A SURVEY INTO THE CURRY-HOWARD ISOMORPHISM & TYPE SYSTEMS

by

Phillip Mates

A Senior Honors Thesis Submitted to the Faculty of
The University of Utah
In Partial Fulfillment of the Requirements for the

Honors Degree in Bachelor of Science

In

Computer Science

Approved:

Matthew Might Supervisor	Al Davis Chair, School of Computing
Dianne Leonard College of Engineering Honors Advisor	Thomas G. Richmond Interim Dean, Honors College

December 2011

ABSTRACT

The Curry-Howard Isomorphism is the correspondence between the intuitionistic fragment of classical logic and simply typed lambda calculus ($\lambda_{\rightarrow}$). It states that the type signatures of functions in $\lambda_{\rightarrow}$ correspond to logical propositions, and function bodies are equivalent to proofs of those propositions. In this survey we will explore the implications of this isomorphism and how it scales to higher order logic and type systems, such as System F and Dependent Types. Furthermore, we will look at how the Curry-Howard Isomorphism is being put to use in modern programming languages and academic research.

TABLE OF CONTENTS

INTRODUCTION

The Curry-Howard Isomorphism states that there is a direct correspondence between intuitionistic logic and typed lambda calculus. More specifically, a type declaration given in a programming language corresponds to a logical proposition and an implementation of that type declaration is a proof of that proposition. Figure 1.1 shows the Haskell type signature for *const* and the corresponding logical proposition via the Curry-Howard Isomorphism. Since intuitionistic logic is a special subset of classical logic, providing a well-typed implementation to the function's type signature is the same as giving a logical proof. The proof of *const* is `const a b = a`. The example in figure 1.1 is somewhat uninteresting since it deals with first-order logic. However, as will be shown, the Curry-Howard Isomorphism holds for higher levels of logic and has led to many useful concepts in Programming Languages research.

1.1 How is it useful?

One might ask how useful the Curry-Howard Isomorphism is? While it isn't directly applicable to the average programmer, it is heavily used in research. Programming Language researchers are coming up with crazy new programming languages that make use of Curry-Howard Isomorphism. For example, Cayenne and Omega, two relatively new languages use dependent types, which correspond to higher-order logic. The Curry-Howard Isomorphism enables Mathematicians and Computer Scientists to benefit from each other's discoveries. Computer scientists have traditionally researched reductions in lambda calculus,

```
const :: a -> b -> a
```

(a) Haskell type signature

$$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \to \beta \to \alpha}$$

(b) Intuitionistic logical proposition

Figure 1.1: Corresponding *const* type signature and logical proposition

while mathematicians have worked on normalizations in proof theory. Thus, a discovery in one domain directly translates into the other domain [19].

The remainder of this survey will follow accordingly: Section 2 introduces intuitionistic logic and how it differs from classical logic. Section 3 explores the second order type system known as System F. Section 4 delves further into the intricacies of System F by way of parametric polymorphism. Section 5 introduces type operators and dependent types.

INTUITIONISTIC LOGIC

The Curry-Howard Isomorphism uses a fragment of classical logic called intuitionistic logic. What differentiates intuitionistic logic from other forms of logic?

Simply stated, intuitionistic logic is designed such that proving a theorem is done by constructing an example that satisfies that theorem from a set of givens. Intuitionistic logic originates from mathematicians concerned about unsound applications of classical logic. From Weyl 1946:

> "According to his view and reading of history, classical logic was abstracted from the mathematics of finite sets and their subsets. ... Forgetful of this limited origin, one afterwards mistook that logic for something above and prior to all mathematics, and finally applied it, without justification, to the mathematics of infinite sets."

This is troubling indeed, so Mathematicians began studying the intuitionistic fragment of classical logic, and in essence the distinction of this fragment is how it treats the infinite. In classical logic, "the infinite is treated as *actual* or *completed* or *extended* or *existential*." While in intuitionistic logic "the infinite is treated only as *potential* or *becoming* or *constructive*." [5]

What sort of implications does this have? There is no longer the classical notions of a single Truth ($\top$) or Falsity ($\bot$). Logical propositions are True if a construction can be shown for them, that is if we can derive a construction from a rule system and a set of knowns. False becomes the non-constructable or the absurd. Thus the law of excluded middle and double negation elimination do not hold.

2.1 Rejection of double negation elimination

Negation of a proposition, $\neg P$, in intuitionistic logic can be thought of as $P \to \bot$, that is every construction of P is turned into a non-existent object. Thus to prove double negation in intuitionistic logic, we must show that $P \to \neg\neg P$ and $\neg\neg P \to P$. The proof of the first, which should be expanded to $P \to ((P \to \bot) \to \bot)$ is as follows :

> Given a proof of P, here is a proof of $P \to ((P \to \bot) \to \bot)$: Take a proof $P \to \bot$. It is a method to translate proofs of P into proofs of $\bot$. Since we have a proof of P, we can use this method to obtain a proof of $\bot$. [19]

Going the other way is where we encounter problems. $((P \to \bot) \to \bot) \to P$ doesn't hold because we don't have a construction of P.

2.2 Rejection of the law of excluded middle

According to [8]

> "Brouwer [1908] observed that LEM was abstracted from finite situations, then extended without justification to statements about infinite collections. For example, if x, y range over the natural numbers $0, 1, 2, \ldots$ and $B(x)$ abbreviates the property (there is a $y > x$ such that both y and $y + 2$ are prime numbers), then we have no general method for deciding whether $B(x)$ is true or false for arbitrary x, so $\forall x.(B(x) \vee \neg B(x))$ cannot be asserted in the present state of our knowledge. And if A abbreviates the statement $\forall x.B(x)$, then $(A \vee \neg A)$ cannot be asserted because neither A nor $\neg A$ has yet been proved."

SYSTEM F

System F is a second-order typed lambda calculus that allows universal quantification over types. The Curry-Howard Isomorphism extends to System F, providing a direct correspondence to second order intuitionistic logic. It was discovered independently by the Logician Jean-Yves Girard and Computer Scientist John C. Reynolds. Girard developed System F as a proof notation for second-order logic, while Reynolds invented System F when trying to build a type system for a polymorphic programming language. [19]

3.1 A Quick Intuition on System F

The practical idea of System F is to add polymorphism to typed lambda calculus. For instance if we want to define the untyped function $composition = \lambda f.\lambda g.\lambda x.f\ (g\ x)$ in $\lambda_{\rightarrow}$, we would potentially need to define an infinite number of functions in order to satisfy all possible types:

$$\begin{array}{l} compositionInt = \lambda f : Int \rightarrow Int.\ \lambda g : Int \rightarrow Int.\ \lambda x : Int.\ f\ (g\ x) \\ compositionBool = \lambda f : Bool \rightarrow Bool.\ \lambda g : Bool \rightarrow Bool.\ \lambda x : Bool.\ f\ (g\ x) \\ \ldots \end{array}$$

To circumvent this, we add universal quantification ($\forall$) over types to the calculi and call it λ_2. The composition function now looks like this:

$$composition = \lambda X.\ \lambda f : X \rightarrow X.\ \lambda g : X \rightarrow X.\ \lambda x : X.\ f\ (g\ x)$$

The idea being that the type is abstracted out of the term, allowing for type variables to later be applied to it: $composition[Int] \rightarrow [X \mapsto Int]composition$. More formally, type abstractions and applications are added to the syntax and reduction rules of $\lambda_{\rightarrow}$ (see pg. 343 of [10] for the rules).

Adding universal quantification over types means that we're now working with second order logic, which we will explore now.

3.2 Fun with Quantifiers

First-order logic only uses variables that range over individuals (elements of the domain of discourse) [23]. For instance, $\forall x \in S \;.\; (x + 1 > 12) \; where \; S = \{1, 2, \ldots, 10\}$

In second-order logic variables can also range over sets of individuals. As in it allows for quantification over not only individuals (terms in System F), but also over predicates (types in System F). For example: $\forall x \in Odds(S) \;.\; \exists y \in Evens(S) \;.\; (x + y = 11) \; where \; S = \{1, 2, \ldots, 10\}$.

We must remember that we are still playing with intuitionistic logic, so it is worth exploring what quantifiers really mean in this context. [19] does a good job of explaining the difference between quantification in classical and intuitionistic logics:

> "The intended meaning of $\forall p \varphi(p)$ is that $\varphi(p)$ holds for all possible meanings of p. The meaning of $\exists p \varphi(p)$ is that $\varphi(p)$ holds for some meaning of p. Classically, there are just two possible such meanings: the two truth values. Thus, the statement $\forall p \varphi(p)$ is classically equivalent to $\varphi(\top) \wedge \varphi(\bot)$, and $\exists p \varphi(p)$ is equivalent to $\varphi(\top) \vee \varphi(\bot)$. . . . In the intuitionistic logic, there is no finite set of truth-values, and the propositional quantifiers should be regarded as ranging over some infinite space of predicates."

3.3 Correspondence in the Girard-Reynolds Isomorphism

According to Wadler in [9], in addition to System F, Girard proved a Representation Theorem: every function on natural numbers that can be proved total in second-order predicate calculus can be represented in System F. While Reynolds proved an Abstraction Theorem: every term in System F satisfies a suitable notion of logical relation.

Girard's Representation Theorem results in a projection from second order logic into System F, while Reynolds' Abstraction Theorem results in an embedding of System F into second order logic. Quoting Wadler loosely from [9]: Second order logic is larger than System F's image under the Curry-Howard isomorphism. Second order logic contains quantifiers over individuals, types, and predicates, while System F only allows for quantification over types. Girard's projection from second order logic onto System F is similar to the Curry-Howard isomorphism, in that it takes propositions to types and proofs to terms, but differs in that it erases information about individuals. This mapping preserves reductions, hence it is no mere surjection but a true epimorphism.

Thus the Reynolds embedding followed by the Girard projection is the identity. While Girard's projection followed by Reynolds' embedding is not since the projection discards all information about individuals.

3.4 System F in use

System F enables us to reason about programs with polymorphism. This power comes with a downside, type inference becomes undecidable. To mitigate this many modern languages use weaker forms of polymorphism, such as *Rank-2 Polymorphism*, in which type inference is decidable. In addition to formalizing polymorphism, System F has yielded interesting results in language security ([22]) and lead to the notion of free theorems.

PARAMETRICITY

With universal quantification ($\forall$) over types, System F allows us to reason about polymorphic functions. System F's polymorphism is called *parametric polymorphism*, as compared to the *ad-hoc polymorphism* of method overloading in most imperative languages. The generality of this parametric polymorphism has much to offer.

4.1 Theorems for Free

In "Theorems for Free!" [21], Wadler refines some ideas on parametric polymorphism presented by Reynolds in [13]. The overarching idea is that parametrically polymorphic functions behave uniformly across all types. With this uniformity we can derive certain simple theorems from only the type signature.

Take, for instance, this polymorphic function type $f : \forall A.A \rightarrow A$. If we exclude the ability to do runtime type analysis, there is no way to do operations that distinguish between types. Hence, the only possible function body that could inhabit f is the identity function $f = \lambda A \mathrel{.} \lambda x : A \mathrel{.} x$

Extending this idea to functions that operate over lists, a function type such as $g : \forall A.[A] \rightarrow [A]$ can only operate over the elements provided to it. Wadler, making use of this, shows that polymorphic functions in System F commute with all functions that operate on more specific types. For instance, given function signatures $f : \forall A.[A] \rightarrow [A]$ and $g : Char \rightarrow Int$, Wadler shows that $map\ g \circ f_{Char} \equiv f_{Int} \circ map\ g$.

Figure 4.1 is a continuation of this example in Haskell using the ExistentialQuantification extension. We let f be the $reverse$ function, and g be ord, which takes characters to their ascii values.

```
{-# LANGUAGE ExistentialQuantification #-}

f :: forall a . [a] -> [a]
f = reverse

g :: Char -> Int
g = ord

> map g . f $ ['a'..'f']
[102,101,100,99,98,97]

> f . map g $ ['a'..'f']
[102,101,100,99,98,97]
```

Figure 4.1: Example demonstrating a free theorem

4.2 Pffft, is this useful?

While trivial looking, these simple commuting theorems are actually useful. They enable us to add function commutativity to our arsenal of algebraic manipulations when we are reasoning about programs [21]. In addition, [18] states that the reordering of function applications can lead to improved efficiency of code by enabling additional transformations and analysis.

Can we scale the idea of free theorems to more complicated languages, such as those in use today? It turns out that there is a lot of recent research making use of free theorems.

4.3 Parametricity with Haskell

You may have noticed by now that languages like Haskell have runtime type analysis, as well as imprecise error semantics, both of which prevent us from deriving free theorems from polymorphic functions. By imprecise error semantics, we mean that the error raised by a program is not guaranteed to be the same exception that would be encountered by a straight forward sequential execution [4]. This means that when making use of free theorems in Haskell, there is no guarantee that new errors will won't be introduced.

```
-- produces ***Exception: divide by zero
l = [[i] | i <- [1..(div 1 0)]]

-- can produce *** Exception: Prelude:tail: empty list
tail

-- can only propagate errors
null
takeWhile

-- only propagates errors
map tail

-- can possibly produce empty list or divide by zero errors
(takeWhile (null . tail) l)

-- by parametric free theorems the following should hold:
map tail (takeWhile (null . tail) l) ==
takeWhile null (map tail l)

-- but this could produce 2 different errors
map tail (takeWhile (null . tail) l)

-- while this could only produce 1 error
takeWhile null (map tail l)
```

Figure 4.2: Example of Haskell's imprecise error semantics

Let us look at the example in Figure 4.2, which is derived from [18]. This example shows that Haskell's non-deterministic error semantics might introduce new errors when free theorems are used. [18] is able to formulate a way to use free theorems in the presence of imprecise errors.

As for runtime type analysis, the essence of the problem is that polymorphic functions can treat inputs of different types non-uniformly. Hence there is no straight forward way to derive free theorems from polymorphic functions that analyze types, such as the function f' from [20], which is shown in Figure 4.3. The authors of [20] remedy this problem by using representation types. Languages such as λ_R use term representation of types to simulate runtime type analysis [2]. The idea is to have terms represent types, so if term e represents

```
-- Without runtime type analysis functions of type a -> a
-- can only be inhabited by the identity
f :: forall a . a -> a
f x = x

-- With runtime type analysis anything is possible!
-- NOTE: the folloing function is psuedo code
f′ :: forall a . a -> a
f′ x = if (typeOf x) == (typeOf 1)
         then succ x
         else x

g :: forall a . a -> Int
g _ = 42

-- The free theorem for f and g:
f . g == g . f

-- No free theorem for f′ and g since:
f′ . g /= g . f′
```

Figure 4.3: Free theorems aren't possible in the presence of runtime type analysis

type t, then term e has type $R\ t$.

The introduction of GADTs (generalized algebraic datatypes) has enabled representation types to be used with parametricity. This is best seen in Figure 4.4, which is an example from [20] that uses Haskell's GADT language extension. In Figure 4.4 the general free theorem $(g[\tau]\ r) \circ h \cong h \circ (g[\tau]\ r)$ doesn't hold because of R_{int}, yet if we are analyzing R_{any} it holds: $(g[\tau]\ (R_{any}[\tau])) \circ h \cong h \circ (g[\tau]\ (R_{any}[\tau]))$ [20]

The work in [20] starts to lay the theoretical foundations needed to start reasoning about complex languages such as Haskell in terms of parametricity.

4.4 Additional Research into Parametricity

We've shown how the generality provided by parametric polymorphism is useful for deriving simple theorems over functions. While novel, recent research has been working to

```
-- representation type of Int, Func, and Any
data R a where
  Rint :: R Int
  Rarrow :: R a -> R b -> R (a -> b)
  Rany :: R a

-- runtime type analysis using representation types
g :: (R a) -> a -> a
g t y = case t of
             Rint -> succ y
             Rany -> y

> Rint 1
2

> Rany 'a'
'a'

> Rint 'a'
***Couldn't match expected type `Int' with actual type `Char'
```

Figure 4.4: Representation Types in Haskell using GADTs

make parametricity more applicable to modern languages such as Haskell. In addition to the research presented in this section, there are a number of other interesting projects in this area. "Generalizing Parametricity Using Information-flow" [22] looks to provide security and confidentiality to module writers utilizing parametric polymorphism in the face of runtime type analysis. "A Logic for Parametric Polymorphism" [12] introduces a logic that formalizes parametric polymorphism and follows in the steps of the Logic for Computable Functions, taking the first steps towards creating a full fledged logic for polymorphic programs.

DEPENDENT TYPES

We've covered second order lambda calculus and its connection to second order logic, but how much more expressive can we make type systems? It turns out there are two other extensions to $\lambda_{\rightarrow}$ that we can make, ω and P. λ_{ω} formalizes type operators, enabling the creation of Algebraic Data Types. Exploring ADT's generalization, Generalized Algebraic Data Types, will then carry us into Dependent Types (λ_{P} or λ_{Π}).

5.1 Dependent Types in a Nutshell

So far we have seen terms that depend on terms (lambda abstractions), terms that depend on types (polymorphism and simple type abstractions), and soon we'll see types that depend on types (type operators). This leaves one additional possibility, types that depend on terms, or dependent types. Having types that depend on terms lets us move a lot of a program's computation into the type checker, yielding crazy possibilities: lists that carry their length in their type [7], an AVL tree implementation whose type guarantees certain balanced properties [15], a sorting algorithms whose type proves well-ordered output [1], and so on. But first let's explore the λ_{ω} system.

5.2 Type Operators

Suppose you want to create a new type from existing types. The lambda systems we've explored so far don't support this expressive of type level operations. System F lets us use simple type applications and abstractions to model polymorphism, but it doesn't allow for the creation of new types. For instance, let's try constructing Haskell's Either type using an ADT: `data Either a b = Left a | Right b`. On the left is the type, where any two types can be substituted for a and b. On the right are the two terms that inhabit that type. Formally, the type looks like $Either\ Y\ Z = \forall X.(Y \rightarrow X) \rightarrow (Z \rightarrow X) \rightarrow X$ or $\lambda Y.\lambda Z.\forall X.(Y \rightarrow X) \rightarrow (Z \rightarrow X) \rightarrow X$, where Y and Z are type parameters and X

is the new type produced. $Either$ is a type that depends on other types. If we add some simple type application and abstraction rules we are good to go, right?

Not really. We now have two types of types, proper types with arity of 0, as well as partially applied types, or type families, with arity greater than 0. Using type families in type signatures doesn't make sense, so we need to ensure that our type signatures are themselves well typed. So enters the notion of kinds.

5.3 Kinds

What is the type of a type? Let us denote it as a kind ($*$). The kind of a proper type is denoted $Int :: *$. Things get interesting now that we have type operators. The type constructor $Either$ is of the kind $* \to * \to *$. And of course you can curry type constructors: $Either\ Int\ :: * \to *$,

Going deeper, what is the type of a kind? According to [6]:

> "As the kind of $*$ is itself $*$, we can encode a variation of Russell's paradox, known as Girards paradox. This allows us to create an inhabitant of any type. To fix this, the standard solution is to introduce an infinite hierarchy of types: the type of $*$ is $*1$, the type of $*1$ is $*2$, and so forth."

So basically it is turtles all the way down...

5.4 Generalized Algebraic Data Types (GADT)

ADTs are great, but we might ask, can we squeeze more expressiveness out them? The answer is yes, and it comes in the form of GADTs. The key distinction between ADTs and GADTs is that pattern matching causes type refinement. Take, for example, the ADT in Figure 5.1 (a). The construction of a $Lit1$ term in Figure 5.1(b) doesn't lead to the type refinement of a into Int in the type $Term'\ a$. $Lit2$ does refine a to $Char$ but this

```
                              > :t Lit1 1
data Term' a = Lit1 Int       Lit1 1 :: Term' a
             | Lit2 a
             | Succ' Int      > :t Lit2 'a'
                              Lit2 'a' :: Term' Char
```

(a) (b)

Figure 5.1: datatype exemplifying ADT type refinement

```
data Term a where
  Lit    :: Int -> Term Int
  Succ   :: Term Int -> Term Int
  IsZero :: Term Int -> Term Bool
  If     :: Term Bool -> Term a -> Term a -> Term a
  Pair   :: Term a -> Term b -> Term (a,b)
```

(a)

```
> :t Lit 1
Lit 1 :: Term Int

> :t IsZero $ Lit 1
IsZero $ Lit 1 :: Term Bool
```

(b)

Figure 5.2: Demonstrating the expressiveness possible with GADTs

type refinement is too restrictive. Let's remedy this by using the GADT from [3], $Term\ a$ (Figure 5.2 (a)), which can be used to define a well-typed eval function. a is refined to Int in 5.2(b), and some expressiveness is gained since a is allowed to be refined to $Bool$ in $IsZero$.

5.5 Faking dependent types with GADT

In some instances we can use GADTs to fake dependent typing. Since GADTs separate values from types, that is, terms aren't allowed in type operations, we have to use singleton types that represent values. This is the approach taken by Sheard et. al. in the design of Ωmega [14].

Take, for instance, this series of ADTs:

```
-- Phantom Types
data Zero
data Succ n

type One   = Succ Zero
type Two   = Succ One
type Three = Succ Two
type Four  = Succ Three

data Vec n a where
  Nil  :: (Show a) =>                  Vec Zero a
  Cons :: (Show a) => a -> Vec n a -> Vec (Succ n) a

instance Show (Vec n a) where show = showVec

showVec :: Vec n a -> String
showVec Nil = ""
showVec (Cons x xs) = show x ++ " " ++ showVec xs

foo = Cons 3 (Cons 2 (Cons 1 Nil))

> show foo
"3 2 1"
```

Figure 5.3: Mimicking dependent typing with GADTs

$$data\, Vec0\ \alpha = Vec0$$

$$data\, Vec1\ \alpha = Vec1\ \alpha$$

$$data\, Vec2\ \alpha = Vec2\ \alpha\ \alpha$$

$$data\, Vec3\ \alpha = Vec3\ \alpha\ \alpha\ \alpha$$

These ADTs are trying to capture length information of the vector. There is a pattern here, and it would be nice to create an abstraction for it, such as: $\forall\alpha :: *.\forall n :: Nat.Vec\, n\, \alpha$, where n represents the length, and α the data [6]. Using GADTs we can create types to represent the natural numbers and successfully achieve the abstraction above. See Figure 5.3 for an instance of this that has been modified from [7].

While this is nice, things can get cumbersome when you have to create parallels between the type and term worlds.

5.6 Pi Types

In literature, the Π type represents the dependent product type, and is a generalization of λ in simply typed lambda calculus. It abstracts arrows, accepting type-level arguments, as well as term-level arguments. Hence $S \to T$ is the same as $\Pi x : S \,.\, T$, where x does not appear free in T.

According to [11], 1st order predicate logic is isomorphic to dependent types. A predicate B over type A is viewed as a type value function over A, and hence universal quantification is the same as dependent product: $\forall x : A.B(x)$ is equivalent to $\Pi x : A.B(x)$

5.7 Challenges with Type Checking

Dependent typing lets us move a lot of computation to the type checking phase, giving us the ability to construct more expressive proofs via the Curry-Howard Isomorphism. Unfortunately, we can't get all this proof-carrying code without incurring a cost.

The main problem is that now there are terms in our types, so to check type equality we need to evaluate our types to a normal form. If we aren't careful and encode too powerful of a language into our types the result is a non-terminating type checker. Modern dependently type languages remedy this problem differently: "Ωmega type-level programs are written as term rewriting systems, that have to terminate; Epigram and Coq programs are written in total languages - every program terminates; in Agda, programs are interrupted when they get stuck in a loop - soundness is regained through a separate termination check (I think). In practice, non-termination doesn't seem to be a big problem." [17]. Where total languages are those that provably terminate, and hence are not Turing-Complete, while partial languages are Turing-Complete.

Language designers must tread lightly. Total languages are restrictive and require a certain approach to problems, while partial languages require tricks to ensure termination and soundness of type checking.

5.8 Are Dependent Types the Way to the Future?

There is a lot of stigma against dependently typed languages:

> "Most Haskell programmers are hesitant to program with dependent types. It is said that type checking becomes undecidable; the phase distinction between type checking and evaluation is lost; the type checker will always loop; and that dependent types are really, really hard.
>
> The same Haskell programmers, however, are perfectly happy to program with a ghastly hodgepodge of generalized algebraic data types, multi-parameter type classes with functional dependencies, impredicative higher-ranked types, and even data kinds. They will go to great lengths to avoid dependent types." [6]

At the same time, a lot of smart people are pushing dependent types. For instance, [16] sees a dependently typed, pay-as-you-go approach to language verification as the next logical step in the progression of languages.

What is certain is that there is a long ways to go, in terms of both programmer intellect and language development, before we start seeing the adoption of these mind bending type systems.

CONCLUSION

The Curry-Howard Isomorphism relates intiutionistic logic with programming language type systems. We've explored the ideas behind intiutionistic logic and how it differs from the notions of classical logic. With System F, we got a feel for type systems that allow for quantification over predicates, or types. In addition to polymorphism, quantification over predicates enabled us to derive certain theorems about our programs for free. We concluded with a study of dependent types, which move much of the program's computation into the type level, yielding proof-carrying code. Throughout the survey special attention was paid to demonstrating how these ideas have found their way into modern programming language implementations such as Haskell. The Curry-Howard Isomorphism has yielded rich results in both programming language theory and implementation. We must wonder, what else does it have to offer?

BIBLIOGRAPHY

[1] Thorsten Altenkirch, Conor Mcbride, and James Mckinna. Why dependent types matter. In *In preparation, http://www.e-pig.org/downloads/ydtm.pdf*, 2005.

[2] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics, 2002.

[3] haskell.org. Generalised algebraic data types, 2011. [Online; accessed 5-December-2011].

[4] Simon Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. A semantics for imprecise exceptions. *SIGPLAN Not.*, 34:25–36, May 1999.

[5] Stephen C. Kleene. *Introduction to Metamathematics*. Ishi Press International, March 2009.

[6] Andres Löh, Conor McBride, and Wouter Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundam. Inf.*, 102:177–207, April 2010.

[7] David Menendez. Fixed-length vectors in haskell, part 1: Using gadts, 2005. [Online; accessed 5-December-2011].

[8] Joan Moschovakis. Intuitionistic logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2010 edition, 2010.

[9] Philip and Wadler. The girardreynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1-3):201 – 226, 2007. ¡ce:title¿Festschrift for John C. Reynoldss 70th birthday¡/ce:title¿.

[10] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[11] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[12] Gordon Plotkin and Martn Abadi. A logic for parametric polymorphism. In *Theoretical Computer Science*, pages 361–375. Springer-Verlag, 1993.

[13] John C. Reynolds. Types, abstraction and parametric polymorphism. *Information Processing '83*, pages 513–523, 1983.

[14] Tim Sheard, James Hook, and Nathan Linger. GADTs + extensible kind system = dependent programming. Technical report, Portland State University, 2005. `http://www.cs.pdx.edu/~sheard`.

[15] Tim Sheard and Nathan Linger. Programming in omega. In Zoltn Horvth, Rinus Plasmeijer, Anna Sos, and Viktria Zsk, editors, *CEFP*, volume 5161 of *Lecture Notes in Computer Science*, pages 158–227. Springer, 2007.

[16] Tim Sheard, Aaron Stump, and Stephanie Weirich. Language-based verification will change the world. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 343–348, New York, NY, USA, 2010. ACM.

[17] Manuel Simoni. Axis of eval: Notes on dependent types, 2010. [Online; accessed 5-December-2011].

[18] Florian Stenger and Janis Voigtländer. Parametricity for haskell with imprecise error semantics. In *Proceedings of the 9th International Conference on Typed Lambda Calculi and Applications*, TLCA '09, pages 294–308, Berlin, Heidelberg, 2009. Springer-Verlag.

[19] Morten Heine B. Srensen and Pawel Urzyczyn. Lectures on the curry-howard isomorphism, 1998.

[20] Dimitrios Vytiniotis and Stephanie Weirich. Free theorems and runtime type representations. *Electron. Notes Theor. Comput. Sci*, 173:2007, 2007.

[21] Philip Wadler. Theorems for free! In *FUNCTIONAL PROGRAMMING LANGUAGES AND COMPUTER ARCHITECTURE*, pages 347–359. ACM Press, 1989.

[22] Geoffrey Washburn. Generalizing parametricity using information-flow. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, pages 62–71, Washington, DC, USA, 2005. IEEE Computer Society.

[23] Wikipedia. Second-order logic — wikipedia, the free encyclopedia, 2011. [Online; accessed 5-December-2011].