypothesis we have derivations of $P \mid \Gamma \circledast \Delta \vdash M : \phi\,\upsilon\,\tau$ and $P \mid \Gamma' \circledast \Delta \vdash N : \upsilon$ we check for $\texttt{Used}(\Gamma) = \texttt{Used}(\Gamma')$ and if it is true and apply [$\twoheadrightarrow$E] or check for $\texttt{Used}(\Gamma)\#\texttt{Shared}(\Gamma') \wedge \texttt{Shared}(\Gamma')\#\texttt{Used}(\Gamma)$ and if it is true we apply [$\multimap$E] to reuse derivations of $\texttt{ShFun}\ \phi$ or $\texttt{SeFun}\ \phi$ respectively to construct the derivation of $P \mid (\Gamma \oplus \Gamma') \circledast \Delta \vdash MN : \tau$ or $P \mid (\Gamma \circledast \Gamma') \circledast \Delta \vdash MN : \tau$.

- Case [Let^s]. By induction hypothesis have a derivation of $P \mid \Gamma \circledast \Delta \vdash M : \tau$ and $P \mid \Gamma' \sqcup x : \tau \circledast \Delta \vdash N : \tau$ Applying [$\forall$I] and [$\Rightarrow$I] on the first hypothesis we derive $\varnothing \mid \Gamma \circledast \Delta \vdash M : \sigma$. Now by applying the [LET] rule and reusing $P \vdash \Delta$ un we construct the derivation of $P \mid \Gamma \sqcup \Gamma' \circledast \Delta \vdash$ let $x = M$ in $N : \tau$. ∎

Theorem B.2 (Completeness of $\vdash^s$)**.** *If $P \mid \Gamma \vdash M : \sigma$ then $\exists Q, \tau$ such that $Q \mid \Gamma \vdash^s M : \tau$ and $(P \mid \sigma) \sqsubseteq \mathit{Gen}(\Gamma, Q \Rightarrow \tau)$*

We first go over few intermediate results and definitions that will help us prove the final result.

Lemma B.1. *If $P \mid \Gamma \vdash M : \tau$, $\forall_{y \in dom(\Delta)} y$ is not free in M and $P \vdash \Delta$ un then $P \mid \Gamma \circledast \Delta \vdash M : \tau$.*

Proof. By induction on derivation of $P \mid \Gamma \vdash M : \tau$,

- Case [VAR]. M is an expression x and $y \notin \mathtt{fvs}(x)$ also, $P \vdash \sigma$ un for $(y : \sigma)$, thus using [VAR^s] we can obtain the required derivation.

- Case others. [LET], [$\twoheadrightarrow$I], [$\twoheadrightarrow$E], [$\multimap$I], [$\multimap$E] are all straightforward. ∎

Lemma B.2. *If $P \mid \Gamma \vdash M : \tau$, $\forall_{y \in dom(\Delta)} y$ is not free in M then $P \mid \Gamma \oplus \Delta \vdash M : \tau$.*

Proof. By induction on derivation of $P \mid \Gamma \vdash M : \tau$,

- Case [VAR]. M is an expression x and $y \notin \mathtt{fvs}(x)$ also, $\Gamma \oplus \{y : \sigma\}$ thus, by using [WKN-SH] and [VAR^s] the required derivation is obtained.

- Case others. [LET], [$\twoheadrightarrow$I], [$\twoheadrightarrow$E], [$\multimap$I], [$\multimap$E] are all straightforward. ∎

Lemma B.3. *If $P \mid \Gamma \vdash^s M : \tau$ and $\sigma = \mathit{Gen}(\Gamma, P \Rightarrow \tau)$, then for any $P' \Rightarrow \tau' \sqsubseteq \sigma$, $P' \mid \Gamma \vdash^s M : \tau'$*

Proof. Let $\sigma = \forall \vec{t}.Q \Rightarrow \upsilon$. By definition, there are some $\vec{u}$, such that $\tau' = [\vec{u}/\vec{t}]\upsilon$ and $P' \Rightarrow [\vec{u}/\vec{t}]Q$. Hence, we have $P' \mid [\vec{u}/\vec{t}]\Gamma \vdash M : \tau'$, and as $\vec{t}$ is free in Γ, we get the desired result, $P' \mid \Gamma \vdash M : \tau'$ ∎

Definition 6 ($\Gamma \sqsubseteq \Gamma'$)**.** If $\mathtt{dom}(\Gamma) = \mathtt{dom}(\Gamma')$ and $\forall_{x \in \mathtt{dom}(\Gamma)} \Gamma(x) \sqsubseteq \Gamma'(x)$ then we say $\Gamma \sqsubseteq \Gamma'$.

Lemma B.4. *If $P \mid \Gamma \vdash^s M : \tau$, and $\Gamma \sqsubseteq \Gamma'$ then $P \mid \Gamma' \vdash^s M : \tau$.*

Proof. By induction on derivation of $P \mid \Gamma \vdash^s M : \tau$.

- Case [Var^s]. If $(P \Rightarrow \tau) \sqsubseteq \sigma$ and $\sigma \sqsubseteq \sigma'$ then $(P \Rightarrow \tau) \sqsubseteq \sigma'$. Now by applying [$\forall$E] we get the required derivation.

- Case others. All other cases, [Let^s],[$\twoheadrightarrow$I^s],[$\multimap$I^s], and [App^s] are trivial. Essentially use [$\forall$E] rule to get the correct instance of the type to get the required derivation. ■

Proof of Theorem B.2. By induction on derivation of $P \mid \Gamma \vdash M : \sigma$.

- Case [ID]. We have $(x : \sigma) \in \Gamma$, where $\sigma = (\forall \vec{t}.Q \Rightarrow \tau)$. Pick fresh type variables $\vec{u}$ so that $([\vec{u}/\vec{t}]Q \Rightarrow [\vec{u}/\vec{t}]\tau) \sqsubseteq \sigma$, thus $P, [\vec{u}/\vec{t}]Q \mid \Gamma \vdash x : [\vec{u}/\vec{t}]\tau$ using [VAR^s].

$$\begin{aligned}\sigma' &= \texttt{Gen}(\Gamma, (P, [\vec{u}/\vec{t}]Q \Rightarrow [\vec{u}/\vec{t}]\tau)) \\ &= \forall u.(P, [\vec{u}/\vec{t}]Q) \Rightarrow [\vec{u}/\vec{t}]\tau\end{aligned}$$

  as $\vec{u}$ are fresh i.e. $\vec{u} \notin \texttt{dom}(\Gamma)$ and $(P \mid \sigma) \sqsubseteq \sigma'$.

- Case [CTR-UN]. The newly duplicated type assignments are in Δ which is separate from Γ and $P \vdash \Delta$ un. The induction hypothesis gives the required derivation of $P \mid \Gamma \circledast \Delta \vdash M : \tau$ such that $(P \mid \sigma) \sqsubseteq (\varnothing \mid \texttt{Gen}(\Gamma \circledast \Delta, Q \Rightarrow \tau))$

- Case [WKN-UN]. This follows directly from Lemma B.1 and the induction hypothesis.

- Case [CTR-SH]. The newly duplicated type assignments are in Δ which is in sharing with Γ. The induction hypothesis gives the required derivation of $P \mid \Gamma \oplus \Delta \vdash M : \sigma$ such that $(P \mid \sigma) \sqsubseteq (\varnothing \mid \texttt{Gen}(\Gamma \oplus \Delta, Q \Rightarrow \tau))$

- Case [WKN-SH]. This follows directly from Lemma B.2 and the induction hypothesis.

- Case [$\multimap$I]. By induction hypothesis and Lemma B.3, we have a derivation of $Q \mid \Gamma \vdash^s M : \tau'$ also, $P \mid \Gamma \geq \phi$ thus by [$\multimap$I^s] we have the required derivation.

- Case [⇀E]. By induction hypothesis and Lemma B.3, we have a derivation of $Q \mid \Gamma \vdash^s M : \phi\,\tau\,\tau'$ and $Q \mid \Gamma' \vdash^s N : \tau$. We can partition the environment into separating contexts $\Gamma \circledast \Gamma'$ and apply [App^s] to obtain the required derivation.

- Case [⇁I]. By induction hypothesis and Lemma B.3, we have a derivation of $Q \mid \Gamma \vdash^s M : \upsilon$ also, $Q \mid \Gamma \geq \phi$ thus by using [⇁I^s] we obtain the required derivation.

- Case [⇁E]. By induction hypothesis and Lemma B.3, we have a derivation of $Q \mid \Gamma \vdash^s M : \phi\,\tau\,\tau'$ and $Q \mid \Gamma' \vdash^s N : \tau$. We can partition the environment into sharing contexts $\Gamma \oplus \Gamma'$ and apply [App^s] to get the required derivation.

- Case [⇒E]. By induction hypothesis we have $Q \mid \Gamma \vdash^s M : \tau$ such that, letting $\sigma = \mathtt{Gen}(\Gamma, Q \Rightarrow \tau)$, $(P, \pi \mid \rho) \sqsubseteq \sigma$. As $(P \mid \pi \Rightarrow \rho) \sqsubseteq (P, \pi \mid \rho)$, we also have $(P \mid \pi \Rightarrow \rho) \sqsubseteq \sigma$

- Case [⇒I]. By induction hypothesis we have $Q \mid \Gamma \vdash^s M : \tau$ such that, letting $\sigma = \mathtt{Gen}(\Gamma, Q \Rightarrow \tau)$, $(P \mid \pi \Rightarrow \rho) \sqsubseteq \sigma$. Since $P \Rightarrow \pi$, we have $(P \mid \rho) \sqsubseteq (P \mid \pi \Rightarrow \rho)$, and hence, $(P \mid \rho) \sqsubseteq \sigma$.

- Case [∀I]. By the induction hypothesis, we have $Q \mid \Gamma \vdash^s M : \tau$ such that, letting $\sigma = \mathtt{Gen}(\Gamma, Q \Rightarrow \tau)$, $(P \mid \pi \Rightarrow \rho) \sqsubseteq \sigma$

- Case [∀E]. By the induction hypothesis we have $Q \mid \Gamma \vdash^s M : \tau$ such that, letting $\sigma = \mathtt{Gen}(\Gamma, Q \Rightarrow \tau)$, $(P \mid \pi \Rightarrow \rho) \sqsubseteq \sigma$, As $(P \mid [\tau/t]\sigma) \sqsubseteq (P \mid \sigma)$, $(P \mid [\tau/t]\sigma) \sqsubseteq \sigma$ (becuase $\sigma = (\varnothing \mid \sigma)$.

- Case [Let]. By induction hypothesis, we have

$$Q \mid \Gamma \circledast \Delta \vdash^s M : \upsilon$$

and

$$Q' \mid \Gamma' \circledast \Delta \sqcup \{x : \sigma\} \vdash^s N : \tau$$

such that, letting $\sigma' = \mathtt{Gen}(\Gamma, Q \Rightarrow \upsilon)$, $(P \mid \forall t.\sigma) \sqsubseteq \sigma'$. Thus we conclude that $\Gamma \sqcup \{x : \forall t.\sigma\} \sqsubseteq \Gamma \sqcup \{x : \sigma'\}$. Now by applying Lemma B.4, the induction hypothesis, and Lemma B.3, we

have a derivation of $Q' \mid \Gamma \sqcup \{x : \sigma'\} \vdash N : \tau$. Finally applying [LET^s] we get the required derivation. ■

Theorem B.3 (Soundness of $\mathcal{M}$.)**.** *If* $\mathcal{M}(S, \Psi, \Gamma \vdash M : \tau) = P, S', \Sigma, \Psi'$ *then* $S'P \mid S'\Gamma \vdash M : S'\tau$

We go over a few intermediate results to prove the soundness result.

Lemma B.5. *If* $P \mid \Delta \vdash^s M : \tau$ *and* $Q \Rightarrow P$*, then* $Q \mid \Delta \vdash^s M : \tau$

Lemma B.6. *If* $P \mid \Gamma \vdash^s M : \tau$*,* $\forall_{y \in dom(\Delta)} y$ *is not free in* M *and* $P \vdash^s \Delta$ *un then* $P \mid \Gamma \circledast \Delta \vdash M : \tau$*.*

Proof. We see that the syntax directed system is complete with respect to the original type system hence, due to Lemma B.1 and Theorem B.2, we can prove the required result. ■

Lemma B.7. *If* $P \mid \Gamma \vdash^s M : \tau$*,* $\forall_{y \in dom(\Delta)} y$ *is not free in* M *then* $P \mid \Gamma \oplus \Delta \vdash^s M : \tau$*.*

Proof. Proof is similar to Lemma B.6. We see that the syntax directed system is complete with respect to the original type system hence, due to Lemma B.2 and Theorem B.2, we can prove the required result. ■

Lemma B.8. *If* $P \mid \Gamma \vdash^s M : \tau$ *then* $SP \mid S\Gamma \vdash^s S\tau$*.*

Proof. Proof is by a simple induction on derivation of $P \mid \Gamma \vdash^s M : \tau$ ■

Proof of Theorem B.3. Proof by induction on structure of M

- Case x. We have

$$(x^{\vec{y}} : \forall \vec{t}.P \Rightarrow \tau) \in \Gamma$$

where $\vec{y} \subseteq \mathtt{dom}\Gamma$ and $\Sigma = \{x\}$. We see that

$$([\vec{u}/\vec{t}]P \Rightarrow [\vec{u}/\vec{t}]\tau) \sqsubseteq (\forall \vec{t}.P \Rightarrow \tau)$$

So, we can apply [VAR^s] to construct a derivation of $[\vec{u}/\vec{t}]P \mid \{x : \forall \vec{t}.P \Rightarrow \tau\} \vdash^s x : [\vec{u}/\vec{t}]\tau$

- Case $\lambda^{\twoheadrightarrow} x.M$. We have

$$P; S'; \Sigma; \Psi' = \mathcal{M}(\mathcal{U}(\tau, u_1 u_2 u_3) \circ S, \Psi, \Gamma, x : u_2 \vdash M : u_3)$$

and by induction hypothesis

$$S'P \mid S'((\Gamma \sqcup \{x^{\mathtt{dom}(\Gamma)} : u_2\})|_\Sigma) \vdash^s M : S'u_3$$

Let $Q = \{\mathtt{ShFun}\ u_1\} \cup \mathtt{Leq}(u_1, \Gamma|_\Sigma) \cup \mathtt{Weaken}(x, u_2, \Sigma) \cup P$ and let $\Sigma' = \Sigma \backslash \{x\}$. By Lemma B.5 and Lemma B.6 we can consruct a derivation of

$$S'Q \mid S'((\Gamma \sqcup \{x : u_2\})|_{\Sigma'}) \vdash^s M : S'u_3$$

Now, using $[\twoheadrightarrow \mathrm{I}^s]$ we get the required derivation

- Case $\lambda^{\twoheadrightarrow} x.M$. We have

$$P; S'; \Sigma; \Psi' = \mathcal{M}(\mathcal{U}(\tau, u_1 u_2 u_3) \circ S, \Psi, \Gamma, x : u_2 \vdash M : u_3)$$

and by induction hypothesis

$$S'P \mid S'((\Gamma \sqcup \{x^{\varnothing} : u_2\})|_\Sigma) \vdash^s M : S'u_3$$

Let $Q = \{\mathtt{SeFun}\ u_1\} \cup \mathtt{Leq}(u_1, \Gamma|_\Sigma) \cup \mathtt{Weaken}(x, u_2, \Sigma) \cup P$ and let $\Sigma' = \Sigma \backslash \{x\}$. By Lemma B.5 and Lemma B.6 we can consruct a derivation of

$$S'Q \mid S'((\Gamma \sqcup \{x : u_2\})|_{\Sigma'}) \vdash^s M : S'u_3$$

Now, using $[\twoheadrightarrow \mathrm{I}^s]$ we get the required derivation

- Case MN. We have that

$$P;R;\Sigma;\Psi' = \mathcal{M}(S,\Psi,\Gamma \vdash M : u_1 u_2 \tau)$$

$$P';R';\Sigma';\Psi'' = \mathcal{M}(S \circ R,\Psi',S\Gamma \vdash N : u_2)$$

We have two sub-cases here depending on whether the application is shared or separating. If we can prove $\mathcal{C}(S\Gamma,\Sigma,\Psi)\#\mathcal{C}(S \circ R\Gamma,\Sigma',\Psi')$ then we let $Q = P \cup P' \cup \texttt{SeFun}\ u_1$ or if we can prove $\mathcal{C}(S\Gamma,\Sigma,\Psi) = \mathcal{C}((S \circ R)\Gamma,\Sigma',\Psi')$ then we let $Q = P \cup P' \cup \texttt{ShFun}\ u_1$ (else the inference algorithm fails). Let $\Delta = \texttt{dom}(\Gamma) \backslash \mathcal{C}(\Gamma,\Sigma \cup \Sigma',\Psi \circ \Psi')$ Now by Lemma B.5 and Lemma B.8 and the induction hypothesis we get

$$R'Q' \mid R'\Gamma \circledast R'\Delta \vdash^s M : R'(u_1 u_2 \tau)$$

$$R'Q' \mid R'\Gamma \circledast R'\Delta \vdash^s N : u_2$$

where $Q' = P \cup P' \cup Q \cup \{\texttt{Un}\ \Delta\}$. We can now build the desired derivation using [APP^s].

- Case $\texttt{let}\ x = M\ \texttt{in}\ N$. We have

$$P;R;\Sigma;\Psi' = \mathcal{M}(S,\Psi,\Gamma \vdash M : u_1)$$

$$P';R';\Sigma';\Psi'' = \mathcal{M}(R,\Psi',\Gamma,x : \sigma \vdash N : \tau)$$

where $\sigma = \texttt{GenI}(R\Gamma;R(P \Rightarrow u_1))$

■

Theorem B.4 (Incompleteness of $\mathcal{M}$.)**.**

Proof. The algorithm $\mathcal{M}$ is incomplete as there would be cases where a term can be typed in two ways. $\mathcal{M}$ would compute only one of them and leave out the other one. Consider the type for $\lambda^{-*} f.\lambda^{-*} x.fxx$. This term can be typed in two ways in QuB type system.

- $\mathtt{Un}\ A \mid \varnothing \vdash \lambda^{-\!\!*} f.\lambda^{-\!\!*} x.fxx : (A \multimap A \multimap B) \multimap A \multimap B$
- $\varnothing \mid \varnothing \vdash\vdash \lambda^{-\!\!*} f.\lambda^{-\!\!*} x.fxx : (A \multimap A \twoheadrightarrow B) \multimap A \twoheadrightarrow B.$

Hence Algorithm $\mathcal{M}$ is incomplete. ■