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.

Dr. J. Garrett Morris, Chairperson

Committee members

Dr. Perry Alexander

Dr. Andy Gill

Dr. Prasad Kulkarni

Date defended:

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:

Abstract

Modern programming languages treat resources as normal values. The static semantics of resources in such languages does not match their runtime semantics. 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 (pronounced: cube), a Curry-Howard interpretation of logic of bunched implications (***BI***) (O'Hearn and Pym, 1999). We distinguish two kinds of values—restricted and unrestricted—and two kinds of function implications—sharing and separating. The restricted values model resources while the unrestricted values model program objects that do not contain any resources. Sharing functions denote that functions share resources with its arguments, while separating functions denote that functions do not share resources with its arguments. We show how the use of monads with sharing and separating functions helps in modeling patterns, such as exception handling, that are difficult to express in linear languages.

Acknowledgements

I am grateful to my parents and my brother. Their persistent support and encouragement has been a crucial part of my education and my life otherwise.

I am thankful to my advisor, Dr. Garrett Morris. Without his immense patience, wholesome guidance, and critical insights, it would have been impossible to make this journey.

I am appreciative of my friends, who helped me stay sane by showing me the world outside work.

And of course, those countless cups of coffee that subsisted my morale in the writing task.

Contents

List of Figures

Chapter 1

Introduction

Resources are treated as normal values in Hindley-Milner based type systems. Statically typed modern programming languages such as Haskell and ML do little to track resource usage at compile time. Runtime errors, such as those caused due to closing a file handle more than once, or not closing an open file handle at all, are difficult to track and debug in evolving production code. 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 achieved mainstream success due to practical limitations. In this thesis we develop a type system based on the logic of bunched implications (***BI***) and implement a type inference algorithm for a language that is an extenstion of lambda-calculus. We later extend the system to allow users to define their own datatypes, expressing sharing and separation between fields.

The contributions of this thesis are:

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

- QuB incorporates distributive laws admissible in ***BI*** and internalizes the transformations on the context that are explicit in ***BI***.

- Design a sound and complete syntax directed type system and implement a sound type inference algorithm, based on modified Algorithm $\mathcal{M}$, with formal proofs.

- Examples that illustrate how QuB's type system enhances expressivity in comparison to standard Hindley-Milner type system by tracking resources at compile time and to detect anomalies.

This thesis is organized in the following manner: Chapter 2 details some necessary background work that is related to our work. Chapter 3 illustrates how programming in QuB is different than programming in other languages like Haskell by giving concrete examples. Chapter 4 describes the core language of type and terms and and Chapter 5 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. Finally, Chapter 7 conclues the thesis by giving a direction to future work.

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 (i.e restricted version of System F (Girard et al., 1989)) forms the basis of many modern functional programming languages such as Haskell and ML. Fig. 2.1 shows the type language containing type variables, primitive types (such as integers, floats), the type constructor (→)—which constructs function types—and type scheme (σ). The expression language contains variables, function abstraction and applications, and the polymorphic `let` construct. Type inference algorithm $\mathcal{W}$ (Damas and Milner, 1982) and its variant type checking algorithm $\mathcal{M}$ (Lee and Yi, 1998) is decidable in the sense, the algorithm always terminates with either a success or failure. Both the algorithms compute a most general, or principal typing scheme for an expression.

$t, u \in$ Type Variables

$$\begin{aligned}
\text{Types} \quad & \tau, \upsilon ::= t \mid \iota \mid \tau \to \tau \\
\text{Typing Scheme} \quad & \sigma ::= \tau \mid \forall t.\tau \\
\text{Typing Context} \quad & \Gamma ::= \varepsilon \mid \Gamma, x : \sigma \\
\text{Expressions} \quad & M, N ::= x \mid \lambda x.M \mid MN \mid \texttt{let } x = M \texttt{ in } N
\end{aligned}$$

Figure 2.1: Hindley-Milner Type and Expression Language

Robinson's (1965) unification algorithm plays a key role in computation of well-formed principal types. Its purely syntactic approach in creating substitutions to unify types keeps the complete process elegant. Algorithm $\mathcal{M}$ can either be used to infer the types of all well-typed expressions or it 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 **HM** type system are shown in Fig. 2.2. Γ is the collection of assumptions, or context, in which the expression M is typed. It can be thought of as a collection–list or set–of an ordered pair of identifier and its type scheme. Γ_x denotes the type assignment excluding the type variable x. The [VAR] rule is tautology; a simple lookup of variable x in context Γ for the type scheme. [→I] and [→E] type lambda terms and application respectively. Rules for parametric polymorphism are implicit in expression language. [∀I] rule generalizes the type scheme by adding universal quantifiers and [∀E] generates an instance of the type scheme by substituting free type variables. We write $\sigma'\sqsubseteq\sigma$ to mean σ' is an instance of σ. While generalizing types, the rule [∀I], has a side condition that ensures the new type variable introduced should not be free in the typing context. The $\texttt{fvs}(\Gamma)$ denotes free type variables in Γ. The [LET] rules allows implicit parametric polymorphism. For example, the expression, `g = λf.(f True, f 1)` is ill-typed. The type unification will fail for `f` as the expression `f True` asserts $\texttt{f:Bool}\to\texttt{u}_1$, while the expression `f 1` asserts $\texttt{f:Int}\to\texttt{u}_2$. However, there indeed exists a polymorphic type $\forall\texttt{u}_1.\texttt{u}_1\to\texttt{u}_1$ that types `f`. The `let` construct makes this possible. 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 $\texttt{f}:\forall\texttt{u}_1.\ \texttt{u}_1\to\texttt{u}_1$, using [→I] and [∀I] rules and `g` is assigned a type `(Bool, Int)` as intended.

Using the Curry-Howard correspondence, the **HM** type system corresponds to the second order

$$
\begin{aligned}
A, B, C &::= x \mid A \supset B \mid \forall x.B \mid A \wedge B \mid A \vee B \\
\Gamma, \Delta &::= \varepsilon \mid \Gamma, A
\end{aligned}
$$

Figure 2.3: Grammar for Intuitionistic Logic

implication fragment of intuitionistic propositional logic. The grammar of intuitionistic logic is shown in Fig. 2.3 where $\Gamma \supset \Delta$ denotes implication, $A \wedge B$ denotes conjunction and $A \vee B$ denotes disjunction.

$$
\frac{}{A \vdash A}\,[\text{Ax}] \qquad \frac{\Gamma \vdash B}{\Gamma, A \vdash B}\,[\text{WKN}] \qquad \frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}\,[\text{CTR}]
$$

$$
\frac{\Gamma \vdash B \qquad x \notin \Gamma}{\forall x.B}\,[\forall\text{I}] \qquad \frac{\Gamma \vdash \forall x.B \qquad \Gamma \vdash A}{B[x/A]}\,[\forall\text{E}]
$$

$$
\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B}\,[\supset\text{I}] \qquad \frac{\Gamma \vdash A \supset B \qquad \Gamma \vdash A}{\Gamma \vdash B}\,[\supset\text{E}]
$$

$$
\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,[\wedge\text{I}] \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\,[\wedge\text{E}_1] \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\,[\wedge\text{E}_2]
$$

$$
\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\,[\vee\text{I}_1] \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\,[\vee\text{I}_2] \qquad \frac{\Gamma \vdash A \supset C \qquad \Gamma \vdash B \supset C \qquad \Gamma \vdash A \vee B}{\Gamma \vdash C}\,[\vee\text{E}]
$$

Figure 2.4: Logic rules: Intuitionistic Propositional Logic System

The rules of the logic system are shown in Fig. 2.4 in Gentzen style natural deduction where Γ, Δ are contexts that keep track of assumptions. The [Ax] rule corresponds to [VAR] rule while [$\rightarrow$I] and [$\rightarrow$E] correspond to [$\supset$I] and [$\supset$E] respectively. The weakening [WKN] and contraction [CTR] rules are implicit in the term structue of **HM** type-system. The weakening rule states that we can add unrelated assumptions to proof derivations without affecting them, while contraction states that we can discard duplicate assumptions from our proof derivations and the proof will still hold. The [$\forall$I] introduces the universal quantifier for a proposition and [$\forall$E] instantiates the quantifier by replacing the variable with given proposition. The rules [$\forall$I] and [$\forall$E] correspond to parametric polymorphism introduction [$\forall$I] and elimination [$\forall$E] rules in Fig. 2.2. [$\wedge$I], [$\wedge$E$_2$] and [$\wedge$E$_2$] introduces and eleminates conjunction. Conjunction is equivalent to pairs or product types.

[$\vee$I$_2$], [$\vee$I$_2$] and [$\vee$E] introduces and eleminates disjunction respectively. Disjunction is equivalent to sum types. While, **HM** type system does not treat disjunction and conjunction types as first class citizens, they can be Church encoded using lambda expressions(Pierce, 2002).

$$\boxed{\mathcal{M}(\Gamma \vdash M : \tau) = S}$$

$$\begin{aligned} \mathcal{M}(\Gamma \vdash x : \tau) &= \mathcal{U}(\tau, [\vec{u}/\vec{t}]\upsilon) \\ \text{where} \quad & \forall \vec{t}.\upsilon = \Gamma(x) \end{aligned}$$

$$\begin{aligned} \mathcal{M}(\Gamma \vdash \lambda x.M : \tau) &= S \circ S' \\ \text{where} \quad S &= \mathcal{U}(\tau, u_1 \to u_2) \\ S' &= \mathcal{M}(S\Gamma, x : Su_1 \vdash M : Su_2) \end{aligned}$$

$$\begin{aligned} \mathcal{M}(\Gamma \vdash MN : \tau) &= S \circ S' \\ \text{where} \quad S &= \mathcal{M}(\Gamma \vdash M : u \to \tau) \\ S' &= \mathcal{M}(S\Gamma \vdash N : Su) \end{aligned}$$

$$\begin{aligned} \mathcal{M}(\Gamma \vdash (\texttt{let}\ x = M\ \texttt{in}\ N) : \tau) &= S \circ S' \\ \text{where} \quad S &= \mathcal{M}(\Gamma \vdash M : u) \\ \sigma &= \texttt{Gen}(S\Gamma, Su) \\ S' &= \mathcal{M}(S\Gamma, x : \sigma \vdash N : \tau) \end{aligned}$$

$$\boxed{\text{Auxiliary Definitions}}$$

$$\begin{aligned} \texttt{Gen}(\Gamma, \tau) &= \forall \vec{t}.\tau \\ \text{where} \quad \vec{t} &= \texttt{fvs}(\tau) \backslash \texttt{fvs}(\Gamma) \end{aligned}$$

$$\begin{aligned} \texttt{fvs}(t) &= \{t\} \\ \texttt{fvs}(\forall \vec{t}.\tau) &= \texttt{fvs}(\tau) \backslash \vec{t} \\ \texttt{fvs}(\Gamma) &= \bigcup_{\forall (x:\sigma) \in \Gamma} \texttt{fvs}(\sigma) \end{aligned}$$

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

The type inference algorithm $\mathcal{M}$ is given in figure Fig. 2.5. $\mathcal{U}$ is Robinson's unification algorithm that computes the most general unifier required to unify two types and $\texttt{Gen}(\Gamma, \tau)$ generalizes a type to a principle type scheme. All the type variables denoted by u are fresh, and hence do not shadow the existing type variables in the context. Substitutions, denoted by S, can be applied on types or type schemes. They are combined using $\circ$ operator. When substitutions are applied to a context Γ, they are applied to each type scheme contained by Γ. $\vec{t}$ is shorthand for a set of type variables $\{t_1, t_2, \ldots, t_n\}$ and $\vec{t} \backslash \vec{u}$ denotes the set difference.

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 addresses resource and resource control problem. The core idea is that propositions cannot be freely duplicated or discarded in contrast to intuitionistic logic. In terms of formal logic system, the contraction and weakening rules are restricted. Propositions are modeled as resources. In real world software applications, if resources—such as database connections, file handles or even in-memory shared state—are freely copied or dropped from a program context, it can introduce bugs or crashes. If contraction and weakening are completely abandoned, the system gets overly restrictive. As a work around, the modality operator ! is introduced for controlled use of contraction and weakening. Wadler(1993) describes a refinement of linear logic based on Girard's Logic of Unity(1993). It is a disjoint union of linear logic and intuitionistic logic: $[A]$ means that A is an intuitionistic assumption and the rules of weakening and contraction are admissible, while $\langle A \rangle$ would mean that it is a linear assumption and weakening and contraction are prohibited.

$$
\begin{aligned}
A, B, C &::= X \mid {!A} \mid A \multimap B \mid A \,\&\, B \mid A \otimes B \mid A \oplus B \\
\Gamma, \Delta &::= \varepsilon \mid \Gamma, \langle A \rangle \mid \Gamma, [A]
\end{aligned}
$$

Figure 2.6: Grammar for Intuitionistic Linear Logic

In a linear logic, the assumptions can not be used more than once. In the proposition $A \multimap B$, the assumption A cannot be reused after it has been used to obtain B. This gives a new view of implication as consumption. Due to the absense of weakening and contraction rules in this system we obtain two fragments—multiplicative and additive—of connectives. Multiplicative conjunction is represented as $A \otimes B$, and additive conjunction and disjunction as $A \,\&\, B$ and $A \oplus B$ respectively. The syntax of linear logic is shown in Fig. 2.6.

The structural and connective logical rules for the complete system is given in Fig. 2.7. The

Structural Rules

$$\frac{}{[A] \vdash A}\,[\mathrm{ID}_{[]}] \qquad \frac{}{\langle A \rangle \vdash A}\,[\mathrm{ID}_{\langle\rangle}]$$

$$\frac{\Gamma, \Delta \vdash A}{\Delta, \Gamma \vdash A}\,[\mathrm{EXCH}] \qquad \frac{\Gamma, [A], [A] \vdash B}{\Gamma, [A] \vdash B}\,[\mathrm{CTRN}] \qquad \frac{\Gamma \vdash B}{\Gamma, [A] \vdash B}\,[\mathrm{WKN}]$$

$$\frac{[\Gamma] \vdash A}{[\Gamma] \vdash\, !A}\,[!\mathrm{I}] \qquad \frac{\Gamma \vdash\, !A \qquad \Delta, [A] \vdash B}{\Gamma, \Delta \vdash B}\,[!\mathrm{E}]$$

Connective Rules

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\,[\multimap I] \qquad \frac{\Gamma \vdash A \multimap B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B}\,[\multimap E]$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}\,[\& I] \qquad \frac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash A}\,[\& E_1] \qquad \frac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash B}\,[\& E_2]$$

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\,[\otimes I] \qquad \frac{\Gamma \vdash A \otimes B \qquad \Delta, A, B \vdash C}{\Gamma, \Delta \vdash C}\,[\otimes E]$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}\,[\oplus I_1] \qquad \frac{\Delta \vdash B}{\Delta \vdash A \oplus B}\,[\oplus I_2] \qquad \frac{\Gamma \vdash A \oplus B \qquad \Delta, A \vdash C \qquad \Delta, B \vdash C}{\Gamma, \Delta \vdash C}\,[\oplus E]$$

Figure 2.7: Intuitionistic Linear Logic Rules

exponential modality ! is used to relax linearity constratins. It 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 implication $A \supset B$ can be encoded in linear logic by using the modality operator as $!A \multimap B$. We clearly see that this is a more powerful system in comparison to intuitionistic logic because of its enhanced expressivity for handling resources. However, there is an awkward asymmetry between the connectives. The absense of structural rules enables conjuntion to have multiplicative and additive fragments. To express additive implication the use of modality is necessary.

There have been several prototype languages based on linear logic. L^3(Ahmed et al., 2007) is an intermediate language that is built on a linear type system and supports strong updates. Lolliproc(Mazurak and Zdancewic, 2010) explores the use of linear logic in concurrent functional programming while F°(Mazurak et al., 2010) uses kinds to distinguish between linear and unrestricted types for general purpose programming. 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. Qualified types are the types that satisfy all the predicates for the term. A modification of Damas-Milner algorithm $\mathcal{W}$ to incorporate predicates ensures that type inference is sound and complete. Qualified types are powerful enough to expresses type classes with functional dependencies(Jones, 2000), record types(Gaster and Jones, 1996), sub-typing(Jones, 1994) and first class polymorphism(Jones, 1997).

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

Figure 2.8: Qualified Types and Expression Language

The type language from Fig. 2.1 is modified 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. 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 $P \Rightarrow \pi$ asserts that the predicate π can be inferred from the predicates in P. The typing rules in **HM** type system are slightly modified and two 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 new rules are highlighted. The predicate set P is threaded throughout the other rules but is not used anywhere except [⇒I] and [⇒E].

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

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

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

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

Figure 2.9: Typing Rules for Qualified Types

2.4 Quill: Linear Logic with Qualified Types

Quill(Morris, 2016) uses the framework of qualified types to implement a linear type system with a sound and complete type inference. 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 $\tau \geq \tau'$ holds 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, `Un` can be thought 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 weakening and contraction. The predicate `Fun` τ implies that the type τ is of a function type. The function may or may not capture resources in its closure and so the functions themselves can be of restricted or unrestricted type.

Simple types such as integers and Booleans are all of unrestricted type, as they can be duplicated or dropped freely. Program resources such as file handles, database connections are 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 $\lambda x.\mathcal{F}x$ would depend on the linearity of $\mathcal{F}$. To generalize, we can say that

```
class Un where
    dup  :: t →! (t ⊗ t)
    drop :: t →! 1
```

Figure 2.10: `Un` as a Typeclass

the linearity of the lambda expression depends on its closure. The type of $\lambda f.\lambda x.fx$ can be written as $(\tau \xrightarrow{f} \upsilon) \rightarrow \tau \xrightarrow{g} \upsilon$. This function would be well typed only if f is less restricting i.e. admits more structural rules than g, so to say $\forall fg\tau\upsilon.\{f \geq g\} \Rightarrow (\tau \xrightarrow{f} \upsilon) \rightarrow \tau \xrightarrow{g} \upsilon$.

2.5 Logic of Bunched Implications (***BI***) 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 two connectives comma (,) or a semicolon (;). The logic of ***BI*** combines substructural 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 for propositions A, B and C: $A,(B;C) \neq A,B;A,C$ and $A;(B,C) \neq A;B,A;C$.

O'Hearn and Pym (1999) introduce two 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 introduction rules for implication:

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

The weakening rule is prohibited on Γ, A so nither A or Γ can be dropped out of context. Similarly the contraction rule is prohibited that disallows formation of Γ, A, A from Γ, A 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 (or vice versa) 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 additive implication as there is there is no modality introduced in the system. The logic of ***BI*** combines the additive logic i.e. classical intuitionistic logic with the multiplicative side i.e. intuitionistic substructural 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 express the non-resource intuitionistic parts.

$$A, B, C ::= A -\!\!* B \mid A \twoheadrightarrow B \mid A \& B \mid A \otimes B \mid A \oplus B$$
$$\Gamma, \Delta ::= \{\}_m \mid \{\}_A \mid \Gamma, A \mid \Gamma; A$$

Figure 2.11: Grammar for Logic of ***BI***

The complete grammar for logic of ***BI*** is shown in Fig. 2.11. The contexts $\{\}_m$ represents a multiplicative empty context while the $\{\}_a$ represents an additive empty context. The rules for the logic system are shown in Fig. 2.12. $\Gamma(\Delta)$ means that Δ is a sub-tree within Γ. We give more details about sub-trees in Chapter 4.

$\alpha\lambda$-calculus(O'Hearn, 1999; Pym, 2002) is the Curry-Howard interpretation of the logic of ***BI***. It views implication in terms of sharing rather than the number of times it is used. It introduces 2 kinds of arrows by modifying the the syntax of lambda calculus:

1. $-\!\!*$: Functions that do not share resources with their arguments

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

The types and terms of $\alpha\lambda$-calculus are shown in Fig. 2.13. The structural and connective rules for $\alpha\lambda$-calculus are summarized in Fig. 2.14. We have left out the terms for additive and multiplicative conjunction and additive disjunction for the sake of simplicity.

Structural

$$\frac{}{A \vdash A}\,[\mathrm{ID}] \qquad \frac{\Gamma(\Delta) \vdash A}{\Gamma(\Delta;\Delta') \vdash A}\,[\mathrm{WKN}] \qquad \frac{\Gamma(\Delta;\Delta) \vdash A}{\Gamma(\Delta) \vdash A}\,[\mathrm{CTR}]$$

Multiplicative Connectives

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \mathrel{-\!\!*} B}\,[-\!\!*\mathrm{I}] \qquad \frac{\Gamma \vdash A \mathrel{-\!\!*} B \quad \Delta \vdash A}{\Gamma, \Delta \vdash B}\,[-\!\!*\mathrm{E}]$$

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\,[\otimes I] \qquad \frac{\Gamma(A,B) \vdash C \quad \Delta \vdash A \otimes B}{\Gamma(\Delta) \vdash C}\,[\otimes E]$$

Additive Connectives

$$\frac{\Gamma; A \vdash B}{\Gamma \vdash A \twoheadrightarrow B}\,[\twoheadrightarrow\mathrm{I}] \qquad \frac{\Gamma \vdash A \twoheadrightarrow B \quad \Delta \vdash A}{\Gamma; \Delta \vdash B}\,[\twoheadrightarrow\mathrm{E}]$$

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma; \Delta \vdash A \,\&\, B}\,[\&\mathrm{I}] \qquad \frac{\Gamma(A;B) \vdash C \quad \Delta \vdash A \,\&\, B}{\Gamma(\Delta) \vdash C}\,[\&\mathrm{E}]$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}\,[\oplus\mathrm{I}_1] \qquad \frac{\Delta \vdash B}{\Delta \vdash A \oplus B}\,[\oplus\mathrm{I}_2] \qquad \frac{\Gamma \vdash A \oplus B \quad \Delta(A) \vdash C \quad \Delta(B) \vdash C}{\Gamma(\Delta) \vdash C}\,[\oplus\mathrm{E}]$$

Figure 2.12: Rules for Logic of ***BI***

The logic of ***BI*** patches up the awkward asymmetry experienced in linear logic. The multiplicative conjunction $\otimes$ has a right adjoint counterpart as $-\!\!*$ while additive conjunction & has a right adjoint counterpart $\twoheadrightarrow$. In other words, $A \otimes B \vdash C$ iff $A \vdash B \mathrel{-\!\!*} C$ and $A \,\&\, B \vdash C$ iff $A \vdash B \twoheadrightarrow C$. This relieves us from introducing modality into the system.

Due to the rules of $\alpha\lambda$-calculus $f : \tau \mathrel{-\!\!*} \tau'; x : \tau \nvdash fx : \tau'$, as f needs an argument that does not share any resources with it-self. The term $\lambda x.\alpha f.fxx : \tau \mathrel{-\!\!*} (\tau \twoheadrightarrow \tau \twoheadrightarrow \tau') \twoheadrightarrow \tau'$ is a typable term in $\alpha\lambda$-calculus as shown in Fig. 2.15. This illustrates the difference between logic of ***BI*** and linear logic as even when the argument is separate from the function, it may be used twice. In linear logic this expression will be ill-typed.

The use of logic of ***BI*** as a type inference system is an active 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

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

Figure 2.13: $\alpha\lambda$-Calculus Types and Terms

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

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

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

Figure 2.14: Typing Rules for $\alpha\lambda$-Calculus

on separation of resources used by objects. Collinson et al. (2005) designs a polymorphic variant of $\alpha\lambda$-calculus.

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

Figure 2.15: Multiplicative Argument used Twice in $\alpha\lambda$-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 no files that remain open when they are no longer in use. This seemingly trivial responsibility becomes tedious and error prone as the program gets more 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` to keep track of the resource. 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 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 second `goto fail;` on line 6 would always 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 the operating system closes the file handles that are not closed when the program exits. In a long running process the operating system would run out of space to allocate new file handles. The whole process would crash with an error that it cannot

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

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

Figure 3.4: Goto Fail bug

open any more file handles. Abnormal exit from the process may cause of loss of information as would interfere in the write process. 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 might 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 they would cause programs to crash 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 `-*>`

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

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

```
(>>=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 (Desugared)

to mean $\multimap$ and -&> to mean $\twoheadrightarrow$. The file handling functions will have 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`. This would be a type error as the bind operation should have a type 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 the 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. In Haskell, error handing is done using `MonadError`(Liang et al., 1995). 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 or rethrow 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 clean-up in a systematic way in QuB with the help of the type system based on ***BI***. 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. The function throw can be used by another function to start the processing of the exception. The function `catch` can convert a code that can fail, to a code that does not fail. It gives a way to clean up resources after the code throws an exception. The sharing arrow between the two arguments of `catch` enables the clean up block of code to be executed on the same resources that were used its first argument i.e. the same file handle will be shared between the arguments.

```
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

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) `catch`
           (\e -> do closeFile fh
                     return $ Left "Could not read file")
```

Figure 3.12: Exception Handling in QuB

Chapter 4

Core Language Syntax and Types

In this chapter we give the formal description of QuB's 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.

$t, u \in$ Type Variables

$P, Q \in$ Finite Predicate Set

$$\begin{array}{rl} \text{Types} & \tau, \upsilon, \phi ::= t \mid \iota \mid \tau \to \tau \\ \text{where} & \to \in \{\overset{!}{\twoheadrightarrow\!\!*}, \multimap\!\!*, \overset{!}{\twoheadrightarrow}, \twoheadrightarrow\} \\ \text{Predicates} & \pi, \omega ::= \texttt{Un}\ \tau \mid \texttt{SeFun}\ \tau \mid \texttt{ShFun}\ \tau \mid \tau \geq \tau' \\ \text{Qualified Types} & \rho ::= \tau \mid \pi \Rightarrow \rho \\ \text{Type schemes} & \sigma ::= \rho \mid \forall t . \sigma \end{array}$$

Figure 4.1: Types QuB

The type language consists of type variables (t, u), built-in types such as integers, booleans (ι), and four function types the sharing arrow ($\twoheadrightarrow$) and the separating arrow ($\multimap\!\!*$) and unrestricted version of both the function types ($\overset{!}{\twoheadrightarrow}, \overset{!}{\multimap\!\!*}$ respectively). 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.

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 do not have any resources or whose resources can be duplicated or deleted easily. Built-in types such as `Int`, `Char` are considered unrestricted types. We write $\texttt{ShFun}\ \phi$ to describe that type ϕ shares 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. they may 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 $\overset{!}{-\!\!*}$ and $\overset{!}{\twoheadrightarrow}$ 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 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.

$$\frac{\pi \in P}{P \Rightarrow \pi} \qquad \frac{\bigwedge_{\pi \in Q} P \Rightarrow \pi}{P \Rightarrow Q} \qquad \frac{}{P \Rightarrow \mathtt{Un}\ (\tau \overset{!}{-\!\!*} \tau')} \qquad \frac{}{P \Rightarrow \mathtt{Un}\ (\tau \overset{!}{\twoheadrightarrow} \tau')}$$

$$\frac{}{P \Rightarrow \tau \geq (\upsilon -\!\!* \upsilon')} \qquad \frac{}{P \Rightarrow \tau \geq (\upsilon \twoheadrightarrow \upsilon')} \qquad \frac{P \Rightarrow \mathtt{Un}\ \tau}{P \Rightarrow \tau \geq (\upsilon \overset{!}{-\!\!*} \upsilon')} \qquad \frac{P \Rightarrow \mathtt{Un}\ \tau}{P \Rightarrow \tau \geq (\upsilon \overset{!}{\twoheadrightarrow} \upsilon')}$$

$$\frac{\tau = -\!\!* \vee \tau = \overset{!}{-\!\!*}}{P \Rightarrow \mathtt{SeFun}\ \tau} \qquad \frac{\tau = \twoheadrightarrow \vee \tau = \overset{!}{\twoheadrightarrow}}{P \Rightarrow \mathtt{ShFun}\ \tau} \qquad \frac{P \Rightarrow \tau \geq \phi t \qquad t \text{ fresh}}{P \Rightarrow \tau \geq \phi}$$

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, the relation is denoted by $\Gamma(\Delta)$. The bunches have a restriction that no identifier appears more than once. Certain structural rules are restricted on the context depending on the connectives used. If contexts are combined using a comma (,), contraction and weakening is not admissible, but if the contexts are combined using a semicolon (;), it may undergo contraction and weakening. Exchange rule is admissible in both the connectives. This distinction enables a special treatment for resources in within the language. By adding resources with a comma constructor in the typing envrionment, the type system will not dispose them off by using the contraction rule, while non-resourceful objects (or normal propositions) can be added using the semi-colon constructor. An example bunch is shown in Fig. 4.3. a and b are in sharing while, c is separate from the a and b. If Γ represents the complete context of Fig. 4.3, $\Delta \equiv (a : A; b : B)$ and $\Delta' \equiv (c : C)$ then $\Gamma \equiv \Delta, \Delta'$ and $\Gamma(\Delta)$. In general, two bunches are said to be equivalent ($\Gamma \equiv \Delta$) if they can be transformed into another by re-associativity.

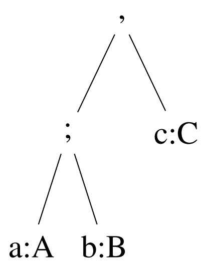


Figure 4.3: Bunch in ***BI***

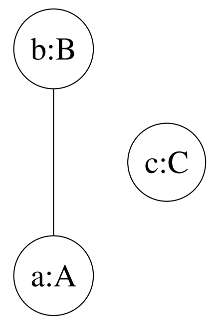


Figure 4.4: Sharing Graph in QuB

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. They also internalize the transformations that are made explicit in the logic of ***BI***. For example, the context tree in ***BI*** $(a : A, a : B), c : C$ is equivalent to $a : A, (b : B, c : C)$ while in QuB the sharing graphs of both the trees would be equal. The sharing graph also preserves the distributive law where $a : A; (b : B, c : D) \vdash (a : A; b : B), (a : A; c : C)$

We define sharing relation, Ψ, as a mapping 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 $\mathtt{dom}(\Psi) = \{x \mid (x, \vec{y}) \in \Psi\}$, where $\vec{y}$ is a shorthand for the denoting collection of variables that are 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, symmetric and non-transitive. The sharing relation is non-transitive due to the natural notion of sharing. For example, given three variables, x, y, z: If x is in sharing with y and y is in sharing with z need not imply x is in sharing with z. So to say,

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

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

$$\forall_{x,y,z \in \mathtt{dom}(\Psi)}\ \text{if } y \in \Psi(x) \text{ and } z \in \Psi(y) \not\Rightarrow z \in \Psi(x) \qquad \text{(non-transitive)}$$

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 $x^{\vec{y}} : \tau$ 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 for our system as shown in Fig. 4.5.

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

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. $\mathtt{Used}(\Gamma)$ 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. With respect to sharing graphs it would mean adding edges between the nodes.

$$\begin{aligned} \texttt{Vars}(\Gamma, x^{\vec{y}} : \tau) &= \texttt{Vars}(\Gamma) \cup \{x\} \\ \texttt{Shared}(\Gamma, x^{\vec{y}} : \tau) &= \texttt{Shared}(\Gamma) \cup \{\vec{y}\} \\ \texttt{Used}(\Gamma) &= \texttt{Vars}(\Gamma) \cup \texttt{Shared}(\Gamma) \end{aligned}$$

$$(\Gamma, x^{\vec{y}} : \tau)^{[a \mapsto \vec{b}]} = \begin{cases} a \notin \vec{y} & (\Gamma^{[a \mapsto \vec{b}]}, x^{\vec{y}} : \tau) \\ a \in \vec{y} & (\Gamma^{[a \mapsto \vec{b}]}, x^{(\vec{y} \backslash a) \cup \vec{b}} : \tau) \end{cases}$$

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

Figure 4.6: Auxiliary Functions on Type Assignments

Two type assignments are said to be in disjoint union ($\circledast$) if none of the terms used in the type assignments are common with shared terms of other type assignment. 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.

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

Figure 4.7: Type Assignment Operations

Term Variables	$x, y, z \in \text{Var}$
Expressions	$M, N ::= x \mid \lambda^{\twoheadrightarrow\!\!*} x.M \mid \lambda^{\twoheadrightarrow} x.M \mid MN \mid \texttt{let } x = M \texttt{ in } N$

Figure 4.8: Term Language

Our term language is similar to that of simply typed lambda calculus involving variables and term application but we have two types of lambda expressions, the sharing lambda ($\lambda^{\twoheadrightarrow}$) denotes sharing of the argument term with the expression M and the separating lambda term ($\lambda^{\twoheadrightarrow\!\!*}$) that implies the argument term has a separating context with the expression M. We also have `let` construct to enable parametric polymorphism.

We work in a call-by-value semantics similar to Quill(Morris, 2016) and F°(Mazurak et al., 2010) as it is possible that functions may not be evaluated in call-by-name or call-by-need semantics that would break the linearity property.

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 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. We begin this chapter describing conventions and notations that are followed throughout the thesis.

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. We write $\sigma = \Gamma_\sigma(x)$ for the type scheme σ, assigned to the term x in Γ. $\mathtt{dom}(\Gamma)$ is the set of identifiers in the type assignment i.e. $\mathtt{dom}(\Gamma) = \{x \mid (x^{\vec{y}} : \sigma) \in \Gamma\}$. $\Gamma_\Psi(x)$ denotes the sharing information of the variable x.

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.

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

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 unrestricted which is captured by the antecedent predicate Δ un on the type that is dropped or duplicated.

The [LET] rule is used to enable parametric polymorphism as usual. The rules of [$\twoheadrightarrow$I] and [$-\!\!*$I] describes the abstraction over shared and separating resources respectively, while [$\twoheadrightarrow$E] and [$-\!\!*$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 $\lambda^{-\!*}x.M$ and $\lambda^{\twoheadrightarrow}.M$ would be assigned function predicates ShFun ϕ or SeFun ϕ only if it is more restricting that its environment. This is specified in the judgments $\cdot \geq \cdot$ shown in Fig. 5.4. To avoid name shadowing, we would assume that the binders introduce fresh variables. 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 $\lambda^{-\!*}$ the binder variable is

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

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

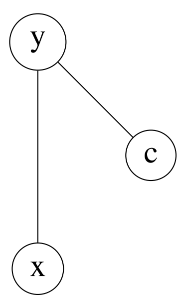
$$\frac{P, \pi \mid \Gamma \vdash M : \rho}{P \mid \Gamma \vdash M : \pi \Rightarrow \rho}\ [\Rightarrow \text{I}] \qquad \frac{P \mid \Gamma \vdash M : \pi \Rightarrow \rho \qquad P \vdash \pi}{P \mid \Gamma \vdash M : \rho}\ [\Rightarrow \text{E}]$$

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

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

Figure 5.2: Connective Typing Rules

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



$$\Gamma = \{c^y : \tau -\!* \upsilon, x^y : \tau, y^{\{c,x\}} : \tau'\}$$

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

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

$$\boxed{P \vdash \cdot \text{ un}}$$

$$\frac{P \Rightarrow \texttt{Un}\ \tau}{P \vdash \tau \text{ un}}\ [\texttt{Un-}\tau] \qquad \frac{P, \pi \vdash \rho \text{ un}}{P \vdash \pi \Rightarrow \rho \text{ un}}\ [\texttt{Un-}\rho]$$

$$\frac{P, \texttt{Un}\ t \vdash \sigma \text{ un}}{P \vdash \forall t.\sigma \text{ un}}\ [\texttt{Un-}\sigma] \qquad \frac{\bigwedge_{x:\sigma \in \Gamma} P \vdash \sigma \text{ un}}{P \vdash \Gamma \text{ un}}\ [\texttt{Un-}\Gamma]$$

$$\boxed{P \vdash \cdot \geq \cdot}$$

$$\frac{P \Rightarrow \tau \geq \phi}{P \vdash \tau \geq \phi}\ [\geq\text{-}\tau] \qquad \frac{P, \pi \vdash \rho \geq \phi}{P \vdash (\pi \Rightarrow \rho) \geq \phi}\ [\geq\text{-}\rho]$$

$$\frac{P, \texttt{Un}\ t \vdash \sigma \geq \phi}{P \vdash (\forall t.\sigma) \geq \phi}\ [\geq\text{-}\sigma] \qquad \frac{\bigwedge_{x:\sigma \in \Gamma} P \vdash \sigma \geq \phi}{P \vdash \Gamma \geq \phi}\ [\geq\text{-}\Gamma]$$

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. [$-\!\!*$I^s] is used in occurence of $\lambda^{-\!*}$, 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 [$-\!\!*$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 separating. 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.

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

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

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

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

Context Operations

$$\Gamma \tilde{\circledast} \Delta = \mathtt{Used}(\Gamma) \,\#\, \mathtt{Vars}(\Delta) \wedge \mathtt{Used}(\Delta) \,\#\, \mathtt{Vars}(\Gamma) \qquad \Gamma \tilde{\oplus} \Delta = \mathtt{Used}(\Gamma) = \mathtt{Used}(\Delta)$$

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 separate from both Γ and Γ'. Thus Γ and Γ' will be completely separate from each other while Γ'' would be in sharing with both Γ and Γ'. Δ would be unrestricted as it is not being shared by either M or 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 $\tau' \twoheadrightarrow \tau$. The conditions outlined for [App^s] do not 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. $\Gamma \tilde{\otimes} \Delta$ is an assertion that there exists a proof either possible to find by inspecting the type assignment, or provided by the user that $\Gamma \otimes \Delta$ is defined. Similarly, $\Gamma \tilde{\oplus} \Delta$ would mean 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* $(P \mid \sigma) \sqsubseteq Gen(\Gamma, Q \Rightarrow \tau)$

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. The theorems 5.1 and 5.2 together provides an assurance of both the systems being equivalent.

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 and 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, 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 subscripts denote fresh type variables.

We define some auxilary functions shown 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. $\mathtt{Weaken}(x,\sigma,\Sigma,\Psi)$ adds the unrestricted predicate to the type σ if x does not belong to used variables Σ and is not in sharing with any used variables. $\mathtt{Un}(\Gamma)$ adds an unrestricted predicate to all the types of the variables that are in the domain of Γ. $\mathtt{GenI}(\Gamma, P \Rightarrow \tau)$ generalizes the qualified type to a type scheme. $\mathcal{C}(\Gamma,\Psi,\Sigma)$ computes the closure of sharing information of variables in Γ restricted to the variables in Σ.

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

$$\mathtt{GenI}(\Gamma, P \Rightarrow \tau) = \forall(\mathtt{fvs}(P) \cup \mathtt{fvs}(\tau) \backslash \mathtt{fvs}(\Gamma)).P \Rightarrow \tau \qquad \mathcal{C}(\Gamma,\Psi,\Sigma) = \bigcup_{x\in\mathtt{dom}(\Gamma)\wedge x\in\Sigma} \Psi(x)$$

$$\mathtt{Weaken}(x,\sigma,\Sigma,\Psi) = \begin{cases} P & \text{if } x \notin \Sigma \wedge \Psi(x) \# \Sigma, P \vdash \sigma\ \mathtt{un} \\ \varnothing & otherwise \end{cases}$$

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 υ 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 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 used in it to avoid recursive definition leading 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 type inference algorithm inferes a type that the original type system would have produced. The proof is given in Appendix B.

This type inference algorithm is however not complete. There would be cases where a term can be typed in two ways. Algorithm $\mathcal{M}$ would compute only one of them and leave out the other one. Consider the type for $\lambda^{\twoheadrightarrow\!\!*} f.\lambda^{\twoheadrightarrow\!\!*} x.fxx$. This term can be inferred to have two types by using algorithm $\mathcal{M}$.

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

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

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

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

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

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

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

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

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

Chapter 6

QuB Extention and Datatypes

In this chapter we describe an extension of original QuB type system to include kinds. §6.1 gives an account of how this extension makes the type language generic enough to accept user defined types. We also discuss how we can encode sum types and multiplicative and additive product types in the extended system in §6.2 and how algebriac data types are more expressive due to sharing and separation in §6.3

6.1 Kind System

In the original system, it would be tedious to extend the system with new types. Introducing new types would mean having to introduce new syntax to represent the types 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 originally introduced by Barendregt(1991). Jones(1993) adapted the idea for a system of qualified types. We follow Jones' approach to add the language of type constructors and kinds in our system.

The complete type system is shown in Fig. 6.1. We annotate each type with its kind in superscript. T^{κ} denotes type constructors and its kind κ depends on its arity. $\tau^{\kappa' \to \kappa} \tau^{\kappa'}$ denotes application of types. $-\!\!*$, $\overset{!}{-\!\!*}$, $\twoheadrightarrow$ and $\overset{!}{\twoheadrightarrow}$ 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 $\star$. 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

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

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

would result in a kind κ.

The unification of types has to be done using 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 kind preserving *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 S in a form of $R \circ S$ where R is 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 T's 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 $\kappa' \to \kappa$.

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

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

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

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

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

Figure 6.3: Kind Preserving Unification of Type Constructors

New terms are added in the expression language to support the new types. $\langle M,N \rangle$ denotes a multipicative or separating pair with $\mathtt{let}\ \langle x,y \rangle = M\ \mathtt{in}\ N$ as its deconstructor. The first component of the pair would be bound to x while the second component will be bound to y. Both x and y would have to be used in the expression N for the pair to be well typed. $\langle M;N \rangle$ denotes an additive or sharing pair. $\mathtt{fst}\ M$ and $\mathtt{snd}\ N$ would denote the first and the second components of the pair respectively. The $\mathtt{case}\ M\ \mathtt{of}\ \{\mathtt{inl}\ x \mapsto N; \mathtt{inr}\ y \mapsto N'\}$ denotes pattern matching on the sum type, where $\mathtt{inl}\ x$ and $\mathtt{inr}\ y$ denotes the construction of the sum type.

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

Figure 6.4: Extended QuB Language Syntax

6.2 Pairs and Sums in QuB

Introducing sharing and separating implication in our type system leads to two different flavors of pairs. This distinction cannot be made in intuitionistic logic as the structural rules allow re-use of propositions. Restrictions in weakening and contraction allows us to 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

$$
\begin{aligned}
\otimes &\in \mathcal{T}^{\star \to \star \to \star} \\
\tau \otimes \tau' &= \tau \mathrel{-\!\!*} \tau' \mathrel{-\!\!*} (\tau \mathrel{-\!\!*} \tau' \mathrel{-\!\!*} \upsilon) \mathrel{-\!\!*} \upsilon \\
(\,,\,) &= \lambda^{-\!*} x.\lambda^{-\!*} y.\lambda^{-\!*} f.f x y
\end{aligned}
$$

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

Lambda encoding of multiplicative pairs is given in Fig. 6.5. The typing rules are given in Fig. 6.6. We give the proof of derivation 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 individually. Failure to refer both the components of the multiplicative pair, would raise a type error regarding the resources not being unrestricted.

6.2.2 Additive Pair

$$
\begin{aligned}
\& &\in \mathcal{T}^{\star\to\star\to\star} \\
\tau \,\&\, \tau' &= \tau \mathrel{-\!\!\ast} \tau' \twoheadrightarrow (\tau \mathrel{-\!\!\ast} \tau' \twoheadrightarrow \upsilon) \twoheadrightarrow \upsilon \\
(\,;\,) &= \lambda^{\mathrel{-\!\!\ast}} x.\lambda^{\twoheadrightarrow} y.\lambda^{\twoheadrightarrow} f.fxy
\end{aligned}
$$

Figure 6.7: Additive Pair

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

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

Figure 6.8: Derivable Typing Rules for Additive Pair

The lambda encoding for additive pairs is given in Fig. 6.7. 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. Application of these projection functions on multiplicative pairs would result in implicitly discarding one of the components of the multipicative pair and would raise a type error.

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

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

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

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` $\{f;g\}$ term is the deconstructor that would decide which function to use f or g depending on the type 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.

6.3 Generic Type Constructors

To leverage the full power of sharing and separating in a programming language, we want to enable the users to explicitly define datatypes with sharing and separation between its constituent resources. The [APP^s] rule will ensure that the types are well-formed using the modified Robinson's unification algorithm for user defined 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.

We illustrate using examples how the type system works. The sharing pair is shown in Fig. 6.11. The keyword `data` introduces the type constructor in to the language. The (!!) denotes that `a` and `b` are in sharing. The sharing pair has shared resources and using `fst` and `snd` works to get the individual constituents as shown in Fig. 6.11 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 6.11: Shared Pairs

Separating pair is defined in Fig. 6.12. The functions `fst` and `snd` do not type check as one of the resources is being implicitly dropped. This would force the type inference algorithm to mark the resource to as unrestricted, but as both `a` or `b` are resources, a type mismatch error would be raised at compile time. The `swap` function would type check as expected as both `a` and `b` are used.

```
data Pair a b = SeP a b

fst :: Pair' a b -> a -- Does not type check
fst (SeP x y) = x

snd :: Pair' a b -> b -- Does not type check
snd (SeP x y) = y

swap :: Pair a b -> Pair b a -- Type checks
swap SeP x y = SeP y x
```

Figure 6.12: Separating Pair

Chapter 7

Conclusion and Future Work

We have designed and implemented QuB, a type system based on logic of ***BI***. It offers a conglomeration of **HM** type system and intuitionistic linear type system. We have shown using examples how it is more expressive than existing type systems by statically tracking resources. We have implemented a prototype of QuB(Ingle, 2018) based on Habit(Diatchki, 2007; Morris, 2013), a statically typed call-by-value Haskell-like functional programming language.

Type systems based on sub-structural logics are an active area of research. For future work, we would like to solve the incompleteness problem for QuB type inference algorithm as terms can have two incomparable types depending on the predicates and its context. We would want to have a clear way of solving the dilemma and be able to detect appropriate types automatically in the type inference algorithm. We have not given any formal semantic model for our language to rigorously prove soundness with respect to original system of logic of ***BI*** and neither have we tried to analyze the system with respect to categorical models. We would want to pursue them from a theoretical perspective.

Finally, static type systems provide two benefits, compile time guarantee of program correctness and identifying opportunities for code optimization. We hope to implement larger programs in QuB to be able push the limits of both the guarantees. We conjuncture that low-level programs that are sensitive to resource management will be a good fit to leverage the expressiveness of QuB and detect a larger subset of runtime errors at compile time.

References

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

Atkey, R. (2004). A λ-calculus for resource separation. In Díaz, J., Karhumäki, J., Lepistö, A., and Sannella, D., editors, *Automata, Languages and Programming*, volume 3142, pages 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., and 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., and Robinson, E. (2005). On bunched polymorphism. In Ong, L., editor, *Computer Science Logic*, Lecture Notes in Computer Science, pages 36–50. Springer, Berlin, Heidelberg.

Damas, L. and 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, pages 207–212. ACM.

Diatchki, I. S. (2007). *High-Level Abstractions for Low-Level Programming*. PhD thesis, OGI School of Science & Engineering at Oregon Health & Science University.

Gaster, B. R. and Jones, M. P. (1996). A polymorphic type system for extensible records and variant. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, University Park, Nottingham NG7 2RD.

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., and Lafont, Y. (1989). *Proofs and Types*. Cambridge University Press.

Ingle, A. (2018). Qub. `https://github.com/habit-lang/alb/tree/work/qub`.

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

Liang, S., Hudak, P., and Jones, M. (1995). Monad transformers and modular interpreters. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 333–343. ACM.

Mazurak, K. and Zdancewic, S. (2010). Lolliproc: To concurrency from classical linear logic via curry-howard and control. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 39–50. ACM.

Mazurak, K., Zhao, J., and Zdancewic, S. (2010). Lightweight linear types in system F°. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, pages 77–88. ACM.

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

Morris, J. G. (2013). *Type Classes and Instance Chains: A Relational Approach*. PhD thesis, Portland State University.

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, pages 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, pages 258–279. Springer-Verlag.

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

Pierce, B. C. (2002). *Types and Programming Languages*. MIT Press.

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 Borzyszkowski, A. M. and Sokolowski, S., editors, *Mathematical Foundations of Computer Science 1993: 18th International Symposium, MFCS'93 Gdańsk, Poland, August 30–September 3, 1993 Proceedings*, pages 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.

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

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

$$\boxed{\tau \,\&\, \tau' = (\tau \twoheadrightarrow \tau' \twoheadrightarrow \upsilon) \twoheadrightarrow \upsilon}$$

We now define sharing (additive) pair constructor as:

$$; : \tau \mathrel{-\!\!*} \tau' \twoheadrightarrow (\tau \,\&\, \tau')$$

$$; = \lambda^{-\!*} x.\lambda^{\twoheadrightarrow} y.\lambda^{\twoheadrightarrow} f.fxy$$

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.

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

We assign a type symbol (⊗) for the multiplicative pair

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

We can now define separating (multiplicative) pair as:

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

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

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.

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

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

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

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

A.2 Derivable Typing rules for Sum Types

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \mid g^{cf} : (B \twoheadrightarrow E) \vdash g : (B \twoheadrightarrow E)}\;[\mathrm{ID}] \qquad \zeta}{\varnothing \mid c^{fg} : ((A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E) \oplus f^{cg} : (A \twoheadrightarrow E) \oplus g^{cf} : (B \twoheadrightarrow E) \vdash cfg : E}}{\varnothing \mid c^{f} : ((A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E) \oplus f^{c} : (A \twoheadrightarrow E) \vdash \lambda^{\twoheadrightarrow} g.cfg : (B \twoheadrightarrow E) \twoheadrightarrow E}\;[\twoheadrightarrow \mathrm{I}]}{\varnothing \mid c^{\varnothing} : ((A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E) \vdash \lambda^{\twoheadrightarrow} f.\lambda^{\twoheadrightarrow} g.cfg : (A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E}\;[\twoheadrightarrow \mathrm{I}]}{\varnothing \mid I \vdash \lambda^{-\!\!*} c.\lambda^{\twoheadrightarrow} f.\lambda^{\twoheadrightarrow} g.cfg : ((A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E) -\!\!* (A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E}\;[-\!\!*\mathrm{I}]$$

where ζ is defined below:

$$\dfrac{\dfrac{}{\varnothing \mid f^{cg} : (A \twoheadrightarrow E) \vdash f : (A \twoheadrightarrow E)}\;[\mathrm{ID}] \qquad \dfrac{}{\varnothing \mid c^{fg} : ((A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E) \vdash c : ((A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E)}\;[\mathrm{ID}]}{\varnothing \mid c^{fg} : ((A \twoheadrightarrow E) \twoheadrightarrow (B \twoheadrightarrow E) \twoheadrightarrow E) \oplus f^{cg} : (A \twoheadrightarrow E) \vdash cf : (B \twoheadrightarrow E) \twoheadrightarrow E}\;[\twoheadrightarrow \mathrm{E}]$$

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.

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

Left injection defined below as:

$$\texttt{inl} : A \mathrel{-\!*} A \oplus B$$

$$\texttt{inl} = \lambda^{-\!*} x.\lambda^{\twoheadrightarrow} f.\lambda^{\twoheadrightarrow} g.fx$$

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

Right injection defined below as:

$$\texttt{inr} : B \mathrel{-\!*} A \oplus B$$

$$\texttt{inr} = \lambda^{-\!*} y.\lambda^{\twoheadrightarrow} f.\lambda^{\twoheadrightarrow} g.gy$$

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.

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

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

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

Appendix B

Proofs

Theorem 5.1 (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]. 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 `ShFun` ϕ and $\Gamma \geq \phi$ to construct a derivation of $P \mid \Gamma \vdash \lambda^{\twoheadrightarrow} x.M : \phi \tau \tau'$.

- Case [$-\!\!*$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 [$-\!\!*$I] and reuse the derivations for `SeFun` ϕ and $\Gamma \geq \phi$ to construct a derivation of $P \mid \Gamma \vdash \lambda^{-\!\!*} x.M : \phi \tau \tau'$.

- 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 [$-\!\!*$E] to reuse derivations of `ShFun` ϕ or `SeFun` ϕ 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 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 $(P \mid \sigma) \sqsubseteq \mathtt{Gen}(\Gamma, Q \Rightarrow \tau)$*

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] and [WKN-UN] 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] 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 $\sigma = \mathtt{Gen}(\Gamma, P \Rightarrow \tau)$, 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 5.2. 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].

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

  as $\vec{u}$ are fresh i.e. $\vec{u} \notin \mathtt{dom}(\Gamma)$ 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 $(P \mid \sigma) \sqsubseteq (\varnothing \mid \mathtt{Gen}(\Gamma \circledast \Delta, Q \Rightarrow \tau))$
- 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 $(P \mid \sigma) \sqsubseteq (\varnothing \mid \mathtt{Gen}(\Gamma \oplus \Delta, Q \Rightarrow \tau))$
- 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 [$\multimap$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 [$\twoheadrightarrow$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 [$\twoheadrightarrow$I^s] we obtain the required derivation.

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

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

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

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

- 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 $\sigma' = \texttt{Gen}(\Gamma, Q \Rightarrow \upsilon)$, $(P \mid \forall t.\sigma) \sqsubseteq \sigma'$. 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 5.3 (Soundness of $\mathcal{M}$.)**.** *If $\mathcal{M}(S, \Psi, \Gamma \vdash M : \tau) = P, S', \Sigma, \Psi'$ then $S'P \mid S'\Gamma \vdash M : S'\tau$*

We go over a few intermediate results and definitions 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 $P \vdash^s \Delta$ $\mathtt{un}$ 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 5.2, 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 5.2, 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 $P \mid \Gamma \vdash^s M : \tau$ ■

Definition 7 (Simple Constraint)**.** A simple constraint in t is of the form $\mathtt{Un}\ t$, $\mathtt{ShFun}\ t$, $\mathtt{SeFun}\ t$ or $\tau \geq t$ where t is a type variable.

Lemma B.9 (Constraint Simplification)**.** *Suppose there is a non-trivial entailment $P \Rightarrow Q$.*

$$\exists Q' \text{ such that } P \Rightarrow Q' \Rightarrow Q$$

and Q' is simple and Q' is called simplification of Q.

Proof. Proof is by induction on derivation on $P \Rightarrow Q$. ■

Definition 8 (Non-trivial Entailment)**.** The entailment $P \Rightarrow Q$ is non-trivial if only uses of the assumption rule are for simple constraints.

Definition 9 (Improving Substitution)**.** Suppose P is simple, and X is a set of type variables. A substitution S is called improving substitution for X in P if:

- If $\mathtt{Un}\ t \in P \land \mathtt{ShFun}\ t \in P$ then $St = \overset{!}{\twoheadrightarrow}$
- If $\mathtt{Un}\ t \in P \land \mathtt{SeFun}\ t \in P$ then $St = \overset{!}{\rightarrow\!\!\ast}$
- If $\mathtt{Un}\ t \notin P \land \mathtt{ShFun}\ t \in P$ then $St = \twoheadrightarrow$
- If $\mathtt{Un}\ t \notin P \land \mathtt{SeFun}\ t \in P$ then $St = \rightarrow\!\!\ast$

Lemma B.10. *If $P \mid \Gamma \vdash^s M : \tau$ and $X = fvs(P) \backslash (fvs(\Gamma) \cup fvs(\tau))$, $P' \Rightarrow P$ and S is an improving substitution for $X \in P'$, then $SP \mid \Gamma \vdash^s M : \tau$*

Proof. Proof by induction on derivation of $P \mid \Gamma \vdash^s M : \tau$. The improving substitution will be bound to a type variable ϕ if the rules used are $[\twoheadrightarrow \mathrm{I}^s]$, $[\rightarrow\!\!\ast\, \mathrm{I}^s]$ or $[\mathrm{APP}^s]$, the substitution $S\phi$ is a suitable type for further use. ■

Proof of 5.3. 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 $[\mathrm{VAR}^s]$ to construct $[\vec{u}/\vec{t}]P \mid \{x : \forall \vec{t}. P \Rightarrow \tau\} \vdash^s x : [\vec{u}/\vec{t}]\tau$

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

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

Now, using [$\twoheadrightarrow$I^s] we get the required result.

- Case $\lambda^{-*}x.M$. 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 $Q = \{\mathtt{SeFun}\, u_1\} \cup \mathtt{Leq}(u_1, \Gamma|_{\Sigma}) \cup \mathtt{Weaken}(x, u_2, \Sigma, \Psi') \cup P$ and let $\Sigma' = \Sigma \backslash \{x\}$. By Lemma B.5 and Lemma B.6 we can construct a derivation of

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

Now, using [$-\!*$I^s] we get the required result.

- 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 $Q = P\cup P'\cup \mathtt{SeFun}\ u_1$ or if we can prove $\mathcal{C}(S\Gamma,\Sigma,\Psi) = \mathcal{C}((S\circ R)\Gamma,\Sigma',\Psi')$ then we let $Q = P\cup P'\cup \mathtt{ShFun}\ u_1$ (else the inference algorithm fails). Let $\Delta = \mathtt{dom}(\Gamma)\backslash\mathcal{C}(\Gamma,\Sigma\cup\Sigma',\Psi\circ\Psi')$ 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 $Q' = P\cup P'\cup Q\cup\{\mathtt{Un}\ \Delta\}$. We can now build the desired result using [APP^s].

- Case $\mathtt{let}\ x = M\ \mathtt{in}\ N$. 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 $\sigma = \mathtt{GenI}(R\Gamma;R(P\Rightarrow u_1))$. Let T improve $\mathtt{fvs}(P)$ $(\mathtt{fvs}(\Gamma)\cup\mathtt{fvs}(R\tau))$ in P, then there is a partition of Γ into Γ_M, Γ_N and Δ such that, by Lemma B.10, Lemma B.8 and induction hypothesis we have

$$(T\circ R')P \mid R'(\Gamma_M\circledast\Delta)\vdash^s M : Ru_1)$$

$$R'P \mid R'(\Gamma_N\circledast\Delta\sqcup\{x:\sigma\})\vdash^s N : R'\tau)$$

now, using $R'P\vdash\Delta$ un and [LET^s] we get the desired result. ■