The Kernel of Ad Hoc Polymorphism

by

Stephan Boyer

B.Sc. EECS, Massachusetts Institute of Technology, 2013

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2016

Author .... Signature redacted ....
Department of Electrical Engineering and Computer Science
February 9, 2016

Certified by.................... Signature redacted
Christopher J. Terman
Senior Lecturer, Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by.. Signature redacted ....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

The Kernel of Ad Hoc Polymorphism

by

Stephan Boyer

Submitted to the Department of Electrical Engineering and Computer Science on February 9, 2016, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Ad hoc polymorphism allows a value to take on multiple types, with a separate definition of the value provided for each type. We offer a new formalization of this old concept as a typed lambda calculus. Motivated by the aspiration of extending System F with ad hoc constraints, we introduce a new mechanism for implicit parameter passing. Putting these ideas together, we present a practical replacement for bounded type quantification with simpler metatheory.

Acknowledgments

I'm enormously grateful to Chris Terman, my thesis supervisor. Dr. Terman was immeasurably patient with me as I slowly developed the ideas in this thesis. Chris, I really appreciate your unending support.

Thanks to everyone in the Programming Languages and Verification Group at CSAIL for your company and for confirming my long-held suspicion that software can be formally verified in practice, not just in theory. I especially want to thank Adam Chlipala and C.J. Bell for teaching me how to use the Coq interactive theorem prover and introducing me to various formal systems.

My undergraduate advisor, Hal Abelson, guided me on my path from freshman to graduate student. Hal, it's an honor to have known you over the years. My current academic advisor, Vikash Mansinghka, urged me not to take on more work than I can manage—rightly so. Thank you, Vikash, for your valuable wisdom.

I'm grateful to many others for their encouragement and guidance. These include, but are certainly not limited to, Gustavo Goretkin, Katie Szeto, Divya Bajekal, Anne Hunter, Vera Sayzew, Kyle Gilpin, Erin Shea, my friends and colleagues from the 5W hall of East Campus, and my family.

Contents

List of Figures

Chapter 1

Introduction

A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

—Benjamin C. Pierce in [4]

The challenge for a type system is to be expressive enough to type any program one might reasonably want to write, yet simple enough for a programmer to use effectively. Polymorphism is a prevailing source of both expressivity and complexity in type systems. Suppose we wish to write a generic function

$$\mathit{square}\ x \triangleq x \times x$$

defined over any kind of value that can be "multiplied" by itself. The work presented in this thesis is motivated by a simple question: what type should we ascribe to *square*?

To make sense of this question, we must have some mechanism for defining multiplication over various types. One would expect $\times$ to be implemented differently for integers and matrices, for example. The ability for multiple definitions of $\times$ to coexist in the same scope and be disambiguated by their type signatures is called *ad hoc polymorphism*.[1] Our first contribution is to formalize ad hoc polymorphism in the lambda calculus, the *lingua franca* of type theory research.

The simply typed lambda calculus of §2.2 can only ascribe monomorphic types to *square*. For example, we can prove the judgment

$$\times : \mathit{Integer} \to \mathit{Integer} \to \mathit{Integer} \vdash \mathit{square} : \mathit{Integer} \to \mathit{Integer},$$

which informally reads "assuming $\times$ takes two *Integer*s and returns an *Integer*,[2] *square* takes an *Integer* and returns an *Integer*." Symmetrically, we can also prove

$$\times : \mathit{Matrix} \to \mathit{Matrix} \to \mathit{Matrix} \vdash \mathit{square} : \mathit{Matrix} \to \mathit{Matrix},$$

but there is no way to express that *square* can be used on *any* type for which there is a binary $\times$ operator.

Parametric polymorphism allows types to be parametrized by other types. System F, described in §2.3, extends the simply typed lambda calculus with universal quantification, the most common form of parametric polymorphism. Suppose we define *square* to accept a *multiply* function as well as the operand to be squared:

$$\mathit{square}\ \mathit{multiply}\ x \triangleq \mathit{multiply}\ x\ x$$

Then in System F[3] we can show

$$\vdash \mathit{square} : \forall X.\, (X \to X \to X) \to X \to X.$$

Thus, System F can successfully type *square* as a generic function, but at the cost of adding *multiply* as a formal parameter. For larger programs, the ceremony of

[1]Ad hoc polymorphism is also called *overloading*.

[2]More formally, $\times$ takes one *Integer* and returns a function which maps the second *Integer* to the final *Integer*. Functions in the lambda calculus only accept a single formal parameter. Polyadic and variadic functions are emulated by currying.

[3]Technically, System F as presented in §2.3 would require that *square* also take the type of the operand as a formal parameter. We elect to use a simplified presentation here, deferring the details of the formalism to Chapter 2.

explicitly abstracting over every such function is too cumbersome. Our second contribution is to introduce a mechanism to mitigate this inconvenience.

Subtyping is the object-oriented approach to polymorphism. We can represent an object which can be multiplied by itself as a record containing a value and a *multiply* method. Such a record can be given the recursive type[4]

$$T \triangleq \mu X \,.\, \{x : \top, \mathit{multiply} : X \to X \to X\}\,.$$

Now we can define[5] *square* as

$$\mathit{square}\ x \triangleq x.\mathit{multiply}\ x\ x$$

and show

$$\vdash \mathit{square} : T \to T.$$

We can define a subtyping relation $<:$ over types, and any value whose type is a subtype of T can be squared. This may seem like a success, but it has several disadvantages. First, the metatheory is significantly more complex. We have to introduce the concepts of records, recursive types, subtyping, etc. Second, it is not always sensible to package a polyadic operator as a method of one of its operands. Finally, *square* is now "lossy": $\mathit{square}\ x$ where $x : S <: T$ is an expression of type T, not S. The details of S not shared by T are forgotten.

That last concern is remedied by *bounded quantification.* The idea is to combine subtyping with parametric polymorphism, such that we can show

$$\vdash \mathit{square} : \forall S <: T \,.\, S \to S.$$

With bounded quantification, $\mathit{square}\ x$ where $x : S <: T$ has type S, not T. No type information is lost, at the expense of additional metatheoretic complexity.

[4]This definition is somewhat problematic because any subtype S of T would require that its *multiply* method accept values of type T, not just S. This is due to *contravariance,* another intricacy of subtyping.

[5]The notation $x.\mathit{multiply}$ means the field *multiply* of record x.

Type classes and implicits, from Haskell and Scala, respectively, offer compelling solutions for ad hoc polymorphism. We believe ours is a more fundamental "kernel" which these heavier frameworks approximate.

Chapter 2 reviews the background knowledge on which our work is built. Chapter 3 describes our approach to ad hoc polymorphism. Chapter 4 introduces a mechanism for implicit parameter passing to make programming with polymorphism more convenient in practice. Finally, Chapter 5 summarizes our contributions and suggests directions for future work.

Chapter 2

Prerequisites

This chapter surveys the main ideas upon which our work builds. We use c to stand for constants, s and t for arbitrary terms, x, y, z, and w for variables, S, T, and P for types, X and Y for type variables, and Γ for contexts. The names of types are capitalized, and the names of terms are lowercase.

2.1 The untyped lambda calculus (λ)

The lambda calculus is a model of computation developed by Alonzo Church in the 1930s. Originally intended to be a logical system, it was proven inconsistent by [1] and thus unsuitable as a foundation of mathematics—but it found other purpose. Church used it in [2] to prove the impossibility of constructing an algorithm which correctly decides for any given statement whether it is provable in first-order logic. Peter Landin discovered its use as a programming language in [3], and it has been the kernel of functional programming languages ever since.

2.1.1 Syntax

The syntax of the lambda calculus is given in Figure 2-1. Every *term* is understood to be a function, and there are three syntactic forms. A variable such as x is a term. An abstraction $\lambda x \,.\, t$ of a variable x from a term t is a term. Finally, the application

$t_1\ t_2$ of one term to another is a term. The body of an abstraction extends as far to the right as possible, meaning $\lambda x\ .\ y\ z$ is interpreted as $\lambda x\ .\ (y\ z)$. Application is left-associative, meaning $t_1\ t_2\ t_3$ is interpreted as $(t_1\ t_2)\ t_3$.

t ::=		(term)
	x	(variable)
	$\lambda x\ .\ t$	(abstraction)
	$t\ t$	(application)

Figure 2-1: Syntax of terms for λ

For example, the term $\lambda x\ .\ x$ represents the identity function. With parentheses to clarify the precedence, the term $(\lambda x\ .\ x)\ y$ is the identity function applied to the variable y. The term $\lambda x\ .\ \lambda y\ .\ x$ is a function with formal parameter x, returning a function which ignores its parameter and returns x.[1]

2.1.2 Free and bound variables

A variable x is *bound* if it appears as the formal parameter of an abstraction $\lambda x.t$. A *free* variable is one which is not bound. More formally, the free variables of a term are defined inductively as

1. $free\,(x) = \{x\}$

2. $free\,(\lambda x\ .\ t) = free\,(t) - \{x\}$

3. $free\,(t_1\ t_2) = free\,(t_1) \cup free\,(t_2)$

where $-$ and $\cup$ are set difference and union, respectively.

2.1.3 Substitution

The notion of substitution turns out to be both subtle and paramount in our work, so we pay special attention to the details here. Substitution is tricky is because

[1]While not formally correct, it is useful in practice to think of this term as a function accepting two arguments, x and y, and returning x.

two things can go wrong:

1. A bound variable becomes free or bound to a different binder.
2. A free variable becomes bound.[2]

Thus, substitution must be done carefully such that bindings (and the lack thereof) are preserved. The rules for substitution are given in Figure 2-2.

$$\boxed{\begin{aligned} x\,[x \mapsto s] &= s && \\ y\,[x \mapsto s] &= y && \text{if } y \neq x \\ (\lambda y\ .\ t)\,[x \mapsto s] &= \lambda y\ .\ t\,[x \mapsto s] && \text{if } y \neq x \text{ and } y \notin \mathit{free}(s) \\ (t_1\ t_2)\,[x \mapsto s] &= t_1\,[x \mapsto s]\ \ t_2\,[x \mapsto s] && \end{aligned}}$$

Figure 2-2: Capture-avoiding substitution

The third rule is not "total", meaning not every case is covered. For example, $\lambda a\ .\ x\,[x \mapsto a]$ is not defined above. The resolution is to rename bound variables to "fresh" variables, such that the meaning of the terms is preserved.[3] Since the set of variables is infinite, there will always be an unused variable available to use. For example, $(\lambda a\ .\ x)\,[x \mapsto a]$ can be rewritten as $(\lambda b\ .\ x)\,[x \mapsto a]$, which becomes $\lambda b\ .\ (x\,[x \mapsto a])$ and finally $\lambda b\ .\ a$.

2.1.4 Operational semantics

A *redex*[4] is a term of the form $(\lambda x\ .\ t_1)\ t_2$. We can rewrite this term by substituting t_2 for x in t_1. For example, $(\lambda x\ .\ x)\ \ y$ can be rewritten as y. Rewriting redexes in this way is called *beta reduction*.

It is natural to ask: in what order are redexes in a term reduced? There are several strategies. We will describe a *call-by-value strategy* here; the choice of

[2]This is called *variable capture.*

[3]This is called *alpha conversion.*

[4]*Redex* stands for *reducible expression.*

reduction strategy is inconsequential for our work. The operational semantics is given in Figure 2-3.

For our purposes, a redex will be reduced only when it is in an outermost position and its right side cannot be reduced any further. Here, *outermost* means the redex is not in the body of any abstraction. In the term $t_1\ (t_2\ (\lambda x\ .\ t_3))$, for example, the subterms t_1, t_2, and $\lambda x\ .\ t_3$ are outermost, but t_3 is not. In the case that both sides of an outermost application are redexes, we choose to reduce the left side first. With this reduction strategy, arguments to functions are evaluated before they are substituted into the function body. This strategy is *strict*, meaning that an argument is always evaluated before the body of the function, even if it is never referenced.

To demonstrate beta reduction, consider the term $(\lambda x\ .\ (\lambda y\ .\ x\ y))\ ((\lambda s\ .\ t)\ u)$. With one step of beta reduction, the term is rewritten as $(\lambda x\ .\ (\lambda y\ .\ x\ y))\ t$. A second step gives $\lambda y\ .\ t\ y$, and no more reductions are possible.

Let v represent any variable x or abstraction $\lambda x\ .\ t$, but not an application $t_1\ t_2$.

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ \text{(E-App1)}$$

$$\frac{t \longrightarrow t'}{v\ t \longrightarrow v\ t'}\ \text{(E-App2)}$$

$$\frac{}{(\lambda x\ .\ t)\ v \longrightarrow t\,[x \mapsto v]}\ \text{(E-AppAbs)}$$

Figure 2-3: Operational semantics for λ

2.2 The simply typed lambda calculus ($\lambda_\rightarrow$)

We can extend the untyped lambda calculus with primitive data types, such as *Integer* and *Boolean*, and functions operating on them, such as *successor i* and *not* b.[5] But this allows for some uncomfortable possibilities, for example, *successor true* or *not* 5. In this section, we review a formal system for ruling out such meaningless expressions.

2.2.1 Syntax of types

The idea is to design a set of rules to ascribe *types* to meaningful terms, leaving no possibility of typing a nonsensical program. The syntax of types is shown in Figure 2-4. We start with a set of base types B, for example $B = \{Integer, Boolean\}$. In addition to those, we can build types for functions such as $Boolean \rightarrow Boolean$ and $Integer \rightarrow Integer \rightarrow Integer$. The $\rightarrow$ operator is right-associative, so $Integer \rightarrow Integer \rightarrow Integer$ is interpreted as $Integer \rightarrow (Integer \rightarrow Integer)$.

$$\boxed{\begin{array}{llll} T ::= & & & \text{(type)} \\ & P & \text{where } P \in B & \text{(base type)} \\ & T \rightarrow T & & \text{(arrow type)} \end{array}}$$

Figure 2-4: Syntax of types for $\lambda_{\rightarrow}$

2.2.2 Syntax of terms

The syntax for terms, given in 2-5, is almost identical to that of the untyped lambda calculus from §2.1.1. There are two differences:

1. There are new terms for constants.

2. Abstractions $\lambda x : T \,.\, t$ now have type annotations for their arguments.

2.2.3 Typing rules

Each syntactic category of terms has a typing rule dictating what type, if any, to ascribe to terms of that category. For example, the typing rule for constants is:

[5]These types and functions can be implemented without any additions to the untyped lambda calculus, but it is convenient to have them as primitives. See §5.2 of [4] for details.

$t ::=$		(term)
	c	(constant of type $P \in B$)
	x	(variable)
	$\lambda x : T \,.\, t$	(abstraction)
	$t\ t$	(application)

Figure 2-5: Syntax of terms for $\lambda_{\rightarrow}$

$$\frac{c \text{ is a constant of type } P}{\Gamma \vdash c : P} \text{ (T-CONST)}$$

The judgment above the line is the premise, and the judgment below is the conclusion. In the case of multiple premises above the line, they are interpreted as a conjunction. Γ is a set of typing assumptions, which can be thought of as a map from variables "in scope" to their types. Here, the judgment $\Gamma \vdash c : P$ means c has type P for any typing assumptions Γ, since Γ was not referenced in any premise. So this rule is understood to mean: *under the premise that c is a constant of type P, we know c : P in any context* Γ.

The rule for variables states that, under the premise that if the context Γ contains the assumption $x : T$, then we know $x : T$ in that context. This rule may seem redundant, but it is necessary to formalize the semantics of Γ:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-VAR)}$$

The rule for abstractions states that if we can prove the body of an abstraction has type T_2 assuming the argument has type T_1, then we know the function has type $T_1 \rightarrow T_2$:

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_2 \,.\, T_2 : T_1 \rightarrow T_2} \text{ (T-ABS)}$$

The syntax $\Gamma, x : T_1$ means Γ amended such that x maps to T_1. If x was already in Γ, then it is overwritten to map to T_1.[6] Note that the T-ABS rule does not

[6]Many treatments of the simply typed lambda calculus require that x not already exist in Γ. The choice of allowing *variable shadowing* is inconsequential for our purposes.

uniquely determine the type for an abstraction. For example, $\lambda x \ . \ x$ has type $\mathit{Boolean} \rightarrow \mathit{Boolean}$ and also type $\mathit{Integer} \rightarrow \mathit{Integer}$ according to this rule.

Finally, the rule for applications requires that the argument type for the abstraction match the type of the right side of the application:

$$\frac{\Gamma \vdash t_1 : X \rightarrow Y \qquad \Gamma \vdash t_2 : X}{\Gamma \vdash t_1\ t_2 : Y}\ (\text{T-App})$$

The rules for the simply typed lambda calculus are summarized in Figure 2-6. They can be chained to form proofs about the types of programs. For example, here is a proof that $\lambda x : \mathit{Boolean} \ . \ 5$ has type $\mathit{Boolean} \rightarrow \mathit{Integer}$:

$$\frac{\dfrac{5 \text{ is a constant of type } \mathit{Integer}}{\{x : \mathit{Boolean}\} \vdash 5 : \mathit{Integer}}\ (\text{T-Const})}{\vdash \lambda x : \mathit{Boolean} \ . \ 5 : \mathit{Boolean} \rightarrow \mathit{Integer}}\ (\text{T-Abs})$$

$$\frac{c \text{ is a constant of type } P}{\Gamma \vdash c : P}\ (\text{T-Const})$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ (\text{T-Var})$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_2 \ . \ T_2 : T_1 \rightarrow T_2}\ (\text{T-Abs})$$

$$\frac{\Gamma \vdash t_1 : X \rightarrow Y \qquad \Gamma \vdash t_2 : X}{\Gamma \vdash t_1\ t_2 : Y}\ (\text{T-App})$$

Figure 2-6: Typing rules for $\lambda_{\rightarrow}$

Theorem 1 (Uniqueness of types in $\lambda_{\rightarrow}$). *In the simply typed lambda calculus, a term has at most one type.*

Proof. The proof is by induction on typing derivations. The induction principle is that the type of a term is fixed given a context and the term itself. There are two base cases. The T-Const rule is the first base case. Observe that the type of a constant depends only on the constant itself. The second base case is the T-Var rule, which fixes the type of a variable given a context which contains it. For the T-ABS rule, the argument type is fixed because it is written in the term itself (e.g., $\lambda x : Integer \, . \, x$). The return type is fixed by induction on the premise. So the whole arrow type is fixed. Finally, for the T-APP rule, the type of the abstraction and the type of the argument are both fixed by induction, so the resulting type is fixed as the return type of the function. Thus, a term cannot have more than one type. $\square$

2.2.4 Operational semantics

The operational semantics for $\lambda_{\rightarrow}$ is identical to that of the untyped lambda calculus, given in Figure 2-3, except abstractions $\lambda x : T \, . \, t$ now have type annotations. However, there are two interpretations:

1. In the *Curry style*, the operational semantics is defined for all terms, including those which are ill-typed. The type system then eliminates undesirable terms, but typing comes after semantics.

2. In the *Church style*, only well-typed terms are given semantics. It does not make sense to speak of the semantics of an ill-typed expression. Typing comes before semantics.

In our work, we will adopt the Church style by necessity, as semantics will depend on types.

2.3 System F

Recall the function $square \; multiply \; x \triangleq multiply \; x \; x$ from Chapter 1. For any type we choose for $multiply$, we end up with a corresponding type for $square$:

$$\vdash square : (Integer \rightarrow Integer \rightarrow Integer) \rightarrow Integer \rightarrow Integer,$$

$$\vdash square : (Matrix \rightarrow Matrix \rightarrow Matrix) \rightarrow Matrix \rightarrow Matrix,$$

$$\dots$$

These types are identical modulo the choice of operand type. We would like to be able to abstract the operand type out, such that this infinite set of types collapses into one *parametric* type. System F, discovered independently by Jean-Yves Girard in [5] and John C. Reynolds in [6], provides that mechanism. With System F, we can define

$$\mathit{square} \triangleq \lambda X \,.\, \lambda f : X \to X \to X \,.\, \lambda x : X \,.\, f\ x\ x.$$

The X argument is a type, and the f and x parameters are constrained by that type. Written in this way, *square* can be shown to have type

$$\vdash \mathit{square} : \forall X \,.\, (X \to X \to X) \to X \to X.$$

2.3.1 Syntax of types

The syntax of types is shown in Figure 2-7. It is identical to that of the simply typed lambda calculus, except with two new forms:

1. X (type variable)
2. $\forall X \,.\, T$ (universal type)

Universal types are the types of polymorphic values, that is, values which take a type as a parameter. Type variables represent those type parameters on the type level.

2.3.2 Syntax of terms

System F adds two new forms to the syntax of terms:

1. $\lambda X \,.\, t$ (type abstraction)
2. $t\ T$ (type application)

$T ::=$			(type)
	P	where $P \in B$	(base type)
	$T \rightarrow T$		(arrow type)
	X		(type variable)
	$\forall X \,.\, T$		(universal type)

Figure 2-7: Syntax of types for System F

Type abstractions are a new kind of function which take a type as a parameter and return a term. The full syntax is given in Figure 2-8.

$t ::=$		(term)
	c	(constant of type $P \in B$)
	x	(variable)
	$\lambda x : T \,.\, t$	(abstraction)
	$t\ t$	(application)
	$\lambda X \,.\, t$	(type abstraction)
	$t\ T$	(type application)

Figure 2-8: Syntax of terms for System F

2.3.3 Typing rules

The typing rules for System F are given in Figure 2-9. The two new rules are T-TAbs and T-TApp, for type abstraction and application, respectively.

2.3.4 Operational semantics

The operational semantics for System F is given in Figure 2-9. The rules for type application and abstraction take the same form as their term-level counterparts.

$$\frac{c \text{ is a constant of type } P}{\Gamma \vdash c : P} \text{ (T-CONST)}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-VAR)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_2 \,.\, T_2 : T_1 \to T_2} \text{ (T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : X \to Y \qquad \Gamma \vdash t_2 : X}{\Gamma \vdash t_1\ t_2 : Y} \text{ (T-APP)}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \lambda X \,.\, t : \forall X \,.\, T} \text{ (T-TABS)}$$

$$\frac{\Gamma \vdash t : \forall X \,.\, T_1}{\Gamma \vdash t\ T_2 : T_1\,[X \mapsto T_2]} \text{ (T-TAPP)}$$

Figure 2-9: Typing rules for System F

Let v represent any variable x, abstraction $\lambda x : T \,.\, t$, or type abstraction $\lambda X \,.\, t$, but not an application $t_1\ t_2$ or type application $t\ T$.

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \text{ (E-APP1)}$$

$$\frac{t \longrightarrow t'}{v\ t \longrightarrow v\ t'} \text{ (E-APP2)}$$

$$\frac{}{(\lambda x \,.\, t)\ v \longrightarrow t\,[x \mapsto v]} \text{ (E-APPABS)}$$

$$\frac{t \longrightarrow t'}{t\ T \longrightarrow t'\ T} \text{ (E-TAPP1)}$$

$$\frac{}{(\lambda X \,.\, t)\ T \longrightarrow t\,[X \mapsto T]} \text{ (E-TAPPTABS)}$$

Figure 2-10: Operational semantics for System F

Chapter 3

The lambda-cross calculus $(\lambda_\times)$

Recall our goal of ascribing a type to the generic function *square* $x \triangleq x \times x$. First, we must devise a mechanism to allow the $\times$ operator to be defined over various types; this is called *ad hoc polymorphism*. More precisely, it should be allowed for multiple values to be bound to a single name in the same scope, as long as they have different types. In this chapter, we formalize this idea as a typed lambda calculus. To give the system a name, we choose $\lambda_\times$, pronounced *the lambda-cross calculus*. The $\times$ symbol was chosen to suggest an overloaded operator.

3.1 Syntax of types and terms

The typing rules and operational semantics give $\lambda_\times$ its unique personality, whereas the syntax of types and terms is the same as that of the simply typed lambda calculus. For convenience, the syntax is reproduced in Figures 3-1 and 3-2.

$$
\begin{array}{llll}
T ::= & & & \text{(type)} \\
 & P & \text{where } P \in B & \text{(base type)} \\
 & T \rightarrow T & & \text{(arrow type)}
\end{array}
$$

Figure 3-1: Syntax of types for $\lambda_\times$

$t ::=$		(term)
	c	(constant of type $P \in B$)
	x	(variable)
	$\lambda x : T \,.\, t$	(abstraction)
	$t\ t$	(application)

Figure 3-2: Syntax of terms for $\lambda_\times$

3.2 Typing rules

The typing rules for $\lambda_\times$ are syntactically identical to those of the simply typed lambda calculus, with one semantic difference: the interpretation of the context. In the simply typed lambda calculus, the context is a map from variables to types, and a variable must map to at most one type. In $\lambda_\times$, the context is a *multimap*, that is, a structure allowing for a single variable to simultaneously map to multiple types. The syntax $\Gamma, x : T$ adds $x : T$ to the multimap Γ, but does not remove any existing mapping in the case that x was already in Γ.

This subtle difference allows a term to take on multiple types in the same scope. For example, consider the term $\lambda x : \mathit{Integer} \,.\, \lambda x : \mathit{Boolean} \,.\, x$. The innermost x takes on the types *Integer* and *Boolean* simultaneously, and this leads to two distinct type derivations:

$$\dfrac{\dfrac{\dfrac{x : \mathit{Integer} \in x : \mathit{Integer}, x : \mathit{Boolean}}{x : \mathit{Integer}, x : \mathit{Boolean} \vdash x : \mathit{Integer}}\,(\text{T-Var})}{x : \mathit{Integer} \vdash \lambda x : \mathit{Boolean} \,.\, x : \mathit{Boolean} \to \mathit{Integer}}\,(\text{T-Abs})}{\vdash \lambda x : \mathit{Integer} \,.\, \lambda x : \mathit{Boolean} \,.\, x : \mathit{Integer} \to \mathit{Boolean} \to \mathit{Integer}}\,(\text{T-Abs})$$

$$\dfrac{\dfrac{\dfrac{x : \mathit{Boolean} \in x : \mathit{Integer}, x : \mathit{Boolean}}{x : \mathit{Integer}, x : \mathit{Boolean} \vdash x : \mathit{Boolean}}\,(\text{T-Var})}{x : \mathit{Integer} \vdash \lambda x : \mathit{Boolean} \,.\, x : \mathit{Boolean} \to \mathit{Boolean}}\,(\text{T-Abs})}{\vdash \lambda x : \mathit{Integer} \,.\, \lambda x : \mathit{Boolean} \,.\, x : \mathit{Integer} \to \mathit{Boolean} \to \mathit{Boolean}}\,(\text{T-Abs})$$

This is in contrast to the simply typed calculus, in which a term can have at most one type (cf. Theorem 1).

$$\frac{c \text{ is a constant of type } P}{\Gamma \vdash c : P}\ (\text{T-Const})$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ (\text{T-Var})$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_2 \,.\, T_2 : T_1 \rightarrow T_2}\ (\text{T-Abs})$$

$$\frac{\Gamma \vdash t_1 : X \rightarrow Y \qquad \Gamma \vdash t_2 : X}{\Gamma \vdash t_1\ t_2 : Y}\ (\text{T-App})$$

Figure 3-3: Typing rules for $\lambda_\times$

3.3 Type-aware substitution

The operational semantics of $\lambda_\times$ depends on types. We have to modify the mechanics of substitution given in §2.1.3 to be *type-aware*. Now that a variable can simultaneously refer to multiple values of different types, we have to ensure beta reduction does not accidentally apply substitutions to variables which have the right name but are bound to the wrong value. For example, when substituting in the definition of integer multiplication for $\times$, we don't want to affect any instances where $\times$ refers to matrix multiplication. The rules for type-aware substitution are given in Figure 3-4. Just as in Figure 2-2, the rules in Figure 3-4 are partial. In the case that no rule applies, bound variables must be alpha-renamed to fresh variables.

The type-aware substitution rules deserve some explanation. The first thing to notice is that the syntax $t\,[\Gamma | x : X \mapsto s]$ includes a type context Γ and a type annotation for the variable. The meaning of this syntax is "in a capture-avoiding manner, replace any free instances of x in term t with s if the instance can be shown to have unique type X in context Γ." As with typing rules, the context is used to keep track of the variables in scope and their types. For consistency with the typing rules, the context is a multimap. The syntax $\Gamma - x, x : X$ means Γ with any annotations for x replaced with the single annotation $x : X$.

$$
\begin{aligned}
x\,[\Gamma | x : X \mapsto s] &= \begin{cases} s & \text{if } \Gamma \vdash x : X \text{ and } \Gamma \not\vdash x : Y \text{ for all } Y \neq X \\ x & \text{otherwise} \end{cases} \\
y\,[\Gamma | x : X \mapsto s] &= y \text{ if } y \neq x \\
(\lambda y : T \,.\, t)\,[\Gamma | x : X \mapsto s] &= \lambda y : T \,.\, t\,[\Gamma, y : T | x : X \mapsto s] \\
&\quad \text{if } y \neq x \text{ and } y \notin \mathit{free}(s) \\
(z\; w)\,[\Gamma | x : X \mapsto s] &= \\
&\quad z\,[\Gamma - z, z : T_1 \rightarrow T_2 | x : X \mapsto s] \\
&\quad w\,[\Gamma - w, w : T_2 | x : X \mapsto s] \\
&\quad \text{if } \Gamma \vdash z : T_1 \rightarrow T_2 \text{ and } \Gamma \vdash w : T_2 \\
(z\; t)\,[\Gamma | x : X \mapsto s] &= z\,[\Gamma - z, z : T_2 \rightarrow T_2 | x : X \mapsto s]\;\; t\,[\Gamma | x : X \mapsto s] \\
&\quad \text{if } \Gamma \vdash z : T_1 \rightarrow T_2 \text{ and } \Gamma \vdash t : T_2 \\
&\quad \text{and } t \text{ is not a variable} \\
(t\; w)\,[\Gamma | x : X \mapsto s] &= t\,[\Gamma | x : X \mapsto s]\;\; w\,[\Gamma - w, w : T_2 | x : X \mapsto s] \\
&\quad \text{if } \Gamma \vdash t : T_1 \rightarrow T_2 \text{ and } \Gamma \vdash w : T_2 \\
&\quad \text{and } t \text{ is not a variable} \\
(t_1\; t_2)\,[\Gamma | x : X \mapsto s] &= t_1\,[\Gamma | x : X \mapsto s]\;\; t_2\,[\Gamma | x : X \mapsto s] \\
&\quad \text{if } t_1 \text{ and } t_2 \text{ are not variables}
\end{aligned}
$$

Figure 3-4: Type-aware substitution

The first rule specifies that a variable will only be substituted if it can be shown to have the target type and no other type.

$$
x\,[\Gamma | x : X \mapsto s] = \begin{cases} s & \text{if } \Gamma \vdash x : X \text{ and } \Gamma \not\vdash x : Y \text{ for all } Y \neq X \\ x & \text{otherwise} \end{cases}
$$

The second rule is identical to that of the lambda calculus.

$$
y\,[\Gamma | x : X \mapsto s] = y \text{ if } y \neq x
$$

The third rule resembles that of the lambda calculus, with the bound variable

y added to the context in the recursive substitution.

$$(\lambda y : T \,.\, t)\,[\Gamma|x : X \mapsto s] = \lambda y : T \,.\, t\,[\Gamma, y : T|x : X \mapsto s] \text{ if } y \neq x \text{ and } y \notin \mathit{free}(s)$$

The fourth, fifth, and sixth rules give rise to ad hoc polymorphism. In an application, either the left subterm or the right subterm (or both) may be polymorphic, and these rules choose an implementation such that the types link together. For example, in the expression *negate* 3, the fifth rule will ensure the correct version of *negate* is applied if multiple are in scope.

$$\begin{aligned}
(z\ w)\,[\Gamma|x : X \mapsto s] &= z\,[\Gamma - z, z : T_1 \to T_2|x : X \mapsto s]\ w\,[\Gamma - w, w : T_2|x : X \mapsto s] \\
&\qquad \text{if } \Gamma \vdash z : T_1 \to T_2 \text{ and } \Gamma \vdash w : T_2 \\
(z\ t)\,[\Gamma|x : X \mapsto s] &= z\,[\Gamma - z, z : T_2 \to T_2|x : X \mapsto s]\ \ t\,[\Gamma|x : X \mapsto s] \\
&\qquad \text{if } \Gamma \vdash z : T_1 \to T_2 \text{ and } \Gamma \vdash t : T_2 \text{ and } t \text{ is not a variable} \\
(t\ w)\,[\Gamma|x : X \mapsto s] &= t\,[\Gamma|x : X \mapsto s]\ \ w\,[\Gamma - w, w : T_2|x : X \mapsto s] \\
&\qquad \text{if } \Gamma \vdash t : T_1 \to T_2 \text{ and } \Gamma \vdash w : T_2 \text{ and } t \text{ is not a variable}
\end{aligned}$$

The typing rules guarantee that these substitution rules will always find an appropriate implementation for z or w, but they don't guarantee uniqueness. For example, consider the expression *negate x*. If *negate* is implemented for both integers and rationals and x has both an integer and rational definition in scope, then substitution is nondeterministic. If deterministic behavior is required, such expressions should be rejected.

Finally, the seventh rule has the same form as the corresponding rule from capture-avoiding substitution.

$$\begin{aligned}
(t_1\ t_2)\,[\Gamma|x : X \mapsto s] &= t_1\,[\Gamma|x : X \mapsto s]\ \ t_2\,[\Gamma|x : X \mapsto s] \\
&\qquad \text{if } t_1 \text{ and } t_2 \text{ are not variables}
\end{aligned}$$

3.4 Operational semantics

Once the details of type-aware substitution have been fleshed out, the operational semantics for $\lambda_\times$ is straightforward. The semantics is identical to that of the lambda calculus, except the E-AppAbs rule uses type-aware substitution instead of vanilla capture-avoiding substitution.

Let v represent any variable x or abstraction $\lambda x \,.\, t$, but not an application $t_1\ t_2$.

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ \text{(E-App1)}$$

$$\frac{t \longrightarrow t'}{v\ t \longrightarrow v\ t'}\ \text{(E-App2)}$$

$$\frac{}{(\lambda x : T \,.\, t)\ v \longrightarrow t\,[\varnothing | x : T \mapsto v]}\ \text{(E-AppAbs)}$$

Figure 3-5: Operational semantics for $\lambda_\times$

3.5 Example: overloading negation

In $\lambda_\times$, we can define negation (n) over integers (I) and rationals (Q), with separate implementations. With both definitions in scope, suppose we want to negate the integer 5. First, we write the term

$$(\lambda n : I \to I \,.\, (\lambda n : Q \to Q \,.\, n\ 5)\ n_Q)\ n_I,$$

where n_I and n_Q are the implementations of negation for integers and rationals, respectively.

Next, we construct a typing derivation:

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{
          \dfrac{n : I \to I \in n : I \to I, n : Q \to Q}{n : I \to I, n : Q \to Q \vdash n : I \to I}\;(\text{T-Var})
          \quad
          \dfrac{5 \text{ is a constant of type } I}{n : I \to I, n : Q \to Q \vdash 5 : I}\;(\text{T-Const})
        }{n : I \to I, n : Q \to Q \vdash n\,5 : I}\;(\text{T-App})
      }{n : I \to I \vdash \lambda n : Q \to Q \,.\, n\,5 : (Q \to Q) \to I}\;(\text{T-Abs})
      \quad
      n : I \to I \vdash n_Q : Q \to Q
    }{n : I \to I \vdash (\lambda n : Q \to Q \,.\, n\,5)\; n_Q : I}\;(\text{T-App})
  }{\vdash \lambda n : I \to I \,.\, (\lambda n : Q \to Q \,.\, n\,5)\; n_Q : (I \to I) \to I}\;(\text{T-Abs})
  \quad
  \vdash n_I : I \to I
}{\vdash (\lambda n : I \to I \,.\, (\lambda n : Q \to Q \,.\, n\,5)\; n_Q)\; n_I : I}\;(\text{T-App})
$$

To evaluate the term, we first apply the E-AppAbs rule.

$$(\lambda n : Q \to Q \,.\, n_I\, 5)\; n_Q$$

Notice how the innermost n was substituted with n_I. Finally, we apply the E-AppAbs rule once more.

$$n_I\, 5$$

We see that the correct implementation of n was chosen to negate 5.

Chapter 4

The lambda-delta calculus (λ_δ)

Recall the parametrically polymorphic definition of *square* from §2.3:

$$\vdash \lambda X \,.\, \lambda f : X \to X \to X \,.\, \lambda x : X \,.\, f\ x\ x : \forall X.\, (X \to X \to X) \to X \to X$$

As we mentioned in Chapter 1, we'd rather not explicitly pass the multiplication function f at every call site for *square*. In this chapter, we will introduce a calculus which provides a mechanism for implicit parameter passing. We call this system λ_δ, pronounced *the lambda-delta calculus*. It is named after one of its syntactic constructs.

4.0.1 Syntax of terms

The syntax of terms comes from the simply typed lambda calculus, but with a new kind of function called an implicit abstraction $\delta x : T \,.\, t$. The full syntax of terms is given in Figure 4-1.

An implicit abstraction $\delta x : T \,.\, t$ represents a parameter which is automatically retrieved by name from the context. For example, the *square* function might be refactored to use an implicit abstraction to abstract over the multiplication operator:

$$square \triangleq \lambda X \,.\, \delta f : X \to X \to X \,.\, \lambda x : X \,.\, f\ x\ x$$

$t ::=$		(term)
	c	(constant of type $P \in B$)
	x	(variable)
	$\lambda x : T \,.\, t$	(explicit abstraction)
	$\delta x : T \,.\, t$	(implicit abstraction)
	$t\ t$	(application)

Figure 4-1: Syntax of terms for λ_δ

4.0.2 Operational semantics

The operational semantics, shown in Figure 4-2, are nearly identical to the untyped lambda calculus with one exception: two substitutions are performed in the E-AppAbs rule. The first substitution supplies implicit parameters to any implicit abstractions, and the second substitution performs beta reduction.

Let v represent any variable x or abstraction $\lambda x \,.\, t$, but not an application $t_1\ t_2$.

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ \text{(E-App1)}$$

$$\frac{t \longrightarrow t'}{v\ t \longrightarrow v\ t'}\ \text{(E-App2)}$$

$$\frac{}{(\lambda x : T \,.\, t)\ v \longrightarrow t\,[\delta y : T \,.\, t \mapsto t]\,[x \mapsto v]}\ \text{(E-AppAbs)}$$

Figure 4-2: Operational semantics for λ_δ

4.0.3 Syntax of types

λ_δ adds a new kind of type, implicit arrow types $x : T \rightarrow T$. This type is unusual in that it remembers the name of the argument to the abstraction—this is because the argument is implicitly fetched by name. The full syntax of types is given in Figure 4-3.

$T ::=$			(type)
	P	where $P \in B$	(base type)
	$T \rightarrow T$		(explicit arrow type)
	$x : T \rightarrow T$		(implicit arrow type)

Figure 4-3: Syntax of types for λ_δ

4.0.4 Typing rules

The T-IAbs rule for implicit abstraction mirrors the rule for explicit abstractions. Curious, however, is the T-IApp rule for implicit application. It is a rule of *subsumption,* meaning that it recasts a value of one type as having another type (by eliminating the implicit parameter). The full set of rules is given in Figure 4-4.

$$\frac{c \text{ is a constant of type } P}{\Gamma \vdash c : P} \text{ (T-Const)}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{ (T-Var)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_2 \,.\, T_2 : T_1 \rightarrow T_2} \text{ (T-EAbs)}$$

$$\frac{\Gamma \vdash t_1 : X \rightarrow Y \qquad \Gamma \vdash t_2 : X}{\Gamma \vdash t_1\ t_2 : Y} \text{ (T-EApp)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \delta x : T_2 \,.\, T_2 : x : T_1 \rightarrow T_2} \text{ (T-IAbs)}$$

$$\frac{\Gamma \vdash t_1 : x : X \rightarrow Y \qquad \Gamma \vdash x : X}{\Gamma \vdash t_1 : Y} \text{ (T-IApp)}$$

Figure 4-4: Typing rules for λ_δ

Chapter 5

Conclusion

5.1 Summary

We have presented two calculi which combine to offer a minimal kernel of ad hoc polymorphism. We believe our approach to formalizing ad hoc polymorphism in the lambda calculus is novel, and it provides a solid foundation for future research.

5.2 Future work

We have not attempted to construct type inference algorithms for any of the systems presented in this thesis, and type inference would be the natural next step to make these systems easier to use. The author suspects that one or both of $\lambda_\times$ and λ_δ could be compiled down to the simply typed lambda calculus with little effort, where a Damas-Hindley-Milner strategy may yield a tractable type inference solution.

Another direction for further work is investigating the interaction of these calculi with higher-order polymorphism, such as System F_ω. In fact, we suspect that many type systems could be combined with our work to yield interesting research.

References

[1] Kleene, S. C., & Rosser, J. B. (1935). The inconsistency of certain formal logics. *Annals of Mathematics*, 630-636.

[2] Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics, 58*(2), 345-363.

[3] Landin, P. J. (1964). The mechanical evaluation of expressions. *The Computer Journal, 6*(4), 308-320.

[4] Pierce, B. C. (2002). *Types and Programming Languages*. MIT press.

[5] Girard, J. Y. (1972). Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.

[6] Reynolds, J. C. (1974). Towards a theory of type structure. In *Programming Symposium* (pp. 408-425). Springer Berlin Heidelberg.