FMCt
Making A Type System for the Functional Machine Calculus

Vlad Posmangiu Luchian

MSc in Computer Science
The University of Bath
2021

FMCt
Making A Type System for the Functional Machine Calculus

Submitted by: Vlad Posmangiu Luchian

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Acknowledgements

I would like to thank Willem Heijltjes for providing guidance throughout the writing of this dissertation, and Guy McCusker for allowing me to audit the *Logic and semantics of programming languages* module.

Furthermore I would like to thank Carmen, Alex, and Rebecca for their support and understanding - even when faced with yet another Masters Degree.

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Context

The Functional Machine Calculus, put forward by Heijltjes (2021) (referred to as the *FMC*) is a novel, lambda-calculus like model of higher-order computation integrating computational effects while maintaining confluence. In the unpublished paper the author puts forward ideas about a potential type system for the language, which the following thesis explores. The thesis discusses an implementation and strategy for the type system, together with a strategy for an inference algorithm.

Motivation As highlighted by Heeren, Hage and Swierstra (2002), type systems are an indispensable tool present in contemporary higher-order, polymorphic languages. Type systems enable the detection of ill-typed expressions at compile-time, making a major contribution towards the popularity of languages like *Haskell* and *ML*. By enabling *language safety* (as defined by Cardelli (1996)) and adding an ergonomic dimension to the use of a language, type systems are a major contributor to the (fearless) use of models of computation.

Chapter 2

Background

The following chapter introduces a selection of the literature, and research undertaken building towards the type system proposal.

2.1 Expressing Computation

2.1.1 Early History

As pointed out by Barendregt Henk (1994), the search for a *universal language* can be summarised by Leibniz's ideal:

1. *Create a "universal language" in which all possible terms can be stated.*
2. *Find a decision method to solve all the problems stated in the universal language.*

Historically, a formal notation for abstraction in computation can be traced back to Giuseppe Peano (1889). In his book on the axioms or principles for arithmetic, he uses the notation $\alpha[x]$ to represent the term α as a function depending on the variable x. Peano proposes the notation $\phi = \alpha[x]$ and the equation $\phi x' = \alpha[x]x'$, with the right hand side representing the result of substituting x' for x in ϕ. However, this notation, did not gain momentum, with Peano proposing new notations including $\alpha\bar{x}$ and $\alpha|x$.

In subsequent years further systems have been proposed by mathematicians in their writings, with notable mentions Gottlob Frege (1891), Burali-Forti (1894) and, Russell and Whitehead (1913). However, as made clear by Cardone and Hindley (2016), none of the mentioned authors offer a formal definition for the operations of substitution and conversion.

In the 1920's, Moses Ilyich Schönfinkel (1924) sets the foundations of combinatorics, a mathematical study interested in removing the need for quantified variables in mathematical logic. Following Schönfinkel's writing, J. von Neumann (1925) publishes his PhD thesis on the axiomatisation of set theory, and in Curry (1930) further develops the concept of a combinator.In his thesis, Curry includes the first formal definition of conversion, and a finite set of axioms form which he proved the admissibility of rule:

$$(\zeta) \text{ if } Ux = Vx \text{ and } x \text{ does not occur in } UV, \text{ then } U = V.$$

2.1.2 Untyped Lambda Calculus

Published by Church (1932), the Lambda Calculus (λ calculus) is a type-free logic with unrestricted quantification, and no law of excluded middle. As pointed out in Cardone and Hindley (2016) the motivation behind its development was Church's search for a foundation for logic more natural than Russell's type theory or Zermelo's set theory, that would not contain free variables. Shortly following the publishing, a contradiction was found in the paper and was subsequently revised by Church (1933).

Formally, the λ calculus is a mathematical system of expressing computation based on a minimal expression (or term) based language. The expressions are built up from inductively defined terms which can take the form of an abstraction, an application or a variable. Written in the Backus-Naur form (BNF) these are:

Definition 2.1.1.

$$M, N ::= \quad x \;\mid\; \lambda x.M \;\mid\; MN,$$

where x is a variable, M, N are terms, $\lambda x.M$ is an abstraction, and MN is an application. In a non-λ calculus context the abstraction $\lambda x.M$ can be though of as an anonymous function $f_{(x)} \to M$ while the application MN can be though as replacing the x of the anonymous function of N, (i.e. $f_{(N)}$ where $f_{(x)} \to M$).

Computation in the λ calculus is described by the following rules:

Definition 2.1.2. α **conversion** is the method of replacing bound variables with *fresh* (unused) ones. Through the use of α conversion, λ calculus establishes a natural equivalence between terms called α **equivalence** noted as $=_\alpha$. Two terms are said to be α equivalent if they are of the same form.

$$\lambda x.\lambda y.xyz =_\alpha \lambda y.\lambda x.yxz =_\alpha \lambda m.\lambda n.mnz$$

Through the use of α conversion *variable capture* is avoided - substituting term with α equivalent ones, terms can avoid wrongfully binding free variables.

Definition 2.1.3. β **reduction** is the equivalent of computation in the λ Calculus. Terms of the form $(\lambda x.M)N$ (called *redexes*) are β reduced through the substitution of all bound occurrences of variable x in M with N. The operation of substitution is noted as:

$$(\lambda x.M)N \to_\beta M[N/x],$$

Where: $\to_\beta$ reads as *one β reduction step* and, $M[N/x]$ reads as *replace all bound occurrences of x with N in M*. Note that the substitution is done avoiding variable capture.

$$\underbrace{(\lambda x.\lambda y.(\lambda x.x)yx)a}\, b \to_\beta \underbrace{(\lambda y.(\lambda x.x)ya)b} \to_\beta \underbrace{(\lambda x.x)b}\, a \to_\beta ba$$

Definition 2.1.4. η **reduction** is the dropping of an abstraction over a function, resulting in an α equivalent term to the term we started from.

$\lambda x.\, fx \to_\eta f \mid (x \notin FV(f))$, where $FV\ (f)$ is the set containing all the *free variables* of f.

β reduction is *confluent* when working up to α conversion - meaning that terms can be reduce in any order up to α equivalence, without affecting the final outcome.

Definition 2.1.5. Having discussed β reduction, we can now define the β **normal form** of a λ term. which is reached when a term can no longer be reduced. Thus, the normalisation of a λ term can be expressed as:

$$T_1 \to_\beta T_2 \to_\beta \ ... \to_\beta T_n$$

T_n is the normal form of T_1 if $\nexists T_{n+1}$ such that $T_n \to_\beta T_{n+1}$

Based on their property to normalise, we can now define two classes of terms:

Definition 2.1.6. Weakly normalising terms have a terminating sequence, that after a finite amount of steps can be reached. Thus $\forall\ w$, with w a λ term with w weakly normalising, $\exists w'$ such that $w \to_{\beta *} w'$ and w' is in β normal form.

Definition 2.1.7. The second class is that of **strongly normalising terms**, which do not have an infinite sequence of terms the initial term β reduces to. Strongly normalising terms also have the property of weak normalising terms of having a normal form. Thus we can write:

a term M is strongly normalising if:
$\nexists$ an infinite sequence of terms $M_1, M_2, \ldots$ such that
$M \to_\beta M_1 \to_\beta M_2$ ... .

Not all terms are normalising in the untyped lambda calculus, leading to its non-deterministic property. Fixed point combinators are a good example of a term without normal form.

Definition 2.1.8. At the end of Church (1933) introduced the idea of the integers as λ terms:

$$1 \equiv \lambda x.\,\lambda y.\,xy, \qquad n =_\beta \lambda x.\,\lambda y.\,\underbrace{x(\ldots(xy)\ldots)}_{n\ times}, \qquad Succ = \lambda x.\,\lambda y.\,\lambda z.\,y(xyz).$$

Definition 2.1.9. Similarly he introduced notions for Church booleans:

$$\lambda x.\,\lambda y.\,x = True \qquad \lambda x.\,\lambda y.\,y = False$$

Definition 2.1.10. And for the Church **if** operator:

$$\lambda b.\,\lambda x.\,\lambda y.\,bxy = if$$

Example 2.1.11. We can see how applying **if** to **True** works with an example. Let M, P be two λ terms in:

$$\textbf{if True}\ M\ P = (\lambda bxy.\,bxy)(\lambda xy.\,x)MP \to_\beta (\lambda xy.\,(\lambda xy.\,x)xy)MP \to_{\beta^*} (\lambda xy.\,x)MP \to_{\beta^*} M$$

Definition 2.1.12. **Fixed points** of the form $Yf = f.\,Yf$ use recursion to achieving looping in the λ calculus.

Church proved that the λ calculus is a universal model of computation, with capabilities equivalent to that of a Turing Machine.

Definition 2.1.13. As pointed out in Barendregt (1984) although β reduction is non-deterministic - λ calculus maintains **confluence**. As illustrated in Figure 2.1.13, this property of the λ calculus means that the order in which the terms are β reduced does not make a difference to the outcome of the calculation. Although not an intuitively evident fact, the property was proven in Church and Rosser (1936) and is also known as the Church-Röser Theorem.

Given A, A', B, C are all λ terms:

$$(A \to_{\beta^*} B) \wedge (A \to_{\beta^*} C) \Rightarrow \exists\, A', (B \to_{\beta^*} A') \wedge (C \to_{\beta^*} A').$$

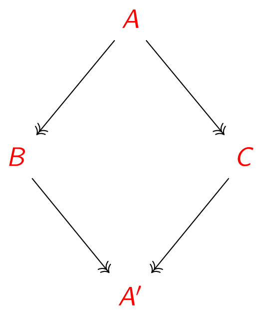


Figure 2.1: Confluence of Lambda Calculus

Theorem 2.1.14. *Confluence of the λ calculus is lost with the addition of side-effects.*

Example 2.1.15. One example is the addition of **rnd**, a function that returns a random church numeral.

$$rnd \to_\beta N_x,$$
$$x \in \mathbb{N},\ N_x \in \text{Church Numeral}.$$

Looking at the application of **rnd** to the combinator $\lambda x.\, xx$, which depending on evaluation strategy (as defined at 2.1.17, 2.1.19) will reduce to two different normal forms - which are then impossible to further reduce to a common term.

$$\begin{aligned}(\lambda x.\, xx)\ \mathbf{rnd} \to_{\beta}^{CBV} (\lambda x.\, xx)\ N_x \quad &\to_{\beta} N_x N_x \to_{\beta} N_{xx}, (1)\\ \to_{\beta}^{CBN} \mathbf{rnd}\ \mathbf{rnd} \quad &\to_{\beta} N_x N_y \to_{\beta} N_{xy}. (2)\end{aligned}$$

Given that there is a probability that $x \neq y$ then $(1) \neq (2)$. By definition (1) and (2) are in their normal form, thus $\nexists\ \lambda$ term N' such that $(1) \to_{\beta^*} N' \wedge (2) \to_{\beta^*} N'$. We can conclude that the confluence of the term calculus has been lost, and furthermore that different reduction strategies yielded different normal forms. $(q.e.d)$

2.1.3 Evaluation

Reduction Strategy

As we have seen, different reduction strategies are confluent as long as we do not introduce side-effects into the λ calculus. Let us define these strategies by first introducing the two major categories and then examples.

Definition 2.1.16. Based on the strictness of the strategy are two main types of evaluation strategies:

1. **Strict evaluation** evaluates all of the redexes inside a term, before the body of the function is evaluated,
2. **Non-Strict evaluation** does not.

Definition 2.1.17. Call-by-name, CBN, lazy-evaluation or **Normal Order** is a non strict evaluation strategy that does not evaluate the terms inside an abstraction before it is applied. The order in which redexes get evaluated is: outer most, left most first.

Theorem 2.1.18. *CBN always produces a normal form if the term has one.*

A drawback of **CBN** is that due to its laziness it can amass *large* (a lot of memory/space needed for a computer, or effort for a human) terms with many nested redexes, that can become *"hard"* to manipulate.

Definition 2.1.19. Call-by-value, CBV or **eager evaluation** is a strict evaluation strategy that evaluates all the redexes of a term, before a term gets to be applied. The evaluation strategy evaluates inner-most left most. As it stands, it is the most commonly used evaluation strategy in current programming languages.

Definition 2.1.20. Weak head normal form (WHNF) is a non strict evaluation strategy that reduces a term to its data constructor (or lambda abstraction) - allowing for sub-expressions to remain unevaluated inside the term.

Example 2.1.21.

(1)	$\lambda x.\, (\lambda x.\, xx)x$	is in WHNF,
(2)	$E(\lambda x.\, (\lambda x.\, xx))$	is not in WHNF.

2.1.4 Computational Effects

At this point the contextualisation of computational effects in both semantics and computers must be introduced.

Definition 2.1.22. A **computational effect** is the result of a computation - i.e. reduction of a *redex*, application.

Definition 2.1.23. A **computational side-effect** as defined by Plotkin and Power (2004) is the result of a computation that is done on *"the side"* while polymorphically computing something else, or in the case of a command nothing at all.

Example 2.1.24. Examples of computational side-effects:

1. Reading from an input, a file, a keyboard, or a mouse,

2. Reading, writing or allocating memory,
3. Controlling program continuation with transfers (*(go to)* and long jumps).

Definition 2.1.25. In Gerald Jay Sussman (1997), a **computer program** is defined as being made from three components: Modularity, Objects, State. If modularity is representative of the logical manner in which a program is divided, and objects are the entities we find within each module, then state is the information stored by each of the objects inside the modules.

Definition 2.1.26. Computational effects in a computer program can then be defined as actions that modify the state of the objects - implicitly modifying the state of the module. Seen in a reverse order, objects and modules are just a higher abstraction and categorisation of state.

Definition 2.1.27. Referential transparency, of an expression or term is the relative property of a term of not introducing side-effects at its evaluation. (as seen at Example 2.1.15).

Working With Effects

Monad

In working with effects, Moggi (1989) proposes that category theory should be taken as the general theory of functions and develop categorical of computations based on monads. This methodology comes from the belief that *"category theory comes, logically before the λ calculus"* - leading to Moggi considering a categorical semantics of computation rather than trying to work on the $\beta\eta$ - conversion rules.

Following this line of thought comes the proposals of using *monads* to allow a pure functional program to maintain referential transparency when modelling functions with computational effects.

Definition 2.1.28. A **monad** is an abstraction(based on a category theory *endofunctor*), that provides two methods: a *bind* operation that wraps the argument within the monad, and a *compose* method that allows it to compose function with monadic output.

With the use of a monads, comes a way to encapsulate information and work with it in a sequential manner (example: *IO Monad* of Haskell) with the information inside the wrapper of the monad itself. In Plotkin and Power (2002) the authors then model these effects algebraically, focussing on the notions of global and local state, giving good examples of proofs of the soundness of these monads of interest.

Thunk

Definition 2.1.29. Thunk is a subroutine used to introduce additional computation into another subroutine. As defined in Ingerman (1961), it can be thought of as a primitive type of *closure*. Thunks are the main method used by (*most*) CBV programming languages to achieve CBN like operational effects.

Definition 2.1.30. A **closure** is a technique of binding a name to a term within a locally defined, or *scoped* context (also known as scope). This allows for terms to be provided with their own environment - for example allowing a function to access captured variables through the use of the locally copied values.

As pointed out in Chapter 1, the FMC proposes a new strategy to close the above gap between the CBV and CBN which is, integrated as part of its syntax.

Semantic Styles

Definition 2.1.31. In order to discuss formally about computational effects, a definition of how the terms are evaluated must be formulated (*language semnantics*). In Moggi (1989) mentions three ways of formalising the semantics, also discussed in Pierce (2002):

1. **Operational semantics** specify the behaviour of the language by defining a *simple abstract machine* for it. A state of the machine is representative of a term in the language, and the transition of the machine from is given by a *transition function* that gives the machine either the next state or a halting state. If given two or more machines for the same language, the resulting terms are equal starting from an equal term, then we have a proof of equality.
2. **Denotational semantics** offers a higher level view of the language, where it gives a mathematical structure to the intended model - for example defining mathematical structures numbers, or

functions. Then equivalence is established by trying to establish equivalence between terms. This approach allows one to argue more about the *domain specifics* and logic of the language, rather than the low-level implementation details of *operational semantics*.

3. **Axiomatic semantics** gives a class of possible models for the language, by taking the axioms of the models and forming a language out of them. Then, the equivalence is denoted by proving that two terms denote the same object in all the possible models.

Historical context

Pierce (2002) makes it clear that historically (70's, 80's) *Operational semantics* was considered a weaker style of giving the semantics of a language than its counterparts. But with the work of Plotkin (1981), Kahn (1987) and Milner, operational semantics is currently being used with equal consideration, and furthermore it has been proven to avoid many of the mathematical and logic complication that the latter two introduce in the description of a language's semantics.

Furthermore, in Streicher and Reus (1998) discusses how deriving an abstract machine based on the Krivine machine for a language based on its continuation semantics, and giving its denotational semantics is useful in defining the behaviour of a functional programming language. A relevant thing pointed by Moggi (1989) is that the equivalence of a program $A \to B$ with a total function from A to B in denotational vs operational semantics is difficult to prove - since this identification can wipe out the effects (behaviour like non-termination, non-determinism or side-effects) inherent in a program.

2.2 Type Systems

Logistics of Type Systems

Why Types?

An aspect that algorithms, programs, proofs and any system has in common is that with increasing complexity and length, comes an increase in the challenge of keeping errors and mistakes out of the objects themselves. Types and type checking offer an effective, static strategy to check the consistency and well formulation of the above mentioned objects. Pierce (2002) and Cardelli (1996) provide an extensive discussion of why the study of types and type systems matter in the world of programming.

History of Types

As mentioned in Coquand (2018), the theory of types was introduced by Russell, in order to deal with contradictions he found in his account of set theory, and was published in 1903 in "Appendix B: The Doctrine of Types". The addition of types is a natural manner in which one can distinguish between kinds of objects in logical reasoning and computing. Types are an indicator that certain terms (formulas, functions or relations) can only be replaced with terms of an equivalent typing. (*The Lambda Calculus (Stanford Encyclopedia of Philosophy)* (n.d.)) For a time-capsule of the type systems development see Fig. 2.2.

Practical Type Systems Expectations

There are specific expectations of a type system from a practical point of view of a user, for them to be *fit for purpose*. Cardelli (1996) defines the expectations as being:

1. **Decidably verifiable** - there should exist an algorithm (*typechecker*) which can check that the terms are well typed;
2. **Transparent** - upon failing to find a type, it should be clear where and why,
3. **Enforceable** - type checks should be statically checked as much as possible.
4. **Inferable*** - to the above I add the fact that a general expectation is that the typechecker should have the capacity to *infer* the most general type, statically at compile time. This is as touched upon in Damas (1984) a good exemplification of the type growing from the semantics of the language, rather than being an artificial add-on.

1870s	*origins of formal logic*	Frege (1879)
1900s	*formalization of mathematics*	Whitehead and Russell (1910)
1930s	*untyped lambda-calculus*	Church (1941)
1940s	*simply typed lambda-calculus*	Church (1940), Curry and Feys (1958)
1950s	Fortran	Backus (1981)
	Algol-60	Naur et al. (1963)
1960s	*Automath project*	de Bruijn (1980)
	Simula	Birtwistle et al. (1979)
	Curry-Howard correspondence	Howard (1980)
	Algol-68	(van Wijngaarden et al., 1975)
1970s	Pascal	Wirth (1971)
	Martin-Löf type theory	Martin-Löf (1973, 1982)
	System F, F^{ω}	Girard (1972)
	polymorphic lambda-calculus	Reynolds (1974)
	CLU	Liskov et al. (1981)
	polymorphic type inference	Milner (1978), Damas and Milner (1982)
	ML	Gordon, Milner, and Wadsworth (1979)
	intersection types	Coppo and Dezani (1978)
		Coppo, Dezani, and Sallé (1979), Pottinger (1980)
1980s	NuPRL project	Constable et al. (1986)
	subtyping	Reynolds (1980), Cardelli (1984), Mitchell (1984a)
	ADTs as existential types	Mitchell and Plotkin (1988)
	calculus of constructions	Coquand (1985), Coquand and Huet (1988)
	linear logic	Girard (1987) , Girard et al. (1989)
	bounded quantification	Cardelli and Wegner (1985)
		Curien and Ghelli (1992), Cardelli et al. (1994)
	Edinburgh Logical Framework	Harper, Honsell, and Plotkin (1992)
	Forsythe	Reynolds (1988)
	pure type systems	Terlouw (1989), Berardi (1988), Barendregt (1991)
	dependent types and modularity	Burstall and Lampson (1984), MacQueen (1986)
	Quest	Cardelli (1991)
	effect systems	Gifford et al. (1987), Talpin and Jouvelot (1992)
	row variables; extensible records	Wand (1987), Rémy (1989)
		Cardelli and Mitchell (1991)
1990s	higher-order subtyping	Cardelli (1990), Cardelli and Longo (1991)
	typed intermediate languages	Tarditi, Morrisett, et al. (1996)
	object calculus	Abadi and Cardelli (1996)
	translucent types and modularity	Harper and Lillibridge (1994), Leroy (1994)
	typed assembly language	Morrisett et al. (1998)

Figure 2.2: Timeline of types in computer science and logic from Pierce (2002)

Type Systems Formalisms

Type systems are described and based around a particular formalism. The elements of type system formalisms are: *Judgements, Type Rules, and Type Derivations*.

Definition 2.2.1. Judgements are rules of the type $\Gamma \vdash \aleph$, where we say Γ *entails* $\aleph$. Γ is a *typing context* or *typing environment*, that can be represented by a set of variables and their types (see Definition 2.2.9), and $\aleph$ is an *assertion*.

Definition 2.2.2. Type rules assert the validity of an *assertion*. A valid *assertion* is by definition equivalent with a *well typed* term. (see Definition 2.2.6). A collection of *type rules* is called a formal *type system*.

Theorem 2.2.3. *If the typing context Γ does not contain any elements, (i.e. $\Gamma = \emptyset$) then the environment Γ is well formed.*

Definition 2.2.4. Type derivation is a tree of logically connecting judgements stemming from one term. (see Example 2.2.5) They can be created with the use of type variables, which maintain generality - which is the definition of **type polymorphism**.

Example 2.2.5. A *well typed type derivation* for the $\lambda^{\rightarrow}$ term $(\lambda x.\, x)(\lambda x.\, x)$ based on rules defined at Definition 2.2.9. The type variable δ can be replaced with any other type variable, as long as it is consistently replaced across the derivation.

$$\dfrac{\dfrac{}{x : \delta \to \delta \vdash x : \delta \to \delta}}{\vdash \lambda x.\, x : (\delta \to \delta) \to \delta \to \delta} \qquad \dfrac{\dfrac{}{x : \delta \vdash x : \delta}}{\vdash \lambda x.\, x : \delta \to \delta}$$
$$\overline{\vdash (\lambda x.\, x)(\lambda x.\, x) : \delta \to \delta}.$$

The Curry–Howard-Lambek correspondence

A property also called *The Curry-Howard isomorphism* establishes a direct link between three seemingly unrelated fields, namely the correctness of a computer program, mathematical proofs and cartesian closed categories. The correspondence is based on the observation that families of seemingly unrelated formalisms - namely, the proof systems on one hand, and the models of computation on the other - are in fact the same kind of mathematical objects. This correspondence is of high importance when considering programs as proofs.

Definition 2.2.6. The Curry-Howard-Lambek define well-defined morphisms as abiding the following rules where the categorical morphism $f : \alpha \to \beta$ is replaced with *sequent calculus based notation* $f : \alpha \vdash \beta$:

$$\frac{}{id : \alpha \vdash \alpha}(identity) \qquad \frac{t : \alpha \vdash \beta \qquad u : \alpha \vdash \gamma}{u \circ t : \alpha \vdash \gamma}(composition)$$

$$\frac{}{\star : \alpha \vdash \top}(unit\ type) \qquad \frac{t : \alpha \vdash \beta \qquad u : -\alpha \vdash \gamma}{(t, u) : \alpha \vdash \beta \times \gamma}(cartesian\ product)$$

$$\frac{}{\pi_1 : \alpha \times \beta \vdash \alpha}(left\ projection) \qquad \frac{}{\pi_2 : \alpha \times \beta \vdash \beta}(right\ projection)$$

$$\frac{t : \alpha \times \beta \vdash \gamma}{\lambda t : \alpha \vdash \beta \to \gamma}(currying) \qquad \frac{}{eval : (\alpha \to \beta) \times \alpha \vdash \beta}(application)$$

2.2.1 Simply Typed Lambda Calculus

Context

An initial version of the typed λ calculus($\lambda^{\to}$) was introduced by Alonzo Church in 1940. Its creation was an attempt to constrain and avoid paradoxical uses of the untyped lambda calculus. As pointed out by Baxter (2014), the simply typed λ calculus is the theoretical basis for typed, functional programming languages, with most typed systems handling typing similarly to the $\lambda^{\to}$.

Definition 2.2.7. Bakus Naur Form Grammar for a simple type can be written as:

$$\begin{aligned} \tau &::= o \mid \tau \to \tau. \\ Where&: \\ o &\text{ is the base type,} \\ \tau &\text{ is a type,} \\ \tau \to \tau &\text{ is a function type.} \end{aligned}$$

Definition 2.2.8. If use these new constructs to constrain the terms of the λ Calculus we get the definition for the $\lambda^{\to}$ calculus. In Bakus Naur Form:

$$M ::= \ x \mid \lambda x^{\tau}.\, M \mid MN$$

$$\begin{aligned}
&where: \\
&\quad \tau \text{ is a type.} \\
&\quad x \text{ is a variable.} \\
&\lambda x^{\tau}.\, M \text{ is a typed abstraction.} \\
&\lambda x^{\tau}.\, M \Leftrightarrow \lambda x : \tau .\, M (\text{notations are equivalent}). \\
&\quad MN \text{ is an application.}
\end{aligned}$$

Typing rules

Definition 2.2.9. The typing rules for the $\lambda^{\rightarrow}$:

$$\frac{}{\Gamma, \vdash x : \tau \vdash x : \tau} var. \qquad \frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x^{\tau}.M : \tau \rightarrow \sigma} abstr. \qquad \frac{\Gamma \vdash M : \tau \rightarrow \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \sigma} app.$$

$$\begin{aligned}
&Where: \\
M : \tau &\Leftrightarrow M \text{ has type } \tau, \\
\Gamma &\Leftrightarrow \text{context - a finite function from variables to types}, \\
\Gamma &= x_1 : \tau_1, x_2 : \tau_2, x_3 : \tau_3, ..., x_n : \tau_n, \\
\Gamma, x : \tau &\Leftrightarrow \text{context } \tau \text{ extended so } x \text{ has type } \tau, \\
\Gamma \vdash x : \tau &\Leftrightarrow \text{Typing judgment}, \\
M : \tau \rightarrow \sigma &\Leftrightarrow \text{abstraction } M \text{ receives type } \tau \text{ and returns type } \sigma.
\end{aligned}$$

Theorem 2.2.10. *Given the* Subject Reduction *property of $\lambda^{\rightarrow}$ terms (β reduction gives another $\lambda^{\rightarrow}$ term), it has been proven (Tait, 1967) that typeable terms on $\lambda^{\rightarrow}$ are all strongly normalising. This is why $\lambda^{\rightarrow}$ is a deterministic system of computation, and algorithms written in $\lambda^{\rightarrow}$ are decidable, thus not Turing Complete. Furthermore fixed point combinators cannot be captured by a type in the $\lambda^{\rightarrow}$ system.*

2.2.2 Hindley Milner Type System

Motivation

Created by Hindley (1969), and further defined by Miller (1988), the Hindley Milner Type System, is a classical type system with parametric polymorphism, a closed proof formulated in Damas (1984), completeness property, and the ability to infer the most general type without type annotations. As specified by Miller (1988) the system has at its core simplicity, inference, and **polymorphism**.

In the future research, the type system's unification algorithms are a good source of inspiration and an adequate departure point, with multiple inference algorithms and richness in literature.

Language

Definition 2.2.11. As described in Heeren, Hage and Swierstra (2002) we first need to introduce the lambda language that the Hindley Milner type systems works on top of. This is a simple λ calculus language to which we add the *let* construct.

$$\begin{aligned}
Terms, E &:= x && \text{(variable)}, \\
&| E_1 E_2 && \text{(application)}, \\
&| \lambda x \rightarrow E && \text{(abstraction)}, \\
&| let\ x = E_1\ in\ E_2 && \text{(let)}.
\end{aligned}$$

To this simple language we add types.

$$Type, \tau := \alpha |\ Boolean |\ Integer |\ String |\ \tau \rightarrow \tau |\ \forall \vec{\alpha}.\tau (polytype/typescheme).$$

Definition 2.2.12. A **type scheme** is a type vector $\vec{\alpha}$ in which a set of **polymorphic** type variables $\vec{\alpha} = \alpha_1, \alpha_2, \cdots$ are bound to the universal type quantifier. Although the variables have an order in the type scheme, this order is of no significance.

Definition 2.2.13. Hindley Milner typing rules, as presented in Damas (1984) are:

$$\frac{}{\Gamma \vdash x : \tau} Var. \quad (x : \tau \in \Gamma)$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash E_2 : \tau_1}{\Gamma \vdash E_1 E_2 : \tau_2} App.$$

$$\frac{\Gamma / x \cup \{x : \tau_1\} \vdash E : \tau_2}{\Gamma \vdash \lambda x.E : \tau_1 \rightarrow \tau_2} Abs.$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \qquad \Gamma / x \cup \{x : generalise(\Gamma, \tau_1)\} \vdash E_2 : \tau_2}{\Gamma \vdash let\ x = E_1\ in\ E_2 : \tau_2} Let.$$

Damas (1984) provides an extensive description of the inference procedure of finding adequate substitutions, and a proof of how this finds the most general type for the term. The manner in which this proof is developed offers a good point of reference.

2.2.3 $\lambda\mu$ calculus

The proposal of Parigot (1992) is to decompose the λ calculus into two types of variables: λ variables and μ variables, with the latter being used to name terms in the first. $\lambda\mu$ calculus maintains confluence and is able to be typed with the same rules as the $\lambda^{\rightarrow}$ with the addition of a naming rule.

$$\frac{t : \Pi \vdash A, \Sigma}{[\alpha]t : \Pi \vdash A^{\alpha}, \Sigma} \qquad \frac{e : \Gamma \vdash A^{\alpha}, \Delta}{\mu\alpha.e : \Gamma \vdash A, \Delta}$$

The study of the calculus is of interest due to its similar nature to the poly-lambda calculus of Heijltjes (2021) and could provide an intermediary step to the fully dependent type system for the FMC. Most importantly, the $\lambda\mu$ as defined in Parigot (1992) calculus provides a bridge between constructive and classical proofs - and understanding the proof of this could lead to a similar property being embedded into the FMC.

2.2.4 Dependently Typed Systems

Definition 2.2.14. A **dependent type system** allows type constructors to depend on different terms or other type constructors.

In the analysis of type systems, Barendregt (1993) distinguishes between several typologies of type systems (graphically portrayed at Figure 2.3), all stemming from the $\lambda^{\rightarrow}$ (simply typed lambda calculus):

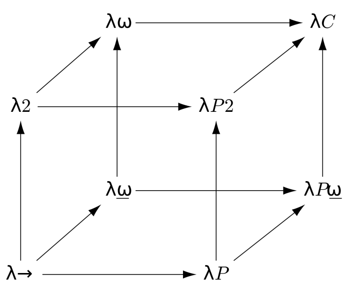


Figure 2.3: Barendregt's Lambda Cube as depicted by **?**

1. systems where terms can bind types (polymorphism) - on the y axis,
2. systems where types can bind terms (dependent types) - on the x axis,
3. systems where types can bind types (type operators or type constructors) - on the z axis.

Motivation

As programming languages and the field of computer science expands, so does the need for reliability and correctness. As discussed, in Subsection 2.2 type systems are a proven static method of achieving the two aforementioned goals. As systems become more complex, the expressive requirements needed from the type system also increase. In an ideal scenario, types would become a first *class citizen* of the language, allowing the programmer to freely mix and use terms and types.

It should be observed, that the minimal syntax of the FMC and its ease of parametrisation across multiple variables (types of locations, types of variables, types of machines, types of output etc.), seems to offer the perfect background for a fully dependent type system; making use of the creative possibilities of the FMC.

2.2.5 Idris

Dependent types and specifically *full dependent types* offer no restriction on the values that a type can be defined by, thus allowing for complete flexibility in type definitions. Thus using, lessons learned from the implementation of other dependently-typed programming languages like *Coq, Agda, or* $\lambda\Pi$ Baxter (2014).

To give an example of this mixing of types and terms we can look at the syntax proposed by Brady (2013) in the implementation of *Idris*, a dependently typed programming language.

$$
\begin{aligned}
Terms, t ::=\ & c && \text{(constant)}\\
\mid\ & x && \text{(variable)}\\
\mid\ & b.\, t && \text{(binding)}\\
\mid\ & tt && \text{(application)}\\
\mid\ & T && \text{(type constructor)}\\
\mid\ & D && \text{(data constructor)}
\end{aligned}
$$

$$
\begin{aligned}
Constants, c ::=\ & Type_i && \text{(type universes)}\\
\mid\ & i && \text{(integer literal)}\\
\mid\ & str && \text{(string literal)}
\end{aligned}
$$

$$
\begin{aligned}
Binders, b ::=\ & \lambda x : t && \text{(abstraction)}\\
\mid\ & let\ x \to t : t && \text{(let binding)}\\
\mid\ & \forall x : t && \text{(function space)}
\end{aligned}
$$

Type inference in Idris is done by using a *cumulativity rule*, while Girard's paradox is avoided by parametrising the Type of Types with the help of a universe level, and an ordering of types based on higher or lower levels. The type checker normalises all of the terms and compares them, and the default normalisation rules is based on a CBV strategy. Furthermore, during typechecking, Iris has a method of checking for totality of functions, while similarly to Haskell, partial functions are allowed to run (a divergence from most dependently typed languages).

There are many aspects not touched upon and embedded complexity which is apparent in the differences between the *FMC* and *Idris* - the completely different syntax and structure being obvious. But studying existing systems and learning from the development process can offer an insight into good strategies.

The manner in which these strategies could be applied for the sequential types, is one of the proposed objectives of the research.

2.3 FMC - A new λ calculus

2.3.1 Semantics

Definition 2.3.1. As defined in Heijltjes (2021), the **FMC**'s simple syntax:

$$M, N ::= \star \mid x.\, N \mid [M]a.\, N \mid a\langle x\rangle.\, N.$$

Where:

$$\begin{aligned} \star &: \text{ an end or nil,} \\ x.\, N &: \text{ a (sequential) variable } x, \\ [M]a.\, N &: \text{ an application or a push action on location } a, \\ a\langle x\rangle.\, N &: \text{ an abstraction or a pop action on location } a \text{ which binds variable } x \text{ in } N. \end{aligned}$$

Definition 2.3.2. The **Functional Abstract Machine** (FAM) is a Krivine Machine that has states (S, N) where N is a FMC term and $S : A \to FMC^{\mathbb{N}}$ is the memory function assigning to each location $a \in A$ a stack of FMC terms $S_a \in FMC^{\mathbb{N}}$. Empty stacks are given as ε_a, and a stack with top element M and remaining stack S_a is given as $S_a.M$. The stack S_a at position a is separated from the remaining memory S as $S; S_a$.

Definition 2.3.3. β **Rewrite** reduction in the FMC and is given by the rule:

$$[M]a.\, A_1 \ldots A_n.\, a\langle x\rangle.\, N \to_\beta A_1 \ldots A_n.\, \{M/x\}N,$$

where actions $A_1 \ldots A_n$ are not on the location a, and substitution $\{M/x\}N$ is a *capture avoiding substitution* replacing variable x with term M in term N defined by the rules at Definition 2.3.6 and capture avoiding application of $M.\, N$ as defined at Definition 2.3.7.

Definition 2.3.4. Reduction takes place separately on each location and the regular λ-calculus is embedded via a reserved location λ, which is usually omitted for brevity.

Example 2.3.5. Using Definition 2.3.8 we can permute a term passed all the terms that do not occur on the same location:

$$[M]a.\, A_1 \ldots A_n.\, a\langle x\rangle.\, N \sim A_1 \ldots A_n.\, [M]a.\, a\langle x\rangle.\, N,$$
$$\text{if } A_1 \ldots A_n \text{ do not occur on location } a.$$

Definition 2.3.6. Capture avoiding substitution in the FMC is defined as:

$$\begin{aligned} \{L/y\}\star &\triangleq \star, \\ \{L/y\}y.\, N &\triangleq L.\, \{L/y\}N, \\ \{L/y\}x.\, N &\triangleq x.\, \{L/y\}N, \\ \{L/y\}[M]a.\, N &\triangleq [\{L/y\}M]a.\, \{L/y\}N, \\ \{L/y\}a\langle y\rangle.\, N &\triangleq a\langle y\rangle.\, N, \\ \{L/y\}a\langle x\rangle.\, N &\triangleq a\langle z\rangle.\, \{L/y\}\{z/x\}N \text{ where } z \text{ is fresh.} \end{aligned}$$

Definition 2.3.7. Capture avoiding application in the FMC is defined as:

$$\begin{aligned} \star\,.\, N &\triangleq N, \\ (x.\, M).\, N &\triangleq x.\, (M.\, N), \\ ([L]a.\, M).\, N &\triangleq [L]a.\, (M.\, N), \\ (a\langle x\rangle.\, M).\, N &\triangleq a\langle z\rangle.\, ((\{z/x\}M).\, N) \text{ where } z \text{ is fresh.} \end{aligned}$$

Definition 2.3.8. Terms are considered **modulo α equivalent**, if after permuting operations on other stacks, the terms are reflexively equal. The operation of permuting non interacting terms is notated with $\sim$.

$$\begin{aligned}&\text{Iff location } a \neq \text{ location } b,\\&[M]a.\,[N]b.\,P \sim [N]b.\,[M]a.\,P,\\&a\langle x\rangle.\,[N]b.\,P \sim [N]b.\,a\langle x\rangle.\,P \text{ if } x \notin freeVar(N),\\&a\langle x\rangle.\,b\langle y\rangle.\,P \sim b\langle y\rangle.\,a\langle x\rangle.\,P.\end{aligned}$$

Example 2.3.9. An example of modulo α equivalent terms would be terms M, N where

$$M = [1]a.\,[1]b.\,[1]c \text{ and } N = [1]b.\,[1]c.\,[1]a.$$

Definition 2.3.10. For brevity we omit the trailing $\star$ of a term - thus $x.\,\star$ is written as x and $M.\,P.\,\star$ as $M.\,P$.

Definition 2.3.11. We call **sequentiality** the decomposition of variable x into a *variable with continuation* $x.\,N$ and an *end of instructions* construct $\star$ - so that the original variable constructor is recovered as $x.\,\star$.

Example 2.3.12. *Sequentiality* is one of the main features of the FMC, allowing the interfacing of CBN and CBV and the easy choice between the two. This is best portrayed by revisiting the example highlighting the non confluent manner in the absence of the sequentiality property:

$$\begin{aligned}a := 2; (\lambda x.!a)(a := 3; 5) &\mapsto^*_{cbn} 2\\&\mapsto^*_{cbv} 3\end{aligned}$$

With *sequentiality*, we can now build the term specifically to get either of the results, as we wish.

$$a := 2\,.\,t := (\backslash\backslash x.!a)\,.\,p := (a := 3.5)\,.\,?p\,.\,?t\ p\,.\,?a\,.\,print \rightarrow_{\beta^*} [2]out, \qquad (cbn)$$
$$a := 2\,.\,t := (\backslash\backslash x.!a)\,.\,p := (a := 3.5)\,.\,!p\,.\,?t\ p\,.\,?a\,.\,print \rightarrow_{\beta^*} [3]out;\ 5 \text{ on the spine.} \qquad (cbv)$$

Theorem 2.3.13. *The constructs of* location *(2.3.1) and sequentiality (2.3.11) are independent and conservative and can be negated by forcing $A = \{\lambda\}$ (where λ is the location of the main stack), respectively forcing sequential variables and $\star$ to always occur together.*

Theorem 2.3.14. *The FMC maintains confluence under both **cbn** and **cbv** reduction strategies.*

2.3.2 Encoding Effects

Definition 2.3.15. Effects are encoded in the *FMC* calculus as operations on pre-defined locations.

Definition 2.3.16. Input is encoded as a pop action on location $in \in A$, and is notated as $in\langle x\rangle$ where x is a variable in the main stack.

Definition 2.3.17. Output is encoded as a push action on location $out \in A$, and is notated as $[x]out$ where x is a variable in the main stack. No pop action can be effectuated on out.

Definition 2.3.18. Higher Order Mutable Store is a subset $C \subseteq A$ of locations designated as storage cells, whose stack can only hold at most one value. The operations are *update* $c := M.\,N$ which will set the cell c with value M and *read!* $!c$ which reads and executs the value at location c. The encodings are

$$c{:=}M.\,N \triangleq c\langle\_\rangle.\,[M]c.\,N$$
$$c \triangleq c\langle x\rangle.\,[x]c.\,N$$

Where _ is a *fresh* variable that does not occur in M or N which is immediately discarded.

Example 2.3.19. A good example would be the encoding of a function that takes to arguments and returns the sum.

$$f:\ = (\backslash\backslash x.\,\backslash\backslash y.\,x + y).\,!f\ 2\ 3.\,print \text{ would print } 5$$
$$\text{where the term parses to}$$
$$f\langle f\rangle.\,[\langle x\rangle.\,\langle y\rangle.\,[y].\,[x].\,+]f.\,[3].\,[2].\,f\langle f\rangle.\,[f]f.\,f.\,\langle print\rangle.\,[print]out$$

Definition 2.3.20. Non-deterministic and probabilistic computation is encoded as a pop action on locations $nd \in A$ respectively $rnd \in A$. The actions $nd\langle x\rangle. N$ and $rnd\langle x\rangle. N$ bind the variable x to a Boolean value in Church encoding $T = \lambda x. \lambda y. x$ for *True* or $F = \lambda x. \lambda y. y$ for *False* with the one from location nd being *non-determistically* generated and the one from location rnd being *deterministically* generated.

Definition 2.3.21. By using nd, rnd locations we can then encode **traditional non-deterministic sum** $+$ and **fair probabilistic sum** $\oplus$.

$$N + M \triangleq nd\langle x\rangle. xMN$$

$$N \oplus M \triangleq rnd\langle x\rangle. xMN$$

Example 2.3.22. Writing a random number to the standard output would be done by the term:

$$rnd\langle x\rangle. [x]out,$$

while reading from the *stdin* and storing the information in a location b would be done by the term

$$in\langle a\rangle. [a]b.$$

Definition 2.3.23. Values and commands are encoded as follows:

- Values - are characterised by the machine terminating with an abstraction, or a term.
- Commands - are characterised by the machine terminating with an end ($\star$).

To be able to distinguish between errors and normal execution, the proposal is that these returned values should be located on the λ location, but not on the main sequence of the term - *the spine*.

Example 2.3.24. The execution of the term $a\langle -\rangle. [1]a. a\langle x\rangle. [x]. \langle print\rangle. [print]out$ would:

1. $a\langle -\rangle$ Initialise position a,
2. $[1]a$ Push 1 to position a,
3. $a\langle x\rangle$ Bind 1 (from the last position of stack a) to variable x,
4. $[x]$ Push the contents of variable x to the λ stack,
5. $\langle print\rangle$ Bind the contents from the λ stack to variable $print$ and
6. $[print]out$ Push the contents of variable $[print]$ to location out.

At the end of this run the machine would terminate with $\star$ on its λ stack, representing a successful operation of the type $\star \Rightarrow out(int)$.

Example 2.3.25. The evaluation of the term $M = in\langle x\rangle. + 2\, x$ on input 3 would terminate with a 5 on the λ stack which is representative of an integer result, thus an integer **value**. The type of the operation would be $(Int)in \Rightarrow (Int)$. This term could be composed with a term $N = \langle print\rangle. [print]out$ of type $(Int) \Rightarrow (Int)out$ leading to an operation of the type $(Int)in \Rightarrow (Int)out$ characterised by $\star$ on the λ stack.

Example 2.3.26. Finally the current proposal highlights that the operation $in\langle x\rangle. x. y$ could be treated as an error, as both x and free variable y remain on the main spine. To this proposals of treating errors as a separate location $error$ or a new action parametrised location at position out of the type $error.\$out$ could be added.

Chapter 3

FMC Type System

Overview of pre-existing proposal

The Heijltjes (2021) proposed type system requires full typing information to be added to the definition of a program, in order for a type check. This is not practical, and furthermore could prove cumbersome going forward. This observation leads naturally into the need for a inference based static type system - which intuitive and precedent based information (the success of Haskell, and Rust) would lead to a better ergonomic and safety of programming in the language.

3.1 Overview

The dissertation's proposed type system builds upon the **Poly Types** as defined by Heijltjes (2021). To make clear the similarities and differences, the paper will reintroduce some of the basic concepts, building towards the implementation and design decisions.

Lambda calculus simple types are not suitable for the FMC. But, following operational considerations leads to a simple conjunction-implication system without primitive monadic functors. The system is parametrised on locations - adequately modelling FMC's operational semantics. Finally, the proposed system semantically defines a cartesian closed category.

Definition 3.1.1. Sequent is a mathematical general condition assertion of the form: $A_1, A_2, ...A_m \vdash B_1, B_2, ...B_n$, where: $A_1, A_2, ...A_m$ are called *"antecendents"* and $B_1, B_2, ...B_n$ are called *"consequents"*. The expression is read as: *if all the antecendents are true, then at least one of the consequents are true*. This style of logical reasoning has its roots in Sequent Calculus.

3.1.1 Typed Sequential Lambda Calculus

Definition 3.1.2. sequential types are an appropriate proposal to be used with the *sequential λ-calculus*:

$$\rho, \sigma, \tau ::= \sigma_n ... \sigma_1 \Rightarrow \tau_1 ... \tau_n$$

Where: ρ is a type, consisting of a vector σ_n ... σ_1 of antecedents and a vector τ_1 ... τ_n of precedents. The concatenation of types can be interpreted with the use of standard implication and conjunction as:

$$\rho = \sigma_n \land ... \land \sigma_1 \rightarrow \tau_1 \land ... \land \tau_n.$$

Definition 3.1.3. The new typing rules for the sequential type system are:

$$\frac{}{\Gamma \vdash \star : \vec{\tau} \Rightarrow \vec{\tau}}\star$$

$$\frac{\Gamma, x : \overleftarrow{\rho} \Rightarrow \vec{\sigma} \vdash N : \overleftarrow{\sigma}\overleftarrow{\tau} \Rightarrow \vec{\upsilon}}{\Gamma, \vdash x : \overleftarrow{\rho} \Rightarrow \vec{\sigma} \vdash x.\,N : \overleftarrow{\rho\tau} \Rightarrow \vec{\upsilon}}\,var.$$

$$\frac{\Gamma, x : \rho \vdash N : \overleftarrow{\sigma} \Rightarrow \vec{\tau}}{\Gamma \vdash \langle x \rangle.\,N : \rho\overleftarrow{\sigma} \Rightarrow \vec{\tau}}\,abs.$$

$$\frac{\Gamma \vdash M : \rho \qquad \Gamma \vdash N : \rho\overleftarrow{\sigma} \Rightarrow \vec{\tau}}{\Gamma \vdash MN : \overleftarrow{\sigma} \Rightarrow \vec{\tau}}\,app.$$

epsilon

Example 3.1.4. The intuitive manner in which the types can be understood is: **a term** will have a type, and **a location** will have a vector of types. If N has type $\overleftarrow{\sigma} \Rightarrow \vec{\tau}$ and S has the type $\vec{\sigma}$, then the machine run (S, N) will produce stack $(T, \star)$ with the type $\vec{\tau}$. We can observe how, the types present the net behaviour of the abstract machine and not the intermediate stack use.

A further property of the sequential types is the ability to type terms of the type $\lambda x.\,xx$ by assigning x a type of the form $\Rightarrow \vec{\tau}$. This property is novel, as it completely diverges from the $\lambda^{\rightarrow}$, yet fixed point combinators are still not able to be typed.

Theorem 3.1.5. *Terms of the type $\lambda x.\,xx$ satisfy: expansion, composition, subject substitution, and subject reduction.*

3.1.2 Poly Typed Functional Machine Calculus

Poly-types are a further parametrisation of the sequential type, analogous to the change from a single stack to a multiple-stack abstract machine.

Definition 3.1.6. **Poly-types** $\rho, \sigma, \tau, \upsilon$ are given by the language:

$$\tau ::= \overleftarrow{\sigma}_A \Rightarrow \vec{\tau}_A$$
$$\vec{\tau}_A ::= \{\vec{\tau}_a | a \in A\}$$
$$\vec{\tau}_a ::= \tau_1 ... \tau_n$$

Where:

A is a set of locations.

$\vec{\tau}_a$ is vector $\vec{\tau}$ parametrised on location A

The strong normalising properties of terms typed in the poly-type system are maintained, based on a proof analogous to that of the sequential types.

Definition 3.1.7. **Sequential types** are a basic structure leading to the *FMC* type system and are of the form:

$$\rho,\ \sigma,\ \tau ::= \sigma_n ... \sigma_1 \Rightarrow \tau_1 ... \tau_n$$

Where: ρ is a type, consisting of a vector σ_n ... σ_1 of antecedents and a vector τ_1 ... τ_n of precedents. The concatenation of types can be interpreted with the use of standard implication and conjunction as:

$$\rho = \sigma_n \wedge ... \wedge \sigma_1 \rightarrow \tau_1 \wedge ... \wedge \tau_n.$$

Definition 3.1.8. **Poly-types** are defined by parameterising the sequential types with the addition of a location variable.

Poly-types $\rho, \sigma, \tau, \upsilon$ are given by the language:

$$\tau ::= \overleftarrow{\sigma}_A \Rightarrow \overrightarrow{\tau}_A$$
$$\overrightarrow{\tau}_A ::= \{\overrightarrow{\tau}_a | a \in A\}$$
$$\overrightarrow{\tau}_a ::= \tau_1 ... \tau_n$$

Where A is a set of locations, and $\overrightarrow{\tau}_a$ is a vector $\overrightarrow{\tau}$ parametrised on location a.

Definition 3.1.9. Types can be intuitively understood as the net behaviour of the FMC machine, where the antecedents represent the input that the machine requires, and the precedents represent the output of the machine. The dissertation proposes a further expansion of the standard poly-type. The BNF of the proposed *FMC* types is:

$$\mathbb{T} ::= \ \mathbb{C} \mid \mathbb{V} \mid \epsilon \mid \mathbb{T}\mathbb{T} \mid \mathbb{T} \Rightarrow \mathbb{T} \mid l(\mathbb{T})$$

$\mathbb{C}$ are **constants**, a terminal type modelling the behaviour of constants in computation. Although useful, **constants** can be omitted without any impact on the integrity of the type-system. $\mathbb{V}$ are **variables** that can be cast to any other type $\mathbb{T}$ through the use of substitutions. It is important to note that substitutions are consistent on equal **variables**. ϵ is the **empty** type, a representation similar to that of *void*. $\mathbb{T}\mathbb{T}$ is a **concatenation** of types, representative of the sequential property of the *FMC* . $\mathbb{T} \Rightarrow \mathbb{T}$ is a **function** type, similar to the one found in the $\lambda^{\rightarrow}$. Lastly $l(\mathbb{T})$ is the location parametrised type, the natural way of capturing the locations of the *FMC* .

Although the first four types (with the exception of ϵ - which is a special case in itself) seem to not be parametrised on a location, in reality they are representative of types present on the home stack (referred to as the γ location) of the FMC machine.

For notation conventions used refer to Figure 3.1.2.

Figure 3.1: Type notation conventions

For consistency the notation conventions used through the thesis, and taken forward to the parser implementation:

1. **Constant types** $\mathbb{C}$ are written as words beginning with a capital letter, for example Int, or $Bool$.
2. **Variable types** $\mathbb{V}$ are written as words beginning with a lower case letters, for example a, or $bA1$.
3. **Location types** $_l\mathbb{T}$ are written location first followed by the bracketed type, for example $in(a)$ or $a(b(A))$.
4. **Concatenated types** $\mathbb{T}\mathbb{T}$ are written surrounded by brackets with the types separated by a comma or a space, for example $(a, B, l(C))$. The position of the types in the vector is read from left to right, with left being the first type in the vector.
5. **Function types** $\mathbb{T} \Rightarrow \mathbb{T}$ are as $\mathbb{T} \Rightarrow \mathbb{T}$. For example $a \Rightarrow b$, $x \Rightarrow l(a \Rightarrow b)$ or $x \Rightarrow n(() \Rightarrow l(a \Rightarrow b))$.
6. **Empty type** ϵ can be omitted from a function type, writing $\epsilon \Rightarrow a$ as $\Rightarrow a$, or (parser specific) as $() \Rightarrow a$.

Definition 3.1.10. A well typed FMC_t term N is typed by a context $\Gamma = x : a \Rightarrow b...$ where $\Gamma \vdash N$ based on the following typing rules:

$$\frac{}{\Gamma \vdash \star : \Rightarrow} \textit{star}$$

$$\frac{\overline{\Gamma \vdash M : d \Rightarrow e}}{\Gamma / x \cup x : a \Rightarrow b \vdash x \,;\, M : a \dotplus b \gg d \Rightarrow b \dotplus b \ll d} \textit{variable}$$

$$\frac{\overline{\Gamma \vdash M : a \Rightarrow b} \qquad \overline{\Gamma \vdash N : d \Rightarrow e}}{\Gamma \vdash [M]l.\, N : d \Rightarrow (e,\, l(a \Rightarrow b))} \textit{application}$$

$$\frac{\overline{\Gamma \vdash M : c \Rightarrow d}}{\Gamma / x \cup x : a \Rightarrow b \vdash l\langle x \rangle.\, M : (l(a \Rightarrow b),\, c) \Rightarrow d} \textit{abstraction}$$

A **saturated location** is an empty location, or in other words a location which holds the type ϵ. The function *loc* $: \mathbb{T} - > \mathbb{L}$ returns a set of all the non empty locations of a type.

The fusion law is the equivalent of function composition in the *FMC* where $f.x$ is equivalent to $x; f$. The fusion/composition of terms can only happen on the home location γ. Furthermore, any concatenation of γ types gets fused into one (resulting) γ type with an input type and an output type.

Example 3.1.11. Given the terms $M : (a \Rightarrow b)$ and $N : (b \Rightarrow c)$ their sequencing into one term $M ; N$ could wrongly be represented by the concatenation of their types $((a \Rightarrow b), (b \Rightarrow c))$. This is an example of not applying the rule of fusion/composition, and would represent delaying the evaluation of the two terms; essentially chaining unevaluated (thunks). This is not how the *FMC* behaves, where a sequencing of γ terms must fuse/compose into one resulting $\gamma((\mathbb{T} \Rightarrow \mathbb{T}))$ type - describing the net behaviour of the machine. Note that γ is an arbitrary location, and can be replaced by any other location, but that the *FMC* machine only evaluates terms parametrised at this location.

Definition 3.1.12. Given the term $M. N$, where M is of the type $(a \Rightarrow b)$ and N is of the type $(c \Rightarrow d)$, for a well typed term to be able to apply the law of fusion one of the following conditions must stand:

1. c is of the form (b, e), i.e. $b \subseteq c$. The output of the first term is fully consumed by the second term. The remaining types from the second type get concatenated to the input of the first type, with its output type now becoming the output type of the term.
2. b is of the form (c, e), i.e. $c \subseteq b$. The input of the second term is fully consumed by the output of the first term. Case in which the remaining output of the first term is concatenated to the end of the second output.
3. Both rule 1. and 2. are special conditions of a more general rule, specifically that of location independence. In order for the fusion of two terms to take place, the necessary condition is that if any location is unsaturated in either the left or right type at the end of the unification, it must be saturated in the location of the other type. Thus if the output of term M is of the form $(a, \lambda(b), \eta(c), \mu(d))$ and the input of term N is of the form $(m, \lambda(n), \eta(p))$ then fusion can only take place iff $((a \subseteq m) \vee (a \supseteq m)) \wedge ((b \subseteq n) \vee (b \supseteq n)) \wedge ((c \subseteq p) \vee (c \supseteq p))$ but $\mu(d)$ does not make a difference in this instance, as location μ is already saturated in the opposing type - any omitted location is saturated.

Theorem 3.1.13. *Any well typed FMC term is defined by a function type* $\mathbb{T} \Rightarrow \mathbb{T}$*.*

Proof. Any well type term M is of the form $(a \Rightarrow b)$ if upon its evaluation by the *FMC* machine with a type a on its γ stack, it would terminate with an element of type b on its γ stack. Similar to *arrows* defined by Hughes, John (2000), *FMC* terms are lifted functions, or morphisms from one type to another. Thus the only way in which the the γ stack could be holding an element of type a is if the machine evaluated a term of the type $(\Rightarrow a)$. And following the rule of fusion the sequencing of the two would give rise to the type $(\Rightarrow b)$.

Any other term is not considered well typed as it cannot be evaluated by the *FMC* machine. By the typing rules defined at 3.1.10, all the other terms are dependant on a well typed term, thus by induction, any well typed term is of the form $(a \Rightarrow b)$. □

Example 3.1.14. Nevertheless terms at other locations can still be nested inside the γ type, with the **modulus equivalence** property standing true:

$$M : (\Rightarrow \lambda(a)); N : (\Rightarrow \mu(b)) = M ; N : (\Rightarrow (\lambda(a), \mu(b)) = N ; M : (\Rightarrow (\lambda(a), \mu(b))$$

The sequencing of terms M,N did not result in a concatenation of the two γ types - i.e. $(\Rightarrow ((\Rightarrow \lambda(a)), (\Rightarrow \mu(b)))$. But rather in their fusion into a new γ type. The fusion of the output types of the two terms did result in a new term which is based on the concatenation of the two output types. This is due to the fact that the *FMC* machine *delays* the evaluation those terms.

Proposition 3.1.15. *Juxtaposition* $\dotplus :: \mathbb{T} \rightarrow \mathbb{T} \rightarrow \mathbb{T}$ *defines the operation of concatenation between two FMC types. The triple* $(\mathbb{T}, \dotplus, \epsilon)$ *forms a monoid.*

3.2 Merging/Unification

While assessing the equality of most types is trivial, assessing the equality of types containing variables requires more thought, as variables can expand by splitting into new variables, or contract by becoming an empty type to create equivalent types.

Definition 3.2.1. The **cardinality** of a type is the number of concatenated terms it has at a give location, or inside a location parametrised type. The value of a $\mathbb{C}, \mathbb{T}, \mathbb{V}, \mathbb{T} \Rightarrow \mathbb{T}$ is one when counting at a location, and the cardinality of $(\mathbb{T})$ is equal to the inner cardinality of the wrapped type $\mathbb{T}$.

Definition 3.2.2. The **expansion** and **contraction** of a type is the property of a type variable $\mathbb{V}$ to expand and contract by substituting itself with ϵ.

$$Int \overset{contract}{\leftarrow} a,\ Int,\ a \overset{expand}{\rightarrow} a1,\ a2,\ \cdots,\ an,\ Int,\ a1,\ a2,\ \cdots,\ an,$$

Note that substitutions apply in a consistent manner on equal variables.

To help in determining the equivalence of two types, the function $merge :: (\mathbb{S} \times \mathbb{T} \times \mathbb{T}) \rightarrow (\mathbb{S} \times \mathbb{T} \times \mathbb{T})$ takes a substitution list together with two types and creates a list of substitutions needed to merge the two types, while also keeping track of the remaining unmerged types at each step. The merging algorithm acts both as a unification algorithm and equivalence test.

The functions $\ll$ and $\gg$ typed $\mathbb{T} \rightarrow \mathbb{T} \rightarrow \mathbb{T}$ are specialisation of *merge* which use an empty list of substitutions and only return the remaining elements from the first element respectively the second element. Both resulting types have all the substitutions applied.

Definition 3.2.3. The high level description of the merge algorithm is:

---

1. Apply the substitutions to both terms.
2. Recursively normalise the types - sorting on a per location, per type basis, respecting the modulo equivalence property. Example:

$$\begin{aligned} t_1 &= ((\Rightarrow Int),\ a1,\ \lambda((a1 \Rightarrow Int)),\ b1,\ c1,\ d1) && \mapsto ((\Rightarrow Int),\ a1,\ b1,\ c1,\ d1,\ \lambda((a1 \Rightarrow Int))) \\ t_2 &= ((\Rightarrow Int),\ a2,\ b2,\ d2,\ \lambda(e2)) && \mapsto ((\Rightarrow Int),\ a2,\ b2,\ d2,\ \lambda(e2)) \end{aligned}$$

3. Check the cardinality of the two types, if the types do not contain variables, or are of minimum cardinality difference $\mathbb{V}$, proceed to point 4. with the current terms. Otherwise proceed to the **general type pattern finding** algorithm as follows: (also diagrammatically portrayed in Figure 3.4)
   (a) Create all the variable substitution variations for the expansion or contraction of terms, with cardinality between the smallest cardinality and the highest cardinality;
   (b) Filter out all the variations except the ones resulting in the smallest total cardinality difference between the two new terms.
   (c) Run the merging algorithm from step 4 on all of the resulting terms. Return the best result, which is defined in decreasing order as: the result with both types fully merged, the result with one fully merged type and smallest total cardinality, the result with the smallest combined cardinality.
4. Start the merging process on a per type, per location basis, from left to right - keeping track of what remains unmerged in both left and right types, and the substitutions gathered up to that point;
5. The two types are of the same kind and are equal, at the same position and location. If a different, non-variable term is found then the two terms are not equivalent (see Figure 3.3). Note that function types are equal iff both their input and output types are equal.

$$\begin{gathered} (a \Rightarrow b) = (d \Rightarrow c) \Leftrightarrow a = d, b = c \\ A = A \\ A \neq B \\ a = A \Rightarrow \mathbb{S}(a \rightarrow A) \\ p(a) = k(b) \Leftrightarrow a = b, la = ka \end{gathered}$$

6. One or both types are variables - in which case the algorithm casts the variable to the the other type by creating a substitution. Then the rest of the type is continued to be merged after applying the new substitution list to the terms (see Figure 3.2).
7. if a full merge cannot be found, the algorithm returns the remaining types to be merged together with the substitutions at that point.

Proposition 3.2.4. *The merging algorithm is guaranteed to find a solution if one exists, or the first smallest difference between the types.*

Proof. The algorithm creates all the possible expansion, contractions of the two terms, and attempts merging each of them. If the two terms are equivalent, their equivalent form lies in one of the possible expansion, contractions, or casting of the intermediary types. The solution is not space efficient, with a complexity estimated at $(k^d)^l$ where k is the cardinal of unique variables in both terms, d is the maximum cardinal difference between terms at any location and l is the number of locations in the types. □

proposition

Figure 3.2: Example of merging process on two equivalent terms

$$\begin{array}{r} t_1 = ((\Rightarrow Int), a1, \lambda((a1 \Rightarrow Int))) \\ t_2 = ((\Rightarrow Int), a2, \lambda(e2, f2)) \\ s = \{\} \end{array} \mapsto \left| \begin{array}{r} t_1 = (a1, \lambda((a1 \Rightarrow Int))) \\ t_2 = (a2, \lambda(e2, f2)) \\ s = \{\} \end{array} \right. \mapsto \left\| \begin{array}{r} t_1 = \epsilon \\ t_2 = \epsilon \\ s_{final} = \left\{ \begin{array}{l} a1 \to a2 \\ e2 \to (a2 \Rightarrow Int) \\ f2 \to \epsilon \end{array} \right\} \end{array} \right\} (*)$$

From $(*)$ we can deduce that the types t_1, t_2 are equivalent, given the sequential application of the substitutions at s_{final}. In their merged form the two terms are:

$$t_1 \equiv t_2 \equiv (\Rightarrow Int), a2, \lambda((a2 \Rightarrow Int))$$

Figure 3.3: Example of merging process on two non equivalent terms

$$\begin{array}{r} t_1 = ((\Rightarrow a2), Bool, \lambda((a1 \Rightarrow Int))) \\ t_2 = ((\Rightarrow a1), Int, \lambda(e2, f2)) \\ s = \{\} \end{array} \mapsto \left| \begin{array}{r} t_{1final} = (Bool, \lambda((a2 \Rightarrow Int))) \\ t_{2final} = (Int, \lambda(e2, f2)) \\ s_{final} = \{a1 \to a2\} \end{array} \right. \quad (**)$$

From $(**)$ we can deduce that the types t_1, t_2 are not equivalent. We also know that by applying the substitution s_{final} we could partially merge the two types to obtain: t_{1final}, t_{2final}. Although not relevant in this example, keeping track of the partial results is important for the algorithm as a hole.

3.3 Fusion

The ***fusion*** $:: (\mathbb{S} \times \mathbb{T}_t \times \mathbb{T}_t) \to (\mathbb{S} \times \mathbb{T}_t)$ *algorithm is used to determine the type of sequencing FMC terms. The function captures the behaviour of well typed FMC_t terms behave. The intuitive principle behind it is that the FMC machine can consume fully, or partially types which are on the location parametrised stacks, while maintaining certain laws.*

Definition 3.3.1. The function's high level description is:

Figure 3.4: Example of expansion and contraction generating minimum cardinal difference types, and the empty type.

minimum cardinality difference

$$\Delta^{min.}_{card}(t_1, t_2) = 1$$

$$\begin{aligned} t_1 &= (\underbrace{a, b, a, (\Rightarrow Int)}_{card.4}, \lambda(\underbrace{\Rightarrow b}_{card.1})) \\ t_2 &= (\underbrace{c, d}_{card.2}) \end{aligned} \rightarrow$$

$$\{a \rightarrow \epsilon\} \rightarrow \begin{cases} t_1 = (\underbrace{b, (\Rightarrow Int)}_{card.2}, \lambda(\underbrace{\Rightarrow b}_{card.1}) \\ t_2 = (\underbrace{c, d}_{card.2}) \end{cases}$$

$$\{b \rightarrow \epsilon, d \rightarrow (d1, d2)\} \rightarrow \begin{cases} t_1 = (\underbrace{a, a, (\Rightarrow Int)}_{card.3}, \lambda(\underbrace{\Rightarrow b}_{card.1}) \\ t_2 = (\underbrace{(c, d1, d2)}_{card.3}) \end{cases}$$

$$\{b \rightarrow \epsilon, c \rightarrow (c1, c2)\} \rightarrow \begin{cases} t_1 = (\underbrace{a, a, (\Rightarrow Int)}_{card.3}, \lambda(\underbrace{\Rightarrow b}_{card.1}) \\ t_2 = (\underbrace{(c1, c2, d)}_{card.3}) \end{cases}$$

$$\{c \rightarrow \epsilon, d \rightarrow (d1, d2, d3, d4)\} \rightarrow \begin{cases} t_1 = (\underbrace{a, b, a, (\Rightarrow Int)}_{card.4}, \lambda(\underbrace{\Rightarrow b}_{card.1}) \\ t_2 = (\underbrace{(d1, d2, d3, d4)}_{card.4}) \end{cases}$$

$$\{d \rightarrow (d1, d2, d3)\} \rightarrow \begin{cases} t_1 = (\underbrace{a, b, a, (\Rightarrow Int)}_{card.4}, \lambda(\underbrace{\Rightarrow b}_{card.1}) \\ t_2 = (\underbrace{(c, d1, d2, d3)}_{card.4}) \end{cases}$$

$$\{c \rightarrow (c1, c2), d \rightarrow (d1, d2)\} \rightarrow \begin{cases} t_1 = (\underbrace{a, b, a, (\Rightarrow Int)}_{card.4}, \lambda(\underbrace{\Rightarrow b}_{card.1}) \\ t_2 = (\underbrace{(c1, c2, d1, d2)}_{card.4}) \end{cases}$$

$$\{c \rightarrow (c1, c2, c3)\} \rightarrow \begin{cases} t_1 = (\underbrace{a, b, a, (\Rightarrow Int)}_{card.4}, \lambda(\underbrace{(\Rightarrow b)}_{card.1})) \\ t_2 = (\underbrace{(c1, c2, c3, d)}_{card.4}) \end{cases}$$

$$\{d \rightarrow \epsilon, c \rightarrow (c1, c2, c3, c4)\} \rightarrow \begin{cases} t_1 = (\underbrace{a, b, a, (\Rightarrow Int)}_{card.4}, \lambda(\underbrace{(\Rightarrow b)}_{card.1})) \\ t_2 = (\underbrace{(c1, c2, c3, d)}_{card.4}) \end{cases}$$

1. The function receives two FMC_t types, and a list of initial substitutions. The substitutions are applied to the two types, and the new version of the types is taken forward. Example:

$$\frac{\begin{aligned} type1 &= (x \Rightarrow y) \\ type2 &= (a \Rightarrow x) \\ subs &= \{x \rightarrow Int\} \end{aligned}}{\begin{aligned} type1' &= (Int \Rightarrow y) \\ type2' &= (a \Rightarrow Int) \end{aligned}}$$

2. The **merge** algorithm is applied to the output type of the left type and the input type of the right type, using an empty list of substitutions. Example:

$$\frac{\begin{aligned} t_1' &= Int \Rightarrow y \\ t_2' &= Int \Rightarrow b \end{aligned}}{merge(\{\}, y, Int) = (\{y \rightarrow Int\}, \epsilon, \epsilon) = result}$$

3. Depending on the *result* of the **merge** the two types can or cannot be fused:

(a) Iff the first element is fully consumed then the remaining of the second element is concatenated to the end of the first element input, and the output of the first element is replaced with the output of the second element.

$$\frac{\dfrac{\begin{aligned} t_1 &= a \Rightarrow b \\ t_2 &= c \Rightarrow z \end{aligned}}{merge(\{...\}, b, c) = (\{...\}, \epsilon, v)}}{fusion(\{...\}, t_1, t_2) = (\{...\}, a \Rightarrow z)}$$

(b) Iff the second element is fully consumed then the output type becomes the remaining of the first element concatenated to the end of the second element's output type, with the input type remaining the same.

$$\frac{\dfrac{\begin{aligned} t_1 &= a \Rightarrow b \\ t_2 &= c \Rightarrow z \end{aligned}}{merge(\{...\}, b, c) = (\{...\}, v, \epsilon,)}}{fusion(\{...\}, t_1, t_2) = (\{...\}, a \Rightarrow z \dotplus v)}$$

(c) Iff both elements are partially consumed and the remaining elements are all on different locations (do not interfere with one another) then both are added as previously described.

$$t_1 = a \Rightarrow b$$
$$t_2 = c \Rightarrow z$$

$$merge(\{...\}, b, c) = (\{...\}, v, m,)$$
with: $loc(v) \cap locm = \emptyset$

$$fusion(\{...\}, t_1, t_2) = (\{...\}, a \dotplus m \Rightarrow z \dotplus v)$$

(d) Otherwise, the two terms cannot be merged, which should return an error indicating what types are left after the merging attempt. This also means that the term which was meant to fusion, is badly typed.

Intuitively, the merging and fusion algorithms, mimic the manner in which the *FMC* machine operates. Terms of a specific type are *"consumed"* by the *FMC* machine to produce new terms. Analogous to the manner in which terms on different locations can permute freely - partially consumed terms with non-shared locations can compose. This behaviour of the *FMC* machine is similar to partial application in the λ calculus.

Proposition 3.3.2. *Typed terms are not proof of machine termination.*

Proof. Given the term $[x].\langle x: \_\rangle.x$, the FMC_t machine would enter an infinite loop. But as can be seen from the type of the term x the type is correct. Furthermore it can easily be inferred. Terms of the type $\epsilon \Rightarrow \epsilon$ are not proof of termination. Another example term is shown in the following derivations:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash x ; \star : \epsilon \Rightarrow \epsilon} \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash x ; \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash \langle x : e1 \Rightarrow f1 \rangle ; x ; \star : \lambda(\epsilon \Rightarrow \epsilon) \Rightarrow \epsilon}}{\Gamma \vdash [x ; \star] ; \langle x : e1 \Rightarrow f1 \rangle ; x ; \star : \epsilon \Rightarrow \epsilon}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash 1 ; \star : \epsilon \Rightarrow Int} \quad \dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash \langle y : \epsilon \Rightarrow Int \rangle ; \star : \lambda(\epsilon \Rightarrow Int) \Rightarrow \epsilon}}{\Gamma \vdash [1 ; \star] ; \langle y : \epsilon \Rightarrow Int \rangle ; \star : \epsilon \Rightarrow \epsilon} \quad \dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash \langle x : e1 \Rightarrow f1 \rangle ; \star : \lambda(\epsilon \Rightarrow \epsilon) \Rightarrow \epsilon}}{\Gamma \vdash [[1 ; \star] ; \langle y : \epsilon \Rightarrow Int \rangle ; \star] ; \langle x : e1 \Rightarrow f1 \rangle ; \star : \epsilon \Rightarrow \epsilon}$$

□

Proposition 3.3.3. *Any typed term except for* $\Rightarrow$ *is proof of machine termination.*

Alternative Typing Rules

During the research phase, a system based on alternative typing rule was considered, portrayed in Figure 2.2. The system works by breaking down FMC_t terms into smaller terms and fusing them one by one starting from the first term on the left. In comparison, the current typing laws derive a type from the last sequential term (always a $\star$), building the derivation backwards. The differences are similar to traversing and folding the term from the left or from the right, with the alternative typing strategy traversing from the left.

3.4 FMC$_t$ *Syntax*

To ergonomically work with the proposed type system and to allow expressing types in the *FMC* 's syntax, the term of the bind/*pop* term is altered. This constitutes the basis of the FMC_t .

Definition 3.4.1. The *BNF* of the FMC_t is:

$$\begin{aligned} N ::= {} & \star && (star) \\ & \mid x\,;N && (variable) \\ & \mid l\langle x : \mathbb{T}_t\rangle\,;N && (pop) \\ & \mid [M]l\,;N && (push) \end{aligned}$$

The *FMC* and the FMC_t are identical, with the FMC_t introducing some additional concepts, namely native constants, and type constraining. The syntax of the location parameters contains the same list of reserved locations. The location variable $\mathbb{L}$ can take any *string* value with the exception of the reserved locations $\mathbb{L} = \{x | x \in string, x \notin \{in, out, rnd, nd\}\}$. The syntax of the location variable l is described by the BNF:

$$l ::= \; in \mid out \mid rnd \mid nd \mid \mathbb{L}$$

Proposition 3.4.2. *Binding can be inferred in the FMC_t without any additional information if the term is well typed.*

Proof. As seen in the typing laws 3.1.10 the *bind* operator *pops* a term from a specific location and binds it. Given that the type at the specific location is either known or empty in any well typed term, means that no further information to the bind is needed to infer the type. All that is needed is to type the variable with a pair of fresh type variables in a function type, i.e. $\langle x{:}\_\rangle \Leftrightarrow \langle x{:}a1 \Rightarrow a2\rangle$ where $a1$, $a2$ are fresh. If the popped location is empty then the type of x remains general, until a term tries to unify the variables. If the popped location is occupied, then the variables $a1$, $a2$ get unified in the context, taking forward the new type. As seen in the following examples.

$$\dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash \langle x\!: a1 \Rightarrow b1\rangle\,; \star : \lambda(a1 \Rightarrow \epsilon) \Rightarrow \epsilon}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash x\,; \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash \langle x\!: a1 \Rightarrow b1\rangle\,; x\,; \star : \lambda(\epsilon \Rightarrow \epsilon) \Rightarrow \epsilon}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash 1\,; \star : \epsilon \Rightarrow Int} \qquad \dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash \langle x\!: \epsilon \Rightarrow Int\rangle\,; \star : \lambda(\epsilon \Rightarrow Int) \Rightarrow \epsilon}}{\Gamma \vdash [1\,;\, \star]\,; \langle x\!: \epsilon \Rightarrow Int\rangle\,; \star : \epsilon \Rightarrow \epsilon}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash 1\,; \star : \epsilon \Rightarrow Int} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash x\,; \star : \epsilon \Rightarrow Int}}{\Gamma \vdash \langle x\!: \epsilon \Rightarrow Int\rangle\,; x\,; \star : \lambda(\epsilon \Rightarrow Int) \Rightarrow Int}}{\Gamma \vdash [1\,;\, \star]\,; \langle x\!: \epsilon \Rightarrow Int\rangle\,; x\,; \star : \epsilon \Rightarrow Int}$$

Note, the current inference algorithm infers the type of ☐

Theorem 3.4.3. *It is sufficient to type all variables to establish the type of a well-typed term.*

Proof. Proof is analogous to 3.4.2. ☐

Primitives

In the syntax of the *FMC* primitives had to be encoded, and the only constant primitive available was $\star{:}(\Rightarrow)$ and other terms built upon it. For example the pushing of $\star$ to a location:

$$[\star].\,\langle x\rangle.\,\star \Rightarrow x : (\Rightarrow)$$
$$[[\star]l.\,\star].\,\langle x\rangle.\,\star \Rightarrow x : (\Rightarrow l((\Rightarrow)))$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon} \quad \dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash [\,\star\,];\star : \epsilon \Rightarrow \lambda(\epsilon \Rightarrow \epsilon)} \quad \dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash \langle x\colon e1 \Rightarrow f1\rangle;\star : \lambda(\epsilon\Rightarrow\epsilon)\Rightarrow\epsilon}}{\Gamma \vdash [[\,\star\,];\,\star\,];\langle x\colon e1\Rightarrow f1\rangle;\star : \epsilon\Rightarrow\epsilon}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon} \quad \dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash [\,\star\,];\star : \epsilon \Rightarrow \lambda(\epsilon \Rightarrow \epsilon)} \quad \dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash \langle x\colon q1 \Rightarrow r1\rangle;\star : \lambda(\epsilon\Rightarrow\epsilon)\Rightarrow\epsilon}}{\Gamma \vdash [[\,\star\,];\,\star\,];\langle x\colon q1\Rightarrow r1\rangle;\star : \epsilon\Rightarrow\epsilon} \quad \dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash [[[\,\star\,];\,\star\,];\langle x\colon q1\Rightarrow r1\rangle;\,\star\,];\star : \epsilon\Rightarrow\lambda(\epsilon\Rightarrow\epsilon)}$$

with ϵ representing the constant. To make working with constants easier, the FMC_t introduces some primitives, of the type $(\Rightarrow\mathbb{C})$. These are *pre-bound* to their terms and present in any FMC_t typing context.

Definition 3.4.4. FMC_t primitives:

$$\begin{aligned} 0, 1, 2\ldots &: (\Rightarrow \mathit{Int}) \\ \mathit{True}, \mathit{False} &: (\Rightarrow \mathit{Bool}) \\ +, - &: ((\mathit{int}, \mathit{int}) \Rightarrow \lambda(\Rightarrow \mathit{int})) \\ \mathit{if} &: ((\mathit{bool}, \mathit{if}(a), \mathit{if}(a)) \Rightarrow \lambda(a)) \\ = &: ((\mathit{eq}(a), \mathit{eq}(a)) \Rightarrow \lambda(\mathit{bool})) \end{aligned}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash 1;\star : \epsilon \Rightarrow \mathit{Int}} \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash 2;\star : \epsilon \Rightarrow \mathit{Int}} \quad \dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash \mathit{if};\star : (\mathit{Bool}, (\mathit{if}(\epsilon \Rightarrow \mathit{Int}), \mathit{if}(\epsilon \Rightarrow \mathit{Int}))) \Rightarrow \lambda(\epsilon \Rightarrow \mathit{Int})}}{\Gamma \vdash [2;\,\star\,]\mathit{if};\mathit{if};\star : (\mathit{Bool}, \mathit{if}(\epsilon \Rightarrow \mathit{Int})) \Rightarrow \lambda(\epsilon \Rightarrow \mathit{Int})}}{\Gamma \vdash [1;\,\star\,]\mathit{if};[2;\,\star\,]\mathit{if};\mathit{if};\star : \epsilon \Rightarrow (\lambda(\epsilon \Rightarrow \mathit{Int}), \mathit{Bool})}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash 1;\star : \epsilon \Rightarrow \mathit{Int}} \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash 2;\star : \epsilon \Rightarrow \mathit{Int}} \quad \dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash \mathit{if};\star : (\mathit{Bool}, (\mathit{if}(\epsilon \Rightarrow \mathit{Int}), \mathit{if}(\epsilon \Rightarrow \mathit{Int}))) \Rightarrow \lambda(\epsilon \Rightarrow \mathit{Int})}}{\Gamma \vdash [2;\,\star\,]\mathit{if};\mathit{if};\star : (\mathit{Bool}, \mathit{if}(\epsilon \Rightarrow \mathit{Int})) \Rightarrow \lambda(\epsilon \Rightarrow \mathit{Int})}}{\Gamma \vdash [1;\,\star\,]\mathit{if};[2;\,\star\,]\mathit{if};\mathit{if};\star : \epsilon \Rightarrow (\lambda(\epsilon \Rightarrow \mathit{Int}), \mathit{Bool})}}{\Gamma \vdash \mathit{True};[1;\,\star\,]\mathit{if};[2;\,\star\,]\mathit{if};\mathit{if};\star : \epsilon \Rightarrow (\lambda(\epsilon \Rightarrow \mathit{Int}), (\mathit{Bool}, \mathit{Bool}))}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash \langle x\colon a1 \Rightarrow b1\rangle;\star : \lambda(a1 \Rightarrow \epsilon) \Rightarrow \epsilon} \quad \dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash \mathit{if};\star : (\mathit{Bool}, (\mathit{if}(\lambda(a1 \Rightarrow \epsilon) \Rightarrow \epsilon), \mathit{if}(\lambda(a1 \Rightarrow \epsilon) \Rightarrow \epsilon))) \Rightarrow \lambda(\lambda(a1 \Rightarrow \epsilon) \Rightarrow \epsilon)}}{\Gamma \vdash [\langle x\colon a1 \Rightarrow b1\rangle;\,\star\,]\mathit{if};\mathit{if};\star : (\mathit{Bool}, \mathit{if}(\lambda(a1 \Rightarrow \epsilon) \Rightarrow \epsilon)) \Rightarrow \lambda(\lambda(a1 \Rightarrow \epsilon) \Rightarrow \epsilon)}$$

if The type of *if* is worth discussing, as it showcases many features of the FMC_t , and a few design considerations. The type of the term shows the net behaviour of the term itself. Described, *if* will take an evaluated *bool* and two unevaluated (necessarily of the same type) terms from the *if* location. Then, it places the an element of type a in location λ. It is important that the element of type a is in location λ because if the type was $((\mathit{bool}, \mathit{if}(a), \mathit{if}(a)) \Rightarrow a)$ a could not be recaptured or bound, and it would also be executed upon its creation - potentially leading to unwanted results. Some examples, for an intuition on how *if* works, together with their type:

Example 3.4.5.

$$\begin{aligned} [1.\star]\mathit{if}.\,[2.\star]\mathit{if}.\,\mathit{True}.\,\mathit{if}.\,\star &: (\Rightarrow\lambda((\Rightarrow \mathit{int}))) \\ [1.\star]\mathit{if}.\,[2.\star]\mathit{if}.\,\mathit{if}.\,\star &: (\mathit{Bool} \Rightarrow \lambda((\Rightarrow \mathit{int}))) \\ [1.\star].\,\mathit{if}.\,\star &: ((\mathit{bool}, \mathit{if}(a), \mathit{if}(a)) \Rightarrow (\lambda(\Rightarrow a), \lambda(\Rightarrow \mathit{Int}))) \end{aligned}$$

As can also be seen, *if* also offers a good example of polymorphism and casting.

scoop Example 3.4.5 gives rise to a new intricacy of the FMC_t . There is no *direct* access to the evaluated output, i.e. terms on the γ location. If we have a term M of the type $(\Rightarrow Int)$ there is no way to *"pick up"* the result from the term $M\,;M : (\Rightarrow Int,\ Int)$. One way would be to push it from the start to a location $[M\,;M\,;\star] : (\Rightarrow\lambda((\Rightarrow Int,\ Int)))$ but in some instances this is not a feasible way of programming. The proposal is a new location $!$ (called *"scoop"*) $\langle x{:}\_\rangle ! : (\Rightarrow)$ that binds to a term the entire pre-entered state of the machine, while leaving the state of the FMC_t machine unchanged, (with the exception of the new bind).

Example 3.4.6.

$$M; \langle x : \_\rangle ! \Leftrightarrow [M]; \langle x : \_\rangle$$
$$M; M; \langle x : \_\rangle ! \Leftrightarrow [M; M]; \langle x : \_\rangle$$
$$N; \langle x : \_\rangle !; M; M; \langle y : \_\rangle ! \Leftrightarrow [[N]; \langle x : \_\rangle; M; M]; \langle y : \_\rangle$$

3.5 Dealing with effects

reading To discuss the type system's behaviour with regards to reading from a location, some further examples are useful:

Example 3.5.1.

$$in\langle x : (\Rightarrow Int)\rangle\,;\star$$
$$rnd\langle x : (\Rightarrow Bool)\rangle\,;\star$$
$$nd\langle x : \_\rangle\,;\star$$

The first terms are well behaved, as it is clear what the FMC_t machine is expecting from the in, rnd locations, but the third type is less clear. Thus a first constraint, should be not allowing the infer action to take place from the in, rnd, nd locations. The solution is to accept that these locations have special conditions, with regards to pushing and popping, that should be captured and enforced by the type system - and reflected in the behaviour of the evaluator.

writing Writing to the output imposes a different type of issue, that of unevaluated thunks:

Example 3.5.2.

$$[1]\,;\langle x : \_\rangle\,;[x]out : (\Rightarrow out((\Rightarrow Int)))$$

As we can see from the type of the term, out does not hold the $\mathbb{C}$ Int but rather an unevaluated term that would resolve to an Int. Although consistent with the behaviour of the FMC_t this is most probably not the way in which a user would expect the out location to work. Thus the typechecker can impose some extra conditions to the location out and pushing to the location could behave slightly differently. Thus the typechecer should ensure only terms of the type $(\Rightarrow a)$ can be pushed to the out location. This could allow a second evaluator to run the term and display the output.

streams Streams in the FMC_t are typed $(\Rightarrow\mathbb{T})$ and are the equivalent of constants, i.e. constant functions. As the type system stands at the moment, no further addition is needed.

3.6 Dependently Typed FMC_t

As seen in subsection 2.2.5 a first step towards dependently typing the FMC_t is to create a term for the type constructor.

$$\begin{aligned} N ::= &\ \star && (star) \\ |&\ x\,;N && (variable) \\ |&\ l\langle x : \mathbb{T}_t\rangle\,;N && (pop) \\ |&\ [M]l\,;N && (push) \\ |&\ \{x : \mathbb{T}_t\}\,;N && (let) \end{aligned}$$

Figure 3.5: Alternative typing rules, separating the fusion rule.

$$\frac{}{\Gamma \vdash \star : (\Rightarrow)}\ \textit{star}$$

$$\frac{}{\Gamma/x \cup x : (a \Rightarrow b) \vdash x : (a \Rightarrow b)}\ \textit{variable}$$

$$\frac{\overline{\Gamma \vdash M : a \Rightarrow b}}{\Gamma \vdash [M]l : (\Rightarrow l(a \Rightarrow b))}\ \textit{application}$$

$$\frac{\overline{\Gamma/x \cup x : (a \Rightarrow b) \vdash x : (a \Rightarrow b)}}{\Gamma \vdash l\langle x\rangle : (l(a \Rightarrow b)\Rightarrow)}\ \textit{abstraction}$$

$$\frac{\overline{\Gamma \vdash M : (a \Rightarrow b)} \qquad \overline{\Gamma \vdash N : (c \Rightarrow d)}}{\Gamma \vdash \ \textit{in}\ M\, ;N : (a \dotplus (b \gg c) \Rightarrow d \dotplus (b \ll c))\ \textit{iff}\ \textit{loc}((b \gg c)) \cap \textit{loc}((b \ll c)) = \emptyset}\ \textit{fusion}$$

Chapter 4

Implementation

4.1 Overview

The research was undertaken through both theoretical and practical means, and most of the progress was captured through an empirical testing of the proposed algorithms in a fresh Haskell implementation of the *Evaluator, Parser, Type-Checker* and auxiliary modules i.e. *WEB-FMCt* and *Latex-converter*. The intuition behind the FMC and FMC_t is closely tied to experimental analysis and testing. The experimental process, together with *tagged* iterations are documented on the project's public GitHub page, and are open to consultation. To maintain consistency across the project, all the development has been implemented in Haskell. For reproducibility the builds have been written using *NIX* and further containerised. Although the dissertation focuses on the theoretical nature of the Type Checker, much consideration has been given to the way in which the software solution was developed to allow for ease development and expansion. For an overview of the set-up see Figure 4.1.

4.2 Haskell Implementation

Haddock Documentation

A legible, and documented coding style was adopted, that can be automatically parsed by *Haddock*, the documentation generator for *Haskell* code. The documented, code should enable easy refactoring, maintenance, and improved code comprehension. Haddock documentation can be consulted inside a browser, and offers quick searching features, that allows for fast navigation. In the event of a push of the library to *Stackage* (the central repository for Haskell libraries), the documentation of the code for any successful library. For a view of the documentation website, refer to Figure 4.2.

Parser Module

Essential to the process was the development of an easily editable and maintainable parser. As can be seen in the Listing 5.2 the use of the *Parsec* library and parser combinators, allowed for a legible implementation, that can be further customised and extended as the FMC_t language develops and matures.

Web-Interface

A basic web interface *FMCt-WEB* was set up to allow for easy interaction with the FMC_t and its type-checker without the need of locally building or installing. The interface makes use of the Haskell *Scotty* library to serve static web pages that are pre-computed on the server-side. The web-pages are built using custom components, set up with the combinator library called *Lucid*. The deployment of the site is done on a free instance of *Heroku* which runs a *Docker* containerised version of the *FMCt-WEB* executable. The testing, build, and deployment of the *Docker* container is done automatically by a *CI/CD* pipeline set up in *GitHub Actions*, which makes sure that the on-line version is up to-date, and working, without any need for maintenance.

Figure 4.1: Architecture of type-checker and implementation.

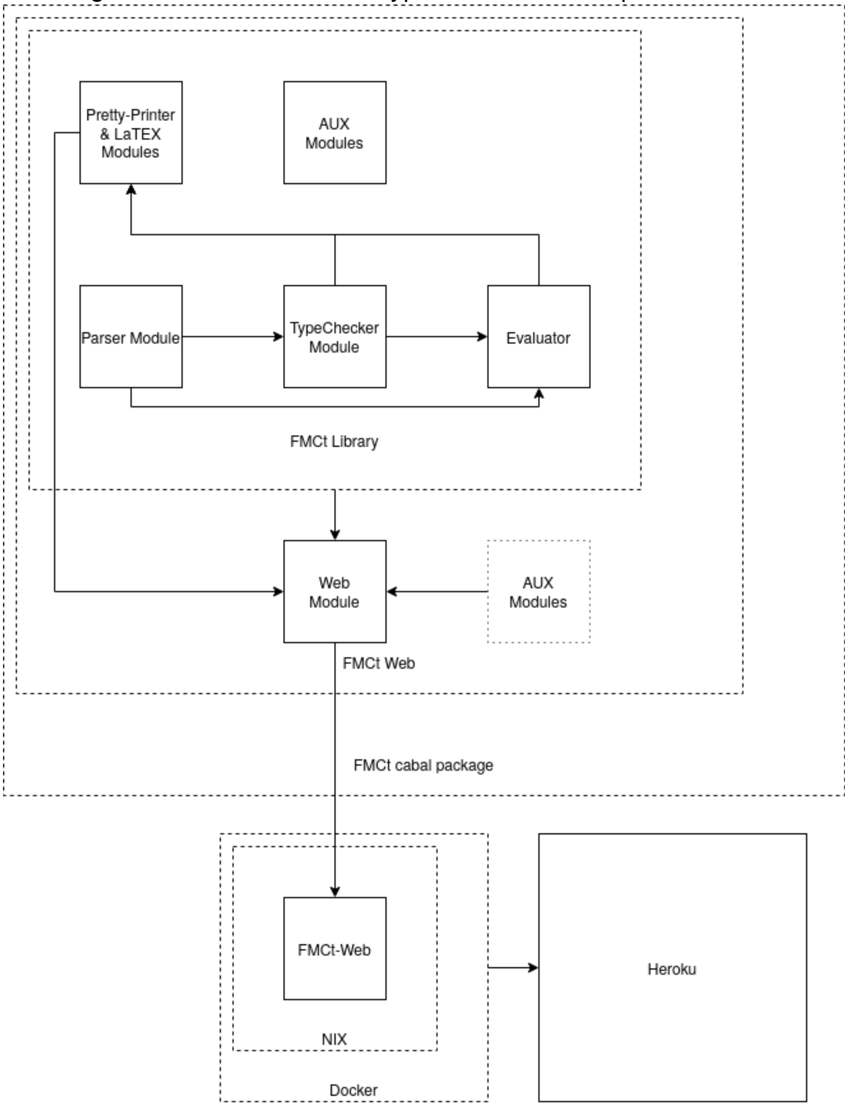
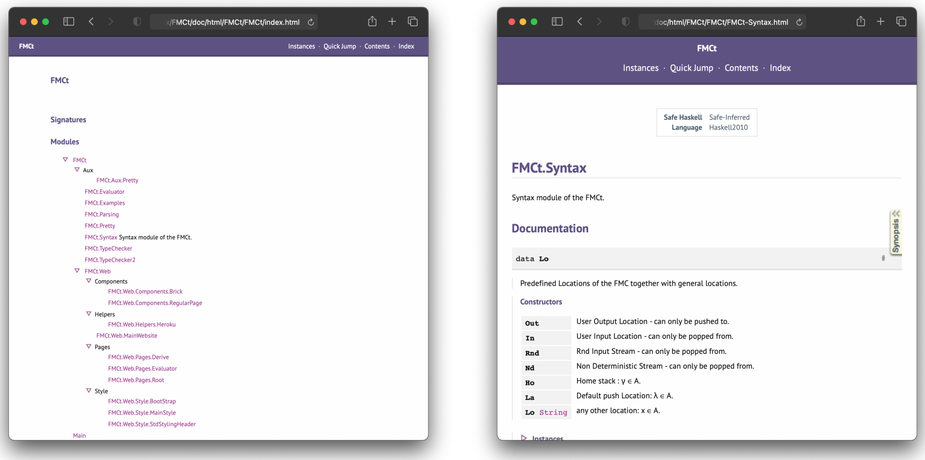


Figure 4.2: Example of Haddock documentation, generated from source code.



The design of the web-interface, although not-aesthetical pleasing, is modular enough to allow for further development. The infrastructure, is robust enough to allow for easy re-deployment (changing provider), or scaling. Finally the web interface is also sufficiently useful to provide a proof-of-concept, and easy interaction with the subject of the dissertation.

Latex Derivation Converter

To allow for the easy type-up of type-checker derivations, a Latex module was developed that translates successful FMC_t typing derivations to latex code. The derivations used in the dissertation, are end products of the module.

Type Checker

The typechecker module allows for the easy building of type derivations based on the laws previously defined. As all the partial functions in the Haskell implementation, the functions make use of an *Either* data-type.

The current inference mechanism relies on a first-collision-first-substitution basis, where each type is cast as the fusion algorithm acts upon it - limiting the amount of types it can infer.

Chapter 5

Critical Analysis

5.1 Theoretical

The thesis' initial scope has been achieved, and can be summarised to the following objectives. The primary scope was to research the feasibility of the proposed type system, and assess if typing each variable is sufficient to derive types.

A secondary scope was to research the feasibility of type inference, and an ergonomic way of integrating types into the *FMC* 's syntax - responded through the FMC_t . The proposed fusion/merging algorithms provide decidable and tractable ways of inferring types without annotations, through the use of fresh type variables. Lastly the research touches on notion of type streams and constant functions.

In addition to the original scope, the research proposed a novel way of integrating constants (the like of *Int* and *Bool*) into the calculus, while maintaining the properties of the original *FMC* .

Further directions into the study of the *FMC* would be to continue and propose an equivalent of type-schemes for the language, together with a generalisation algorithm. Further study can expand and extend into methods of integrating types, and type constructors into the language itself, together with the entailing analysis of the language's properties.

5.2 Practical

From a practical software point of view, the dissertation achieved the delivery of a new modular Haskell implementation of the FMC, expanded with the proposed type-system.

The parser, evaluator, type-checker, web-interface are all written under under an open-source license and are available at the link https://github.com/cstml/FMCt. The web-interface is hosted at https://fmct-web.herokuapp.com/ and there is a functional CI/CD pipeline that integrates, builds, and deploys changes pushed to the repository. The design of the system is thought for ease of refactoring, with the build integrating contemporary methods for deployment.

In terms of further work, the current implementation does not make use of the **general type pattern finding** algorithm from Definition 3.2.3 which would be essential for the inference of any type.

Further limitations are the lack of a type-scheme like behaviour of polymorphism. As type variables are consistent and substituted consistently across terms, once a binder type is established it cannot be polymorphically changed. Thus, if the inference mechanism sets the type variable $if1$ of term if to be Int, then the current implementation will not allow if to accept any other type subsequently.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash 2 ; \star : \epsilon \Rightarrow Int} \qquad \dfrac{\dfrac{}{\Gamma \vdash \star : \epsilon \Rightarrow \epsilon}}{\Gamma \vdash \langle x \colon i1 \Rightarrow j1 \rangle \lambda ; \star : \lambda(\epsilon \Rightarrow \epsilon) \Rightarrow \epsilon}}{\dfrac{\Gamma \vdash \lambda[2 ; \star] ; \langle x \colon i1 \Rightarrow j1 \rangle \lambda ; \star : \epsilon \Rightarrow \epsilon}{\Gamma \vdash 1 ; \lambda[2 ; \star] ; \langle x \colon i1 \Rightarrow j1 \rangle \lambda ; \star : \epsilon \Rightarrow Int}}$$

Bibliography

Barendregt, H.P., 1984. *The lambda calculus: its syntax and semantics*, Studies in logic and the foundations of mathematics.

Barendregt, H.P., 1993. Lambda calculi with types.

Barendregt Henk, 1994. Introduction to lambda calculus.

Baxter, S., 2014. *An ML implementation of the dependently typed lambda calculus*. Ph.D. thesis.

Brady, E., 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming*.

Cardelli, L., 1996. Type systems. *Digital equipment corporation systems research center*.

Cardone, F. and Hindley, J.R., 2016. History of lambda-calculus and combinatory logic.

Church, A., 1932. A set of postulates for the foundation of logic [Online]. [Online], 33(2). Available from: https://doi.org/10.2307/1968337.

Church, A., 1933. A set of postulates for the foundation of logic (second paper). *The annals of mathematics*.

Church, A. and Rosser, J.B., 1936. Some properties of conversion. p.11.

Coquand, T., 2018. Type theory. *The stanford encyclopedia of philosophy*.

Curry, H.B., 1930. Grundlagen der kombinatorischen logik. *American journal of mathematics*, (3).

Damas, L.M.M., 1984. Type assignment in programming languages. p.144.

Gerald Jay Sussman, H., 1997. Structure and interpretation of computer programs, (second edition). *Computers & mathematics with applications*.

Giuseppe Peano, 1889. *Arithmetices principia: nova methodo*.

Heeren, B., Hage, J. and Swierstra, D., 2002. Generalizing Hindley-Milner Type Inference Algorithms.

Heijltjes, W., 2021. Reimagining the lambda calculus. *Unpublished*.

Hindley, R., 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146.

Hughes, John, 2000. Generalizing monads to arrows.

Ingerman, P.Z., 1961. Thunks: a way of compiling procedure statements with some comments on procedure declarations. (1).

J. von Neumann, 1925. Eine axiomatizierung der mengenlehre. *Journal für die reine und angewandte mathematik*.

Kahn, G., 1987. Natural semantics.

The lambda calculus (stanford encyclopedia of philosophy), n.d. [Online]. Available from: https://plato.stanford.edu/entries/lambda-calculus/#Non-Extensionality [Accessed 2021-03-01].

Miller, J.S., 1988. Implementing a scheme-based parallel processing system. *International journal of parallel programming* [Online], 17(5), pp.367–402. Available from: `https://doi.org/10.1007/BF01383881` [Accessed 2021-02-02].

Moggi, E., 1989. Computational lambda-calculus and monads. *[1989] proceedings. fourth annual symposium on logic in computer science*. IEEE Comput. Soc. Press.

Moses Ilyich Schönfinkel, 1924. Über die bausteine der mathematischen logik. *Mathematische annalen,*.

Parigot, M., 1992. A/z-CALCULUS: AN ALGORITHMIC INTERPRETATION OF CLASSICAL NATURAL DEDUCTION. p.12.

Pierce, B., 2002. Types and programming languages.

Plotkin, G. and Power, J., 2002. Notions of computation determine monads. *Foundations of software science and computation structures*. Springer Berlin Heidelberg. Series Title: Lecture Notes in Computer Science.

Plotkin, G. and Power, J., 2004. Computational effects and operations: An overview. 73.

Plotkin, G.D., 1981. A structural approach to operational semantics.

Streicher, T. and Reus, B., 1998. Classical logic, continuation semantics and abstract machines. *J. funct. prog.*, (6).

Appendix

Parser Module Source Code

```
module FMCt.Parsing (
    parseFMC,
    parseType,
    parseFMCtoString,
    parseFMC’,
    PError (..),
) where

import Control.Exception (Exception)
import qualified Control.Exception as E
import Control.Monad (void)
import FMCt.Syntax (Lo (..), T, Tm (..), Type (..))
import Text.ParserCombinators.Parsec

data PError
    = PTermErr String
    | PTypeErr String
    deriving (Show)

instance Exception PError

-- | Main Parsing Function. (Unsafe)
parseFMC :: String -> Tm
parseFMC x = either (E.throw . PTermErr . show) id $ parse term "FMC Parser" x

-- | Main Parsing Function. (Safe)
parseFMC’ :: String -> Either ParseError Tm
parseFMC’ x = parse term "FMCParser" x

-- | Utility Parsing Function used for the FMCt-Web.
parseFMCtoString :: String -> String
parseFMCtoString x = either show show $ parse term "FMCParser" x

-- | Type Parser.
parseType :: String -> T
parseType x = either (E.throw . PTypeErr . show) id $ parse termType "TypeParser" x

-- | Term Parser.
term :: Parser Tm
term = choice $ try <$> [ application, abstraction, variable, star]

-- | Abstraction Parser.
-- Example: lo<x:a>
abstraction :: Parser Tm
abstraction = do
    l <- location
    v <- char ’<’ >> spaces >> many1 alpha <> many alphaNumeric
    t <- spaces >> char ’:’ >> spaces >> absTy <* spaces <* char ’>’
    t2 <- (spaces >> sepparator >> spaces >> term) <|> omittedStar
    return $ B v t l t2
      where
        absTy = try higherType <|> try uniqueType

application :: Parser Tm
application = do
    t <- between (char ’[’) (char ’]’) (term <|> omittedStar)
```

```
    l <- location
    t2 <- (spaces >> sepparator >> spaces >> term) <|> omittedStar
    return $ P t l t2

variable :: Parser Tm
variable = do
    x <- spaces >> (many1 alphaNumeric <|> many1 operators)
    t2 <- (spaces >> sepparator >> spaces >> term) <|> omittedStar
    return $ V x t2

star :: Parser Tm
star =  (eof >> return St)
        <|> (void (char '*') >> return St)

omittedStar :: Parser Tm
omittedStar = (string "") >> return St

location :: Parser Lo
location = choice $
  try <$> [ string "out" >> return Out
          , string "in"  >> return In
          , string "rnd" >> return Rnd
          , string "nd"  >> return Nd
          , string "  "   >> return La
          , string "^"   >> return La
          , string "_"   >> return Ho
          , string "  "   >> return Ho
          , Lo <$> many1 alphaNumeric
          , string ""    >> return La
          ]

-- |  Type
-- Strings beginning with a small letter
-- Example:
-- >> a
-- >> b
variableType :: Parser T
variableType = do
    x <- many1 smallCapsAlpha <> many alphaNumeric
    return $ TVar x

-- | Unique Variable type
-- Just an underscore "_"
-- Example: _
uniqueType :: Parser T
uniqueType = do
    _ <- between spaces spaces $ char '_'
    return $ TVar "inferA" :=> TVar "inferB"  -- this gets changed to a unique variable at typecheck time
    -- TODO: preparser that changes these to fresh vars

-- | Constant Type
-- Strings beginning with a capital letter
-- Example: Int, A, B
constantType :: Parser T
constantType = do
    x <- many1 capsAlpha <> many alphaNumeric
    return $ TCon x

-- | Location Types are Types at a specific location
--
-- Examples
-- >> In(Int)
-- >> In(Int=>Int)
locationType :: Parser T
locationType = do
    l <- location
    t <- between (spaces >> char '(') (spaces >> char ')') termType
    return $ TLoc l t

-- | Vector Types are a list of types.
--
-- Examples
```

```
-- >> a,b,c
-- >> a b c
vectorType :: Parser T
vectorType = do
    t <- between
         (spaces >> (char '('))
         (spaces >> (char ')'))
         (termType `sepBy1` (((char ' ') <* spaces) <|> (spaces *> char ',' <* spaces)))
    return $ TVec t

-- | Empty type is empty
--
-- Examples: e => e,  ()=>e
emptyType :: Parser T
emptyType = do
    _ <-  (spaces >> string "e") <|> string "()"
    return $ TEmp

higherType :: Parser T
higherType = do
  --between (char '(') (char ')') $ do
      t1 <- termType'
      _  <- spaces >> string "=>" >> spaces
      t2 <- termType'
      return $ t1 :=> t2

-- | All Types
termType :: Parser T
termType = try higherType
           <|> try emptyType
           <|> try vectorType
           <|> try locationType
           <|> try constantType
           <|> try variableType
           <|> try uniqueType

-- | Selected types
termType' :: Parser T
termType' = try vectorType
           <|> try emptyType
           <|> try locationType
           <|> try constantType
           <|> try variableType



--------------------------------------------------------------------------------
-- Aux
sepparator :: Parser ()
sepparator = eof <|> void (between spaces spaces (oneOf ".;"))

alpha :: Parser Char
alpha = oneOf $ ['a' .. 'z'] ++ ['A' .. 'Z']

capsAlpha :: Parser Char
capsAlpha = oneOf $  ['A' .. 'Z']

smallCapsAlpha :: Parser Char
smallCapsAlpha = oneOf $  ['a' .. 'z']

numeric :: Parser Char
numeric = oneOf ['0' .. '9']

alphaNumeric :: Parser Char
alphaNumeric = alpha <|> numeric

operators :: Parser Char
operators = oneOf "+-/%=!?"
```

Listing 5.1: Parser module for the FMCt

Typechecker Module Source Code

```
{-# OPTIONS_GHC -Wno-unused-imports #-}
```

```
{-# OPTIONS_GHC -Wno-unused-top-binds #-}
{-# OPTIONS_GHC -Wno-unused-matches #-}
{-# LANGUAGE TupleSections #-}

module FMCt.TypeChecker2
  (
    Derivation(..),
    Judgement,
    Context,
    derive0,
    derive1,
    testD0,
    testD1,
    testD2,
    derive2,
    getTermType,
    pShow',
  ) where
import FMCt.Syntax
import FMCt.Parsing
import FMCt.TypeChecker (
  freshVarTypes,
  splitStream,
  TError(..),
  normaliseT,
  buildContext,
  Operations(..),
  )
import Control.Monad
import FMCt.Aux.Pretty (pShow,Pretty)
import Data.Set
import Control.Exception
import Data.List (nub)

type Context = [(Vv, T)]

type Judgement = (Context, Term, T)

type Term = Tm

type TSubs = (T,T)

data Derivation
    = Star        !Judgement
    | Variable    !Judgement !Derivation
    | Abstraction !Judgement !Derivation
    | Application !Judgement !Derivation !Derivation
    deriving (Show, Eq)

emptyCx :: Context
emptyCx = [("*",mempty :=> mempty)]

normalForm :: T -> T
normalForm = \x -> case x of
  TEmp -> TEmp
  TVar _ -> x
  TCon _ -> x
  TVec [] -> TEmp
  TVec (m:n:p) -> case m of
    TLoc l t -> case n of
      TLoc k t' -> if l < k then TLoc l (normalForm t) <> normalForm (TVec (n:p))
                   else TLoc k (normalForm t') <> normalForm (TVec (m:p))
      _ -> (normalForm n) <> normalForm (TVec (m:p))
    _ -> (normalForm m) <> normalForm (TVec (n:p))
  TVec [x'] -> normalForm x'
  TLoc l t -> TLoc l (normalForm t)
  m :=> n -> normalForm m :=> normalForm n


derive0 :: Term -> Derivation
derive0 term = derive0' freshVarTypes term
  where

```

```
    pBCx = either (const emptyCx) id $ buildContext emptyCx term

    exCx = []
    derive0' :: [T] -> Term -> Derivation
    derive0' stream = \case

      St -> Star (pBCx, St, ty)
        where ty = TEmp :=> TEmp

      x@(V bi t') -> Variable (pBCx', x, ty') nDeriv
        where
          ty = normaliseT $ head stream
          ty' = either (const ty) id $ getType x pBCx
          pBCx' :: [(Vv,T)]
          pBCx' = toList $ fromList pBCx `union` singleton (bi,ty')
          nDeriv = derive0' (tail stream) t'

      x@(B bi bTy lo t') -> Abstraction (nCx, x, ty) nDeriv
        where
          ty = TLoc lo bTy :=> mempty
          nCx = [(bi,bTy)]
          nDeriv = derive0' (tail stream) t'

      xx@(P ptm lo t') -> Application (exCx, xx, ty) deriv nDeriv
        where
          ty = mempty :=> TLoc lo abvT
          deriv = derive0' (tail stream) ptm
          abvT = getDerivationT  deriv
          nDeriv = derive0' (tail stream) t'

derive1 :: Term -> Derivation
derive1 term = snd $ derive1' freshVarTypes pBCx emptySb term
  where
    emptySb = []
    pBCx1    = either (const emptyCx) id $ buildContext emptyCx term -- add constants
    pBCx2    = parseBinders term
    pBCx     = chkUnique $ pBCx1 ++ pBCx2
    chkUnique :: Context -> Context
    chkUnique x = if length x == length (nub $ fmap fst x) then x else error "Variable double bind."

    parseBinders = \case
      St          -> []
      B bi t _ t' -> (bi,t) : parseBinders t'
      P t _ t'    -> parseBinders t ++ parseBinders t'
      V _ t'      -> parseBinders t'

    derive1' :: [T] -> Context -> [TSubs] -> Term -> ([TSubs],Derivation)
    derive1' stream exCx exSb = \case

      St -> (exSb,Star (pBCx, St, ty))
        where ty = TEmp :=> TEmp

      x@(V bi t') -> (,) nSb (Variable (nCx, x, rTy') nDeriv)
        where
          uRes   = derive1' (tail stream) exCx exSb t'
          nDeriv = snd $ uRes
          upSb   = fst $ uRes

          upCx   = applySubsC upSb exCx
          ty     = either (error.show) id $ getType (V bi St) upCx

          upType = getDType nDeriv

          fusion = ty `fuse` upType

          cast   = either (error.show) fst $ fusion
          rTy    = either (error.show) snd $ fusion

          nSb    = upSb ++ cast

          nCx    = applySubsC nSb upCx
          rTy'   = applyTSub  nSb rTy
```

```haskell


      x@(B bi _ lo t') -> (,) nSb (Abstraction (nCx, x, nTy) nDeriv)
        where
          uRes   = derive1' (tail stream) exCx exSb t'
          nDeriv = snd uRes
          upSb   = fst uRes

          upCx   = applySubsC upSb exCx
          upType = getDType nDeriv

          ty'    = either (error.show) id $ getType (V bi St) upCx
          ty     = TLoc lo ty' :=> mempty

          nTy    = either (error.show) (snd) $ ty  `fuse` upType
          cast   = either (error.show) (fst) $ ty' `fuse` upType

          nCx    = applySubsC cast upCx
          nSb    = exSb ++ cast

      xx@(P pTm lo sTm) -> (,) cSb (Application (sCx, xx, nTy') pDeriv sDeriv)
        where
          pRes    = derive1' (tail stream) exCx exSb pTm
          pDeriv  = snd pRes
          pSb     = fst pRes

          sRes    = derive1' (tail stream) exCx pSb sTm
          sDeriv  = snd sRes
          sSb     = fst sRes

          sTy     = getDType sDeriv
          pTy     = getDType pDeriv

          npTy    = applyTSub sSb pTy

          npTy'   = TEmp :=> TLoc lo npTy

          nTy     = either (error.show) snd $ npTy' `fuse` sTy
          cast    = either (error.show) fst $ npTy' `fuse` sTy

          cSb     = sSb ++ cast
          sCx     = applySubsC cSb exCx
          nTy'    = applyTSub cSb nTy

type Result a = Either TError a

-- | Same as "derive1" but safe, and applies all substitutions at the end.
derive2 :: Term -> Result Derivation
derive2 term = do
  let (ppTerm,lTStream) = replaceInfer freshVarTypes term
  bCx           <- pBCx ppTerm                         -- pre build context
  result        <- derive2' lTStream bCx emptySb ppTerm    -- derive
  let derivation = snd result                          -- take final derivation
  let casts      = fst result                          -- take the final casts
  return $ applyTSubsD casts derivation                -- apply them to the derivation and return it

  where
    emptySb = []

    -- | Pre builds the context by adding the constants and the binder types to the context.
    pBCx termR   =  do
      t1 <- buildContext emptyCx termR -- add constants
      let t2 = parseBinders termR
      chkUnique $ t1 ++ t2

    -- | Replace the infer types with new fresh types so they do not overlap.
    replaceInfer :: [T] -> Term -> (Term,[T]) -- ^ Return a Tuple formed out of the new pre-processed term and the stream left.
    replaceInfer stream t = case t of
      St       -> (St     , stream)
      V a n    -> (V a nN, rStr)
        where
```

```
        sStr = splitStream stream
        lStr = fst sStr
        rStr = snd sStr
        nN   = fst $ replaceInfer lStr n
    P p l n -> (P nP l nN, lStr)
      where
        sStr  = splitStream stream
        lStr  = snd sStr
        sStr' = splitStream . fst $ sStr
        str1  = fst sStr'
        str2  = snd sStr'
        nP    = fst $ replaceInfer str1 p
        nN    = fst $ replaceInfer str2 n

    B b ty l n -> (B b nT l nN, rStr)
      where
        sStr = splitStream stream
        lStr = fst sStr
        rStr = snd sStr
        nT = case ty of
          TVar "inferA" :=> TVar "inferB" -> head lStr
          _ -> ty
        nN = fst $ replaceInfer (tail lStr) n

  chkUnique :: Context -> Result Context
  chkUnique x = if length x == length (nub $ fmap fst x)
                then pure x
                else  Left $ ErrOverride "Variable double bind."

  parseBinders = \case
    St            -> []
    B bi t _ t' -> (bi,t) : parseBinders t'
    P t _ t'      -> parseBinders t ++ parseBinders t'
    V _ t'        -> parseBinders t'

  derive2' :: [T] -> Context -> [TSubs] -> Term -> Result ([TSubs],Derivation)
  derive2' stream exCx exSb = \case

    St -> do
      let ty = TEmp :=> TEmp
      let pbC = exCx
      return $ (,) exSb (Star (pbC, St, ty))

    x@(V bi t') -> do
      uRes       <- derive2' (tail stream) exCx exSb t'
      let nDeriv = snd uRes
      let upSb    = fst uRes
      let upCx    = applySubsC upSb exCx
      ty         <- getType (V bi St) upCx
      let upType = getDType nDeriv
      fusion     <- ty `fuse` upType
      let cast    = fst fusion
      let rTy     = snd fusion
      let nSb     = upSb ++ cast
      let nCx     = applySubsC nSb upCx
      let rTy'    = applyTSub  nSb rTy
      return $ (,) nSb (Variable (nCx, x, rTy') nDeriv)


    x@(B bi _ lo t') -> do
      uRes    <- derive2' (tail stream) exCx exSb t'
      let nDeriv = snd uRes
      let upSb    = fst uRes
      let upCx    = applySubsC upSb exCx
      let upType = getDType nDeriv
      ty'        <- getType (V bi St) upCx
      let ty      = TLoc lo ty' :=> mempty
      nTy        <- snd <$> ty  `fuse` upType
      cast       <- fst <$> ty' `fuse` upType
      let nCx     = applySubsC cast upCx
      let nSb     = exSb ++ cast
      return $ (,) nSb (Abstraction (nCx, x, nTy) nDeriv)
```

```
      xx@(P pTm lo sTm) -> do
        pRes        <- derive2' (tail stream) exCx exSb pTm
        let pDeriv  = snd pRes
        let pSb     = fst pRes
        sRes        <- derive2' (tail stream) exCx pSb sTm
        let sDeriv  = snd sRes
        let sSb     = fst sRes
        let sTy     = getDType sDeriv
        let pTy     = getDType pDeriv
        let npTy    = applyTSub sSb pTy
        let npTy'   = TEmp :=> TLoc lo npTy
        nTy         <- snd <$> npTy' `fuse` sTy
        cast        <- fst <$> npTy' `fuse` sTy
        let cSb     = sSb ++ cast
        let sCx     = applySubsC cSb exCx
        let nTy'    = applyTSub cSb nTy
        return $ (,) cSb (Application (sCx, xx, nTy') pDeriv sDeriv)

testD1 :: String -> IO ()
testD1 = putStrLn . pShow . derive1 . parseFMC

testD2 :: String -> IO ()
testD2 str = do
  term       <- return $ parseFMC str
  derivation <- return $ derive2 term
  either (putStrLn . show) (putStrLn) $ pShow <$> derivation

testD0 :: String -> IO ()
testD0 = putStrLn . pShow . derive0 . parseFMC

merge :: [TSubs]          -- ^ Substitutions to be made in both types.
        -> T              -- ^ The consuming Type.
        -> T              -- ^ The merged Type.
        -> ([TSubs],T,T)  -- ^ The result containing: (new list of substitutions,
                          -- unmerged types remaining from the consuming type,
                          -- unmerged types remaining from the merged type).
merge exSubs x y =
  let
    x' = normalForm . normaliseT . (applyTSub  exSubs) $ x -- we use the already subtituted form when consuming
    y' = normalForm . normaliseT . (applyTSub  exSubs) $ y -- for both terms
  in
    case x' of
      TEmp -> case y' of
        TVar _ ->  ((y',mempty):exSubs,mempty,y')
        _      ->  (exSubs,mempty,y') -- mempty doesn't change anything else

      TVec [] -> merge exSubs TEmp y

      TCon _ -> case y' of
        TEmp              -> (exSubs,x',mempty)
        TVec []           -> (exSubs,x',mempty)
        TCon _            -> if x' == y' then (exSubs, mempty, mempty) else (exSubs,x',y')
        t1 :=> t2         -> (exSubs,x',y')
        TVar _            -> ((y',x') : exSubs, mempty, mempty)
        TLoc _ _          -> (exSubs,x',y')
        TVec (yy': yys') -> (finalSubs,finalX,remainY <> finalY)
          where
            (interSubs,interX,remainY) = merge exSubs x' yy'
            (finalSubs,finalX,finalY) = merge interSubs interX (TVec yys')

      TVar _ -> case y' of
        TVar _            -> if x' == y' then (exSubs,mempty,mempty) else ((x',y'):exSubs, mempty,mempty)
        _                 -> ((x',y'):exSubs, mempty, mempty)

      TLoc xl' xt' -> case y' of
        TEmp              -> (exSubs,x',mempty)
        TVec []           -> (exSubs,x',mempty)
        TCon _            -> (exSubs,x',y) -- home row and locations don't interact
        TVar _            -> (exSubs,x',y) -- home row variable and locations don't interact
        TVec (yy': yys') -> (finalSubs,finalX,remainY <> finalY)
          where
```

```
            (interSubs,interX,remainY) = merge exSubs x' yy'
            (finalSubs,finalX,finalY) = merge interSubs interX (TVec yys')

        TLoc yl' yt'      -> if xl' == yl' then (finalSubs, TLoc xl' finalX', TLoc yl' finalY')
                             else (exSubs,x',y')
                               where (finalSubs, finalX', finalY') = merge exSubs xt' yt'
        _ :=> _           -> (exSubs,x',y')

      TVec (xx':xxs') -> case y' of
        TEmp              -> (exSubs,x',mempty)
        TVec []           -> (exSubs,x',mempty)
        TVec (_:_)        -> (finalSubs, interXX' <> finalXXs', finalY')
                               where
                                 (interSubs, interXX', interY')  = merge exSubs xx' y'
                                 (finalSubs, finalXXs', finalY') = merge interSubs (TVec xxs') interY'
        _                 -> (finalSubs, interXX' <> finalXXs', finalY')
                               where
                                 (interSubs, interXX', interY')  = merge exSubs xx' y'
                                 (finalSubs, finalXXs', finalY') = merge interSubs (TVec xxs') interY'

      ix' :=> ox' -> case y' of
        TEmp              -> (exSubs,x',mempty)
        TVec []           -> (exSubs,x',mempty)
        TCon _            -> (exSubs,x',y')
        TVar _            -> ((y',x'):exSubs,mempty,mempty)
        TLoc _ _          -> (exSubs,x',y')
        TVec (yy':yys')   -> (finalSubs, finalX', interYY' <> finalYY')
                               where
                                 (interSubs, interX', interYY') = merge exSubs x' yy'
                                 (finalSubs, finalX', finalYY') = merge interSubs interX' (TVec yys')

        iy' :=> oy'       -> if x'' == y'' then (exSubs, mempty, mempty)
                             else if (finalSubs, finalL, finalR) == (finalSubs, TEmp, TEmp)
                                  then (finalSubs, mempty, mempty)
                                  else (exSubs, x'', y'')
                               where
                                 x'' = normalForm x'
                                 y'' = normalForm y'
                                 (intSubs,   leftIX',  leftIY' ) = merge exSubs  ix' iy'
                                 (finalSubs, rightIX', rightIY') = merge intSubs ox' oy'
                                 finalL                          = normaliseT $ leftIX'  <> leftIY'
                                 finalR                          = normaliseT $ rightIX' <> rightIY'

-- | Assess if two terms have no common unsaturated location
diffLoc :: T -> T -> Bool
diffLoc x y = (loc' x `intersection` loc' y) == empty
  where
    loc' = loc . normaliseT . normalForm

loc :: T -> Set Lo
loc = \case
  TEmp -> empty
  TVec [] -> empty
  TCon _ -> singleton Ho
  TVar _ -> singleton Ho
  _ :=> _ -> singleton Ho
  TVec (x:xs) -> loc x `union` loc (TVec xs)
  TLoc l _ -> singleton l

fuse :: T -> T -> Either TError ([TSubs],T)
fuse = \case
  x@(xi :=> xo) -> \case
    y@(yi :=> yo) ->
      let
          res = merge [] yi xo
      in
        case res of
          (subs,rY,TEmp) -> pure $ (,) subs ((xi <> rY) :=> yo)
```

```
          (subs,TEmp,rX) -> pure $ (,) subs (xi :=> (yo <> rX))
          (subs,rX,rY)   -> if diffLoc rX rY
                              then Right $ (,) subs ((xi <> rY) :=> (yo <> rX))
                              else Left . ErrFuse $ "cannot fuse " ++ show x ++ " "  ++ show y ++ " result: " ++ show res
    y@(TVar _) -> Right ([(y,x)],mempty)
    y          -> Left . ErrFuse $ "cannot fuse " ++ show x ++ " and " ++ show y ++ ". Wrong type Types - Use Function Types"
  x -> \y      -> Left . ErrFuse $ "cannot fuse " ++ show x ++ " and " ++ show y

applyTSub :: [TSubs] -> T -> T
applyTSub subs ty = normaliseT $ aux subs ty
  where
    aux = \case
      [] -> id
      xx@((xi,xo):xs) -> \case
        TEmp -> TEmp
        y@(TCon _ ) -> y
        TLoc l t -> TLoc l (applyTSub xx t)
        TVec y -> TVec $ applyTSub xx <$> y
        yi :=> yo -> applyTSub xx yi :=> applyTSub xx yo
        y@(TVar _) -> if y == xi then applyTSub xs xo else applyTSub xs y

getType :: Term -> Context -> Either TError T
getType = \case
    t@(V b St) -> \case
        [] -> Left $ ErrUndefT $
              mconcat [ "Cannot Find type for binder: ", show b
                      , " in context. Have you defined it prior to calling it?" ]
        ((b', ty) : xs) -> if b == b' then pure ty else getType t xs
    St -> \_ -> pure $ mempty :=> mempty
    t -> \_ -> Left . ErrNotBinder $ mconcat ["Attempting to get type of:", show t]

getDType :: Derivation -> T
getDType = \case
  Star        (_,_,t)     -> t
  Variable    (_,_,t) _   -> t
  Abstraction (_,_,t) _   -> t
  Application (_,_,t) _ _ -> t

setDType :: Derivation -> T -> Derivation
setDType d t = case d of
  Star        (a,b,_)     -> Star (a,b,t)
  Variable    (a,b,_) c   -> Variable (a,b,t) c
  Abstraction (a,b,_) c   -> Abstraction (a,b,t) c
  Application (a,b,_) c e -> Application (a,b,t) c e

getContext :: Derivation -> Context
getContext = \case
  Star        (c,_,_)     -> c
  Variable    (c,_,_) _   -> c
  Abstraction (c,_,_) _   -> c
  Application (c,_,_) _ _ -> c

setContext :: Derivation -> Context -> Derivation
setContext = \case
  Star        (c,a,b)     -> \c' -> Star        (c',a,b)
  Variable    (c,a,b) n   -> \c' -> Variable    (c',a,b) n
  Abstraction (c,a,b) n   -> \c' -> Abstraction (c',a,b) n
  Application (c,a,b) u r -> \c' -> Application (c',a,b) u r

setContextR :: Derivation -> Context -> Derivation
setContextR = \case
  Star        (c,a,b)     -> \c' -> Star        (c',a,b)
  Variable    (c,a,b) n   -> \c' -> Variable    (c',a,b) (setContextR n c')
  Abstraction (c,a,b) n   -> \c' -> Abstraction (c',a,b) (setContextR n c')
  Application (c,a,b) u r -> \c' -> Application (c',a,b) (setContextR u c') (setContextR r c')

applyTSubsD :: [TSubs] -> Derivation -> Derivation
applyTSubsD subs = subCx subs . subTy subs
  where
    subCx :: [TSubs] -> Derivation -> Derivation
    subCx s d = do
```

```
      let cx = getContext d
      let nc = applySubsC s cx
      setContextR d nc

    subTy :: [TSubs] -> Derivation -> Derivation
    subTy s d = case d of
      Star _                   -> d
      Variable    (a,b,t) n    -> Variable    (a,b, applyTSub s t) (subTy s n)
      Abstraction (a,b,t) n    -> Abstraction (a,b, applyTSub s t) (subTy s n)
      Application (a,b,t) p n -> Application (a,b, applyTSub s t) (subTy s p) (subTy s n)

getTermType :: Term -> Result T
getTermType t = do
  deriv <- derive2 t
  return $ getDType deriv

applyDCxSubs :: [TSubs] -> Derivation -> Derivation
applyDCxSubs s d = res
  where
    ctx    = getContext d
    newCtx = applySubsC s ctx
    res    = setContext d newCtx

applySubsC :: [TSubs] -> Context -> Context
applySubsC x y = (\(b,bt) -> (b, applyTSub x bt)) <$> y

allCtx :: Derivation -> Context
allCtx x = case x of
  Star _            -> getContext x
  Variable _ _      -> getContext x
  Application _ u r -> getContext x ++ allCtx u  ++ allCtx r
  Abstraction _ d   -> getContext x ++ allCtx d

getDerivationT :: Derivation -> T
getDerivationT = \case
  Star (_,_,t)            -> t
  Variable (_,_,t)     _   -> t
  Application (_,_,t) _ _ -> t
  Abstraction (_,_,t) _   -> t

setDerivationT :: Derivation -> T -> Derivation
setDerivationT = \case
  Star        (a,b,t)       -> \t' -> Star        (a,b,t')
  Variable    (a,b,t) n     -> \t' -> Variable    (a,b,t') n
  Application (a,b,t) u r   -> \t' -> Application (a,b,t') u r
  Abstraction (a,b,t) n     -> \t' -> Abstraction (a,b,t') n

getLocation :: Term -> Lo
getLocation = \case
  P _ l _ -> l
  B _ _ l _ -> l
  x -> error $ "should't be reaching for location in term: " ++ show x ++ ".This should never
    happen."

-- Show Instance
-- Inspired by previous CW.
instance Pretty Derivation where
    pShow d = unlines (reverse strs)
      where
        (_, _, _, strs) = showD d
        showT :: T -> String
        showT = pShow
        showC :: Context -> String
        showC =
            let sCtx (x, t) = show x ++ ":" ++ showT t ++ ", "
             in \case
                    [] -> []
                    c -> (flip (++) " ") . mconcat $ sCtx <$> c
        showJ :: Judgement -> String
        showJ (cx, n, t) = mconcat $ showC cx : "|- " : pShow n : " : " : showT t : []
        showL :: Int -> Int -> Int -> String
        showL l m r = mconcat $ replicate l ' ' : replicate m '-' : replicate r ' ' : []
        showD :: Derivation -> (Int, Int, Int, [String])
```

```
        showD (Star j) = (0, k, 0, [s, showL 0 k 0]) where s = showJ j; k = length s
        showD (Variable j d') =  addrule (showJ j) (showD d')
        showD (Abstraction j d') = addrule (showJ j) (showD d')
        showD (Application j d' e) = addrule (showJ j) (sidebyside (showD d') (showD e))
        addrule :: String -> (Int, Int, Int, [String]) -> (Int, Int, Int, [String])
        addrule x (l, m, r, xs)
            | k <= m =
                (ll, k, rr, (replicate ll ' ' ++ x ++ replicate rr ' ') : showL l m r : xs)
            | k <= l + m + r =
                (ll, k, rr, (replicate ll ' ' ++ x ++ replicate rr ' ') : showL ll k rr : xs)
            | otherwise =
                (0, k, 0, x : replicate k '-' : [replicate (- ll) ' ' ++ y ++ replicate (- rr) ' ' | y <- xs])
          where
            k = length x; i = div (m - k) 2; ll = l + i; rr = r + m - k - i
        extend :: Int -> [String] -> [String]
        extend i strs' = strs' ++ repeat (replicate i ' ')
        sidebyside :: (Int, Int, Int, [String]) -> (Int, Int, Int, [String]) -> (Int, Int, Int, [String])
        sidebyside (l1, m1, r1, d1) (l2, m2, r2, d2)
            | length d1 > length d2 =
                (l1, m1 + r1 + 2 + l2 + m2, r2, [x ++ "  " ++ y | (x, y) <- zip d1 (extend (l2 + m2 + r2) d2)])
            | otherwise =
                (l1, m1 + r1 + 2 + l2 + m2, r2, [x ++ " " ++ y | (x, y) <- zip (extend (l1 + m1 + r1) d1) d2])


pShow' :: Derivation -> String
pShow' d = unlines (reverse strs)
  where
    (_, _, _, strs) = showD d
    showT :: T -> String
    showT = pShow
    showJ :: Judgement -> String
    showJ (cx, n, t) = mconcat $ "    " : "|- " : pShow n : " : " : showT t : []
    showL :: Int -> Int -> Int -> String
    showL l m r = mconcat $ replicate l ' ' : replicate m '-' : replicate r ' ' : []
    showD :: Derivation -> (Int, Int, Int, [String])
    showD (Star j) = (0, k, 0, [s, showL 0 k 0]) where s = showJ j; k = length s
    showD (Variable j d') =  addrule (showJ j) (showD d')
    showD (Abstraction j d') = addrule (showJ j) (showD d')
    showD (Application j d' e) = addrule (showJ j) (sidebyside (showD d') (showD e))
--          showD (Fusion j d' e) = addrule (showJ j) (sidebyside (showD d') (showD e))
    addrule :: String -> (Int, Int, Int, [String]) -> (Int, Int, Int, [String])
    addrule x (l, m, r, xs)
        | k <= m =
            (ll, k, rr, (replicate ll ' ' ++ x ++ replicate rr ' ') : showL l m r : xs)
        | k <= l + m + r =
            (ll, k, rr, (replicate ll ' ' ++ x ++ replicate rr ' ') : showL ll k rr : xs)
        | otherwise =
            (0, k, 0, x : replicate k '-' : [replicate (- ll) ' ' ++ y ++ replicate (- rr) ' ' | y <- xs])
      where
        k = length x; i = div (m - k) 2; ll = l + i; rr = r + m - k - i
    extend :: Int -> [String] -> [String]
    extend i strs' = strs' ++ repeat (replicate i ' ')
    sidebyside :: (Int, Int, Int, [String]) -> (Int, Int, Int, [String]) -> (Int, Int, Int, [String])
    sidebyside (l1, m1, r1, d1) (l2, m2, r2, d2)
        | length d1 > length d2 =
            (l1, m1 + r1 + 2 + l2 + m2, r2, [x ++ "  " ++ y | (x, y) <- zip d1 (extend (l2 + m2 + r2) d2)])
        | otherwise =
            (l1, m1 + r1 + 2 + l2 + m2, r2, [x ++ " " ++ y | (x, y) <- zip (extend (l1 + m1 + r1) d1) d2])
```

Listing 5.2: TypeChecker module for the FMCt