

QuB: A Resource Aware Functional Programming Language

By

Apoorv Ingle

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

Committee members

Dr. J. Garrett Morris, Chairperson

Dr. Perry Alexander

Dr. Andy Gill

Dr. Prasad Kulkarni

Date defended: July 21, 2018

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

QuB: A Resource Aware Functional Programming Language

Dr. J. Garrett Morris, Chairperson

Date approved: July 21, 2018

Abstract

Modern programming languages treat resources as normal values. The static semantics of resources in such languages does not match 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, a Curry-Howard interpretation of logic of bunched implications (*BI*) (O’Hearn, 2003; Pym, 2002). 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 help in statically detecting resource errors while modeling exceptions that is difficult to capture in linear languages.

Acknowledgements

TODO TODO TODO DO DO DO DO.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 2 | Background Work | 3 |
| 2.1 | Hindley-Milner Type System and Type Inference Algorithm | 3 |
| 2.2 | Linear Logic | 7 |
| 2.3 | Qualified Types | 9 |
| 2.4 | Quill: Linear Logic with Qualified Types | 10 |
| 2.5 | Logic of Bunched Implications and $\alpha\lambda$ -Calculus | 11 |
| 3 | Programming in QuB | 15 |
| 3.1 | File Handles | 15 |
| 3.2 | Exception handling | 19 |
| 4 | Core Language Syntax and Types | 22 |
| 4.1 | Conventions and Notations | 27 |
| 5 | Type System and Type Inference | 28 |
| 5.1 | Type System | 28 |
| 5.2 | Syntax Directed Typing rules | 30 |
| 5.3 | Type Inference and Algorithm \mathcal{M} | 33 |
| 6 | QuB Extention and Datatypes | 37 |
| 6.1 | Kind System | 37 |
| 6.2 | Pairs and Sums in QuB | 39 |
| 6.2.1 | Multiplicative Pair Type | 40 |

| | | |
|----------|--|-----------|
| 6.2.2 | Additive Pair Type | 40 |
| 6.2.3 | Sum Type | 41 |
| 6.3 | Generic Type Constructors | 41 |
| 7 | Conclusion and Future Work | 44 |
| A | Derivations for Products and Sums | 48 |
| A.1 | Derivable Typing Rules For Product Types (Additive and Multiplicative Pairs) . | 48 |
| A.1.1 | Additive Pairs | 48 |
| A.1.2 | Multiplicative Pairs | 49 |
| A.2 | Derivable Typing rules for Sum Types | 51 |
| B | Proofs | 55 |

List of Figures

| | | |
|------|--|----|
| 2.1 | Hindley-Milner Type and Expression Language | 3 |
| 2.2 | Typing Rules for HM Type System | 4 |
| 2.3 | Grammar for Intuitionistic Logic | 5 |
| 2.4 | Logic rules: Implication fragment of second order Intuitionistic Propositional Logic | 5 |
| 2.5 | Algorithm \mathcal{M} for HM type system | 6 |
| 2.6 | Grammar for Intuitionistic Linear Logic | 7 |
| 2.7 | Intuitionistic Linear Logic Rules | 8 |
| 2.8 | Qualified Types and Expression Language | 9 |
| 2.9 | Typing Rules for Qualified Types | 10 |
| 2.10 | Un as a Typeclass | 11 |
| 2.11 | Grammar for Logic of BI | 12 |
| 2.12 | Rules for Logic of BI | 12 |
| 2.13 | $\alpha\lambda$ -Calculus Types and Terms | 13 |
| 2.14 | Typing Rules for $\alpha\lambda$ -Calculus | 13 |
| 2.15 | Multiplicative Argument used Twice in $\alpha\lambda$ -calculus | 14 |
| 3.1 | File Handling Functions | 16 |
| 3.2 | Reading from a file in Haskell | 16 |
| 3.3 | Reading from a File and Closing it Twice | 16 |
| 3.4 | Goto Fail bug | 17 |
| 3.5 | Reading from a File and Not Closing File Handle | 17 |
| 3.6 | Reading from a File in Haskell and Not Closing It (Desugared) | 18 |
| 3.7 | Reading from a File in Haskell and Closing Twice (Desugared) | 18 |

| | | |
|------|---|----|
| 3.8 | File Handling and Bind Function Type Signatures in QuB | 18 |
| 3.9 | Closing file twice in QuB | 19 |
| 3.10 | Haskell's MonadError Typeclass | 20 |
| 3.11 | Handling Errors in Haskell | 20 |
| 3.12 | Exception Handling in QuB | 21 |
| 4.1 | Types QuB | 22 |
| 4.2 | Entailment Rules | 23 |
| 4.3 | Bunch in BI | 24 |
| 4.4 | Sharing Graph in QuB | 24 |
| 4.5 | Typing Context | 25 |
| 4.6 | Auxiliary Functions on Type Assignments | 26 |
| 4.7 | Type Assignment Operations | 26 |
| 4.8 | Term Language | 26 |
| 5.1 | Structural Typing Rules | 28 |
| 5.2 | Connective Typing Rules | 29 |
| 5.3 | Sharing graph and typing context for $\lambda^*c.\lambda^*x.\lambda \rightarrow y.cx$ | 30 |
| 5.4 | Entailment Rules for Base cases | 30 |
| 5.5 | Syntax Directed Typing Rules | 32 |
| 5.6 | Auxiliary Functions | 34 |
| 5.7 | Type Inference Algorithm \mathcal{M} | 36 |
| 6.1 | Extended QuB Types and Kinds | 38 |
| 6.2 | Constructor Application Rule | 38 |
| 6.3 | Kind Preserving Unification of Type Constructors | 39 |
| 6.4 | Extended QuB Language Syntax | 39 |
| 6.5 | Multiplicative Pair | 40 |
| 6.6 | Derivable Typing Rules for Multiplicative Pair | 40 |

| | | |
|------|--|----|
| 6.7 | Additive Pair | 41 |
| 6.8 | Derivable Typing Rules for Additive Pair | 41 |
| 6.9 | Sum Type | 42 |
| 6.10 | Derivable Typing Rules for Sum Type | 42 |
| 6.11 | User Defined Datatypes | 42 |
| 6.12 | Shared Pairs | 43 |
| 6.13 | Separating Pair | 43 |

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 to detect errors. Runtime errors caused due to not closing an open file handle or closing the file handle more than once 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 achieved mainstream success due to practical limitations. We develop a type system based on the logic of bunched implications and implement a type inference algorithm for the language. We later extend to allow users to define their own datatypes, expressing sharing and separation between fields.

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 gives a direction to future work and conclusion.

The contributions made in the thesis are as follows:

- 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 other that do not.

- *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 illustrating how QuB is more expressive than standard Hindley-Milner type system, tracking resources at compile time and to detect anomalies.

Chapter 2

Background Work

2.1 Hindley-Milner Type System and Type Inference Algorithm

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

| | | |
|-------------------------------------|----------------|---|
| | Types | $\tau ::= t \mid \iota \mid \tau \rightarrow \tau$ |
| | Typing Scheme | $\sigma ::= \tau \mid \forall t. \tau$ |
| | Typing Context | $\Gamma ::= \varepsilon \mid \Gamma, x : \sigma$ |
| $t, u, v \in \text{Type Variables}$ | Expressions | $M, N ::= x \mid \lambda x. M \mid MN \mid \text{let } x = M \text{ in } N$ |

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 either infer the types of all well-typed expres-

sions 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.

| | |
|--|--|
| $\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} [\text{VAR}]$ | $\frac{\Gamma \vdash M : \sigma \quad \Gamma_{x,x} : \sigma \vdash N : \tau}{\Gamma \vdash (\text{let } x = M \text{ in } N) : \tau} [\text{LET}]$ |
| $\frac{\Gamma \vdash M : \sigma \quad t \notin \text{fvs}(\Gamma)}{\Gamma \vdash M : \forall t. \sigma} [\forall \text{ I}]$ | $\frac{\Gamma \vdash M : \sigma \quad (\sigma' \sqsubseteq \sigma)}{\Gamma \vdash M : \sigma'} [\forall \text{ E}]$ |
| $\frac{\Gamma_{x,x} : \tau \vdash M : \tau'}{\Gamma \vdash \lambda x. M : \tau \rightarrow \tau'} [\rightarrow \text{ I}]$ | $\frac{\Gamma \vdash M : \tau \rightarrow \tau' \quad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \tau'} [\rightarrow \text{ E}]$ |

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. The [VAR] rule is the tautology; a simple lookup of variable x in context Γ for the type scheme. $[\rightarrow \text{I}]$ and $[\rightarrow \text{E}]$ type lambda terms and application respectively. Rules for parametric polymorphism are implicit in expression language. $[\forall \text{I}]$ rule generalizes the type scheme by adding universal quantifiers and $[\forall \text{E}]$ generates an instance of the type scheme by substituting free type variables. While generalizing types, the rule $[\forall \text{I}]$, has a side condition that ensures the new type variable introduced should not be free in the typing context. The $\text{fvs}(\Gamma)$ denotes free type variables in Γ . The [LET] rules allows implicit parametric polymorphism. For example, the expression, $g = \lambda f. (f \text{ True}, f \ 1)$ is ill-typed. The type unification will fail for f as the expression $f \text{ True}$ asserts $f : \text{Bool} \rightarrow u_1$ while the expression $f \ 1$ asserts $f : \text{Int} \rightarrow u_2$. However, there indeed exists a polymorphic type $\forall u_1. u_1 \rightarrow u_1$ that types f . The `let` construct makes this possible. The expression g can be defined as $g = \text{let } f = \lambda x. x \text{ in } (f \text{ True}, f \ 1)$. The type of the expression f is now computed as $f : \forall u_1. u_1 \rightarrow u_1$, using $[\rightarrow \text{I}]$ and $[\forall \text{I}]$ rules and g is assigned a type $(\text{Bool}, \text{Int})$.

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

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

Figure 2.3: Grammar for Intuitionistic Logic

logic is shown in Fig. 2.3 where $\Gamma \supset \Delta$ denotes implication, $A \wedge B$ denotes conjunction and $A \vee B$ denotes disjunction.

$$\begin{array}{c}
\frac{}{A \vdash A} [\text{Ax}] \qquad \frac{\Gamma \vdash B}{\Gamma, A \vdash B} [\text{WKN}] \qquad \frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B} [\text{CTR}] \\
\frac{\Gamma \vdash B \quad x \notin \Gamma}{\Gamma \vdash \forall x. B} [\forall I] \qquad \frac{\Gamma \vdash \forall x. B \quad \Gamma \vdash A}{B[x/A]} [\forall E] \\
\frac{\Gamma, A \vdash B}{\Gamma \vdash A \supset B} [\supset I] \qquad \frac{\Gamma \vdash A \supset B \quad \Gamma \vdash A}{\Gamma \vdash B} [\supset E] \\
\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} [\wedge I] \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} [\wedge E_1] \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} [\wedge E_2] \\
\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} [\vee I_1] \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} [\vee I_2] \qquad \frac{\Gamma \vdash A \supset C \quad \Gamma \vdash B \supset C \quad \Gamma \vdash A \vee B}{\Gamma \vdash C} [\vee E]
\end{array}$$

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

The rules of the logic system are shown in Fig. 2.4 in Gentzen style natural deduction where Γ , Δ and Φ are propositions. The [Ax] rule corresponds to [ID] 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 structure of **HM** type-system. The weakening rule states that we can add unrelated assumptions to derivations without affecting the proofs while contraction states that we can discard duplicate assumptions in our derivations and the proof will still hold. The $[\forall I]$ 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. $[\wedge I]$, $[\wedge E_1]$ and $[\wedge E_2]$ introduces and eliminates conjunction. Conjunction is equivalent to pairs or product types. $[\vee I_1]$, $[\vee I_2]$ and $[\vee E]$ introduces and eliminates 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, 2005).

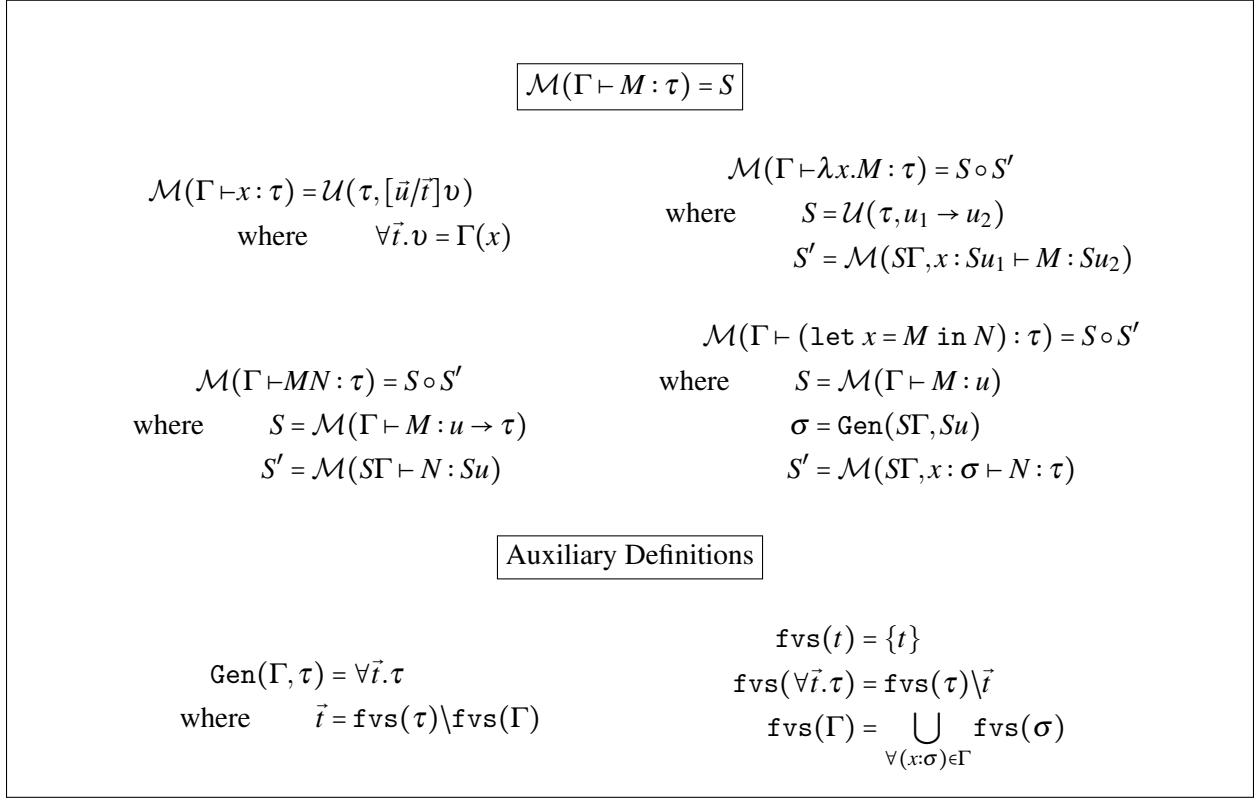


Figure 2.5: Algorithm \mathcal{M} for **HM** type system

The algorithm \mathcal{M} is given in figure Fig. 2.5. \mathcal{U} is the Robinson's unification algorithm that computes the most general unifier required to unify two types and $\text{Gen}(\Gamma, \tau)$ generalizes a type to a principle type scheme. All the type variables denoted by u are fresh and do not shadow the existing type variables in the context. S denotes substitutions that are combined using \circ operator can be applied on types or type schemes. 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 $\{t_1, t_2, \dots, t_n\}$ and $\vec{t} \setminus \vec{u}$ 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 formal terms, 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 in industry grade software. 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 used again 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](#). To

| Structural Rules | | |
|---|---|---|
| $\frac{}{[A] \vdash A} [\text{ID}_{[]}]$ | $\frac{}{\langle A \rangle \vdash A} [\text{ID}_{\langle \rangle}]$ | |
| $\frac{\Gamma, \Delta \vdash A}{\Delta, \Gamma \vdash A} [\text{EXCH}]$ | $\frac{\Gamma, [A], [A] \vdash B}{\Gamma, [A] \vdash B} [\text{CTRN}]$ | $\frac{\Gamma \vdash B}{\Gamma, [A] \vdash B} [\text{WKN}]$ |
| $\frac{[\Gamma] \vdash A}{[\Gamma] \vdash !A} [!I]$ | $\frac{\Gamma \vdash !A \quad \Delta, [A] \vdash B}{\Gamma, \Delta \vdash B} [!E]$ | |
| Connective Rules | | |
| $\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} [\multimap I]$ | $\frac{\Gamma \vdash A \multimap B \quad \Delta \vdash A}{\Gamma, \Delta \vdash B} [\multimap E]$ | |
| $\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} [\&I]$ | $\frac{\Gamma \vdash A \& B}{\Gamma \vdash A} [\&E_1]$ | $\frac{\Gamma \vdash A \& B}{\Gamma \vdash B} [\&E_2]$ |
| $\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} [\otimes I]$ | $\frac{\Gamma \vdash A \otimes B \quad \Delta, A, B \vdash C}{\Gamma, \Delta \vdash C} [\otimes E]$ | |
| $\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B} [\oplus I_1]$ | $\frac{\Delta \vdash B}{\Delta \vdash A \oplus B} [\oplus I_2]$ | $\frac{\Gamma \vdash A \oplus B \quad \Delta, A \vdash C \quad \Delta, B \vdash C}{\Gamma, \Delta \vdash C} [\oplus E]$ |

Figure 2.7: Intuitionistic Linear Logic Rules

relax linearity constraints, exponential modality $!$ is used, which signifies that an assumption can be duplicated or dropped without restriction. $!A$ can be thought of as “as many A ’s as needed”. The intuitionistic implication $A \multimap B$ can be encoded in linear logic by using the modality operator as $!A \multimap B$. We clearly see that this is a much 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 conjunction to have multiplicative and additive fragments but 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. Loliproc(Mazurak & 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. A modification of Damas-Milner algorithm \mathcal{W} to incorporate predicates ensures that type inference is sound and complete. Qualified types are the types that satisfy all the predicates for the term. Qualified types are powerful enough to express type classes with functional dependencies(Jones, 2000), record types(Gaster & Jones, 1996), sub-typing(Jones, 1994) and first class polymorphism(Jones, 1997).

| | |
|--|---|
| | Types $\tau ::= t \mid \iota \mid \tau \rightarrow \tau$ |
| $t, u, v \in \text{Type Variables}$ | Qualified Types $\rho ::= \tau \mid \pi \Rightarrow \rho$ |
| $\pi, \omega \in \text{Predicates}$ | Type Scheme $\sigma ::= \rho \mid \forall t. \sigma$ |
| $P, Q \in \text{Finite Predicate Set}$ | Typing Context $\Gamma ::= \varepsilon \mid \Gamma, x : \sigma$ |
| | Expressions $M, N ::= x : \sigma \mid \lambda x. M \mid MN$ $\mid \text{let } x = M \text{ in } N$ |

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 $P \Vdash \pi$ asserts that the predicate π can be inferred from the predicates in P . The typing rules in **HM** type system are slightly modified and 2 new rules are added as shown in Fig. 2.9. $[\Rightarrow I]$ and $[\Rightarrow E]$ 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 $[\Rightarrow I]$ and $[\Rightarrow E]$.

| | |
|---|---|
| $\frac{x : \sigma \in \Gamma}{P \mid \Gamma \vdash x : \sigma} [\text{VAR}]$ | $\frac{P \mid \Gamma \vdash M : \sigma \quad Q \mid \Gamma_x, x : \sigma \vdash N : \tau}{P, Q \mid \Gamma \vdash (\text{let } x = M \text{ in } N) : \tau} [\text{LET}]$ |
| $\frac{P \mid \Gamma \vdash M : \sigma \quad t \notin \text{fvs}(\Gamma) \cup \text{fvs}(P)}{P \mid \Gamma \vdash M : \forall t. \sigma} [\forall I]$ | $\frac{P \mid \Gamma \vdash M : \forall t. \sigma}{P \mid \Gamma \vdash M : [\tau/t]\sigma} [\forall E]$ |
| $\frac{P \mid \Gamma_x, x : \tau \vdash M : \tau'}{P \mid \Gamma \vdash \lambda x. M : \tau \rightarrow \tau'} [\rightarrow I]$ | $\frac{P \mid \Gamma \vdash M : \tau \rightarrow \tau' \quad P \mid \Gamma \vdash N : \tau}{P \mid \Gamma \vdash MN : \tau'} [\rightarrow E]$ |
| $\frac{P, \pi \mid \Gamma \vdash M : \rho}{P \mid \Gamma \vdash M : \pi \Rightarrow \rho} [\Rightarrow I]$ | $\frac{P \mid \Gamma \vdash M : \pi \Rightarrow \rho \quad P \Vdash \pi}{P \mid \Gamma \vdash M : \rho} [\Rightarrow E]$ |

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 $\tau \geq \tau'$ will hold only if τ admits more structural rules than τ' or, in other words, if τ' is more restricting than τ . The predicate $\text{Un } \tau$ 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 $\text{Fun } \tau$ implies that the type τ is of a function type. The function may or may not capture resources in its closure and 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

```

1  class Un where
2      dup  :: t →! (t ⊗ t)
3      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 $(\tau \xrightarrow{f} v) \rightarrow \tau \xrightarrow{g} v$. This function would be well typed only if f is less restricting i.e. admits more structural rules than g , so to say $f \geq g$.

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

In intuitionistic logic, the context is considered as a list or a set. In the theory of **BI**, the context is treated as a tree. Contexts are called bunched and are syntactically combined using 2 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 & Pym \(1999\)](#) introduce 2 kinds of arrows and use them depending on the connectives used in the context. A multiplicative implication (\multimap) is used when the context is connected with a comma and an additive implication (\multimap) is used when the context is connected using semicolon. This gives rise to two introduction rules for implication:

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

Γ, A cannot under go weakening or contraction to duplicate or get rid of either A or Γ . This hints to a notion that multiplicative implication (\multimap) 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 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 fall back to the non-resource intuitionistic parts.

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

$$\Gamma, \Delta ::= \{\}_m \mid \{\}_A \mid \Gamma, A \mid \Gamma; A$$

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

| | | |
|---|--|--|
| Structural | | |
| $\frac{}{A \vdash A}$ [ID] | $\frac{\Gamma(\Delta) \vdash A}{\Gamma(\Delta; \Delta') \vdash A}$ [WKN] | $\frac{\Gamma(\Delta; \Delta) \vdash A}{\Gamma(\Delta) \vdash A}$ [CTR] |
| Multiplicative Connectives | | |
| $\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}$ [\multimap I] | $\frac{\Gamma \vdash A \multimap B \quad \Delta \vdash A}{\Gamma, \Delta \vdash B}$ [\multimap E] | |
| $\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}$ [\otimes I] | $\frac{\Gamma(A, B) \vdash C \quad \Delta \vdash A \otimes B}{\Gamma(\Delta) \vdash C}$ [\otimes E] | |
| Additive Connectives | | |
| $\frac{\Gamma; A \vdash B}{\Gamma \vdash A \multimap\!\!\multimap B}$ [$\multimap\!\!\multimap$ I] | $\frac{\Gamma \vdash A \multimap\!\!\multimap B \quad \Delta \vdash A}{\Gamma; \Delta \vdash B}$ [$\multimap\!\!\multimap$ E] | |
| $\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma; \Delta \vdash A \& B}$ [&I] | $\frac{\Gamma(A; B) \vdash C \quad \Delta \vdash A \& B}{\Gamma(\Delta) \vdash C}$ [&E] | |
| $\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}$ [\oplus I ₁] | $\frac{\Delta \vdash B}{\Delta \vdash A \oplus B}$ [\oplus I ₂] | $\frac{\Gamma \vdash A \oplus B \quad \Delta(A) \vdash C \quad \Delta(B) \vdash C}{\Gamma(\Delta) \vdash C}$ [\oplus E] |

Figure 2.12: Rules 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 (Chapter 4 for more details).

$\alpha\lambda$ -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. \multimap : Functions do not share resources with their arguments
2. \multimap : Functions 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.

| | |
|-------------------------------------|---|
| $t, u, v \in \text{Type Variables}$ | |
| Context | $\Gamma, \Delta ::= \{\}_m \mid \{\}_a \mid x : \tau \mid \Gamma, \Delta \mid \Gamma; \Delta$ |
| Types | $\tau ::= t \mid \iota \mid \tau \multimap \tau \mid \tau \multimap \tau$ |
| Expressions | $M, N ::= x \mid \lambda x. M \mid \alpha x. M \mid MN$ |

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

| | | |
|---|---|--|
| $\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} [\text{VAR}]$ | $\frac{\Gamma; \Gamma \vdash M : \tau}{\Gamma \vdash M : \tau} [\text{CTRN}]$ | $\frac{\Gamma \vdash M : \tau}{\Gamma; \Delta \vdash M : \tau} [\text{WKN}]$ |
| $\frac{\Gamma_x, x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda x. M : \tau \multimap \tau'} [\multimap\text{I}]$ | | $\frac{\Gamma \vdash M : \tau \multimap \tau' \quad \Delta \vdash N : \tau}{\Gamma, \Delta \vdash MN : \tau'} [\multimap\text{E}]$ |
| $\frac{\Gamma_x, x : \tau \vdash M : \tau'}{\Gamma \vdash \alpha x. M : \tau \multimap \tau'} [\multimap\text{I}]$ | | $\frac{\Gamma \vdash M : \tau \multimap \tau' \quad \Delta \vdash N : \tau}{\Gamma; \Delta \vdash MN : \tau'} [\multimap\text{E}]$ |

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

The logic of **BI** patches up the awkward asymmetry experienced in linear logic. The multiplicative conjunction \otimes has a right adjoint counterpart as \multimap while additive conjunction $\&$ has a

right adjoint counterpart \multimap . In other words, $A \otimes B \vdash C$ iff $A \vdash B \multimap C$ and $A \& B \vdash C$ iff $A \vdash B \multimap C$. This relieves us from introducing modality into the system.

Due to the rules of $\alpha\lambda$ -calculus $f : \tau \multimap \tau'; x : \tau \vdash fx : \tau'$, as f needs an argument that does not share any resources with its context. The term $\lambda x. \alpha f. fxx : \tau \multimap (\tau \multimap \tau \multimap \tau') \multimap \tau'$ 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 though the argument is separate from the function, it may be used twice. In linear logic this expression will be ill-typed.

$$\begin{array}{c}
\frac{f : \tau \multimap \tau \multimap \tau' \vdash f : \tau \multimap \tau \multimap \tau'}{f : \tau \multimap \tau \multimap \tau'; x : \tau \vdash fx : \tau \multimap \tau'} [\text{VAR}] \quad \frac{x : \tau \vdash x : \tau}{x : \tau \vdash x : \tau} [\text{VAR}] \\
\frac{f : \tau \multimap \tau \multimap \tau'; x : \tau \vdash fx : \tau \multimap \tau'}{x : \tau; f : \tau \multimap \tau \multimap \tau'; x : \tau \vdash fxx : \tau'} [\multimap E] \quad \frac{x : \tau \vdash x : \tau}{x : \tau \vdash x : \tau} [\text{VAR}] \\
\frac{x : \tau; f : \tau \multimap \tau \multimap \tau'; x : \tau \vdash fxx : \tau'}{x : \tau; f : \tau \multimap \tau \multimap \tau' \vdash fxx : \tau'} [\text{CTRN}] \\
\frac{x : \tau; f : \tau \multimap \tau \multimap \tau' \vdash fxx : \tau'}{x : \tau \vdash \lambda x. fxx : (\tau \multimap \tau \multimap \tau') \multimap \tau'} [\multimap I] \\
\frac{x : \tau \vdash \lambda x. fxx : (\tau \multimap \tau \multimap \tau') \multimap \tau'}{\vdash \lambda x. \alpha f. fxx : \tau \multimap (\tau \multimap \tau \multimap \tau') \multimap \tau'} [\multimap I]
\end{array}$$

Figure 2.15: Multiplicative Argument used Twice in $\alpha\lambda$ -calculus

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

Chapter 3

Programming in QuB

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

3.1 File Handles

In modern programming languages resources, such as files, are treated as normal variables. It is the programmer's responsibility to check that that files are not closed twice and 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 accidentally close the file handle that may have been reused in the background by another thread. When another thread tries to write on this


```
1  openFile :: FilePath -> IO FileHandle
2  closeFile :: FileHandle -> IO ()
3  readLine :: FileHandle -> IO (String, FileHandle)
4  writeFile :: String -> FileHandle -> IO ((), FileHandle)
```

Figure 3.1: File Handling Functions

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

Figure 3.2: Reading from a file in Haskell

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

```
1  do f  <- openFile "sample.txt"
2      (s, f) <- readLine f
3      () <- closeFile f
4      () <- closeFile f
```

Figure 3.3: Reading from a File and Closing it Twice

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

Another example of incorrect way of using file handle is by not closing the file handle after using it shown in Fig. 3.5. In a short lived process 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 file handles. The whole process would crash with an error that it cannot

```

1  ...
2  if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
3      goto fail;
4  if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
5      goto fail;
6      goto fail;
7  if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
8      goto fail;
9
10 err = sslRawVerify(...);
11 ...

```

Figure 3.4: Goto Fail bug

open any more file handles. Abnormal exit from the process would interfere in the write process and the operating system would close the file handle without waiting for the buffer to be completely written on the file system causing loss of information.

```

1  do f <- openFile "sample.txt"
2      (s, f) <- readLine f
3      return s

```

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

We take a deeper dive into this problem by inspecting the desugared version of the “do” notation. Both the programs would be translated into bind ($>>=$) operations. Recall that type signature of bind function is given as $(>>=) :: (\text{Monad } m) \Rightarrow m \ t \rightarrow (t \rightarrow m \ u) \rightarrow m \ u$. 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 (\Rightarrow) or separated (\rightarrow^*). In QuB program code, we will use \rightarrow^*

```

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

```

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

```

1  (>>=_1) (openFile "sample.txt") (\f ->
2      (>>=_2) (readFile f) (\(s, f) ->
3          (>>=_3) (closeFile f) (\_ -> closeFile f)))

```

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

to mean \rightarrow and \rightarrow to mean \Rightarrow . 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.

```

1  openFile :: FilePath -*> IO FileHandle
2  closeFile :: FileHandle -*> IO ()
3  readLine :: FileHandle -*> IO (String, FileHandle)
4  writeFile :: String -*> FileHandle -*> IO ((), FileHandle)
5
6  (>>=) :: (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 handling 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 $\text{IO } () \text{ -*> } (() \text{ -*> } \text{IO } ()) \text{ -\&> } \text{IO } ()$ as both the arguments share the file variable f . This would be a type error as the bind operation should have a type signature of $\text{IO } () \text{ -*> } (() \text{ -*> } \text{IO } ()) \text{ -*> } \text{IO } ()$. This mismatch in the

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

```

1  (>>=1) (openFile "sample.txt") (\f ->
2      (>>=2) (readFile f) (\(s, f) ->
3          (>>=3) (close f) (\_ -> closeFile f)))
4
5
6  (>>=1) :: IO FileHandle -*> (FileHandle -*> IO ()) -*> IO ()
7  (>>=1) (openFile "sample.txt" ReadMode) (\f -> ...)
8
9  (>>=2) :: IO FileHandle
10         -*> (FileHandle -*> IO (String, Filehandle))
11         -*> IO (String, FileHandle)}
12  (>>=2) (readLine f) (\(s,f) -> ... )
13
14  (>>=3) :: IO () -*> (() -*> IO ()) -&> IO ()
15  (>>=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 unrestricted 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 tagged as an unrestricted type by the QuB type checker. This is a violation of our assumption that resources cannot be of the unrestricted type. Thus the program would not type check due to mismatch of the file handle type to be inferred as unrestricted.

3.2 Exception handling

We expand on the file handling scenario and consider the code that can throw runtime exceptions. The motivation to do so lies in the fact that memory leaks are caused because of runtime exceptions where the part of code that is responsible to clean up resources. In Haskell, error handling 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.

```

1 class Monad m => MonadError e m | m -> e where
2   throwError :: e -> m a
3   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.

```

1 do f <- openFile "sample.txt"
2   ((s, f) <- readLine f
3   let c = caps s
4   () <- closeFile f
5   return $ Right c) `catchError` (\_ ->
6                                   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.

```
1  openFile :: FilePath -> IO FileHandle
2  closeFile :: FileHandle -> IO ()
3  readFile :: FileHandle -> IOF (String, FileHandle)
4  writeFile :: String -> FileHandle -> IOF ((), FileHandle)
5
6  throw :: Exception -> IO a
7  catch :: IOF a -> (Exception -> IO a) -&> IO a
8
9  readFromFile :: FilePath -> IO (Either String String)
10 readFromFile fpath =
11   do fh <- openFile fpath
12     ((s, fh) <- readLine fh
13      let l = caps s
14      () <- closeFile fh
15      return $ Right l) `catch` (\e -> do closeFile fh
16                                         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.

| | |
|---|--|
| $t, u, v, \phi \in \text{Type Variables}$ | Types $\tau ::= t \mid \iota \mid \tau \rightarrow \tau$ |
| $P, Q \in \text{Finite Predicate Set}$ | where $\rightarrow \in \{ \overset{\cdot}{\rightarrow}, \rightarrow^*, \overset{\cdot}{\rightarrow}, \rightarrow \}$ |
| | Predicates $\pi, \omega ::= \text{Un } \tau \mid \text{SeFun } \tau \mid \text{ShFun } \tau \mid \tau \geq \tau'$ |
| | Qualified Types $\rho ::= \tau \mid \pi \Rightarrow \rho$ |
| | Type schemes $\sigma ::= \rho \mid \forall t. \sigma$ |

Figure 4.1: Types QuB

The type language consists of type variables (t, u, v), built-in types such as integers, booleans (ι), and four binary type constructors the sharing arrow (\rightarrow) and the separating arrow (\rightarrow^*) and unrestricted version of both the type constructors ($\overset{\cdot}{\rightarrow}, \overset{\cdot}{\rightarrow}$). The sharing arrow would mean that the function shares resources with its argument and the separating arrow would mean that the function does not share resources with its arguments. We would write both the arrows in an infix notation.

The predicate system enhances the expressivity of the type system. Following the same route taken in Quill (Morris, 2016) we use the predicate $\text{Un } \tau$ to denote that the type τ is unrestricted. Unrestricted types are the ones that do not have any resources or whose resources can be duplicated or deleted easily. Built-in types such as **Int**, **Char** are considered unrestricted types. We write

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

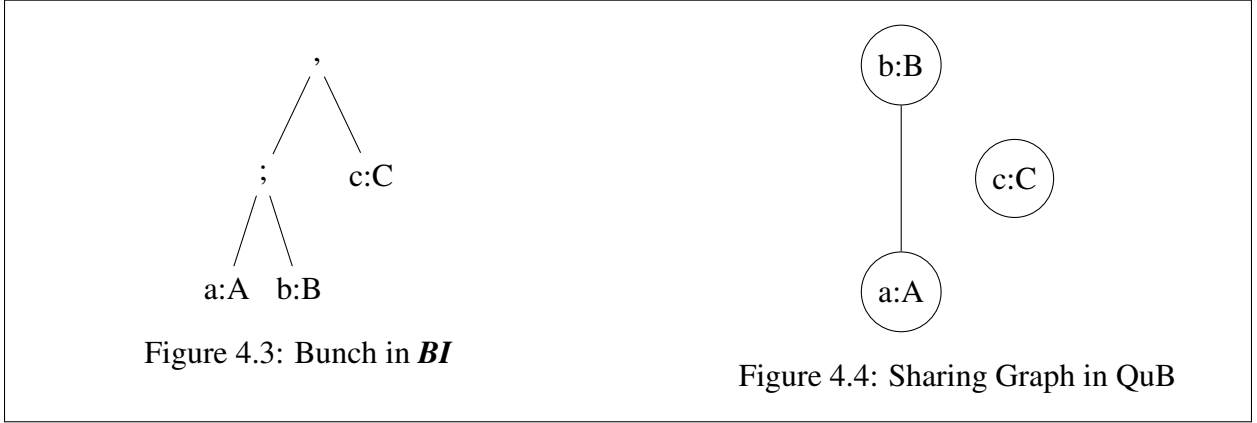
To keep the current system simple we have not included kinds. They are added into this system as a language extension to enable users to define custom types using type constructors. We describe this extension in Chapter 6.

| | | | |
|--|--|---|---|
| $\frac{\pi \in P}{P \Rightarrow \pi}$ | $\frac{\bigwedge \pi \in Q P \Rightarrow \pi}{P \Rightarrow Q}$ | $\frac{}{P \Rightarrow \text{Un}(\tau \dot{\rightarrow} \tau')}$ | $\frac{}{P \Rightarrow \text{Un}(\tau \dot{\rightarrow} \tau')}$ |
| $\frac{}{P \Rightarrow \tau \geq (v \dot{\rightarrow} v')}$ | $\frac{}{P \Rightarrow \tau \geq (v \dot{\rightarrow} v')}$ | $\frac{P \Rightarrow \text{Un} \tau}{P \Rightarrow \tau \geq (v \dot{\rightarrow} v')}$ | $\frac{P \Rightarrow \text{Un} \tau}{P \Rightarrow \tau \geq (v \dot{\rightarrow} v')}$ |
| $\frac{\tau = \dot{\rightarrow} \vee \tau = \dot{\rightarrow}}{P \Rightarrow \text{SeFun} \tau}$ | $\frac{\tau = \dot{\rightarrow} \vee \tau = \dot{\rightarrow}}{P \Rightarrow \text{ShFun} \tau}$ | $\frac{P \Rightarrow \tau \geq \phi t}{P \Rightarrow \tau \geq \phi}$ | $\frac{t \text{ fresh}}{P \Rightarrow \tau \geq \phi}$ |

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 $\Gamma(\Delta)$. Two bunches are equivalent ($\Gamma \equiv \Delta$) if they can be transformed into another by renaming identifiers. The bunches have a restriction that no identifier appears more than once. 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 can undergo contraction and weakening. Exchange rule is admissible in both the connec-

tives. This distinction enables a special treatment for resources in within the language. By adding resources with a comma constructor in the typing environment, the type system will not dispose it 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 our type system, we generalize the tree approach into a graph where each node represents variables or resources and the edges between the nodes represent sharing between them. The example in Fig. 4.3 can be represented as what we would call a sharing graph shown in Fig. 4.4. A graph structure, in general, can represent a binary tree structure and its associated operations. They represent more complex structures than trees, thus will provide more flexibility in accepting well typed terms in our language making it more expressive.

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

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

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

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

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 $x^{\bar{y}} : \tau$ for short. We would write $\Gamma, x^{\bar{y}} : \tau$ to mean $\Gamma \sqcup \{x^{\bar{y}} : \tau\}$. We can now formally define the typing context or environment as shown in Fig. 4.5.

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

Figure 4.5: Typing Context

We define a few auxiliary functions on the type assignments. $\text{Vars}(\Gamma)$ is the set of all the term variables in Γ . $\text{Shared}(\Gamma)$ computes the set of all the term variables that are in sharing with each other. $\text{Used}(\Gamma)$ computes the union of all the term variables in the type assignment and the term variables shared by each of those. We define two partial operators on type assignments as shown in Fig. 4.7. The mapping function $(\Gamma^{\vec{a} \mapsto \vec{b}})$ extends the sharing relation between the terms. In the sharing graph perspective it would mean adding edges between the nodes.

Two type assignments are said to be in disjoint union (\otimes) if either of the type assignments used

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

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

Figure 4.6: Auxiliary Functions on Type Assignments

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

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

Figure 4.7: Type Assignment Operations

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

Figure 4.8: Term Language

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

4.1 Conventions and Notations

We give some standard conventions and notations below that will be used throughout this thesis.

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

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

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

$$\text{fvs}(\sigma) = (\text{fvs}(\tau) \cup \text{fvs}(Q)) \setminus \vec{t}$$

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

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

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

Chapter 5

Type System and Type Inference

In this chapter we describe the type system using the types and terms defined in previous chapter. We describe in QuB's type system §5.1 and then describe a syntax directed type system in §5.2 and give a type inference algorithm \mathcal{M} in §5.3.

5.1 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.

| | | | |
|--|--|--|--|
| $\frac{P \mid \Gamma \otimes \Delta \otimes \Delta \vdash M : \sigma \quad P \vdash \Delta \text{ un}}{P \mid \Gamma \otimes \Delta \vdash M : \sigma} \text{ [CTR-UN]}$ | | $\frac{P \mid \Gamma \oplus \Delta \oplus \Delta \vdash M : \sigma}{P \mid \Gamma \oplus \Delta \vdash M : \sigma} \text{ [CTR-SH]}$ | |
| $\frac{P \mid \Gamma \vdash M : \sigma \quad P \vdash \Delta \text{ un}}{P \mid \Gamma \otimes \Delta \vdash M : \sigma} \text{ [WKN-UN]}$ | | $\frac{P \mid \Gamma \vdash M : \sigma}{P \mid \Gamma \oplus \Delta \vdash M : \sigma} \text{ [WKN-SH]}$ | |
| $\frac{}{P \mid x^{\vec{y}} : \sigma \vdash x : \sigma} \text{ [ID]}$ | | | |

Figure 5.1: Structural Typing Rules

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

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

$$\begin{array}{c}
\frac{P \mid \Gamma \vdash M : \sigma \quad P' \mid \Gamma'_x \sqcup x : \sigma \vdash N : \tau}{P \cup P' \mid \Gamma \sqcup \Gamma' \vdash (\text{let } x = M \text{ in } N) : \tau} [\text{LET}] \\
\\
\frac{P \mid \Gamma \vdash M : \sigma \quad t \notin \text{fvs}(\Gamma) \cup \text{fvs}(P)}{P \mid \Gamma \vdash M : \forall t. \sigma} [\forall \text{ I}] \qquad \frac{P \mid \Gamma \vdash M : \forall t. \sigma}{P \mid \Gamma \vdash M : [\tau \setminus t] \sigma} [\forall \text{ E}] \\
\\
\frac{P, \pi \mid \Gamma \vdash M : \rho}{P \mid \Gamma \vdash M : \pi \Rightarrow \rho} [\Rightarrow \text{ I}] \qquad \frac{P \mid \Gamma \vdash M : \pi \Rightarrow \rho \quad P \vdash \pi}{P \mid \Gamma \vdash M : \rho} [\Rightarrow \text{ E}] \\
\\
\frac{P \Rightarrow \text{ShFun } \phi \quad P \vdash \Gamma \geq \phi \quad P \mid \Gamma[\emptyset \mapsto \{x\}], x^{\text{Vars}(\Gamma)} : \tau \vdash M : \tau'}{P \mid \Gamma \vdash \lambda^{\rightarrow} x. M : \phi \tau \tau'} [\rightarrow \text{ I}] \qquad \frac{P \Rightarrow \text{ShFun } \phi \quad P \mid \Gamma \vdash M : \phi \tau \tau' \quad P \mid \Gamma' \vdash N : \tau}{P \mid \Gamma \oplus \Gamma' \vdash MN : \tau'} [\rightarrow \text{ E}] \\
\\
\frac{P \Rightarrow \text{SeFun } \phi \quad P \vdash \Gamma \geq \phi \quad P \mid \Gamma, x^{\emptyset} : \tau \vdash M : \tau'}{P \mid \Gamma \vdash \lambda^{*} x. M : \phi \tau \tau'} [* \text{ I}] \qquad \frac{P \Rightarrow \text{SeFun } \phi \quad P \mid \Gamma \vdash M : \phi \tau \tau' \quad P \mid \Gamma' \vdash N : \tau}{P \mid \Gamma \oplus \Gamma' \vdash MN : \tau'} [* \text{ E}]
\end{array}$$

Figure 5.2: Connective Typing Rules

The [LET] rule is used to enable implicit parametric polymorphism as usual. The rules of $[\rightarrow \text{I}]$ and $[\rightarrow \text{E}]$ describes the abstraction over shared and separating resources respectively, while $[\rightarrow \text{I}]$ and $[\rightarrow \text{E}]$ is the application rule for shared and separating resources respectively. $[\Rightarrow \text{I}]$ and $[\Rightarrow \text{E}]$ are the rules for qualified types that would add constraints on the type being computed. $[\forall \text{I}]$ introduces polymorphism and $[\forall \text{E}]$ eliminates it. The λ abstractions $\lambda^{*} x. M$ and $\lambda^{\rightarrow} x. M$ have a function type $\text{ShFun } \phi$ or $\text{SeFun } \phi$ only if it is more restricting than its environment. This is specified in the judgments $\cdot \geq \cdot$. To avoid name shadowing, we would assume that the binders introduce fresh names. In case of a λ^{\rightarrow} , the binder variable is added to the sharing information of all terms present in the type assignment and in case of λ^{*} the binder variable is skipped from adding it to the sharing information to imply separation of resources. For example, consider the term $\lambda^{*} c. \lambda^{*} x. \lambda^{\rightarrow} y. c x$. The sharing graph and type assignment is shown in Fig. 5.3.

The rules given in Fig. 5.4 are convenience rules for base cases that compute predicate con-

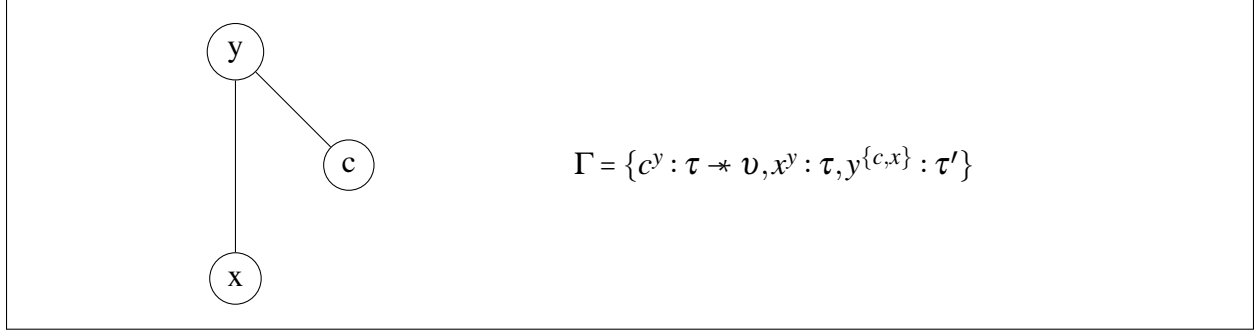


Figure 5.3: Sharing graph and typing context for $\lambda^*c.\lambda^*x.\lambda^*y.cx$

straints for types within a context.

| | |
|---|--|
| $P \vdash \cdot \text{un}$ | |
| $\frac{P \Rightarrow \text{Un } \tau}{P \vdash \tau \text{ un}} [\text{Un-}\tau]$ | $\frac{P, \pi \vdash \rho \text{ un}}{P \vdash \pi \Rightarrow \rho \text{ un}} [\text{Un-}\rho]$ |
| $\frac{P, \text{Un } t \vdash \sigma \text{ un}}{P \vdash \forall t. \sigma \text{ un}} [\text{Un-}\sigma]$ | $\frac{\bigwedge_{x:\sigma \in \Gamma} P \vdash \sigma \text{ un}}{P \vdash \Gamma \text{ un}} [\text{Un-}\Gamma]$ |
| $P \vdash \cdot \geq \cdot$ | |
| $\frac{P \Rightarrow \tau \geq \phi}{P \vdash \tau \geq \phi} [\geq\text{-}\tau]$ | $\frac{P, \pi \vdash \rho \geq \phi}{P \vdash (\pi \Rightarrow \rho) \geq \phi} [\geq\text{-}\rho]$ |
| $\frac{P, \text{Un } t \vdash \sigma \geq \phi}{P \vdash (\forall t. \sigma) \geq \phi} [\geq\text{-}\sigma]$ | $\frac{\bigwedge_{x:\sigma \in \Gamma} P \vdash \sigma \geq \phi}{P \vdash \Gamma \geq \phi} [\geq\text{-}\Gamma]$ |

Figure 5.4: Entailment Rules for Base cases

5.2 Syntax Directed Typing rules

Ideally the typing rules and syntactic forms should have one-to-one correspondence. The type system explained in the §5.1 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 $\text{Gen}(\Gamma, \rho) = \forall (\text{fvs}(\rho) \setminus \text{fvs}(\Gamma)). \rho$.

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 $[\text{Var}^s]$, introduction of polymorphism $[\forall I]$ and qualified types $[\Rightarrow I]$ is done at `let` bindings $[\text{Let}^s]$. This collapses the rules $[\forall E]$, $[\Rightarrow E]$ and $[\text{ID}]$ in one rule $[\text{VAR}^s]$ where we use instantiation or specialization of type variables, and $[\forall I]$, $[\Rightarrow I]$ and $[\text{Let}]$ in one rule $[\text{Let}^s]$ where we use generalization of type variables. $[\rightarrow^* I^s]$ is used in occurrence of λ^* , and $[\rightarrow^s I^s]$ is used in occurrence of λ^{\rightarrow} . We would add the introduced abstraction variable, x , into the sharing context in case of $[\rightarrow^s I^s]$. We collapse the application rules $[\rightarrow^* E]$ and $[\rightarrow E]$ into one rule $[\text{App}^s]$ where we check for sharing of the used variables in both the expressions and then assign a predicate of `ShFun` or `SeFun` depending on whether the variables are shared or not. The `un` predicates are added to the types of the terms that are not used directly in the expression or which are not in sharing with the terms used.

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

$$\begin{array}{c}
\frac{P \vdash \Gamma_{\bar{y}} \text{ un} \quad (P \Rightarrow \tau) \sqsubseteq \sigma}{P \mid \Gamma, x^{\bar{y}} : \sigma \vdash^s x : \tau} [\text{VAR}^s] \\
\\
\frac{Q \mid (\Gamma'_x \oplus \Gamma''_x) \otimes \Delta \vdash^s M : v \quad P \vdash \Delta \text{ un} \quad P \mid (\Gamma_x \sqcup x^\emptyset : \sigma) \oplus \Gamma''_x \otimes \Delta \vdash^s N : \tau \quad \sigma = \text{Gen}(\{\Gamma' \oplus \Gamma'' \otimes \Delta\}, Q \Rightarrow v)}{P \mid (\Gamma \otimes \Gamma') \oplus \Gamma'' \otimes \Delta \vdash^s (\text{let } x = M \text{ in } N) : \tau} [\text{Let}^s] \\
\\
\frac{P \Rightarrow \text{SeFun } \phi \quad P \vdash \Gamma \geq \phi \quad P \mid \Gamma \otimes x^\emptyset : \tau \vdash^s M : v}{P \mid \Gamma \vdash^s \lambda^{*x}. M : \phi \tau v} [\rightarrow^s \text{I}] \quad \frac{P \Rightarrow \text{ShFun } \phi \quad P \vdash \Gamma \geq \phi \quad P \mid \Gamma[\emptyset \mapsto \{x\}] \oplus x^{\text{Vars}(\Gamma)} : \tau \vdash^s M : v}{P \mid \Gamma \vdash^s \lambda^{\rightarrow x}. M : \phi \tau v} [\rightarrow^s \text{I}] \\
\\
\frac{P \mid \Gamma \otimes \Delta \vdash^s M : \phi v \tau \quad P \mid \Gamma' \otimes \Delta \vdash^s N : v \quad P \vdash \Delta \text{ un} \quad (\Gamma \tilde{\otimes} \Gamma' \wedge (P \Rightarrow \text{ShFun } \phi)) \vee (\Gamma \tilde{\otimes} \Gamma' \wedge (P \Rightarrow \text{SeFun } \phi))}{P \mid \Gamma \sqcup \Gamma' \otimes \Delta \vdash^s MN : \tau} [\text{App}^s]
\end{array}$$

Figure 5.5: Syntax Directed Typing Rules

[App^s] do not clearly look as if they are syntax directed and in the cases where the type checker cannot directly infer if the resources used are separate or shared, the user would be expected to provide it using type annotations. $\Gamma \tilde{\otimes} \Delta$ 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, $\Gamma \tilde{\oplus} \Delta$ means that $\Gamma \oplus \Delta$ is well defined.

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

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

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

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

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

asserts that the typing judgement indeed exists.

5.3 Type Inference and Algorithm \mathcal{M}

We now describe the type inference algorithm based on the previously defined syntax directed type system. We use a variation of algorithm \mathcal{M} (Lee & Yi, 1998) for type inference. We address three independent concerns in the type inference algorithm. The first being treatment of polymorphism to be same as Hindley-Milner style. The second, we introduce Un predicates for types that are unrestricted. We track this with the help of carrying a collection of used variables throughout the algorithm which detects whether a variable is discarded or used multiple types. The third being accounting for sharing of the variables. The complete algorithm is outlined in Fig. 5.7. The input to the algorithm includes the term M whose type has to be inferred, τ is the expected type of the term, S is the current substitution and the sharing information Ψ . The output includes the set of new predicates P that are generated, the new set of substitutions S' , the used variables Σ , and the new sharing information Ψ' . The type variables u with a subscript denote fresh variables.

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

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 v 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.

$$\begin{aligned}
Leq(\phi, \Gamma) &= \bigcup_{(x:\sigma) \in \Gamma} \{P \mid P \vdash \sigma \geq \phi\} & Un(\Gamma) &= \bigcup_{(y:\sigma) \in \Gamma} \{P \mid P \vdash \sigma \text{ un}\} \\
Weaken(x, \sigma, \Sigma) &= \begin{cases} P & \text{if } x \notin \Sigma, P \vdash \sigma \text{ un} \\ \emptyset & \text{otherwise} \end{cases} & GenI(\Gamma, P \Rightarrow \tau) &= \\
& & & \forall (fvs(P) \cup fvs(\tau) \setminus fvs(\Gamma)). P \Rightarrow \tau \\
\mathcal{C}(\Gamma, \Psi, \Sigma) &= \bigcup_{x \in \text{dom}(\Gamma) \wedge x \in \Sigma} \Psi(x)
\end{aligned}$$

Figure 5.6: Auxiliary Functions

In next case of sharing function introduction rule $\lambda \rightarrow x.M$, as per the $[\rightarrow I]$ 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 $\text{ShFun } u_1$, where u_1 is a new type variable for the type of function argument x . The second assigning the function to be less restricting than the other variables in the typing assignment Γ . The third, assigning an unrestricted predicate to the binding variable x if it has not been used anywhere in the lambda body M , which is done by the *Weaken* function and the fourth being the predicates generated by recursively type checking the body of the lambda expression M . We create sharing links for x with all the variables within the typing context Γ in updated Ψ'' . The case of separating function $\lambda^* x.M$ is very similar to the previous case of $\lambda \rightarrow x.M$ 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 $\text{SeFun } u_1$ to denote this very separating relation instead of $\text{ShFun } u_1$

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 $\text{Un}(\Gamma|_{\Sigma \cap \Sigma'})$ 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 which would lead to infinite types. We then generalize the computed type to σ and then check the type of expression N by expanding the type assignment Γ with the variable x and type scheme σ .

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

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

The soundness theorem for algorithm \mathcal{M} ensures that the type inference algorithm infers 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 $\lambda^* f. \lambda^* x. fxx$. This term can be inferred to have two types by using algorithm \mathcal{M} .

- $\text{Un } A \mid \emptyset \vdash \lambda^* f. \lambda^* x. fxx : (A \multimap A \multimap B) \multimap A \multimap B$
- $\emptyset \mid \emptyset \vdash \lambda^* f. \lambda^* x. fxx : (A \multimap A \multimap B) \multimap A \multimap B.$

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

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

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

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

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

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

Figure 5.7: Type Inference Algorithm \mathcal{M}

Chapter 6

QuB Extention and Datatypes

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

6.1 Kind System

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

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

Type Variables $t, u, v \in \text{Type Variables}$
 Kinds $\kappa ::= \kappa \mid \kappa' \rightarrow \kappa$
 Types $\tau^\kappa ::= t^\kappa \mid T^\kappa \mid \tau^{\kappa' \rightarrow \kappa} \tau^{\kappa'}$
 Type Constructors $T^\kappa \in \mathcal{T}^\kappa$ where $\{\otimes, \&, \oplus, \dot{*}, \dot{-}, \dot{\rightarrow}, \dot{\Rightarrow}\} \subseteq \mathcal{T}^{* \rightarrow * \rightarrow *}$
 Predicates $\pi, \omega ::= \tau \mid \text{Un } \tau \mid \text{SeFun } \tau \mid \text{ShFun } \tau \mid \tau \geq \tau'$
 Qualified Types $\rho ::= \tau \mid \pi \Rightarrow \rho$
 Type schemes $\sigma ::= \rho \mid \forall t. \sigma$

Figure 6.1: Extended QuB Types and Kinds

$$\frac{\tau :: \kappa' \rightarrow \kappa \quad \tau' :: \kappa'}{\tau \tau' :: \kappa}$$

Figure 6.2: Constructor Application Rule

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

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

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

| | |
|---|---------------|
| $t \stackrel{id}{\sim}_{\kappa} t$ | ([ID-KVar]) |
| $T \stackrel{id}{\sim}_{\kappa} T$ | ([ID-KConst]) |
| $t \stackrel{[T/t]}{\sim}_{\kappa} T, t \notin \text{fvs}(T)$ | ([KVar]) |
| $T \stackrel{[T/t]}{\sim}_{\kappa} t, t \notin \text{fvs}(T)$ | ([KVar']) |
| $\frac{T \stackrel{S}{\sim}_{\kappa' \rightarrow \kappa} T' \quad SH \stackrel{S'}{\sim}_{\kappa'} SH'}{TT' \stackrel{SS'}{\sim}_{\kappa} HH'} \text{([KApply])}$ | |

Figure 6.3: Kind Preserving Unification of Type Constructors

| | |
|----------------|--|
| Term Variables | $x, y, z \in \text{Var}$ |
| Expressions | $M, N ::= x \mid \lambda \multimap x.M \mid \lambda \twoheadrightarrow x.M \mid MN \mid \text{let } x = M \text{ in } N$ $\mid \langle M, N \rangle \mid \text{let } \langle x, y \rangle = M \text{ in } N \mid \langle M; N \rangle \mid \text{fst } M \mid \text{snd } M$ $\mid \text{case } M \text{ of } \{ \text{inl } x \mapsto N; \text{inr } y \mapsto N' \} \mid \text{inl } x \mid \text{inr } y$ |

Figure 6.4: Extended QuB Language Syntax

6.2 Pairs and Sums in QuB

Introduction of two kinds of arrows in our type system leads to different flavors of pairs. This distinction cannot be made in intuitionistic logic as the structural rules allow re-use of propositions. But due to restrictions in weakening and contraction we obtain two kinds of pairs, additive and multiplicative. In this section we illustrate how the extended QuB can be used to introduce new types. We introduce syntax and type constructors for multiplicative pairs in §6.2.1 and the same for additive pairs in §6.2.2. We then introduce the syntax and type constructors for sum types in §6.2.3 and illustrate that they indeed work as expected.

6.2.1 Multiplicative Pair Type

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

$$\begin{aligned} \otimes &\in \mathcal{T}^{* \rightarrow * \rightarrow *} \\ \tau \otimes \tau' &= \tau \multimap \tau' \multimap (\tau \multimap \tau' \multimap v) \multimap v \\ (,) &= \lambda^{-*} x. \lambda^{-*} y. \lambda^{-*} f. fxy \end{aligned}$$

Figure 6.5: Multiplicative Pair

$$\begin{array}{c} \frac{P \mid \Gamma \vdash M : \tau \quad P \mid \Delta \vdash N : \tau'}{P \mid \Gamma \otimes \Delta \vdash \langle M, N \rangle : \tau \otimes \tau'} [\otimes I] \\ \frac{P \mid \Gamma \vdash M : \tau \otimes \tau' \quad P \mid \{x^{\{\bar{z} \subseteq \text{Vars}(\Gamma')\}} : \tau\} \otimes \{y^{\{\bar{z}' \subseteq \text{Vars}(\Gamma')\}} : \tau'\} \sqcup \Gamma'_{x,y} \vdash N : v}{P \mid \Gamma \sqcup \Gamma' \vdash (\text{let } \langle x, y \rangle = M \text{ in } N) : v} [\otimes E] \end{array}$$

Figure 6.6: Derivable Typing Rules for Multiplicative Pair

6.2.2 Additive Pair Type

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. It is worthwhile to note that we do not provide these projection functions for multiplicative pairs so as to restrict the programmer from implicitly discarding one of the components of the multiplicative pair.

$$\begin{aligned}
& \& \in \mathcal{T}^{* \rightarrow * \rightarrow *} \\
& \tau \& \tau' = \tau \multimap \tau' \rightarrow (\tau \multimap \tau' \rightarrow v) \rightarrow v \\
& (;) = \lambda^* x. \lambda^{\rightarrow} by. \lambda^{\rightarrow} f. fxy
\end{aligned}$$

Figure 6.7: Additive Pair

$$\begin{array}{c}
\frac{P \mid \Gamma \vdash M : \tau \quad P \mid \Delta \vdash N : \tau'}{P \mid \Gamma \oplus \Delta \vdash \langle M; N \rangle : \tau \& \tau'} [\& \text{I}] \\
\frac{P \mid \Gamma \vdash M : \tau \& \tau'}{P \mid \Gamma \vdash \text{fst } M : \tau} [\& \text{E}_1] \qquad \frac{P \mid \Gamma \vdash M : \tau \& \tau'}{P \mid \Gamma \vdash \text{snd } M : \tau'} [\& \text{E}_2]
\end{array}$$

Figure 6.8: Derivable Typing Rules for Additive Pair

6.2.3 Sum Type

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

6.3 Generic Type Constructors

To leverage the full power of introducing the sharing concept in a programming language, we would want to have users be able to define datatypes with sharing and separation of its constituent resources. We add two new typing rules as shown in Fig. 6.11 for handling generic datatypes

$$\begin{aligned}
& \oplus \in \mathcal{T}^{* \rightarrow * \rightarrow *} \\
& \tau \oplus \tau' = (\tau \rightarrow v) \rightarrow (\tau' \rightarrow v) \rightarrow v \\
& \text{inl} : \tau \multimap (\tau \oplus \tau') \\
& \text{inl} = \lambda^* x. \lambda^{\rightarrow} f. \lambda^{\rightarrow} g. f x \\
& \text{inr} : \tau' \multimap (\tau \oplus \tau') \\
& \text{inr} = \lambda^* y. \lambda^{\rightarrow} f. \lambda^{\rightarrow} g. g y
\end{aligned}$$

Figure 6.9: Sum Type

$$\begin{array}{c}
\frac{P \mid \Gamma \vdash M : \tau}{P \mid \Delta \vdash \text{inl } M : \tau \oplus \tau'} [\oplus I_l] \qquad \frac{P \mid \Gamma \vdash M : \tau'}{P \mid \Delta \vdash \text{inr } M : \tau \oplus \tau'} [\oplus I_r] \\
\frac{P \mid \Gamma \vdash M : \tau \oplus \tau' \quad P \mid \Gamma \oplus x : \tau \vdash N : v \quad P \mid \Gamma \oplus y : \tau' \vdash N' : v}{P \mid \Gamma \vdash \text{case } M \text{ of } \{\text{inl } x \mapsto N; \text{inr } y \mapsto N'\} : v} [\oplus E]
\end{array}$$

Figure 6.10: Derivable Typing Rules for Sum Type

using type constructors. In the current implementation, we only consider a specific case of type constructors where all the type variables involved are either all shared or all separate.

How do we write about type constructors?

$$\frac{P \mid \Gamma \oplus x : \tau^\kappa \vdash C : \tau^\kappa \rightarrow T}{P \mid \Gamma \vdash Cx : T} [\text{C-sh}] \qquad \frac{P \mid \Gamma \oplus x : \tau^\kappa \vdash C : \tau^\kappa \multimap T}{P \mid \Gamma \vdash Cx : T} [\text{C-se}]$$

Figure 6.11: User Defined Datatypes

We illustrate using examples how the type system works. The sharing pair has shared resources and using `fst` and `snd` works to get both the individual constituents as shown in Fig. 6.12 The `(!!)` denotes that `a` and `b` are in sharing. The `fstp` and `sndp` functions typecheck as expected.

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

```

data Pair' a b = ShP !! a b

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

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

```

Figure 6.12: Shared Pairs

```

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 6.13: Separating Pair

Chapter 7

Conclusion and Future Work

missing pieces here.

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

References

- Ahmed, A., Fluet, M., & Morrisett, G. (2007). L^3 : A linear language with locations. *Fundamenta Informaticae*, 77(4), 397–449.
- Atkey, R. (2004). A λ -calculus for resource separation. In J. Díaz, J. Karhumäki, A. Lepistö, & D. Sannella (Eds.), *Automata, Languages and Programming*, volume 3142 (pp. 158–170). Springer Berlin Heidelberg.
- Barendregt, H. (1991). Introduction to generalized type systems. *Journal of Functional Programming*, 1(2), 125–154.
- Bernardy, J.-P., Boespflug, M., Newton, R. R., Peyton Jones, S., & Spiwack, A. (2017). Linear haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2, 5:1–5:29.
- Collinson, M., Pym, D., & Robinson, E. (2005). On bunched polymorphism. In L. Ong (Ed.), *Computer Science Logic*, Lecture Notes in Computer Science (pp. 36–50). Springer, Berlin, Heidelberg.
- Damas, L. & Milner, R. (1982). Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82 (pp. 207–212).: ACM.
- Gaster, B. R. & 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., & Lafont, Y. (1989). *Proofs and Types*. Cambridge University Press.
- Jones, M. P. (1993). A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture, FPCA '93* (pp. 52–61).: ACM.
- Jones, M. P. (1994). A theory of qualified types. *Science of Computer Programming*, 22(3), 231 – 256.
- Jones, M. P. (1997). First-class polymorphism with type inference. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '97* (pp. 483–496).: ACM.
- Jones, M. P. (2000). Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming: Springer-Verlag LNCS 1782*.
- Lee, O. & Yi, K. (1998). Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4), 707–723.
- Liang, S., Hudak, P., & Jones, M. (1995). Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '95* (pp. 333–343).: ACM.
- Mazurak, K. & Zdancewic, S. (2010). Lollipop: 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* (pp. 39–50).: ACM.
- Mazurak, K., Zhao, J., & 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* (pp. 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. (2016). The best of both worlds: Linear functional programming without compromise. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016 (pp. 448–461).: ACM.
- O’Hearn, P. W. (1999). Resource interpretations, bunched implications and the alpha lambda-calculus. In *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications*, TLCA ’99 (pp. 258–279).: Springer-Verlag.
- O’Hearn, P. W. (2003). On bunched typing. *Journal of functional Programming*, 13(4), 747–796.
- O’Hearn, P. W. & Pym, D. J. (1999). The logic of bunched implications. *The Bulletin of Symbolic Logic*, 5(2), 215–244.
- Pierce, B. C., Ed. (2005). *Advanced Topics in 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 A. M. Borzyszkowski & S. Sokolowski (Eds.), *Mathematical Foundations of Computer Science 1993: 18th International Symposium, MFCS’93 Gdańsk, Poland, August 30–September 3, 1993 Proceedings* (pp. 185–210). Springer Berlin Heidelberg.

Appendix A

Derivations for Products and Sums

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

A.1 Derivable Typing Rules For Product Types (Additive 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.

$$\begin{array}{c}
 \frac{}{\emptyset \mid y^{xf} : \tau' \vdash y : \tau'} [\text{ID}] \quad \frac{\frac{}{\emptyset \mid x^{fy} : \tau \vdash x : \tau} [\text{ID}] \quad \frac{}{\emptyset \mid f^{xy} : \tau \multimap \tau' \multimap v \vdash f : \tau \multimap \tau' \multimap v} [\text{ID}]}{\emptyset \mid x^{xy} : \tau \oplus f^{xy} : \tau \multimap \tau' \multimap v \vdash fx : (\tau' \multimap v)} [\multimap E]} [\multimap E] \\
 \frac{\emptyset \mid y^{xf} : \tau' \oplus x^{yf} : \tau \oplus f^{xy} : \tau \multimap \tau' \multimap v \vdash fxy : v}{\emptyset \mid x^{yf} : \tau \oplus y^{xf} : \tau' \oplus f^{xy} : \tau \multimap \tau' \multimap v \vdash fxy : v} [\text{EXCH}] \\
 \frac{\emptyset \mid x^y : \tau \oplus y^x : \tau' \vdash \lambda \multimap f.fxy : (\tau \multimap \tau' \multimap v) \multimap v}{\emptyset \mid x^\emptyset : \tau \vdash \lambda \multimap y.\lambda \multimap f.fxy : \tau' \multimap (\tau \multimap \tau' \multimap v) \multimap v} [\multimap I] \\
 \frac{\emptyset \mid x^\emptyset : \tau \vdash \lambda \multimap y.\lambda \multimap f.fxy : \tau' \multimap (\tau \multimap \tau' \multimap v) \multimap v}{\emptyset \mid I \vdash \lambda \multimap x.\lambda \multimap y.\lambda \multimap f.fxy : \tau \multimap \tau' \multimap (\tau \multimap \tau' \multimap v) \multimap v} [\multimap I]
 \end{array}$$

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

$$\tau \& \tau' = (\tau \multimap \tau' \multimap v) \multimap v$$

We now define sharing (additive) pair constructor as:

$$\begin{aligned}
& ; : \tau \multimap \tau' \multimap (\tau \& \tau') \\
& ; = \lambda^* x. \lambda^{\multimap} y. \lambda^{\multimap} f. fxy
\end{aligned}$$

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

$$\begin{array}{c}
\frac{}{\emptyset \mid x^y : \tau \vdash x : \tau} \text{[ID]} \\
\frac{}{\emptyset \mid x^y : \tau \oplus y^x : \tau' \vdash x : \tau} \text{[WKN-SH]} \\
\frac{}{\emptyset \mid x^\emptyset : \tau \vdash \lambda^{\multimap} y. x : \tau' \multimap \tau} \text{[} \multimap \text{I]} \\
\frac{}{\emptyset \mid I \vdash \lambda^{\multimap} x. \lambda^{\multimap} y. x : \tau \multimap \tau' \multimap \tau} \text{[} \multimap \text{I]}
\end{array}$$

$$\begin{aligned}
& \text{fst} : \tau \multimap \tau' \multimap \tau \\
& \text{fst} = \lambda^{\multimap} x. \lambda^{\multimap} y. x
\end{aligned}$$

$$\begin{array}{c}
\frac{}{\emptyset \mid y^x : \tau' \vdash y : \tau'} \text{[ID]} \\
\frac{}{\emptyset \mid x^y : \tau \oplus y^x : \tau' \vdash y : \tau'} \text{[WKN-SH]} \\
\frac{}{\emptyset \mid x^\emptyset : \tau \vdash \lambda^{\multimap} y : \tau' \multimap \tau'} \text{[} \multimap \text{I]} \\
\frac{}{\emptyset \mid I \vdash \lambda^{\multimap} x. \lambda^{\multimap} y. y : \tau \multimap \tau' \multimap \tau'} \text{[} \multimap \text{I]}
\end{array}$$

$$\begin{aligned}
& \text{snd} : \tau \multimap \tau' \multimap \tau' \\
& \text{snd} = \lambda^{\multimap} x. \lambda^{\multimap} y. y
\end{aligned}$$

A.1.2 Multiplicative Pairs

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

$$\begin{array}{c}
\frac{}{\emptyset \mid y^\emptyset : \tau' \vdash y : \tau'} [\text{ID}] \quad \frac{\frac{}{\emptyset \mid x^\emptyset : \tau \vdash x : \tau} [\text{ID}] \quad \frac{}{\emptyset \mid f^\emptyset : \tau \multimap \tau' \multimap v \vdash f : \tau \multimap \tau' \multimap v} [\text{ID}]}{\emptyset \mid x^\emptyset : \tau \otimes f^\emptyset : \tau \multimap \tau' \multimap v \vdash fx : (\tau' \multimap v)} [\multimap \text{E}] \\
\frac{}{\emptyset \mid y^\emptyset : \tau' \otimes x^\emptyset : \tau \otimes f^\emptyset : \tau \multimap \tau' \multimap v \vdash fxy : v} [\text{EXCH}] \\
\frac{}{\emptyset \mid x^\emptyset : \tau \otimes y^\emptyset : \tau' \otimes f^\emptyset : \tau \multimap \tau' \multimap v \vdash fxy : v} [\multimap \text{I}] \\
\frac{}{\emptyset \mid x^\emptyset : \tau \vdash \lambda^* y. \lambda^* f. fxy : (\tau \multimap \tau' \multimap v) \multimap v} [\multimap \text{I}] \\
\frac{}{\emptyset \mid I \vdash \lambda^* x. \lambda^* y. \lambda^* f. fxy : \tau \multimap \tau' \multimap (\tau \multimap \tau' \multimap v) \multimap v} [\equiv]
\end{array}$$

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

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

We can now define separating (multiplicative) pair as:

$$\begin{aligned}
& , : \tau \multimap \tau' \multimap (\tau \otimes \tau') \\
& , = \lambda^* x. \lambda^* y. \lambda^* f. fxy
\end{aligned}$$

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 auxiliary definitions from above.

$$\begin{array}{c}
\frac{P \mid \Gamma \vdash M : \tau \quad P \mid \Delta \vdash N : \tau'}{P \mid \Gamma \oplus \Delta \vdash \langle M; N \rangle : \tau \& \tau'} [\& I] \\
\frac{P \mid \Gamma \vdash M : \tau \& \tau'}{P \mid \Gamma \vdash \text{fst } M : \tau} [\& E_1] \qquad \frac{P \mid \Gamma \vdash M : \tau \& \tau'}{P \mid \Gamma \vdash \text{snd } M : \tau'} [\& E_2] \\
\frac{P \mid \Gamma \vdash M : \tau \quad P \mid \Delta \vdash N : \tau'}{P \mid \Gamma \otimes \Delta \vdash \langle M, N \rangle : \tau \otimes \tau'} [\otimes I] \\
\frac{P \mid \Gamma \vdash M : \tau \otimes \tau' \quad P \mid \{x^{\{\vec{z} \subseteq \text{Vars}(\Gamma')\}} : \tau\} \otimes \{y^{\{\vec{z}' \subseteq \text{Vars}(\Gamma')\}} : \tau'\} \sqcup \Gamma'_{x,y} \vdash N : v}{P \mid \Gamma \sqcup \Gamma' \vdash (\text{let } \langle x, y \rangle = M \text{ in } N) : v} [\otimes E]
\end{array}$$

A.2 Derivable Typing rules for Sum Types

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

$$\begin{array}{c}
\frac{}{\emptyset \mid g^{cf} : (B \multimap E) \vdash g : (B \multimap E)} [\text{ID}] \quad \frac{\frac{}{\emptyset \mid f^{cg} : (A \multimap E) \vdash f : (A \multimap E)} [\text{ID}] \quad \frac{}{\emptyset \mid c^{fg} : ((A \multimap E) \multimap (B \multimap E) \multimap E) \vdash c : ((A \multimap E) \multimap (B \multimap E) \multimap E)} [\text{ID}]}{\emptyset \mid c^{fg} : ((A \multimap E) \multimap (B \multimap E) \multimap E) \oplus f^{cg} : (A \multimap E) \vdash cf : (B \multimap E) \multimap E} [\multimap E] \\
\frac{\frac{\frac{}{\emptyset \mid c^{fg} : ((A \multimap E) \multimap (B \multimap E) \multimap E) \oplus f^{cg} : (A \multimap E) \oplus g^{cf} : (B \multimap E) \vdash cf g : E} [\multimap I]}{\emptyset \mid c^f : ((A \multimap E) \multimap (B \multimap E) \multimap E) \oplus f^c : (A \multimap E) \vdash \lambda^{\multimap} g.cfg : (B \multimap E) \multimap E} [\multimap I]}{\frac{\frac{}{\emptyset \mid c^\emptyset : ((A \multimap E) \multimap (B \multimap E) \multimap E) \vdash \lambda^{\multimap} f.\lambda^{\multimap} g.cfg : (A \multimap E) \multimap (B \multimap E) \multimap E} [\multimap I]}{\emptyset \mid I \vdash \lambda^* c.\lambda^{\multimap} f.\lambda^{\multimap} g.cfg : ((A \multimap E) \multimap (B \multimap E) \multimap E) * (A \multimap E) \multimap (B \multimap E) \multimap E} [*I]}
\end{array}$$

We now define sum type to be

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

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

52

$$\begin{array}{c}
\frac{}{\emptyset \mid x^{fg} : A \vdash x : A} [\text{ID}] \quad \frac{}{\emptyset \mid f^{gx} : (A \multimap E) \vdash f : (A \multimap E)} [\text{ID}] \\
\frac{}{\emptyset \mid x^{fg} : A \oplus f^{gx} : (A \multimap E) \vdash fx : E} [\multimap E] \\
\frac{\frac{}{\emptyset \mid x^{fg} : A \oplus f^{gx} : (A \multimap E) \oplus g^{fx} : (B \multimap E) \vdash fx : E} [\text{WKN-UN}]}{\frac{}{\emptyset \mid x^f : A \oplus f^x : (A \multimap E) \vdash \lambda^{\multimap} g.fx : (B \multimap E) \multimap E} [\multimap I]} [\multimap I] \\
\frac{}{\emptyset \mid x^\emptyset : A \vdash \lambda^{\multimap} f.\lambda^{\multimap} g.fx : (A \multimap E) \multimap (B \multimap E) \multimap E} [\multimap I] \\
\frac{}{\emptyset \mid I \vdash \lambda^* x.\lambda^{\multimap} f.\lambda^{\multimap} g.fx : A * (A \multimap E) \multimap (B \multimap E) \multimap E} [*I]
\end{array}$$

Left injection defined below as:

$$\text{inl} : A * A \oplus B$$

$$\text{inl} = \lambda^* x.\lambda^{\multimap} f.\lambda^{\multimap} g.fx$$

$$\begin{array}{c}
\frac{\overline{\emptyset \mid y^{fg} : B \vdash y : B} \text{ [ID]} \quad \frac{\overline{\emptyset \mid g^{yf} : (B \multimap E) \vdash g : (B \multimap E)} \text{ [ID]}}{[\multimap\text{I}]} \\
\frac{\overline{\emptyset \mid y^{fg} : B \oplus g^{yf} : (B \multimap E) \vdash gy : E}}{[\text{WKN-UN}]} \\
\frac{\overline{\emptyset \mid y^{fg} : B \oplus f^{yg} : (A \multimap E) \oplus g^{yf} : (B \multimap E) \vdash gy : E}}{[\multimap\text{I}]} \\
\frac{\overline{\emptyset \mid y^f : B \oplus f^y : (A \multimap E) \vdash \lambda^{\multimap} g.gy : (B \multimap E) \multimap E}}{[\multimap\text{I}]} \\
\frac{\overline{\emptyset \mid y^\emptyset : B \vdash \lambda^{\multimap} f.\lambda^{\multimap} g.gy : (A \multimap E) \multimap (B \multimap E) \multimap E}}{[\multimap\text{I}]} \\
\frac{\overline{\emptyset \mid I \vdash \lambda^{\multimap} y.\lambda^{\multimap} f.\lambda^{\multimap} g.gy : B \multimap (A \multimap E) \multimap (B \multimap E) \multimap E}}{[\multimap\text{I}]}
\end{array}$$

Right injection defined below as:

$$\begin{array}{c}
\text{inr} : B \multimap A \oplus B \\
\text{inr} = \lambda^{\multimap} y.\lambda^{\multimap} f.\lambda^{\multimap} g.gy
\end{array}$$

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

$$\text{case } c \text{ of } \{f;g\} = \lambda^*c.\lambda^*f.\lambda^*g.cfg$$

$$\frac{\frac{\frac{\emptyset \mid \text{inl} : A \multimap A \oplus B \vdash x : A}{\emptyset \mid I \vdash \text{inl } x : A \oplus B} [\oplus I_1] \quad \frac{\frac{\emptyset \mid \text{inr} : B \multimap A \oplus B \vdash y : B}{\emptyset \mid I \vdash \text{inr } y : A \oplus B} [\oplus I_2]}{\frac{\emptyset \mid \Gamma \vdash M : A \oplus B \quad \emptyset \mid \Gamma \oplus x : A \vdash N_1 : E \quad \emptyset \mid \Gamma \oplus y : B \vdash N_2 : E}{\emptyset \mid \Gamma \vdash \text{case } M \text{ of } \{\text{inl } x \mapsto N_1; \text{inr } y \mapsto N_2\} : E} [\oplus E]}$$

Appendix B

Proofs

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

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

- Case $[\text{VAR}^s]$. By induction hypothesis we have a derivation of $P \mid x : \sigma \vdash x : \sigma$ by $[\text{VAR}]$ 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 $[\text{WKN-SH}]$ or $[\text{WKN-UN}]$ respectively for all the bindings in Γ to construct a derivation of $P \mid \Gamma \sqcup \{x : \sigma\} \vdash x : \tau$.
- Case $[\rightarrow I^s]$. By induction hypothesis we have a derivation of $P \mid \Gamma \oplus x : \tau \vdash M : \tau'$. We apply $[\rightarrow I]$ and reuse the derivations for $\text{ShFun } \phi$ and $\Gamma \geq \phi$ to construct a derivation of $P \mid \Gamma \vdash \lambda \rightarrow x.M : \phi \tau \tau'$.
- Case $[\rightarrow I^s]$. By induction hypothesis we have a derivation of $P \mid \Gamma \oplus x : \tau \vdash M : \tau'$. Similar to previous case, we apply $[\rightarrow I]$ and reuse the derivations for $\text{SeFun } \phi$ and $\Gamma \geq \phi$ to construct a derivation of $P \mid \Gamma \vdash \lambda \rightarrow x.M : \phi \tau \tau'$.
- Case $[\text{App}^s]$. By induction hypothesis we have derivations of $P \mid \Gamma \oplus \Delta \vdash M : \phi \nu \tau$ and $P \mid \Gamma' \oplus \Delta \vdash N : \nu$ we check for $\text{Used}(\Gamma) = \text{Used}(\Gamma')$ and if it is true and apply $[\rightarrow E]$ or check for $\text{Used}(\Gamma) \# \text{Shared}(\Gamma') \wedge \text{Shared}(\Gamma') \# \text{Used}(\Gamma)$ and if it is true we apply $[\rightarrow E]$ to reuse derivations of $\text{ShFun } \phi$ or $\text{SeFun } \phi$ respectively to construct the derivation of $P \mid (\Gamma \oplus \Gamma') \oplus \Delta \vdash MN : \tau$ or $P \mid (\Gamma \otimes \Gamma') \oplus \Delta \vdash MN : \tau$.

- Case [Let^s]. By induction hypothesis have a derivation of $P \mid \Gamma \otimes \Delta \vdash M : \tau$ and $P \mid \Gamma' \sqcup x : \tau \otimes \Delta \vdash N : \tau$. Applying [∀I] and [⇒I] on the first hypothesis we derive $\emptyset \mid \Gamma \otimes \Delta \vdash M : \sigma$. Now by applying the [LET] rule and reusing $P \vdash \Delta$ un we construct the derivation of $P \mid \Gamma \sqcup \Gamma' \otimes \Delta \vdash \text{let } x = M \text{ in } N : \tau$. ■

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

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

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

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

- Case [VAR]. M is an expression x and $y \notin \text{fvs}(x)$ also, $P \vdash \sigma$ un for $(y : \sigma)$, thus using [VAR^s] we can obtain the required derivation.
- Case others. [LET], [→I], [→E], [→I], [→E] are all straightforward. ■

Lemma B.2. *If $P \mid \Gamma \vdash M : \tau$, $\forall_{y \in \text{dom}(\Delta)} y$ 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 $y \notin \text{fvs}(x)$ also, $\Gamma \oplus \{y : \sigma\}$ thus, by using [WKN-SH] and [VAR^s] the required derivation is obtained.
- Case others. [LET], [→I], [→E], [→I], [→E] are all straightforward. ■

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

Proof. Let $\sigma = \forall \vec{t}. Q \Rightarrow v$. By definition, there are some \vec{u} , such that $\tau' = [\vec{u}/\vec{t}]v$ 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 $\text{dom}(\Gamma) = \text{dom}(\Gamma')$ and $\forall_{x \in \text{dom}(\Gamma)} \Gamma(x) \sqsubseteq \Gamma'(x)$ 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 [∀E] we get the required derivation.
- Case others. All other cases, [Let^s], [→I^s], [→I^s], and [App^s] are trivial. Essentially use [∀E] rule to get the correct instance of the type to get the required derivation. ■

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

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

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

as \vec{u} are fresh i.e. $\vec{u} \notin \text{dom}(\Gamma)$ 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 $P \mid \Gamma \oplus \Delta \vdash M : \tau$ such that $(P \mid \sigma) \sqsubseteq (\emptyset \mid \text{Gen}(\Gamma \oplus \Delta, Q \Rightarrow \tau))$
- 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 $(P \mid \sigma) \sqsubseteq (\emptyset \mid \text{Gen}(\Gamma \oplus \Delta, Q \Rightarrow \tau))$
- Case [WKN-SH]. This follows directly from Lemma B.2 and the induction hypothesis.
- Case [→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 [→I^s] we have the required derivation.

- Case $[\rightarrow E]$. 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 $\Gamma \oplus \Gamma'$ and apply $[\text{App}^s]$ to obtain the required derivation.
- Case $[\rightarrow I]$. By induction hypothesis and Lemma B.3, we have a derivation of $Q \mid \Gamma \vdash^s M : v$ also, $Q \mid \Gamma \geq \phi$ thus by using $[\rightarrow I^s]$ we obtain the required derivation.
- Case $[\rightarrow E]$. 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 $[\text{App}^s]$ to get the required derivation.
- Case $[\Rightarrow E]$. By induction hypothesis we have $Q \mid \Gamma \vdash^s M : \tau$ such that, letting $\sigma = \text{Gen}(\Gamma, Q \Rightarrow \tau)$, $(P, \pi \mid \rho) \sqsubseteq \sigma$. 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 $\sigma = \text{Gen}(\Gamma, Q \Rightarrow \tau)$, $(P \mid \pi \Rightarrow \rho) \sqsubseteq \sigma$. 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 $\sigma = \text{Gen}(\Gamma, Q \Rightarrow \tau)$, $(P \mid \pi \Rightarrow \rho) \sqsubseteq \sigma$.
- Case $[\forall E]$. By the induction hypothesis we have $Q \mid \Gamma \vdash^s M : \tau$ such that, letting $\sigma = \text{Gen}(\Gamma, Q \Rightarrow \tau)$, $(P \mid \pi \Rightarrow \rho) \sqsubseteq \sigma$, As $(P \mid [\tau/t]\sigma) \sqsubseteq (P \mid \sigma)$, $(P \mid [\tau/t]\sigma) \sqsubseteq \sigma$ (because $\sigma = (\emptyset \mid \sigma)$).
- Case $[\text{Let}]$. By induction hypothesis, we have

$$Q \mid \Gamma \oplus \Delta \vdash^s M : v$$

and

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

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

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

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

We go over a few intermediate results 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 $P \mid \Gamma \vdash^s M : \tau$, $\forall_{y \in \text{dom}(\Delta)} y$ is not free in M and $P \vdash^s \Delta$ un then $P \mid \Gamma \oplus \Delta \vdash M : \tau$.*

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

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

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

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

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

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

Lemma B.9 (Constraint Simplification). *Supppose 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 $\text{Un } t \in P \wedge \text{ShFun } t \in P$ then $St = \dashrightarrow$
- If $\text{Un } t \in P \wedge \text{SeFun } t \in P$ then $St = \dashvstar$
- If $\text{Un } t \notin P \wedge \text{ShFun } t \in P$ then $St = \Rightarrow$
- If $\text{Un } t \notin P \wedge \text{SeFun } t \in P$ then $St = \Rightarrow*$

Lemma B.10. If $P \mid \Gamma \vdash^s M : \tau$ and $X = fvs(P) \setminus (fvs(\Gamma) \cup fvs(\tau))$, $P' \Rightarrow P$ 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 $[\rightarrow I^s]$, $[\rightarrow* I^s]$ or $[\text{APP}^s]$, the substitution $S\phi$ is a suitable type for further use. ■

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

- Case x . We have

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

where $\vec{y} \subseteq \text{dom}(\Gamma)$ 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 $[\text{VAR}^s]$ to construct $[\vec{u}/\vec{t}]P \mid \{x : \forall \vec{t}. P \Rightarrow \tau\} \vdash^s x : [\vec{u}/\vec{t}]\tau$

- Case $\lambda \rightarrow_x M$. 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

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

Let $Q = \{\text{ShFun } u_1\} \cup \text{Leq}(u_1, \Gamma|_{\Sigma}) \cup \text{Weaken}(x, u_2, \Sigma) \cup P$ and let $\Sigma' = \Sigma \setminus \{x\}$. 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 $[\rightarrow I^s]$ 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

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

Let $Q = \{\text{SeFun } u_1\} \cup \text{Leq}(u_1, \Gamma|_{\Sigma}) \cup \text{Weaken}(x, u_2, \Sigma) \cup P$ and let $\Sigma' = \Sigma \setminus \{x\}$. 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 $[\rightarrow I^s]$ 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 $Q = P \cup P' \cup \text{SeFun } u_1$ or if we can prove $\mathcal{C}(S\Gamma, \Sigma, \Psi) = \mathcal{C}((S \circ R)\Gamma, \Sigma', \Psi')$ then we let $Q = P \cup P' \cup \text{ShFun } u_1$ (else the inference algorithm fails). Let $\Delta = \text{dom}(\Gamma) \setminus \mathcal{C}(\Gamma, \Sigma \cup \Sigma', \Psi \circ \Psi')$ Now by Lemma B.5 and Lemma B.8 and the induction hypothesis we get

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

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

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

- Case `let $x = M$ 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 $\sigma = \text{GenI}(R\Gamma; R(P \Rightarrow u_1))$. Let T improve $\text{fvs}(P)$ ($\text{fvs}(\Gamma) \cup \text{fvs}(R\tau)$) 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

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

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

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

■