Dependent Types In Lambda Cube

Ondřej Peterka
Brno University of Technology, Faculty of Information Technology
Božetěchova 2, 612 66 Brno, Czech Republic
ipeterka@fit.vutbr.cz

Abstract. This paper describes some backgrounds for author's further study of dependent types. It covers to certain extent Curry-Howard correspondence, lambda cube and also tries to explain some basics of the intuitionistic logic - especially from type theory point of view and differences to classic logic.

Key words: dependent types, Curry-Howard isomorphism, intuitionistic logic, lambda cube, pure type systems

1. Introduction

This work covers to a certain extent some backgrounds for author's further study of dependent types. Chapter 2 contains basic information about two different special dependent types - *Pi-type* and *Sigma-type*. The chapter also discuss the problem of confusing terminology for these types and tries to explain why there is so many names for the same types. Also a mention of *dependent record types* can be found in this chapter. The chapter 3 gives explanation of *Curry-Howard isomorphism*. It starts with some basics of the *intuitionistic logic* - especially from type theory point of view and differences, when compared to classic logic. The text in chapter then covers the main point of the chapter: Curry-Howard isomorphism as a close relationship between mathematical proofs on one side and computer programs on the other side. The last chapter 4 has as the subject matter the *Barendregt's lambda cube*. The chapter gives some explanation on all eight calculus-es found in the lambda cube and describes how they can be defined using one set of generic rules.

2. Dependent types

2.1 Motivation

In a Hindley-Milner-typed functional language (e.g. Haskell) we can easily define dot product [1] of two one-dimensional vectors represented as lists:

```
dp [] [] = 0
dp (x:s) (y:t) = x * y + dp s t
```

[1] sometimes called *inner product*

December 14, 2007.

The weak point of this definition is apparent: an attempt to multiply two vectors of different lengths fails, but the failure occurs as late as in runtime. We would like to have such incorrect application detected sooner, preferably during the type-checking phase. Thus we want the compatibility of the lengths of the vectors be encoded in the type of function `dp`.

This can be achieved in a language with *dependent types* (e.g. Cayenne):

```
dp :: (n::Nat) -> Vec n -> Vec n -> Float
dp 0 [] [] = 0.0
dp n (x:s) (y:t) = x * y  +  dp (n-1) s t
```

Now `Vec` is a type constructor dependent on a natural number n and representing the type of one-dimensional vectors of length n The type of function `dp` enforces the desired constraint: both arguments have to have the same length.

More closely to type theory we could write the type of the dot product function as follows (for more info about Π notation see the chapter 2.2):

$$Nat : \star, Float : \star, Vec : Nat \to \star \vdash dp : \Pi n : Nat.Vec\, n \to Vec\, n \to Float$$

Similarly we can take advantage of dependent types in matrix multiplication:

```
mmul :: (m::Nat) -> (n::Nat) -> (p::Nat)
                              -> Mat (m,n) -> Mat (n,p) -> Mat (m,p)
```

Other examples could be found in Pierce [2005], Augustsson [1998] or Skarvada [2007].

2.2 Standard Types

The notions of standard types stands here for **Pi-type** ($\Pi x : A.B$) and **Sigma-type** ($\Sigma x : A.B$). There is different terminology in use, and it would be useful to unify it. So called "Pi-types" and "Sigma-types" offer us the power corresponding to the power of quantifiers in the first order and higher order logic. Particularly, Pi-type is an image of the universal quantifier ($\forall$) and Sigma-type stands for the existential quantifier ($\exists$). Why and how they correspond is described for example in Sorensen [1998]. Some explanation on this (along with explanation of confusing terminology) is given also in the paragraphs below.

2.2.1 Pi and Sigma types - different terminology

The reason of the fact, that there is different terminology in use for Sigma- and Pi-types is somewhat unclear. However, the most probable reason of using different names is the difference between the intuitionistic logic and the classic logic. The older terms used were "Dependent function type" and "Dependent product type" for Pi-type and Sigma-type respectively. These older names are rather more clear and they have their origin in intuitionist logic.

If we have

$$\forall x : Nat.P(x),$$

we can read it (intuitionistic way) as *"I have a method for constructing an object of the type $P(x)$ using any given object x of the type Nat"*. So here, it is more like a generalization of the ordinary function type. And that is the reason, why the term "Dependent function type" is used. On the other hand, if we think in terms of the classic logic, we will read the formula differently. It is more like saying *"We have an infinite conjunction $P(x_1) \wedge P(x_2) \wedge P(x_3) \ldots$ "*. The conjuction corresponds to the product, and that is why Pi-type is also called "dependent product".

Now, why is Sigma-type named by some authors "dependent product type" and by other authors "dependent sum". The answer lies again in the difference between the intuitionist logic and the classic logic. Lets consider Sigma-type of the form

$$\Sigma x : Nat.P(x)$$

Intuitionistic meaning could be described as *"I have an object x of the type Nat (but unspecified any further) and I know that it has a property P(x)"* . It is given by the nature of the intuitionism, that we need to record both of these, i.e. the object x and the property $P(x)$. We can do this by creating a pair. And the pair is nothing else than binary product. So, the Sigma type is sometimes called "dependent product type". On the other hand, considering the classic logic point of view, we will read the formula of a sigma type as *"we have an infinite disjunction $P(x_1) \vee P(x_2) \vee P(x_3) \ldots$"*. The disjunction corresponds to the sum types (or variant for Pascal or C++ programmers). Here we can find the reason for naming Sigma-type "Dependent sum".

As the terminology on the dependent stuff is really confusing it could be a good idea to use just names "Pi-type" and "Sigma-type".

2.3 Record Types

Dependent record types are formalized by Betarte [1998]. Basic info could be found in Larrson [1998] and Rysavy [2005] Before introducing *dependent record type* let us review what is a *record type*. A *record type* could be viewed as a sequence of fields and each field is uniquely denoted using a label. Each label (field) is also associated with a certain type. The *record type* could looks like:

$$\langle L_1 : \alpha_1, L_2 : \alpha_2, \ldots L_n : \alpha_n \rangle$$

A dependent record type adds one more property. It is a dependency between the type of each label L_i and types of all preceding labels L_1 till L_{i-1}. In other words the dependent type of a label (field) L_i in *dependent* *record type* could be seen as a function, which takes all preceding labels as arguments and creates a type α_i. The *dependent record type* could looks like:

$$\begin{pmatrix} L_1 & : & \alpha_1() \\ L_2 & : & \alpha_2(L_1) \\ \dots & & \dots \\ L_n & : & \alpha_n(L_1, L_2, \dots, L_{n-1}) \end{pmatrix}$$

For example of using a dependent types for forming algebraic structures (magma, semigroup, . . .) see Skarvada [2007].

3. Curry-Howard correspondence

3.1 Intuitionistic logic

The intuitionism was formerly founded by Luitzen Egbertus Jan Brouwer as an opponent to the then-prevailing formalism of David Hilbert. The intuitionism is an approach to mathematics as the constructive mental activity of humans and as its name could suggest it gives strong emphasis on mathematician's intuition. The intuitionism is sometimes mistaken for constructivism, but the intuitionism is just one kind of constructivism. On the other hand, it could be useful for our purposes not to distuinguish between these two philosophies. The main reason is that the true of a statement in the intuitionism philosophy is somewhat ill-defined.

In Brouwer's original intuitionism, the truth of a statement is taken to be equivalent to the mathematician being able to intuit the statement. As one can probably see, this is not very sufficient definition of the truth for our further study and understanding of the intuitionistic logic. Constructivism, more clearly, asserts that it is necessary to find (or "construct") a mathematical object to prove that it exists. The well known approach of absurdum proof would not be used in constructivism philosophy: when one assumes that an object does not exist and derives a contradiction from that assumption, one still has not found the object and therefore not proved its existence.

The intuitionistic logic was introduced by Arend Heyting to provide formal basis for the intuitionism and as was partially suggested above, the name "constructivist logic" is used interchangebly. In order to understand the intuitionistic logic (and also the intuitionism as the philosophy) one has to forget the classical notion of "true". The true could be defined as a possibility of construction of the statement. In other words, "true" can be identified with the existence of a proof or "construction" of the statement.

First good step to begin to understand the intuitionistic logic is to try to understand the following explanation of propositional connectives in the domain of the intuitionistic logic:

- A construction of $\varphi_1 \wedge \varphi_2$ consist of a construction of φ_1 and a construction of φ_2. *(In other words, you need both proofs (constructions), of φ_1 statement and φ_2 statement.)*
- A construction of $\varphi_1 \vee \varphi_1$ consist of a number $i \in \{1, 2\}$ and φ_i. *(In other words, you need one of proofs (constructions), either of φ_1 statement or of φ_2 statement, but you have to know which construction it is.)*
- A construction of $\varphi_1 \rightarrow \varphi_2$ could be undesrtand as a function transforming every construction of φ_1 into a construction of φ_2. *Transforming means a process here, where new proof of φ_2 is created on the basis of φ_2.*
- There is no construction of $False$.

If we want to consider first-order logic we will have to add more explanation:

- A construction of $\forall x.\varphi(x)$ is a method, which transforms every object x into a construction $\varphi(x)$. *As one can see it is very similar to what is done in the case of implication connective.*
- A construction of $\exists x.\varphi(x)$ is a pair consisting of an object x and a construction $\varphi(x)$. *Again somewhat similar to what we have seen in the case of the conjunction connective. Both similarities are used in designing and understanding of so called Pure type systems (see Chapter 4)*

Another important issue to understand is the meaning of negation. As is known already from classical logic the negation $\neg\varphi$ could be viewed as an implication $\varphi \rightarrow False$. This means, that if the assumption of φ leads to absurd, we can assert and write $\neg\varphi$. In the similar way as was described connective for implication ($\rightarrow$) above, here is the meaning of $\neg\varphi$ in the intuitionistic logic:

- A construction of $\neg\varphi$ is a method that turns every construction of φ into an non-existent object.

Consider for example classical tautology

$$\neg\neg A \rightarrow A$$

This tautology is not valid in the intuitionistic logic. The reason is, that we can not assert A only on the basis of the fact, that we are able to show it *is not true, that A is true.* Also the double negation theorem itself is not valid in the intuitionistic logic. Consider a statement *”It is not the case that it’s not snowing.”* The statement is weaker (by intuition), than statement *”It is snowing”*. The latter requires directly a proof (construction) for the fact, that it is snowing, whereas the former not.

3.1.1 Tertium Non Datur rejection

One of the most interesting as well as the worst understandable difference between the classic and the intuitionistic logic is the rejection of the law of *Aristotelian law of excluded middle (*$\varphi_1 \vee \neg\varphi_2$*)* in the intuitionistic logic. In the intiutionistic point of view there is not so much information in the statement $\varphi_1 \vee \neg\varphi_2$ so it could be acceptable as a theorem. The problem is that, there are situations, where we can not say exactly which of φ_1 and φ_2 is true. Let us see some examples from Sorensen [1998], which could help us to understand this issue better.

EXAMPLE 1. Lets consider the following statement A:
There is seven 7’s in a row somewhere in the decimal representation of the number π*.*

Currently nobody can tell whether the statement is false or true. And it could happen that nobody ever will. So statement $A \vee \neg A$ is not tautology from our point of view, even if we have to admit that one of the options must hold. However, we do not have a proof of any of them.

EXAMPLE 2. For the next example let us consider a statement B:
There are two irrational numbers x and y, such that x^y *is rational.*

Once again, we know there must be solution and we do even know the two possibilities. It is either $x = y = \sqrt{2}$ in the case, that $\sqrt{2}^{\sqrt{2}}$ is rational number, or we put $x = \sqrt{2}^{\sqrt{2}}$ and $y = \sqrt{2}$. However, the problem is, that we do not know, which option to choose. So, yet again $B \vee \neg B$ will not work as expected.

One may object that these two mentioned examples are based on problems, which can be solved as early as today or tommorow. However, there will always be such problems and therefore intuitionstic logic has its part of meaning.

3.2 Curry-Howard isomorphism

Curry-Howard isomorphism describes the close relationship between mathematical proofs on one side and computer programs on the other side. Curry-Howard isomorphism is just one of the names, which are used for the description of the mentioned relationship. One can see in the literature

(Sorensen [1998], Pierce [2005] or Thompson [1991]) also the name *Curry-Howard correspondence*, which reflects the fact, that some mathematicians are not completely sure, that there the real *isomorphism* in mathematical sense of the word exists. Other names are also *propositions as types, formulas as types* or *proofs as programs.* Each of these names already gives advice on what is the correspondence about. The discovery of the isomorphism is thanks to the American mathematician Haskell Curry and logician William Alvin Howard. The isomorphism in mathematical sense of word could be seen between natural deduction in intuitionistic logic and simply typed lambda calculus (for fine description of both of these terms see e.g. Thompson [1991]), where exist bijections between elements of logic and elements of simply typed lambda calculus.

The following described correspondence follows description in Sorensen [1998] and is only valid when considering intuitionistic logic. However, in recent days there are more or less successful attempt to find isomorphism using clasicall logic (some description of this could be found also in Sorensen [1998]).

- **logical formulas and types**: Logical formulas can be viewed as types known from the simply typed lambda calculus. Formula $A \wedge B$ can be seen as a product type, which consists of types A and B, which themselves are again formulas on the side of logic, and types on the side of type theory.
- **connective and type constructor**: We could see already one example above, namely for *conjunction* connective. In the similar manner we can create new type using connective for disjunction. The resulting type will be a sum (variant) type. The sum type is a type with two unary constructors. The implication connective has its counterpart in the arrow type. Any formula $A \rightarrow B$, where A and B are also formulas can be seen in simply typed lambda calculus domain as a function of type $A \rightarrow B$, where A and B are types. The argument of such function is in this way of type A and the result of the function is of type B.
- **assumption and term variable**: When we write the right side $f : A \rightarrow B, x : A$ of the judgement $f : A \rightarrow B, x : A \vdash fx : B$ we introduce two assumption - $f : A \rightarrow B$ and $x : A$. These are seen in lambda calculus as variables and in logic as assumptions.
- **proof and term**: Every term could be viewed as a construction (proof). Term represents an inline form of the natural deduction tree needed to construct the proof (term). This analogy is the reason why the Curry-Howard isomorphism is denoted as *proofs as programs.*
- **propositional variable and type variable**: Briefly it means that there is correspondence between propositional variables as A, B, etc. with types A, B, etc. This analogy is the reason why the Curry-Howard isomorphism is denoted as *propositions as types.*

- **provability and inhabitation**: It basically means, that one can proof the formula A if there exist a type A, which is inhabited. And vice versa - asking whether there exists a term of a given type (inhabitation) corresponds to asking if there exists a construction for the given proposition. One can refer to section 3.3 to find out more about the inhabitation problem.
- **construction representing proof tree with redundancy and redex**: A redex consists of a constructor immediately surrounded by the corresponding destructor and in the proof tree it is reflected as application of introduction rule immediately followed by the corresponding elimination rule. On the other hand a term in the normal form corresponds to a proof tree in the normal form.
- **normalization and reduction**: This two names could be used interchangeably. Reduction (or normalization) of a redex corresponds to normalization (or reduction) of a proof tree.

Defining the correspondence also for first-order predicate logic leads to using Pi- and Sigma-types (see section 2).

- **universal quantification and Pi-type**: Formula $\forall x : A.B(x)$ corresponds under Curry-Howard isomorphism to Pi-type (also called *dependent product type*) $\Pi x : A.B(x)$, which is in turn just generalized arrow type $A \rightarrow B$. The only difference is that in a Pi-type we record dependency between the object of type A and the result object of the type B (x is not free in B). In the case of the plain arrow type, there is no need to record dependency - it is just a plain function.
- **existent quantification and dependent Sigma-type** : Formula with existential quantifier $\exists x : A.B(x)$ corresponds to Sigma-type (also called *dependent sum types*) which is written as $\Sigma x : A.B(x)$ and is in turn generalized form of ordinary product type $A \times B$ (if x is not free in B it is the exact case of an ordinary product type).

3.3 The type inhabitation problem

The inhabitation problem is a problem of finding a $\lambda - expression$ with a given type. The answer to this problem is quite interesting and was partially mentioned in a previous section - there is a $\lambda - expression$ of particular type (i.e. the type is inhabited) only if there is a proof for the type corresponding (by Curry-Howard correspondance) theorem in logic. For inuitionistic logic the correspondence is actually bijection.

As was already mentioned earlier there are some theorems of the classic logic, which are not theorems in the inuitionistic logic. What it means from point of view of the inhabitation problem? Well, for example $A \vee \neg A$ (the law of excluded middle) has no corresponding inhabited type - it is not possible to construct $\lambda - expression$, which has that type. Certainly, there are many more theorems from the classic logic, which has no corresponding inhabited types. Some of them (such as the law of excluded middle or Pierce's law) are actually even tautologies in classic logic sense.

Let's explain on few examples now why formulas which are not theorems can not correspond to an inhabited type.

EXAMPLE 1a. Consider $\alpha \to \beta$, which is clearly no theorem. If there was an $\lambda - expression$ with such type, it would had to be a function having argument of type α and producing as a result an object of a type β, which is not known. β is not known because the function does not receive any description or proof of it.

EXAMPLE 1b. On the other hand consider identity function $\alpha \to alpha$. The identity function is valid, because the type α is well known to the function, so it can be reproduced.

EXAMPLE 2a. Now, it will be more easier to understand, why function of a type $\beta \to (\alpha \to \alpha)$ is inhabited. One can explain it as follows: the β argument is known and is ignored and an identity function which is a theorem is produced.

EXAMPLE 2b. On the other hand if we switch sides of the first implication, we get the non-inhabited type $(\alpha \to \alpha) \to \beta$. The reason is the same as in the case of $\alpha \to \beta$.

EXAMPLE 3. In the last example we have a well known tautology $A \to \neg\neg A$ from classical logic. It is explained why $A \to \neg\neg A$ is not an inhabited type:

$$A \to \neg\neg A$$

which is:

$$A \to ((A \to False) \to False)$$

and term would be

$$\lambda a.\lambda x : (A \to False).(xa)$$

but

$$A \vee \neg A$$

can be rewritten to

$$A \vee (A \to False)$$

and one has to show that A or $(A \to False)$ is true in the empty context (i.e. without hypotheses).

4. Barendregt's lambda cube

The idea of the lambda cube was firstly introduced by the mathematician Henk Barendregt (1991). The invention of the lambda cube was partly motivated by an attempt to explain the structure of the rather complicated calculus of construction. It is a fine tool for investigation of eight forms of lambda calculus-es and their generic (typed) unification in the form of *pure type systems (PTS).*

4.1 Sorts - terms, types, kinds

To be able to understand the lambda cube, we must first understand the relationship among terms, types, kinds and sorts. Very nice explanation for understandning part of the lambda cube could be found in Jones and Meijer [1997]. Also the book Sorensen [1998] gives comprehensive explanation of the lambda cube and PTS. Note please, that it is quite difficult to understand any explanation of the following issues, because of shortage of synonyms for words as "type". In the description of the lambda cube we can identify three levels: *Terms, Types and Kinds.* All these three levels are called *sorts.* So if we say that an expression is a term, we mean that the expression is of the sort *Term.* We will use a similar abbreviation for types and kinds. In general pure type systems we can have more than just three levels, but in the lambda cube there are just three used, which can vary and one constant on the fourth level, $\Box$. Similar is the situation in some related programming languages as Haskell. The fourth level denotes, what we could called the type of kind, but it is quite confusing because the name of *type* is already used for one of our three levels.

So what these levels and their names exactly means? We assert that each well-formed term has its (belongs to) a type. So this is the first relation between the levels and is probably easy to understand. For notation of variables of the sort for *Types* we use Greek lower case alphabet. So, if we have expression $x : \alpha$, we claim x is of the sort *Term* and is of the type *alpha.* The α must be then of the sort *Type.*

The next relation says, that each well-formed type has its kind. For the notation of sorts for *Types* and *Kinds* we will use marks as is shown in the table I. So if we have a type variable $\alpha : \star$, we assert that the type α is of the concrete kind $\star$ (*regular type*). So, if we consider $Int : \star$, we assert that the concrete type Int is of the kind *regular type* ($\star$). However, if we have type constructor $Tree : \star \to \star$, we say that $Tree$ is of the kind $\star \to \star$ (and it is not of the kind *regular type*). It is something, what we could call, let's say, kind *type constructor* , so we could then say, that $\star \to \star$ reflects the concrete kind *type constructors.* On the other hand, both Int and $Tree$ is of the sort *Type.*

And finally the last relation, if we have $\star : \Box$, we assert that the $\star$ is of valid kind, which is of the *type* of $\Box$. $\Box$ is the only possible *type* of kinds. Note please, that the word *type* does not stand here for a sort level.

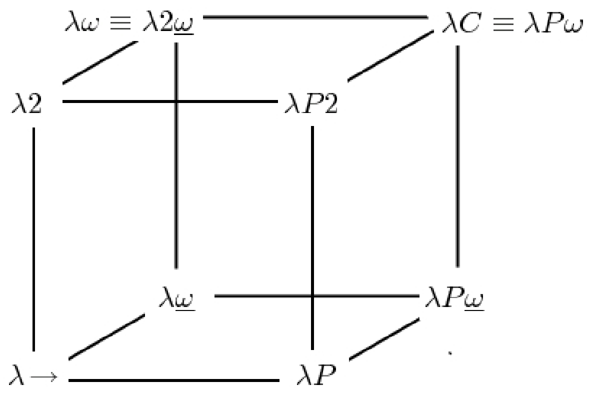
constant	description
$\star$	constant for sort *Type*
$\Box$	constant for sort *Kind*

TABLE I: Table of used constants - types, kinds

Concerning the last two relations, $\star$ means that the symbol on the left side represents a type, $\Box$ says that the symbol on the left of it is some kind of types (we considered so far only kinds of regular types and binary type constructors).

Other explanation could be found in Sorensen [1998], Jones and Meijer [1997] and Pierce [2005].

4.2 λ-cube



$\lambda\omega \equiv \lambda 2\underline{\omega}$
$\lambda C \equiv \lambda P\omega$
$\lambda 2$
$\lambda P2$
$\lambda\underline{\omega}$
$\lambda P\underline{\omega}$
$\lambda\!\rightarrow$
λP

Fig. 1: Caption text.

Lambda cube consists of eight λ-calculus-es formation as one can see on the figure 1. However only four of them are necessary to study to be able to understand all of them. Moreover, it is enough to have only one set of generic inference rules to characterize all eight of them (as we will see). The generic property of the rules is given by using variables s_1 and s_2. The variables could range over a set $\{\star, \Box\}$, where $\star$ and $\Box$ has meaning of the sort *Type* and the sort *Kind* respectively. Using simple combinatoric we receive four possible combination as is depicted in the table II. In the table one can see also the meaning of using such combination for generating a set of rules.

By incorporating the variables s_1 and s_2 to the rules, we can generate appropriate rules for each of the four base calculus-es. By combination of the resulting four sets of rules we can receive rules for the rest four calcu-

$s_1 \rightsquigarrow s_2$	dependency	resulting base system	enabled feature
$\star \rightsquigarrow \star$	terms on terms	$\lambda\!\to$	typebility
$\square \rightsquigarrow \star$	terms on types	$\lambda 2$	polymorphism
$\star \rightsquigarrow \square$	types on terms	λP	dependent types
$\square \rightsquigarrow \square$	types on types	$\lambda\omega \equiv \lambda 2\underline{\omega}$	type constructors

Table II: Table of dependencies

denotation	used for generating rules				name (lang)
$\lambda\!\to$	$\star \rightsquigarrow \star$				Simply typed
$\lambda 2$	$\star \rightsquigarrow \star$		$\square \rightsquigarrow \star$		System F_2
$\lambda\underline{\omega}$	$\star \rightsquigarrow \star$			$\square \rightsquigarrow \square$	Unnamed (Haskell)
$\lambda\omega \equiv \lambda 2\underline{\omega}$	$\star \rightsquigarrow \star$		$\square \rightsquigarrow \star$	$\square \rightsquigarrow \square$	System F_ω
λP	$\star \rightsquigarrow \star$	$\star \rightsquigarrow \square$			Dependent types
$\lambda P2$	$\star \rightsquigarrow \star$	$\star \rightsquigarrow \square$	$\square \rightsquigarrow \star$		Unnamed (Cayenne)
$\lambda P\underline{\omega}$	$\star \rightsquigarrow \star$	$\star \rightsquigarrow \square$		$\square \rightsquigarrow \square$	No name
$\lambda C \equiv \lambda P\omega$	$\star \rightsquigarrow \star$	$\star \rightsquigarrow \square$	$\square \rightsquigarrow \star$	$\square \rightsquigarrow \square$	CC (Coq)

Table III: Table of dependencies

luses. The table III contains all calculus-es and their corresponding used combinations of generated rules.

We will show here as an example only one rule, namely the rule for forming Pi-type - the one which ensures that the Π type is well formed. The rest of the rules could be found in Jones and Meijer [1997] or Sorensen [1998].

Now we will go through all four combinations:

- $\star \rightsquigarrow \star (s_1 = \star, s_2 = \star)$: This is the case, when we get the simple typed lambda calculus (if $\star \rightsquigarrow \star$ is the only combination used). This case means that A and B both must be types. So terms of the Pi-type are terms, which depends on other terms. Here we can claim that term of $\Pi x : A.B$ has laso type $A \to B$. In other words $\Pi x : A.B$ is equal to $A \to B$.
- $\square \rightsquigarrow \star (s_1 = \square, s_2 = \star)$: This combination can lead us to the system $F2$ and grant us the polymorphism feature of our type system. So A is a kind and B is a type and a $\Pi x : A.B$ Pi-type depends on a type x of the kind A.

$$\frac{\Gamma \vdash \lambda A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad \Gamma \vdash s_1 \rightsquigarrow s_2}{\Gamma \vdash (\Pi x : A.B) : s_2}$$

Fig. 2: One of the rules - product rule

- $\star \rightsquigarrow \Box(s_1 = \star, s_2 = \Box)$: Here we enter the domain of the value dependent types. We have now A as a type and B as a kind. What does it mean is, that the checked Pi-type $\Pi x : A.B$ is a kind (it reflects the sort of the B). It gives us the power to construct types which depends on terms.
- $\Box \rightsquigarrow \Box(s_1 = \star, s_2 = \star)$: Both, A and B, are some kinds ($\star$ for example). Again, the checked product type is a kind, as it again reflects B. The difference to dependent types is, that the type of the kind $\Pi x : A.B$ is now dependent on a type instead of a term. Here we get nothing else then the possibility of having type constructors.

4.3 Examples for all eight systems of λ-cube

Examples of all eigth systems follow. For each system there is only one example. If you want more examples see e.g. Sorensen [1998].

- **Example for $\lambda\!\rightarrow$:** $\alpha{:}\star, \beta,{:}\star, y : \beta \vdash \lambda x : \alpha.y : \Pi x : \alpha.\beta$
  Here, we have a demonstration of using Pi-type as a generalization of the plain arrow (function type). As the α and β are well-formed types and we are on the domain of the simple typed calculus (therefore there could be no free occurrence of x in β), we could write $A \rightarrow B$ instead of using Pi type. One can see it is really just an expression of the simply typed calculus.

- **Example for $\lambda 2$:** $\beta : \star \vdash (\lambda\alpha : \star.\lambda x : \alpha.x)\beta : \beta \rightarrow \beta$
  In this example we can see how the polymorphism is possible. We applied the expression $(\lambda\alpha : *.\lambda x : \alpha.x)$ to a type variable β, so the expression is of the type $\beta \rightarrow \beta$. Note, that we could also use a Pi-type instead of the arrow type $\beta \rightarrow \beta$.

- **Example for $\lambda\underline{\omega}$:** $\alpha : \star \vdash (\lambda\beta : \star.\beta \rightarrow \beta)\,\alpha$
  In this case, we observe the mechanism, which is known in functional programming as a type constructor. The expression $\lambda\beta : \star.\beta \rightarrow \beta$ is a parametrized type, which could be applied on other type. The expression has kind $\star \rightarrow \star$. We could use a Pi type instead of the arrow type again.

- **Example for λP:** $\alpha : *, p : \alpha \rightarrow \star, x : \alpha \vdash px$
  Here we have an example on value dependent types. The dependent part of the example is $p : \alpha \rightarrow \star$. It maps terms to types. The whole example shows the construction of a resulting type (which is of the kind regular type $\star$). One can also see here how important it is to understand the context as sequence and not as a set - to claim, that p is well-formed we have to show that α is well-formed type.

- **Example for $\lambda\omega$: $\lambda\alpha : \star.\lambda\beta : \star.\Pi\gamma : \star.\alpha \rightarrow \beta \rightarrow \gamma$.**
  This is rather complicated example, but it is necessary as the $\lambda P\omega$ is composition of two already mentioned systems ($\lambda\underline{\omega}$ and $\lambda 2$). We will start with description from the rear. So, expression $\Pi\gamma : \star.\alpha \rightarrow \beta \rightarrow \gamma$ represents a polymorphic type (notice the $\gamma : \star$) and is the one, which could be created using plain $\lambda 2$ system. As the next we can see, that we abstract over this expression ($\lambda\alpha : \star.\lambda\beta : \star$ part) and to be able to do that we need to have properties of the $\lambda\underline{\omega}$.

- **Example for $\lambda P2$:** $\lambda\alpha : \star.\lambda p : \alpha \rightarrow \star.\lambda x : \alpha.\lambda y : (px).y : \Pi\alpha : \star.\Pi p : \alpha \rightarrow \star.\Pi x : \alpha.px \rightarrow \Pi x : \alpha.px$
  In this example one can see a combination of dependent types and polymorphism. The polymorphism is given by introducing type variable α in the very beginning of the expression. The second abstraction, $p : \alpha \rightarrow \star$ could be created and used only when value dependent types (λP system) is employed. As one can see the expression is an identity function for a variable, which has a polymorphic dependent type as its type.

- **Example for $\lambda P\underline{\omega}$:** $\alpha : \star \vdash \lambda p : \alpha \rightarrow \star : (\alpha \rightarrow \star) \rightarrow (\alpha \rightarrow \star)$ Stars ($\star$) on both sides of the type of the expression leads to the fact, that this expression is creatable only in a system, which is at least as powerful as $\lambda\underline{\omega}$. However, the expression also contains dependent types (p is a dependent type).

- **Example for λC :** $\lambda\alpha : \star.\lambda p : \alpha \rightarrow \star.p : \Pi\alpha : \star.(\alpha \rightarrow \star) \rightarrow (\alpha \rightarrow \star)$
  We can see here an easy example of *Calculus of constructions*. The lambda expression is an type identity function employing all systems, which are covered by the lambda cube. In the beginning of the expression one can see a type variable, which grants us polymorphism ($\lambda 2$). As the next, there is a variable p for a dependent type (λP). And as the last, note the type of the expression (or *kind* better to say) - on both sides of the arrow, there is a star $\star$, which gives us a hint, that also $\lambda\underline{\omega}$ system part is employed.

References

Augustsson, Lennart. 1998. Cayenne – a Language with Dependent Types. In *International Conference on Functional Programming*, 239–250.

Betarte, Gustavo. 1998. *Dependent Record Types and Formal Abstract Reasoning*. PhD thesis, Chalmers University of Technology.

Jones, Simon Peyton and Meijer, Erik. 1997. Henk: a typed intermediate language. In *Henk: a typed intermediate language*.

Larrson, Staffan. 1998. A prolog typechecker for dependent record types. In *Workshop on Types in Compilation*.

Pierce, Benjamin C., Editor. 2005. *Advanced Topics in Types and Programming Languages*. MIT Press.

RYSAVY, ONDREJ. 2005. *Specifying and reasoning in the calculus of objects*. PhD thesis, Brno University of Technology.

SKARVADA, LIBOR. 2007. Can objects have dependent types? In *MEMICS 2007*.

SORENSEN, MORTEN HEINE B. 1998. *Lectures on the Curry-Howard isomorphism*. unknown.

THOMPSON, SIMON. 1991. *Type Theory and Functional Programming*. Addison-Wesley.