SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

Some simple proofs related to the Univalence Axiom formalized in Dedukti

av

Per Nilsson

2018 - No K12

Some simple proofs related to the Univalence Axiom formalized in Dedukti

Per Nilsson

Självständigt arbete i matematik 15 högskolepoäng, grundnivå

Handledare: Erik Palmgren

2018

Acknowledgment

I would like to thank my supervisor Erik Palmgren for his help with all the different versions of this document. I would also like to thank reviewer Johan Lindberg for comments that really made the document better.

1 Introduction

Dedukti is a LF based on the $\lambda\Pi$-calculus. It is similar to the Edinburgh LF [9] but it has been enhanced with higher order rewriting. Higher order rewriting [12] lifts results from term rewriting to typed λ-calculus. This enables Dedukti to include user defined rewrite rules in its type checking algorithm. To ensure subject reduction and uniqueness of types in Dedukti, the user added rewrite rules needs to be confluent. If they also are terminating the type checking becomes decidable [15]. The confluence and termination of the user defined rewrite rules are delegated to external tools.

Logical Frameworks have a long history going back to the Automath project in the 1960s. They are used to implement different kinds of logical systems in a fairly simple way. In [2] a number of examples of implemented logics can be found. This includes Heyting arithmetic, Hol Light proofs and correctness proofs of programming languages. An implementation of Calculus of Construction (CC) has been done in Dedukti, this can be seen as an example of how to implement PTS [4] in a LF. There are also examples of embedding logics in Logical Frameworks in [9] and [14].

Dedukti has no support for implicit arguments, tactics or user defined syntax which is commonly supported in modern proof assistants like for example Coq. To implement a logic in Dedukti a user can define typed constants and rewrite rules. Logical statements are expressed as types and a proof of a statement is a term with the corresponding type. This is the Curry-Howard correspondence between propositions and types. Dedukti has only one accessible universe of types, but it is possible to implement universe hierarchies in similar manner as outlined in [13] and [8]. Since Dedukti is an implementation of $\lambda\Pi$ it can be used to express predicate logic without much effort [4], [2]. Dedukti's $\lambda\Pi$ implementation is minimal so what can be expressed without extra coding is universal quantification and implication. Computation is expressed as rewrite rules which supports higher order patterns.

2 Term rewriting

Term rewriting is a well studied model for computational processes. Maybe the most well known rewriting system is λ-calculus. It is a Higher Order Rewrite System (HRS), since λ-abstractions binds variables. A first order rewriting system is normally introduced in two steps. First by defining the notion of an Abstract Reduction System (ARS), then defining a signature Σ describing the set of terms which are reduced by the ARS. As an informal example of a first order rewrite system consider the following.

Example 1 (Addition of unary numerals)**.** *Unary numerals are generated by the constant Z and the function S. Addition is defined by the following rewrite relations.*

$plus(Z, n) \longmapsto n$
$plus(S(n), m) \longmapsto S(plus(n, m))$

The following reduction of $2 + 1$ to the value 3 illustrates the above mentioned rewrite relation.

1. $plus(S(S(Z)), S(Z)) \longmapsto S(plus(S(Z), S(Z)))$
2. $S(plus(S(Z), S(Z))) \longmapsto S(S(plus(Z, S(Z))))$
3. $S(S(plus(Z, S(Z)))) \longmapsto S(S(S(Z)))$

The terms of first order rewrite systems are simple, they consist of variables and constants. A constant can take zero or more arguments, if it takes at least one argument then the constant is a function symbol. Since we are mainly interested in defining some properties of the reduction relation $\longmapsto$, we will not develop signatures any further. The following is taken from [3] and [11].

Definition 2. *An abstract reduction system is a pair $(A, \longmapsto)$ where $\longmapsto$ is binary relation on a set A. Instead of $(a, b) \in \longmapsto$ we write $a \longmapsto b$ and call b a one step reduct of a.*

1. *The reflexive transitive closure of $\longmapsto$ is denoted $\xrightarrow{*}$.*
2. *The reflexive transitive symmetric closure of $\longmapsto$ is denoted $\overset{*}{\leftrightarrow}$*
3. *$\longmapsto$ is confluent if for all $a, b, c \in A$ there exists a $d \in A$ s.t. $c \xleftarrow{*} a \xrightarrow{*} b$ implies $c \xrightarrow{*} d \xleftarrow{*} b$*

4. *$a \in A$ is a normal form if there are no $b \in A$ s.t. $a \longmapsto b$. If a has a unique normal form it is denoted $a\downarrow$.*

5. *$\xrightarrow{*}$ is normalizing if every element has a normal form.*

6. *$\longmapsto$ is strongly normalizing (SN) if every reduction sequence $a_1 \longmapsto a_2 \longmapsto \ldots$ eventually terminates (also called terminating or Noetherian).*

If a reduction relation is confluent then for a given term a if the term reduces to two different terms, say b and c, then there are reduction paths from b and c to a common term d. One consequence of confluency is that when searching for expressions to reduce, any search strategy will do. The reason for this is that all reduction paths will sooner or later end up in a common term.

Theorem 3 ([3, 2.1.9])**.** *If $\longmapsto$ is confluent and normalizing then $a \overset{*}{\leftrightarrow} b \quad \Leftrightarrow a\downarrow = b\downarrow$*

Proof. Omitted. A proof can be found in [3] □

This means that if a rewrite system over a set A is confluent and normalizing and if also the equality on A is decidable, then there is a procedure to decide the induced equivalence. It's simply to compute normal forms and compare the resulting elements. If the relation is strongly normalizing we also know that this procedure will always terminate.

3 λ-calculus

λ-calculus [4] is a theory of functions invented by Alonzo Church in the 1930s. It has been very successful as a model for computable functions. It exists in both typed and untyped variants.

Typed λ-calculus can be grouped into two distinct variants depending on how types are assigned. Today maybe the most common variant is where types are inferred from untyped terms. This version exist in programming languages like ML and Haskell. It turns out that for more advanced types of λ-calculi it is not possible to infer types without type annotations.

Named after the inventors, systems with type annotations are commonly called Church systems and systems without annotations are called Curry

systems. In this text type annotations will play a crucial role so we will only consider Church systems. The following introduces simply typed λ-calculus with type annotations and is based on [4].

Definition 4. *Let V denote an infinite set of variables and let $x \in V$, then the syntax of simply typed λ-calculus is defined as follows.*

types : $\alpha, \beta, \tau \in T ::= \alpha | (\alpha \to \tau)$

terms : $M, N, P, Q \in \Lambda_T ::= x \mid (\lambda x : \alpha.M) \mid (MN)$

The following conventions reduces the number of parentheses needed to be written.

- Types associates to the right so $(\alpha_1 \to (\alpha_2 \to \cdots \to \alpha_n \dots))$ is normally written $\alpha_1 \to \alpha_2 \to \cdots \to \alpha_n$.
- Applications associates to the left so $(\dots((FM_1)M_2)\dots M_n)$ is normally written $FM_1M_2\dots M_n$.
- Abstractions associates to the right and multiple variable bindings can be expressed using one λ so $(\lambda x_1 : \alpha_1.(\lambda x_2 : \alpha_2 \dots (\lambda x_n : \alpha_n.M)))$ is normally written $\lambda x_1 : \alpha_1.x_2 : \alpha_2 \dots x_n : \alpha_n.M$
- The outermost parentheses are not written if not needed.

Definition 5. *Definition of context and the domain of a context.*

1. *A context Γ (called basis in [4]) is a set of declarations of distinct variables with type annotations.*
2. *Let $\Gamma = x_1 : \sigma_1, \dots, x_n : \sigma_n$ then $dom(\Gamma) = \{x_1, \dots, x_n\}$*

Definition 6. *Let $M \in \Lambda_T$ then $M : \alpha$ is derivable from a context Γ if, M can be produced using the following rules.*

$$(axiom) \quad \Gamma \vdash x : \alpha, \quad if (x : \alpha) \in \Gamma$$

$$(\to -elimination) \frac{\Gamma \vdash M : (\alpha \to \tau) \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash (MN) : \tau}$$

$$(\to -introduction) \frac{\Gamma, x : \alpha \vdash M : \tau}{\Gamma \vdash (\lambda x : \alpha.M) : (\alpha \to \tau)}$$

Using the rules defined in Definition 6 we can now derive well typed λ-terms. The following is an example.

$$\dfrac{\dfrac{x:\alpha, y:\tau \ \vdash \ x:\alpha}{x:\alpha \ \vdash \ (\lambda y:\tau.x):(\tau \to \alpha)}}{\vdash \ (\lambda x:\alpha.\lambda y:\tau.x):(\alpha \to \tau \to \alpha)}$$

Function types have the same visual appearance as logical implication and it is well known that there is a correspondence between types and propositions. The above is actually the derivation of the propositional tautology normally written $A \to B \to A$, and the λ-term is regarded as a proof of it.

Next we introduce the equational theory induced by the β axiom schema, the definition of the schema uses the concept of a substitution. If names of bound variables are chosen so that they differ from the free variables then substitution can be defined without side conditions on free and bound variable.

Definition 7 (Free variables and variable substitution [4, 2.1.5])**.**

1. *The set of free variables of M, (notation FV(M)), is defined inductively as follows.*

   $FV(x) = \{x\}$

   $FV(MN) = FV(M) \cup FV(N)$

   $FV(\lambda x:\alpha.M) = FV(M) - \{x\}$

2. *The result of substituting N for x in M (notation M[x := N]) is inductively defined as follows (Below $x \not\equiv y$).*

   $x[x := N] \equiv N$

   $y[x := N] \equiv y$

   $PQ[x := N] \equiv (P[x := N])(Q[x := N])$

   $(\lambda y:\alpha.P)[x := N] \equiv \lambda y:\alpha.(P[x := N])$

   $(\lambda x:\alpha.P)[x := N] \equiv \lambda x:\alpha.P$ *(Note that $x \notin FV(\lambda x:\alpha.P)$)*

Also worth noting is that λ-terms are identical modulo renaming of bound variables, i.e. $(\lambda x:\alpha.x):\alpha \to \alpha \equiv (\lambda y:\alpha.y):\alpha \to \alpha$. The process of renaming variables of λ-terms is called α-conversion, it will not be used in this

document since it is somewhat redundant when using the above mentioned variable convention.

The following introduces an equational theory induced by the β-axiom schema and is based on the untyped version presented in [5]. It is defined using the standard equality axioms and rules, i.e reflexivity, symmetry, transitivity and congruence.

Definition 8. *Assume $M, N, P, Q \in \Lambda_T$ and $N : \alpha$ then β-rule is defined by the axiom schema* (1)*. The congruence induced by β-rule include congruence with respect to λ-abstraction as defined by* (2)*. It is also well known that the equational theory induced by the β-rule can be seen as a reduction relation as in* (3)*.*

$$(\lambda x : \alpha.M)N = M[x := N] \tag{1}$$

$$P = Q \Rightarrow \lambda x : \alpha.P = \lambda x : \alpha.Q \tag{2}$$

$$(\lambda x : \alpha.M)N \longmapsto_\beta M[x := N] \tag{3}$$

The following is a simple example of β-reduction which also illustrates how multiple arguments to a function is handled.

Example 9. *Assume that there exists a base type "int", with constants 1, 3 of type "int" and also the λ-expression $\lambda x : int.y : int.x$, then the following is a sequence of one step reducts resulting in a β normal form.*

1. $\lambda x : int.y : int.x\ 1\ 3$
2. $\lambda y : int.1\ 3$
3. 1

If $M \in \Lambda_T$ and $M : \tau$ then the property that M still has type τ after the term has been completely reduced is stated in the following theorem. The theorem is called *subject reduction* and gets its name from the convention that in an expression of the form $M : \tau$, M is called subject (τ is called predicate).

Proposition 10 (Subject reduction [4, 3.2.11]). *Let* $M, M' \in \Lambda_T$ *and* $M \xrightarrow{*}_\beta M'$ *then.*

$\Gamma \vdash M : \alpha \Rightarrow \Gamma \vdash M' : \alpha$

Proof. Omitted. A proof for the Curry system can be found in [4] □

The uniqueness of types property states that for a given term all its types are convertible.

Proposition 11 (Uniqueness of types [4, 3.2.12]). *Let* $M, M' \in \Lambda_T$ *then.*

1. *Assume* $\Gamma \vdash M : \alpha$ *and* $\Gamma \vdash M : \alpha'$ *then* $\alpha \equiv \alpha'$
2. *Assume* $\Gamma \vdash M : \alpha$, $\Gamma \vdash M' : \alpha'$ *and* $M \xrightarrow{*}_\beta M'$ *then* $\alpha \equiv \alpha'$

Proof. Omitted. A proof for Curry system can be found in [4] □

It is also worth noting that simply typed λ-calculus is strongly normalizing, in fact this is a property of all systems of the λ-cube [4] (see below).

4 Higher Order Rewrite Systems

HRS are a generalization of first order rewrite systems where the set of terms includes λ-terms as well as constants and variables. Properties like confluence and strong normalization applies to these types of rewrite systems too. The following example taken from [3] illustrates HRS.

Example 12. *Logical expressions including quantifiers can be encoded using higher order functions where* λ*-terms are used to bind variables. The higher order rewrite relation in* (4) *can be seen as a computational form of the logical expression in* (5).

$$all(\lambda x.(P(x) \wedge Q(x))) \longmapsto all(\lambda x.P(x)) \wedge all(\lambda x.Q(x)) \tag{4}$$

$$\forall x.(P(x) \wedge Q(x)) \rightarrow \forall x.P(x) \wedge \forall x.Q(x) \tag{5}$$

In general higher order rewriting is undecidable but restricting λ-terms to higher order patterns makes higher order rewriting tractable [12].

We will now define the notion of a higher order pattern and it uses the concept of η-equivalent terms. This concept can be defined on typed lambda calculus but since it is not used by Dedukti or in [4], which defines properties of λ-cube, it is defined here for untyped terms and it will only be used in this section of the document.

Definition 13 (η-rule [5])**.** *Assume $M, N,$ are untyped λ-terms where $x \notin FV(M)$ then the η-rule is defined by the following axiom schema.*

$\lambda x.(Mx) = M$

Definition 14 (η-equivalence [12])**.** *If two untyped λ-terms s, t are equivalent according to the η-rule we say that s and t are η-equivalent.*

Definition 15 (Higher order pattern [12, 3.1])**.** *A λ-term t in β-normal form is a higher order pattern if every free variable F occurs in a subterm $F\overrightarrow{u}$ of t, s.t. $\overrightarrow{u}$ is a possibly empty η-equivalent list of distinct bound variables.*

The following example taken from [12] might help to interpret Definition 15.

Example 16. *Let c be a constant and F,G and H be free variables then the following are valid patterns.*

1. *$\lambda x.(cx)$, it contains no free variables so it is trivially a pattern.*
2. *F, it is a pattern since the list of bound variables are allowed to be empty.*
3. *$\lambda x.(F\lambda z.(xz))$, in this case F is applied to the term $\lambda z.(xz)$ which is η-equivalent to x, i.e. F is applied to a list of distinct bound variables.*
4. *$\lambda x.\lambda y.(Fyx)$, F is applied to the distinct bound variables $\{x, y\}$.*

The following are not patterns.

1. *(Fc), F is applied to c which is not a bound variable, its a constant.*
2. *$\lambda x.(Fxx)$, in this case the there are two instances of x which makes it a non pattern.*
3. *$\lambda x.\lambda y.(Fyc)$, F is applied to the constant c as well as the bound variable y.*
4. *$\lambda x.(G(Fx))$, in this case G is applied to an application of a free variable which is not allowed.*

5 Dedukti — a Logical Framework.

$\lambda\Pi$ is one of the potential extensions of simply typed λ-calculus. It introduces the concept of types that depends on terms, which is one of the possible type dependencies. When type dependencies are introduced in a calculi there is a need for the concept *Kind*, which is the type of *Types*.

Definition 17. *Let $S = \{*, \Box\}$ denote the set of sorts, $*$ is pronounced "Type" and $\Box$ is pronounced "Kind".*

Sorts corresponds to *universes* in Intuitionistic Type Theory (ITT), with $*$ and $\Box$ we have a two level hierarchy of *universes*. To illustrate term dependent types consider the following example taken from [4].

Example 18. *Assume $A : *$, $a : A$, i.e. A is a type and a has type A. Also assume $f : (A \to *)$ then $(A \to *) : \Box$ and $(fa) : *$, which is a term dependent type.*

Dedukti is an implementation of $\lambda\Pi$ where the congruence relation on types not only considers β-reduction, but also include the congruence relation induced by user defined rewrite rules. This is indicated by subscripting the $\equiv$ symbol with a Γ as well as a β, i.e. $\equiv_{\beta\Gamma}$. Let $\lambda\Pi R$ denote this variant of the $\lambda\Pi$ calculus. The following presentation of the meta theory of $\lambda\Pi R$ is mainly taken form [1], but it is thoroughly investigated in [16]. The following defines the syntax of $\lambda\Pi R$.

Definition 19 (Syntax)**.** *Let V denote an infinite set of variables and let $x \in V$ then the syntax of $\lambda\Pi R$ is defined as follows.*

sorts: $s \in S ::= * \mid \Box$
terms: $M, N, A, B \in T ::= s \mid x \mid \Pi x : A.B \mid \lambda x : A.M \mid MN$
contexts: $\Sigma, \Gamma, \Delta, \in G ::= \emptyset \mid \Gamma, x : A \mid \Gamma, R$
rewrite rules: $R ::= \emptyset \mid R, M \longmapsto N$

Note that if in a type $\Pi x : A.B$, B does not depend on A, then its customary to use the normal function type syntax, i.e. $A \to B$.

In the following typing rules the condition that a rewrite rule needs to be *well formed* is denoted by R *WF*. A rule is *well formed* if it is well typed and and the if the relation $\equiv_{\beta\Gamma}$ is product compatible (see below for details).

Definition 20. *Let $s \in \{*, \Box\}$, then $M : A$ is derivable from a context Γ if M can be produced using the following rules.*

$$(empty \quad context) \quad \emptyset \vdash$$

$$\textit{(well formed rewrite rules)} \frac{\Gamma \vdash R \quad WF}{\Gamma, R \vdash}$$

$$(axiom) \quad \Gamma \vdash * : \Box$$

$$\textit{(Start rule)} \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \textit{ (if } x \notin dom(\Gamma)\textit{)}$$

$$\textit{(Product rule)} \frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x : A.B : s}$$

$$\textit{(Abstraction rule)} \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.B : s}{\Gamma \vdash (\lambda x : A.M) : (\Pi x : A.B)}$$

$$\textit{(Application rule)} \frac{\Gamma \vdash M : \Pi x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$$

$$\textit{(Conversion rule)} \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \equiv_{\beta\Gamma} B}{\Gamma \vdash M : B}$$

Pure $\lambda\Pi$ without user defined rewrite rules, enjoys properties which includes subject reduction, uniqueness of types, strong normalization and decidable type checking. For these properties to hold for $\lambda\Pi R$ some specific conditions must hold for the rewrite rules, they must be well formed. These conditions are described below. Note that product compatibility follows trivially from confluency of rewrite rules.

In the definitions below a context Γ is a global context. In the global context there are definitions of typed constants and their associated rewrite rules. A local context is a set of variable declarations where each variable has an associated type [15]. The free variables of a pattern are declared in a local context i.e. they are not visible globally.

Definition 21 (Product compatibility [1, 3.2.4])**.** *The congruence relation* $\equiv_{\beta\Gamma}$ *is product compatible if* $\Pi x : A_1.B_1 \equiv_{\beta\Gamma} \Pi x : A_2.B_2$ *implies* $A_1 \equiv_{\beta\Gamma} A_2$ *and* $B_1 \equiv_{\beta\Gamma} B_2$*.*

Definition 22 (Rule typing [1, 3.2.5]). *A rewrite rule $M \longmapsto N$ is well typed in Γ if for any substitution σ of variables free in Γ then $\Gamma \vdash \sigma(M) : A$ implies $\Gamma\ \Gamma \vdash \sigma(N) : A$.*

Theorem 23 (Subject reduction [1, 3.2.9]). *If $M \xrightarrow{*}_{\beta\Gamma} M'$ and $\Gamma \vdash M : A$ then $\Gamma \vdash M' : A$*

Proof. Follows from Product compatibility and welltypedness of rewrite rules. A detailed proof can be found in [16]. □

Theorem 24 (Uniqueness of type [1, 3.2.10]). *If $\Gamma \vdash M : A$ and $\Gamma \vdash M : A'$ then $A \equiv_{\beta\Gamma} A'$*

Proof. Follows from Product compatibility and welltypedness of rewrite rules. A detailed proof can be found in [16]. □

The completeness property *decidable type checking* depends on the restriction of HRS to a class of patterns where matching modulo β-reduction is decidable. The following definition is essentially the same as Definition 15 but the Dedukti software does not implement the η-rule so the definition is slightly different.

Definition 25 (Pattern [1, 3.2.13]). *A pattern is a term $P = c\overrightarrow{N}$ where $c \in dom(\Gamma)$ and, for any variable x that is free in Γ, if x appears in P then it only appears in the form $x\overrightarrow{y}$ where y is a list of distinct variables that are bound in P. A rewrite rule is called a "pattern rule" in Γ if its left-hand side is a pattern in Γ.*

The following example illustrates higher order patterns in relation to a context Γ.

Example 26 ([1, 3.2.14]). *Assume a context $\Gamma = \{Real : Type, exp : Real \to Real, D : (Real \to Real) \to (Real \to Real)\}$ then $D(\lambda x.exp(fx))$ is a pattern. The expression $D(\lambda x.g(fx))$ is not a pattern, since g is a free variable that is not applied to a list of distinct bound variables.*

Theorem 27 (Welltypedness [1, 3.2.15]). *A rewrite rule $M \longmapsto N$ is well typed in Γ if M is a pattern and there exist a local context Δ and a type A s.t. $dom(\Delta) \subseteq FV(M)$ and $\Gamma, \Delta \vdash M : A$ and $\Gamma, \Delta \vdash N : A$.*

Proof. Omitted. A proof can be found in [16]. □

Theorem 28 (Decidibility of type checking [1, 3.2.16])**.** *If all rewrite rules in* Γ *are pattern rules and* $\longmapsto_{\beta\Gamma}$ *is confluent and strongly normalizing then type checking is decidable.*

Proof. Omitted. A proof can be found in [16]. □

To prove that the pattern rules are both confluent and strong normalizing is in general undecidable, but see [1] for a discussion. According to [1] rewrite rules that are not confluent can result in false negatives and rewrite rules that are not strongly normalizing can result in non terminating algorithms. In practice the soundness of the system is not compromised as long as product compatibility is maintained (see chapter 3.2 [1]).

6 The λ-cube and Pure Type Systems

The λ-cube [4] is framework for relating eight versions of typed λ-calculi to each other. Simply typed λ-calculus, $\lambda\Pi$, System F and CC are examples of calculi in the λ-cube. In the λ-cube all calculi are based on the same abstract syntax, and β-axiom schema. The typing rules can be grouped in two categories, general rules and a specific rule. The general rules applies to all calculi and are essentially the rules for $\lambda\Pi$ minus the *product rule*. The specific rule is a parametric introduction rule for the dependent product. The following are the typing rules for the λ-cube.

Definition 29. *Let* $s, s_1, s_2 \in \{*, \Box\}$ *then* $M : A$ *is derivable from a context* Γ *if, M can be produced using the following rules.*

$$(axiom) \quad \Gamma \vdash * : \Box$$

$$\textit{(Start rule)} \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A} \textit{ (if } x \notin \Gamma\textit{)}$$

$$\textit{(Weakening rule)} \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B} \ (x \notin \Gamma)$$

$$\textit{(Product rule)} \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.B : s_2}$$

$$\textit{(Abstraction rule)}\ \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi x : A.B : s}{\Gamma \vdash (\lambda x : A.M) : (\Pi x : A.B)}$$

$$\textit{(Application rule)}\ \frac{\Gamma \vdash M : \Pi x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[x := N]}$$

$$\textit{(Conversion rule)}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s \qquad A \equiv_\beta B}{\Gamma \vdash M : B}$$

Example 30. *The following is an example of sort combinations for some well known systems.*

1. $\lambda\Pi$ *is specified with* $(s_1, s_2) \in \{(*, *), (*, \Box)\}$.
2. *System F with* $(s_1, s_2) \in \{(*, *), (\Box, *)\}$.
3. *CC with* $(s_1, s_2) \in \{(*, *), (\Box, *), (*, \Box), (\Box, \Box)\}$.

The different combinations of sorts specifies what can be expressed in the type system.

Example 31. *The following shows some examples of the expressiveness of sort combinations.*

1. *The tuple* $(*, \Box)$ *makes term dependent types possible. Example:* $Vec : N \to *$. *Which is the type of vectors depending of its size.*
2. *The tuple* $(\Box, *)$ *makes polymorphism possible. Example:* $(\lambda A : *.x : A.x) : \Pi A : *.A \to A$. *Which is the polymorphic identity function.*

The λ-cube is generalized to PTS [4], where the specification of a λ-calculi is done by using a triple (S, A, R).

1. S a set of sorts, e.g.: $\{*, \Box\}$
2. A a set of axioms, which relates sorts to each other, e.g.: $\{* : \Box\}$.
3. R a set of rules, which are of the form (s_1, s_2, s_3) where $s_i \in S$. The λ-cube is a restriction to the following tuple (s_1, s_2, s_2).

For PTS the dependent product rule is as follows.

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x : A.B : s_3} \, (s_1, s_2, s_3 \in R)$$

There are several reasons to introduce PTS [4]. Its easier to express subtle differences in λ-calculi in PTS than using the λ-cube framework. Also proving properties of a λ-calculus is easier in the PTS framework. It has already been mentioned that all calculi in the λ-cube are strongly normalizing. This is not in general true for PTS.

Example 32. *Let $\lambda*$ denote the PTS with $(S, A, R) = (\{*\}, \{* : *\}, \{(*, *, *)\})$. $\lambda*$ is not strongly normalizing and all types are inhabited [4].*

The λ-cube and PTS also relates to something called the logical cube [4]. The logical cube is essentially a mapping from λ-calculi to logical systems using the proposition-as-types paradigm.

Example 33. *The following exemplifies the mapping from the λ-cube to the logical cube.*

1. *$\lambda\Pi$ maps to predicate logic.*
2. *System F maps to second order propositional logic.*
3. *CC maps to higher order predicate logic.*

7 Embeddings in Dedukti

The λ-cube and PTS are defined with a two level hierarchy of universes. This is a simplification compared to the standard way to avoid Girard's paradox [6], [4], which is to construct an infinite hierarchy of universes that are closed under type forming operators. There are two styles of representing universes [13], [1].

1. Russell style, in this style there is no distinction between types and terms. An object (type or term) is simply declared to have a type. This style is used by PTS and systems like Coq, Agda, etc.. For example in Coq we have $nat : Set$, $Set : Type_0$ and $Type_0 : Type_1$.
2. Tarski style, in this style there is a set of codes for types in a universe U_i. There is also a decoding function T_i that takes a code and returns a type.

In [1], [7] it is shown how to embed PTS in Dedukti. The main problem when doing the embedding is that in more advanced typed λ-calculi, complex types can be used as arguments to functions and this is not supported in $\lambda\Pi$. In Dedukti's $\lambda\Pi$ implementation only terms of type *Type* can be passed to functions. This problem is solved by using mapping functions similar to the decoding functions in Tarski style universe. This means that a type A will have a term representation $|A|$ as an element (code) in U and a type representation $\|A\|$, and the decoding function maps from term representation to type representation i.e. $T(|A|) = \|A\|$, see [7] for details.

The mapping function is implemented using rewriting and in [1], [7] correctness properties of the embedding is proved. The embedding can be specified with the following. For each sort s declare a variable as in (6) with its associated decoding function (7). For each axiom implement a type as in (8) and its associated rewrite rule (9).

$$U_s : Type \tag{6}$$

$$T_s : U_s \to Type \tag{7}$$

$$u_{s_1} : U_{s_2} \tag{8}$$

$$T_{s_2}(u_{s_1}) \to U_{s_1} \tag{9}$$

For each rule there is also a need to declare a higher order abstract syntax function as in (10) and its associated rewrite rule (11).

$$\pi_{s_1,s_2} : \Pi a : U_{s_1}.(T_{s_1} a \to U_{s_2}) \to U_{s_3} \tag{10}$$

$$T_{s_3}(\pi_{s_1,s_2}(a,b)) \to \Pi x : T_{s_1} a.T_{s_2}(bx) \tag{11}$$

7.1 CC embedding code

In this chapter we will show how the above equations can be translated to Dedukti code. To get an understanding of how logical constants can be implemented we will also implement the *prod*, *Id* and *id* constants.

The implementation of Π-types are part of the Higher Order Abstract Syntax (HOAS) implementation of the PTS rules. It is customary to differentiate between concrete and abstract syntax. The concrete syntax is the syntax of the actual language and the abstract syntax is the syntax generated by the parsing software. I.e. the expression *1 + 2* could be parsed and the

abstract syntax *Plus(1, 2)* could be generated. An abstract syntax is *higher order* when it uses functions in the implementation language to implement functions in the implemented language. As we will see the usage of HOAS will affect the readability of the Dedukti code.

We begin with implementing the sorts of the PTS. As already noted there are two sorts, $*$ and $\Box$. As *Type* is a reserved keyword in Dedukti *Set* is chosen as the the type of the *small types* i.e. bool, nat etc. The type of *Set* is *Kind* so the *Kind* constant is defined too.

Dedukti differentiates between static and dynamic constants. Dynamic constants can have rewrite rules associated to them and since the constants *Set* and *Kind* does not have any associated rewrite rules they are static.

Listing 1: Sorts

```
Set : Type.
Kind : Type.
```

With the definitions of *Set* and *Kind* above we get the following relationship between the sorts and types in the system.

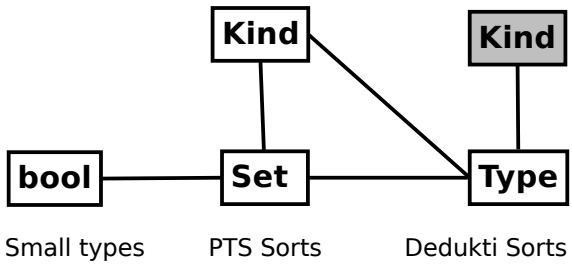

Figure 1: Sorts and types

As we can see Dedukti has two sorts *Type* and *Kind* where *Kind* is internal to Dedukti. The PTS sorts *Set* and *Kind* are implemented as elements of the sort *Type*. The *small types* in the system will as we already have mentioned be elements of the sort *Set*.

The sorts in the PTS need associated decoding functions. The decoding functions are implemented using the *def* keyword. This tells us that they are dynamic constants that can have rewrite rules associated to them. The actual rewrite rules are defined in places where the constants are defined that needs to be rewritten.

Listing 2: Decoding functions

```
def T : Set -> Type.
def K : Kind -> Type.
```

The axiom of CC in PTS states that *Set* has type *Kind*. This is implemented in the embedding as a constant *set* of type *Kind* and an associated rewrite rule for the K decoding function. As we can see the constant *set* is decoded to the constant *Set*. The brackets in the rewrite rule denotes the local context (local variables) in this case it is empty. Note that the extra long arrow in the second line is used to indicate rewriting.

Listing 3: Axiom

```
set : Kind.
[] K set --> Set.
```

The general rules of the PTS are implemented using HOAS. The types below encodes the Π-type for CC. As we can see a is a value of either type *Set* or *Kind*. Note also that Dedukti's concrete syntax uses function type syntax for dependent product types. This can be seen in the type of the variable b. The type depends on a since it is a function from a to either *Set* or *Kind*. There are also associated rewrite rules that maps the abstract syntax to Dedukti's $\lambda\Pi R$ syntax.

To implement any non trivial examples these functions needs to be nested, see below for examples. Also note that the Π-type corresponds to universal quantification when the λ-calculus is viewed as a logic.

Listing 4: Π-type

```
pi_ss : a : Set -> b :  (T a -> Set) -> Set.
pi_sk : a : Set -> b :  (T a -> Kind) -> Kind.
pi_ks : a : Kind -> b :  (K a -> Set) -> Set.
pi_kk : a : Kind -> b :  (K a -> Kind) -> Kind.

[a, b] T(pi_ss a b) --> x : T a -> T(b x).
[a, b] K(pi_sk a b) --> x : T a -> K(b x).
[a, b] T(pi_ks a b) --> x : K a -> T(b x).
[a, b] K(pi_kk a b) --> x : K a -> K(b x).
```

7.2 Product type

The following is an implementation of the product type. The product type can be viewed as the cartesian product of two sets. If it is viewed in a logical context it is simply a conjunction between two propositions. The *prod* constant has the following shape.

prod : Set $\rightarrow$ *Set* $\rightarrow$ *Set*. Since it is a construct from two instances of the the type *Set* to *Set* we need nested uses of the Π-type HOAS with *Kind* as both input and return type. We also need to use the axiom stating that *Set* has type *Kind*. Note that in the expression below "`=>`" is Dedukti's notation of a λ-function and that "`__`" denotes a variable name that is not used.

Listing 5: Product type

```
prod : K (pi_kk set (__ : K set =>
          (pi_kk set (__ : K set => set )))).
```

Since the Π-HOAS is an implementation of the general rule of the CC PTS the above expression can be justified by the following derivation tree of the *prod* constant's type.

$$\dfrac{\Gamma \vdash * : \Box \qquad \dfrac{\dfrac{\Gamma \vdash * : \Box \qquad \Gamma \vdash * : \Box}{\Gamma, x : * \vdash * : \Box}\,(1)}{\Gamma \vdash (\Pi x : *.*) : \Box}\,(2) \qquad \Gamma \vdash * : \Box}{\Gamma, y : * \vdash (\Pi x : *.*) : \Box}\,(3)$$

$$\dfrac{\Gamma \vdash * : \Box \qquad \Gamma, y : * \vdash (\Pi x : *.*) : \Box}{\Gamma \vdash (\Pi y : *.\Pi x : *.*) : \Box}\,(4)$$

As we can see the inferences 2 and 4 are two uses of the general dependent product rule using the tuple $(\Box, \Box)$. The left premises of 1, 2 and 4 are three uses of the axiom $\Gamma \vdash * : \Box$. The inference rules 1 and 3 are instances of the weakening rule which are used to add a variable to the context Γ.

7.3 Rewriting the id type

To get some understanding of how rewriting of HOAS works in Dedukti we will use an expression involving the reflexivity axiom for the identity type and boolean values. *Small types* can be defined in a similar way as inductive types are defined in Agda or Coq. In Dedukti you first define the actual type, in this case its *bool*, then you define the constructors. The following is its implementation.

Listing 6: Boolean type

```
bool : Set.
true : T bool.
false : T bool.
```

Next we implement the *Id* constant. It can be used to express that two values of a *small type* are equal. It has the following shape.

Id : A : Set $\rightarrow$ *A* $\rightarrow$ *A* $\rightarrow$ *Set.* The first argument is a *small type A* and the second and third arguments are values of the type *A*. The following is how it is implemented using HOAS.

Listing 7: Id and bool constants

```
Id : K (pi_kk set (A : K set =>
        (pi_sk A (__ => pi_sk A (__ => set))))).
```

One way to prove that two objects are equal is by using the reflexivity axiom. It has the following shape.

id : A : Set $\rightarrow$ *a : A* $\rightarrow$ *Id A a a.* Given a *small type* we can conclude that an object *a* of that type is equal to itself. It is implemented using HOAS as follows.

Listing 8: id constant

```
id : T (pi_ks set (A : K set =>
        (pi_ss A (a : T A => Id A a a)))).
```

Now that we defined the needed constants we will simulate the rewriting of the expression "`id bool true`". It is an proposition stating that the boolean value *true* is equal to itself. It expands to the following.

```
((T (pi_ks set (A : K set =>
     (pi_ss A (a : T A => Id A a a))))
  bool)
   true)
```

To be able to apply the first argument we need to rewrite the first Π-HOAS, it will be rewritten to the following.

```
((x : K set -> T ((A : K set =>
                  (pi_ss A (a : T A => Id A a a))) x)
  bool)
    true)
```

Now "`K set`" will be rewritten to "`Set`". Since bool is of type "`Set`" this let us to conclude that the value "`bool`" is of the correct type. Substituting "`bool`" for "`x`" results in the following.

```
(T ((A : K set => (pi_ss A (a : T A => Id A a a))) bool)
    true)
```

To be able to be substitute "`bool`" for "`A`" we need first to rewrite "`K set`" again and check the type. Doing the substitution results in the following expression.

```
(T (pi_ss bool (a : T bool => Id bool a a)) true)
```

Rewriting the Π-HOAS will result in the following.

```
(x : T bool -> T ((a : T bool => Id bool a a) x) true)
```

Now we can substitute "`true`" for "`x`" and then "`true`" for "`a`" resulting in.

```
T (Id bool true true)
```

As we can see the variable names "`x`" introduced by the rewrite rules does not interfere with each other since they have different scope. Also the variable names introduced in the HOAS are bound in lambda abstractions so they do not interfere with variable names introduced by the rewrite rules.

7.4 Infinite hierarchies of universes

It turns out that the above embedding is not sufficient for our needs, since it only has two sorts. In [8] and [13] formulations of infinite hierarchies of universes are formalized using simultaneous inductive-recursive definitions. The following will define external universe hierarchies as in [8] and it includes definitions of U_0, T_0 and the introduction rule for Π-formation for U_0 with its associated equality rule.

$$U_0 : Type \tag{12}$$

$$T_0 : U_0 \to Type \tag{13}$$

$$\pi_0 : \Pi u : U_0.(T_0 u \to U_0) \to U_0 \tag{14}$$

$$T_0(\pi_0(u, u')) = \Pi x : T_0 u.T_0(u' x) \tag{15}$$

The next universe U_1 with its introduction rule for Π-formation is defined in a similar way. We also need to relate U_0 with U_1, this is done in the following familiar way. Note that the type defined in (18) "lifts" an object of type U_0 to U_1.

$$u_{01} : U_1 \quad (16) \qquad T_1(u_{01}) = U_0 \quad (17) \qquad t_{01} : U_0 \to U_1 \quad (18)$$

The above definitions are translated to Dedukti using rewriting to implement the equality rules. There is also a identity function defined to get a feel for how the HOAS works. Note how similar the code is to the PTS embedding code. The following is the implementation.

Listing 9: Inductive-recursive universes

```
U0 : Type.
def T0 : U0 -> Type.

pi0 : u :  U0 -> u' : (x : (T0 u) -> U0) -> U0.
[u, u'] T0 (pi0 u u') --> x : T0 u -> T0 (u' x).

U1 : Type.
def T1 : U1 -> Type.

u01 : U1.
[] T1 u01 --> U0.

t01 :  U0 -> U1.
[b] T1 (t01 b) --> T0 b.

def pi1 : u :  U1 -> u' : (x : (T1 u) -> U1) -> U1.
[u, u'] T1 (pi1 u u') --> x : T1 u -> T1 (u' x).

def I : T1 (pi1 u01
        (A : T1 u01 =>
         (pi1 (t01 A)
          (__ : T1 (t01 A) =>  (t01 A))))) :=
         __ => x => x.
```

In [2] there is an embedding defined that is very similar to the code above. The main difference is that the number of universes are not hard coded.

8 Proofs related to the Univalence Axiom in Dedukti

As an example of embedding a logic in Dedukti the following chapter will detail an implementation of some simple proofs related to the UA [17], [10]. The following can be regarded as a naive way to understand the UA.

Assume there are two types A, B in a universe U, then we can define the following function.

$$idtoeqv : (A =_U B) \to (A \simeq B) \tag{19}$$

The function (19) states that if A and B are two types that are identical then there is an equivalence relation between them. In this document we will consider the following notion of equivalence. Assume $f : A \to B$ and $g : B \to A$ then:

$$iso(A, B) := \exists f.\exists g.(\forall a : A.f(ga) =_A a) \wedge (\forall b : B.g(fb) =_B b) \tag{20}$$

If we for types A and B can find functions f and g s.t. (20) holds then we say that the types are are isomorphic.

The function (19) can be regarded as an elimination rule for $A =_U B$ the corresponding introduction rule is the following.

$$ua : (A \simeq B) \to (A =_U B) \tag{21}$$

Here *ua* stands for *univalence axiom*. The function (21) states that if there is an equivalence between two types then they can be regarded as identical. The actual univalence axiom is stated as follows.

Axiom 34 (Univalence)**.** *For any $A, B \in U$ the function* (19) *is an equivalence. In particular we have $(A =_U B) \simeq (A \simeq B)$.*

One of the reasons that the UA is interesting is that it could enable reuse of proofs. To exemplify this consider the natural numbers. The natural numbers can be implemented using unary notation, i.e. with a constant 0 and a successor function but a binary implementation is more commonly used. If we can prove that these two structures are the same then properties proved on one structure can be transported to the other.

Homotopy Type Theory (HOTT) which the UA is part of is a big subject which this document can not possibly cover, for more information on HOTT see [17]. HOTT is based on Martin-Löf's Type Theory which corresponds to Cumulative Type Systems (CTS) [1]. CTS is a generalization of PTS where there is an infinite hierarchy of universes. Using a simple cumulative embedding of Π-types found in [2] as a base, constants for product types, Σ-types and identity types are implemented below.

To get a more or less complete system one should add the type for falsity with no constructor, the type for the true proposition with one constructor and the coproduct type with the two constructors *inl* and *inr*. Since the proofs below does not use these types they are not treated in this text.

8.1 Implementation of universes

We will now implement embedding code for an infinite hierarchy of universes. To be able to do this we need an implementation of natural numbers. Also to make the embedding code a bit more readable we define numeric constants for some numeral values. Note that the symbol ":=" denotes association of constants to terms. In this case the terms are values defined by an inductive type, but the symbol is also used for associating λ-functions with constants.

Listing 10: Natural numbers

```
nat : Type.
Z : nat.
S : nat -> nat.

def 0 := Z.
def 1 := (S 0).
def 2 := (S 1).
```

The following is the implementation of universes, axioms and Π-types they are all implemented using natural numbers as indexes. The code follows a now familiar pattern.

Listing 11: Universes

```
U : nat -> Type.
def T : i : nat -> U i -> Type.

u : i : nat -> U (S i).
[i] T _ (u i) --> U i.

def pi : i :  nat -> a : U i -> b : (T i a -> U i) -> U i.
[i, a, b] T _ (pi i a b) --> x : T i a -> T i (b x).
```

The *pi* constant above is an implementation of the following dependent product rule which ensures that types from different universes are not mixed in an ad hoc fashion.

$$\frac{\Gamma \vdash A : U_i \qquad \Gamma, x : A \vdash B : U_i}{\Gamma \vdash \Pi x : A.B : U_i}$$

Since we want types in a lower universe to be accessible from a higher level we need to be able to "lift" a type to a higher level. This is achieved with the following constant and its associated rewrite rule.

Listing 12: Axiom

```
def L : i : nat -> U i -> U (S i).
[i, a] T _ (L i a) --> T i a.
```

In [1] it was noted that type uniqueness became an issue with a naive implementation of an infinite universe hierarchy. The solution to this subtle issue is the following rewrite rule. For more information see [1].

The rule is also a nice example of a higher order rewrite rule. It is higher order since it includes a λ-expression. Note that the *free* variables i, a and b are defined in the local context. The identifiers *pi* and L have already been defined so they are constants in the global context. Note also that the variable x is bound by the λ-abstraction. During rewriting the variables i, a and b will be instantiated with values using higher order pattern matching. The left hand side of the rewrite rule is a higher order pattern since its free variables are applied to a possible empty list of distinct bound variables.

Listing 13: Rewrite rule

```
[i, a, b] pi _ (L i a) (x => L {i} (b x)) -->
           L i (pi i a (x => b x)).
```

8.2 Product type

We will now implement the product type in similar manner as was done for PTS case. The main difference is that we need an implementation for each universe in our hierarchy, since we want the universes to be closed under set forming operations. In our specific case it turns out that we only need the product type for U_0. It has the following shape.

prod0 : $U_0 \to U_0 \to U_0$. and is implemented using HOAS as follows.

Listing 14: prod0 constant

```
prod0 : T 1 (pi 1 (u 0) (__ =>
             pi 1 (u 0) (__ => (u 0)))).
```

The pair type is implemented using HOAS as below and has the following shape.

pair0 : $A : U_0 \to B : U_0 \to a : A \to b : B \to$ *prod0 A B*.

Listing 15: pair0 constant

```
pair0 :
 T 1 (pi 1 (u 0) (A : T 1 (u 0) =>
       pi 1 (u 0) (B : T 1 (u 0) =>
        pi 1 (L 0 A) (a : T 1 (L 0 A) =>
         pi 1 (L 0 B) (b : T 1 (L 0 B) =>
          L 0 (prod0 A B)))))).
```

After a few moments of looking at these types and the uniformity of the numbers, it might strike us that by using a natural number as an index these constants could have one implementation for all universes. Due to time constraints this is not investigated further.

8.3 Sigma type

The sigma type is a generalization of the product type where the second component depends on the first. From a logical point of view it expresses existential quantification in a constructive sense. Also in this case we only need an implementation for the constant in U_0. The *sig0* constant has the following shape.

sig0 : $A : U_0 \to (A \to U_0) \to U_0$. This constant states that there exist an element in the type specified by A that makes a property true. It is implemented using HOAS as follows.

Listing 16: sig0 constant

```
sig0 :
 T 1 (pi 1 (u 0) (A : T 1 (u 0) =>
      (pi 1
          (pi 1
              (L 0 A)
              (__ : T 1 (L 0 A) => (u 0)))
          (__ => (u 0))))).
```

The introduction rule for the sigma type is commonly called exist. It is used to actually prove an existential statement. For U_0 it has the following shape.

exist0 : $A : U_0 \to B : (A \to U_0) \to a : A \to (B\ a) \to$ *sig0* $A\ B$. The second element B is a dependent type. It depends on the type A. To prove that *sig0* $A\ B$ holds we need to supply a witness a and a proof $(B\ a)$ asserting that B holds for the supplied witness.

It is interesting to note that the proof $(B\ a)$ needs only to be convertible to statement to prove when applied to the witness. The convertibility relation takes into account the rewrite rules defined in the global context and β-reduction. The following is its implementation using HOAS.

Listing 17: exist0 constant

```
exist0 : T 1
 (pi 1 (u 0) (A : T 1 (u 0) =>
  (pi 1
      (pi 1 (L 0 A) (__ : T 1 (L 0 A) => (u 0)))
      (B : T 1 (pi 1 (L 0 A) (__ : T 1 (L 0 A) => (u 0))) =>
   (pi 1 (L 0 A) (a : T 1 (L 0 A) =>
     pi 1 (L 0 (B a)) (__ => L 0 (sig0 A B))))))).
```

8.4 Identity type

The constant *Id0* is similar to the constant *Id* defined for the PTS embedding it has the following shape

Id0 : $A : U_0 \to A \to A \to U_0$. It is implemented using HOAS as follows.

Listing 18: Id0 constant

```
Id0 :
 T 1 (pi 1 (u 0) (A : T 1 (u 0) =>
      (pi 1 (L 0 A) (__ : T 1 (L 0 A) =>
       (pi 1 (L 0 A) (__ : T 1 (L 0 A) => (u 0)))))))).
```

The reflexivity axiom it has the following shape.

id0 : A : $U_0 \rightarrow$ *a : A* $\rightarrow$ *Id0 A a a*. The following is how it is implemented using HOAS.

Listing 19: id0 constant

```
id0 :
 T 1 (pi 1 (u 0) (A : T 1 (u 0) =>
      (pi 1 (L 0 A) (x : T 1 (L 0 A) => L 0 (Id0 A x x))))).
```

In the proofs below we need to be able to state and prove that two *small types* are equal. This leads us to introduce constants *Id1* and *id1* expressing equality in the U_1 universe. Their shape and implementation are the same as for *Id0* and *id0* modulo some number renaming. The following are the constants HOAS implementation.

Listing 20: Id1 and id1 constants

```
Id1 :
 T 2 (pi 2 (u 1) (A : T 2 (u 1) =>
      (pi 2 (L 1 A) (__ : T 2 (L 1 A) =>
       (pi 2 (L 1 A) (__ : T 2 (L 1 A) => (u 1)))))))).

id1 :
 T 2 (pi 2 (u 1) (A : T 2 (u 1) =>
      (pi 2 (L 1 A) (a : T 2 (L 1 A) => L 1 (Id1 A a a))))).
```

8.5 Boolean type

The following is the definition of the inductive type bool representing boolean values.

Listing 21: Boolean type

```
bool : U 0.
true : T 0 bool.
false : T 0 bool.
```

To prove properties of inductive types, but also to do computations, a construct called an eliminator is commonly used. In Dedukti computations can also be defined using rewriting rules. The eliminator for bool has the following shape.

bool_ind : P : (bool → U_0) → P true → P false → x : bool → P x.
And can be explained as follows: Given a property P over the type *bool*, to conclude that it holds for all *bool* values it is enough to show that it holds for the values *true* and *false*. The following is its implementation using HOAS.

Listing 22: Boolean eliminator

```
def bool_ind :
 T 1
   (pi 1 (pi 1 (L 0 bool) (__: T 1 (L 0 bool) => (u 0)))
   (P: T 1 (pi 1 (L 0 bool) (__: T 1 (L 0 bool) => (u 0))) =>
    (pi 1
        (L 0 (P true))
        (__ => (pi 1
                   (L 0 (P false))
                   (__ => L 0 (pi 0 bool P)))))))).
[P, ct, cf] bool_ind P ct cf true --> ct.
[P, ct, cf] bool_ind P ct cf false --> cf.
```

8.6 Iso predicate

The terms *ua* and *idtoeqv* can be formulated using an isomorphism relation between two *small types* A and B. It is implemented below as a parametric function returning as a result a type.

The following is an explanation of the *iso* function as implemented below. The function body starts with "`(A => B => ...)`" which tells us it takes

the two types A and B as arguments. We can also see that it returns a type starting with a *sig0* constant. The first argument to the *sig0* constant is the HOAS definition of the type $A \to B$. The second argument is a λ-function of the shape $(f : A \to B \Rightarrow \dots)$. It states that there exist a function f s.t. ... The next *sig* constant is similar but now the shape of the second argument is $(g : B \to A \Rightarrow (prod0 \dots)$.

The *prod0* constant has two arguments. The first is a *pi* constant with three arguments. The first argument is the universe level, the second is the type variable A and the third is the HOAS definition of a dependent type expressing universal quantification over the type A. The body of the universal quantification is the equality $g(fa) = a$. The second argument to the *prod0* constant is similar but states the equality in the other direction. The constant *iso* has the following shape.

$iso : A : U_0 \to B : U_0 \to U_0$ and is implemented using HOAS as follows.

Listing 23: Iso predicate

```
def iso :
 T 1 (pi 1
         (u 0)
         (__ : T 1 (u 0) =>
          (pi 1 (u 0) (__ : T 1 (u 0) => u 0)))) :=
 (A => B =>
  sig0 (pi 0 A (__ : T 0 A => B))
       (f : T 0 (pi 0 A (__ : T 0 A => B)) =>
   (sig0 (pi 0 B (__ : T 0 B => A))
         (g : T 0 (pi 0 B (__ : T 0 B => A)) =>
          (prod0
           (pi 0 A (a : T 0 A => Id0 A (g (f a)) a))
           (pi 0 B (b : T 0 B => Id0 B (f (g b)) b)))))).
```

8.7 Type two

As an example of the iso predicate we are going to prove that *bool* is isomorphic to the type *two* defined below. Actually it is not hard to see that any types with two constructors that are constants (i.e. not functional types) are isomorphic. Below are the definition of the type *two* with its associated eliminator.

Listing 24: Type two

```
two : U 0.
11 : T 0 two.
00 : T 0 two.

def two_ind :
 T 1
   (pi 1 (pi 1 (L 0 two) (__: T 1 (L 0 two) => (u 0)))
   (P: T 1 (pi 1 (L 0 two) (__: T 1 (L 0 two) => (u 0))) =>
    (pi 1
        (L 0 (P 11))
        (__ => (pi 1
                   (L 0 (P 00))
                   (__ => L 0 (pi 0 two P)))))))).
[P, c1, c0] two_ind P c1 c0 11 --> c1.
[P, c1, c0] two_ind P c1 c0 00 --> c0.
```

We also need two functions mapping the types *bool* and *two* to each other. They are easily defined using rewriting.

Listing 25: Bool two mappings

```
def b_to_t : b : T bool -> T two.
[] b_to_t true --> 11.
[] b_to_t false --> 00.

def t_to_b : s : T two -> T bool.
[] t_to_b 11 --> true.
[] t_to_b 00 --> false.
```

8.8 Bool is isomorphic to two

The lemma that the types *two* and *bool* are isomorphic can easily be stated. The proof text is longer but this is mainly because there is a lot of repetition of types. Also the use of Π-types implemented as HOAS makes the proof hard to follow.

Anyway the following is an explanation of the Dedukti code. Remember that proving statements involving *sig0* types was done using the *exist0* constant. Since there are two nested *sig0* constants in the statement to prove

we need to use two nested instances of *exist0*. The following is the first *exist0* constant's four arguments:

1. The type of the function f which is the argument that should be instantiated with a witness.
2. The actual statement to be proved.
3. The witness function that f will be instantiated with, its name is *t_to_b*.
4. The proof of the statement defined in 2. The proof consists of another *exist0*.

The second *exist0* is similar, but note that the second argument is a bit different. In the statement that needs to be proved f has been instantiated with *t_to_b*. Also the fourth argument, the proof, is a *pair0* constant.

The *pair0* constant is used to prove the *prod0* statement. Its two first arguments are the universal quantification statements showing that the types are equal. Note that at this point f and g has been instantiated with the witness functions. The two last arguments are the proofs by cases using eliminators.

It is interesting to note that the proofs will require rewriting using the rewrite rules of the witness functions. To illustrate this consider the following example using the type *two*. The property P to prove is *(t : T two ⇒ Id0 two (b_to_t(t_to_b t)) t)* and as the proof is by cases then one of the cases is *(P 1)*. Applying P to 1 you get *Id0 two (b_to_t(t_to_b 1)) 1*, and the proof of this is *id0 two 1* which is of type *Id0 two 1 1*.

Since the proof is done by type checking, the type checking algorithm needs to conclude that *Id0 two (b_to_t(t_to_b 1)) 1* and the type *Id0 two 1 1* are convertible. This can be done using rewrite rules defined by *b_to_t* and *t_to_b*.

Listing 26: Bool two isomorphism

```
def iso_two_bool : T 0 (iso two bool) :=
 exist0 (pi 0 two (__ : T 0 two => bool))
  (f : T 0 (pi 0 two (__ : T 0 two => bool)) =>
   (sig0 (pi 0 bool (__ : T 0 bool => two))
    (g : T 0 (pi 0 bool (__ : T 0 bool => two)) =>
     (prod0
      (pi 0 two (t : T 0 two => Id0 two (g (f t)) t))
      (pi 0 bool (b : T 0 bool => Id0 bool (f (g b)) b))))))
     t_to_b
     (exist0
      (pi 0 bool (__ : T 0 bool => two))
       (g : T 0 (pi 0 bool (__ : T 0 bool => two)) =>
        (prod0
         (pi 0 two (t : T 0 two =>
                    Id0 two (g (t_to_b t)) t))
         (pi 0 bool (b : T 0 bool =>
                     Id0 bool (t_to_b (g b)) b))))
       b_to_t
       (pair0
        (pi 0 two (t : T 0 two =>
                   Id0 two (b_to_t (t_to_b t)) t))
        (pi 0 bool (b : T 0 bool =>
                    Id0 bool (t_to_b (b_to_t b)) b))
        (two_ind (t : T 0 two =>
                  Id0 two (b_to_t(t_to_b t)) t)
                 (id0 two 11)
                 (id0 two 00))
        (bool_ind (b : T 0 bool =>
                   Id0 bool (t_to_b(b_to_t b)) b)
                  (id0 bool true)
                  (id0 bool false)))).
```

8.9 Proof of idtoeqv

It turns out that our proof of *idtoeqv* needs an implementation of the identity function for U_0. It has the following shape. $I0 : A : U_0 \to A \to A$. The

following is the HOAS implementation of it.

Listing 27: Identity function

```
def I0 :
 T 1 (pi 1
         (u 0)
         (A : T 1 (u 0) =>
       (pi 1
           (L 0 A)
           (__ : T 1 (L 0 A) =>  (L 0 A))))) :=
       __ => x => x.
```

The constant *idtoeqv* has the following shape.

idtoeqv : $A : U_0 \to B : U_0 \to$ *Id1* U_0 *A B* $\to$ *iso A B*. Below is the HOAS implementation of the type and a proof using pattern matching. It can be explained as follows.

When the types A, B are proved to be equal, then the witnesses for the mapping functions f and g is the identity function for the type A, and proving a statement of the form *Id0 A (I0 A (I0 A a)) a* can be done using the reflexivity axiom *id0 A a*.

Listing 28: Lemma idtoeqv

```
def idtoeqv :
 T 1 (pi 1 (u 0) (A : T 1 (u 0) =>
      (pi 1 (u 0) (B : T 1 (u 0) =>
       pi 1 (Id1 (u 0) A B) (__ => L 0 (iso A B))))))).
 [A] idtoeqv A {A} (id1 (u 0) {A}) -->
  exist0 (pi 0 A (__ : T 0 A => A))
         (f : T 0 (pi 0 A (__ : T 0 A => A)) =>
         (sig0
          (pi 0 A (__ : T 0 A => A))
          (g : T 0 (pi 0 A (__ : T 0 A => A)) =>
           (prod0
            (pi 0 A (a : T 0 A => Id0 A (g (f a)) a))
            (pi 0 A (a : T 0 A => Id0 A (f (g a)) a))))))
          (I0 A)
          (exist0
           (pi 0 A (__ : T 0 A => A))
           (g : T 0 (pi 0 A (__ : T 0 A => A)) =>
            (prod0
             (pi 0 A (a : T 0 A => Id0 A (g (I0 A a)) a))
             (pi 0 A (a : T 0 A => Id0 A (I0 A (g a)) a))))
           (I0 A)
           (pair0
            (pi 0 A (a : T 0 A => Id0 A (I0 A (I0 A a)) a))
            (pi 0 A (a : T 0 A => Id0 A (I0 A (I0 A a)) a))
            (a : T 0 A => id0 A a)
            (a : T 0 A => id0 A a))).
```

To illustrate *idtoeqv* the following states and proves that bool is isomorphic to itself.

Listing 29: bool isomorphic to bool

```
def bool_iso_bool : T 0 (iso bool bool) :=
 idtoeqv bool bool (id1 (u 0) bool).
```

8.10 ua constant

The constant *ua* has the following form in Dedukti syntax.

$ua : A : U_0 \to B : U_0 \to iso\ A\ B \to Id1\ U_0\ A\ B$, and is implemeted using HOAS as follows.

Listing 30: ua constant

```
ua :
 T 1 (pi 1 (u 0) (A : T 1 (u 0) =>
      (pi 1 (u 0) (B : T 1 (u 0) =>
       pi 1 (L 0 (iso A B)) (__ => (Id1 (u 0) A B))))))).
```

Using the *ua* type it is easy show that the types *bool* and *two* are equal.

Listing 31: two is equal to bool

```
def equal_two_bool : T 1 (Id1 (u 0) two bool) :=
 ua two bool iso_two_bool.
```

Having introduced the *ua* constant we should now consider to open chapter 2.1 in [17] and start to implement proofs in a more systematic manner. The first thing to implement would be the Induction principle for identity types. It can be used to prove that for every type A the type $(x =_A y) \to (y =_A x)$ is inhabited.

Before doing more formalization we might want to try to prove that implemented logical constants are adequate, i.e. correctly reflects their logical meaning. But due to time constraints we will stop the work on the formalization here and we will not prove any properties of the implemented logical constants.

9 Conclusion

Dedukti has mainly been targeted as a proof checker for theorems developed in other systems. This explains the lack of features like implicit arguments and user defined syntax. Also the lack of tactics forces the user to derive a λ-term for a given type by hand. For complex types this can be quite a challenge. The usage of HOAS when doing embeddings of logics does not improve usability. In my opinion even with simple proofs readability becomes an issue.

Having said that using Dedukti to prototype a specific variant of a logic must be much faster than doing the implementation in a programming language like for example Haskell. Also Dedukti's usage of user defined rewrite

rules during type checking seems to be a nice feature which is not commonly implemented in proof assistants.

Another feature that is commonly found in proof assistants are some form of *Universe Polymorphism*. With *Universe Polymorphism* universe levels can be deduced, i.e. there is no need to explicitly specify them. Since Dedukti is targeted as proof checker for many of the major proof assistants this feature might be available in Dedukti in the future. Maybe an embedding implementing *Universe Polymorphism* could make the Dedukti code more readable and easier to develop.

References

[1] Ali Assaf. "A framework for defining computational higher-order logics". Phd. Thesis. École polytechnique, Sept. 2015. URL: https://pastel.archives-ouvertes.fr/tel-01235303.

[2] Ali Assaf et al. *Dedukti: a Logical Framework based on the lambda-pi-Calculus Modulo Theory*. Retrieved Jan 2018. URL: http://www.lsv.fr/~dowek/Publi/expressing.pdf.

[3] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, Cambridge, 1998, pp. xii+301. ISBN: 0-521-45520-0; 0-521-77920-0.

[4] H. P. Barendregt. "Lambda calculi with types". In: *Handbook of logic in computer science, Vol. 2*. Oxford Univ. Press, New York, 1992, pp. 117–309.

[5] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013. ISBN: 978-0-521-76614-2. URL: http://www.cambridge.org/de/academic/subjects/mathematics/logic-categories-and-sets/lambda-calculus-types.

[6] Thierry Coquand. "An Analysis of Girard's Paradox". In: *In Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1986, pp. 227–236.

[7] Denis Cousineau and Gilles Dowek. “Embedding pure type systems in the lambda-pi-calculus modulo”. In: *Typed lambda calculi and applications*. Vol. 4583. Lecture Notes in Comput. Sci. Springer, Berlin, 2007, pp. 102–117.

[8] Peter Dybjer. “A general formulation of simultaneous inductive-recursive definitions in type theory”. *J. Symbolic Logic* 65.2 (2000), pp. 525–549. ISSN: 0022-4812.

[9] Robert Harper, Furio Honsell, and Gordon Plotkin. “A framework for defining logics”. *J. Assoc. Comput. Mach.* 40.1 (1993), pp. 143–184. ISSN: 0004-5411.

[10] M. Hötzel Escardó. “A self-contained, brief and complete formulation of Voevodsky’s Univalence Axiom”. *ArXiv e-prints* (Mar. 2018). arXiv: 1803.02294 [math.LO].

[11] J. W. Klop. “Term rewriting systems”. In: *Handbook of logic in computer science, Vol. 2*. Vol. 2. Handb. Log. Comput. Sci. Oxford Univ. Press, New York, 1992, pp. 1–116.

[12] Tobias Nipkow and Christian Prehofer. “Higher-order rewriting and equational reasoning”. In: *Automated deduction—a basis for applications, Vol. I*. Vol. 8. Appl. Log. Ser. Kluwer Acad. Publ., Dordrecht, 1998, pp. 399–430.

[13] Erik Palmgren. “On universes in type theory”. In: *Twenty-five years of constructive type theory (Venice, 1995)*. Vol. 36. Oxford Logic Guides. Oxford Univ. Press, New York, 1998, pp. 191–204.

[14] Frank Pfenning. “Logical frameworks—a brief introduction”. In: *Proof and system-reliability (Marktoberdorf, 2001)*. Vol. 62. NATO Sci. Ser. II Math. Phys. Chem. Kluwer Acad. Publ., Dordrecht, 2002, pp. 137–166.

[15] R. Saillard. “Rewriting Modulo beta in the lambda Pi-Calculus Modulo”. *ArXiv e-prints* (July 2015). arXiv: 1507.08055 [cs.LO].

[16] Ronan Saillard. “Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice”. Phd. Thesis. Ecole Nationale Supérieure des Mines de Paris, Sept. 2015. URL: https://pastel.archives-ouvertes.fr/tel-01299180.

[17] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: `https://homotopytypetheory.org/book`, 2013.