B. Jacobs and T. Melham, 'Translating Dependent Type Theory into Higher Order Logic', in *Typed Lambda Calculi and Applications: Proceedings of the International Conference, Utrecht, March 16–18 1993*, edited by M. Bezem and J. F. Groote, Lecture Notes in Computer Science, vol. 664 (Springer-Verlag, 1993), pp. 209–229.

# Translating Dependent Type Theory into Higher Order Logic[1]

Bart Jacobs[2] and Tom Melham[3]

**Abstract.** *This paper describes a translation of the complex calculus of dependent type theory into the relatively simpler higher order logic originally introduced by Church. In particular, it shows how type dependency as found in Martin-Löf's Intuitionistic Type Theory can be simulated in the formulation of higher order logic mechanized by the HOL theorem-proving system. The outcome is a theorem prover for dependent type theory, built on top of HOL, that allows natural and flexible use of set-theoretic notions. A bit more technically, the language of the resulting theorem-prover is the internal language of a (boolean) topos (as formulated by Phoa).*

## 1 Introduction

In dependent type theory (DTT, for short) terms may occur in types. A typical example is the dependent type $Nat[n]$ of natural numbers less than $n$. This gives DTT greater expressive power than simple type theory, where types cannot depend on the values of terms but only on other types. This advantage of DTT will be illustrated by examples in section 3.2 below.

Church's higher order logic consists of simply typed $\lambda$-calculus with the type constructors $\rightarrow$ for function space and $\times$ for cartesian product and with a special type *bool* of 'truth values' or 'booleans'. It comes equipped with constants $\mathsf{T} : bool$ and $\mathsf{F} : bool$ for true and false and the usual connectives $\supset$, $\wedge$, $\vee$, and $\neg$. Formulas are just boolean terms $\varphi : bool$, and the logic involved is classical. The logic can conveniently be formulated with sequents of the form

$$\Gamma \mid \varphi_1, \ldots, \varphi_m \vdash \psi \qquad (1)$$

which should be read as follows: 'in a context $\Gamma$ containing type declarations of all the term variables free in $\varphi_1, \ldots, \varphi_m, \psi$, the formula $\psi$ is true under the assumption that the formulas $\varphi_1, \ldots, \varphi_m$ are true'. The context $\Gamma$ in such a sequent has the form $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$. The sign '$|$' is used as a separator.

Gordon's HOL theorem prover [4, 5] implements a specific formulation of Church's higher order logic, which will be described in some detail in section 2 below. In HOL, as in Church's original logic, the context $\Gamma$ in a sequent is omitted; types are instead attached to the variables themselves. But in the background, the context can be regarded as still being there, because the information it

[1]Research carried out during 1991–92 at the University of Cambridge under SERC grant GR/F 36675.

[2]Mathematical Institute RUU, P.O.Box 80.010, 3508 TA Utrecht, NL. `bjacobs@math.ruu.nl`

[3]Computer Laboratory, University of Cambridge, New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK. `tfm@cl.cam.ac.uk`

contains can be reconstructed from the formulas of a sequent. We shall therefore feel free to refer to a formulation of HOL with explicit contexts when this has technical or notational advantages.

For example, higher order logic includes universal and existential quantification over the values of each type. Using a formulation based on sequents with an explicit context, the introduction rules for these take the form

$$\frac{\Gamma, x : \sigma \mid \varphi_1, \ldots, \varphi_m \vdash \psi}{\Gamma \mid \varphi_1, \ldots, \varphi_m \vdash \forall x{:}\sigma.\, \psi} \ (x \text{ not in } \varphi_1, \ldots, \varphi_m)$$

$$\frac{\Gamma \vdash a : \sigma \qquad \Gamma \mid \varphi_1, \ldots, \varphi_m \vdash \psi[a/x]}{\Gamma \mid \varphi_1, \ldots, \varphi_m \vdash \exists x{:}\sigma.\, \psi}$$

Note that the side condition '$x$ not in $\varphi_1, \ldots, \varphi_m$' appears natural in this formulation, because in forming $\forall x{:}\sigma.\, \psi$ one has to lift $x$ over the $\varphi_1, \ldots, \varphi_m$.

Another advantage of using explicit contexts is that this highlights the fact that higher order logic is a system of (classical) logic in an environment with *simple types*; a context is just an assignment of simple types to the free variables of a proposition and its assumptions. In this paper, we describe a translation into HOL of a system of classical logic in an environment with *dependent types*. This also has sequents of the form illustrated by (1) above, but the context $\Gamma$ may now contain dependent types. This gives the logic increased expressive power. Our favourite example is the result which says that every injective endofunction on a finite set is surjective. Using such a 'dependent logic', it can be expressed simply as

$$n :: Nat, f :: Nat[n] \to Nat[n] \mid \mathsf{injective}(f) \vdash \mathsf{surjective}(f)$$

where we use :: for inhabitation in DTT.[4] Translating such assertions into the mechanized HOL logic gives us a theorem prover with great expressive power, having a logic with dependent types but based on the underlying HOL theory of simple types.

Our main aim in this work is to obtain this expressive advantage, rather than to use 'propositions-as-types'. Moreover, we are not so much interested in *programming* in DTT, as discussed in [13], as in using the logic of type theory for specification and reasoning. We wish the logic we use to be as expressive and natural as possible. Our motivation for using DTT comes, for example, from the work of Hanna, Daeche and Longley [7], who have made a strong case for the utility of dependent types for hardware specification and verification. Leeser [9] has also shown how the dependent types of the Nuprl theorem-prover can be used to structure the information content of the theorems involved in mechanized reasoning about hardware. The main advantage over working in simple type theory is that typing judgements DTT have increased information content; an inhabitation judgement, for example, can bear the information that a term meets a partial specification.

The translation of DTT into HOL uses the more or less familiar idea of sending a dependent type to a predicate. It is also used in [2] and on a more abstract level in section 4.3.5 of [8]; the translation is extracted from the interpretation of DTT in a topos. A detailed description of this translation forms the basis for an actual implementation, which is briefly described in section 6. Our practical experience with the translated version of DTT is still at the level of playing with examples. In particular, detailed investigation of how best to reason in a mixture of both HOL and DTT has been left for future work.

[4] Strictly speaking, the propositions $\mathsf{injective}(f)$ and $\mathsf{surjective}(f)$ will also make reference to the domain and codomian of $f$. We have left this implicit here.

## 2 Higher order logic and HOL

The higher order logic mechanized by the HOL theorem prover is based on Church's formulation of simple type theory [1]. Gordon's machine-oriented formulation, which we shall call the HOL logic, or just HOL, extends Church's theory in two significant ways: the syntax of types includes the polymorphic type discipline developed by Milner for the LCF logic PP$\lambda$ [6], and the primitive basis of the logic includes formal rules of definition for extending the logic with new constants and new types. The following section gives a quick overview, mainly of the notation we'll be using; see [5] for a complete description, including a set-theoretic semantics.

**Types.** The syntax of types in HOL is given by

$$\sigma \quad ::= \quad c \quad | \quad v \quad | \quad (\sigma_1, \ldots, \sigma_n)op$$

where $\sigma$, $\sigma_1$, ..., $\sigma_n$ range over types, $c$ ranges over type *constants*, $v$ ranges over type *variables*, and *op* ranges over $n$-ary type *operators* (for $n \geq 1$).

In fact, the basic type system is very small; it contains only the primitive types *bool* (the two-element set of truth-values) and *ind* (an infinite set of 'individuals') and one type operator, namely $\rightarrow$ for function space. All other types are formally defined in terms of these primitive ones using one of the rules of definition mentioned above.

Among the types definable in HOL is a singleton type *one* whose sole element will be written $*$. A cartesian product type $\sigma \times \tau$ is also definable, with (surjective) pairing written as $\langle -, - \rangle$ and with projections $\pi, \pi'$. In this paper, we also use a type *num* constant of natural numbers and a polymorphic type constructor $(\alpha)list$, both of which are also formally definable in the HOL logic. Details of the definitions of all these types can be found in [3] or [12].

**Terms.** The syntax of (untyped) terms in the HOL logic is given by

$$\mathrm{M} \quad ::= \quad c \quad | \quad v \quad | \quad (\mathrm{M}\ \mathrm{N}) \quad | \quad \lambda v.\, \mathrm{M}$$

where $c$ ranges over constants, $v$ ranges over variables, and M and N range over terms. Sans serif identifiers (e.g. $\mathsf{a}$, $\mathsf{b}$, $\mathsf{c}$, $\mathsf{Const}$) and non-alphabetical symbols (e.g. $\supset$, $=$, $\forall$) are generally used for constants, and italic identifiers (e.g. $v$, $x$, $x_1$, $f$) are used for variables.

Every well-formed term in higher order logic must be *well-typed*. Writing $\mathrm{M} : \sigma$ indicates explicitly that the term M is well-typed with type $\sigma$. Typing of terms takes place within the context of an assignment of types to constants. Each constant $c$ has fixed *generic* type $Gen(c)$ associated with it. A constant $c$ with a polymorphic generic type $\sigma$ is well typed with any substitution instance of $\sigma$ obtainable by substituting types for type variables. Given an assignment of generic types to constants, the well-typed terms of HOL are defined inductively by the following typing rules.

$$\textbf{var}\ \frac{}{v : \sigma} \qquad\qquad \textbf{con}\ \frac{}{c : \sigma}\ (\sigma \text{ a substitution instance of } Gen(c))$$

$$\textbf{abs}\ \frac{v : \sigma \qquad \mathrm{M} : \tau}{(\lambda v.\mathrm{M}) : \sigma \rightarrow \tau} \qquad\qquad \textbf{app}\ \frac{\mathrm{M} : \sigma \rightarrow \tau \qquad \mathrm{N} : \sigma}{(\mathrm{M}\,\mathrm{N}) : \tau}$$

It follows from these rules that the type of a term is uniquely determined by the types of constants and variables it contains.

**Definitions and Axioms.** The HOL logic is based on the following fundamental definitions for quantifiers and connectives, which we simply list here without further comment.

**D1** $\vdash \mathsf{T} = ((\lambda x{:}bool.\, x) = (\lambda x{:}bool.\, x))$
**D2** $\vdash \forall = \lambda P{:}\alpha \to bool.\, P = (\lambda x.\, \mathsf{T})$
**D3** $\vdash \exists = \lambda P{:}\alpha \to bool.\, P(\varepsilon\ P)$
**D4** $\vdash \mathsf{F} = \forall b{:}bool.\, b$
**D5** $\vdash \neg = \lambda b.\, b \supset \mathsf{F}$
**D6** $\vdash\ \wedge\ = \lambda b_1\, b_2.\forall b.\, (b_1\ \supset\ (b_2\ \supset\ b))\ \supset\ b$
**D7** $\vdash\ \vee\ = \lambda b_1\, b_2.\forall b.\, (b_1\ \supset\ b)\ \supset\ ((b_2\ \supset\ b)\ \supset\ b)$

Note that Hilbert's $\varepsilon$-operator, a primitive constant of HOL, is used in **D3** to define existential quantification. Informally, the semantics of this operator is as follows. If $\mathrm{P}{:}\sigma \to bool$ is a predicate on values of type $\sigma$, then the application '$\varepsilon\,\mathrm{P}$' denotes a value of type $\sigma$ for which P is true. If there is no such value, then the term '$\varepsilon$ P' denotes a fixed but unknown value of type $\sigma$. A consequence is that all types in HOL must be non-empty, since for any predicate $\mathrm{P}{:}\sigma \to bool$, the term $\varepsilon\,\mathrm{P}$ always denotes a value of type $\sigma$. For further discussion of $\varepsilon$, see [10].

Some typical examples of formulas written using this defined logical notation are the five axioms of the HOL logic, which are shown below.

**A1** $\vdash \forall b.\, (b = \mathsf{T}) \vee (b = \mathsf{F})$
**A2** $\vdash \forall b_1\, b_2.\, (b_1\ \supset\ b_2)\ \supset\ (b_2\ \supset\ b_1)\ \supset\ (b_1 = b_2)$
**A3** $\vdash \forall f{:}\alpha \to \beta.\, (\lambda x.\, f\ x) = f$
**A4** $\vdash \forall P{:}\alpha \to bool. \forall x.\, P\ x\ \supset\ P(\varepsilon\ P)$
**A5** $\vdash \exists f{:}ind \to ind.\, (\forall x\ y.\, (fx = fy)\ \supset\ (x = y))\ \wedge\ \neg \forall x. \exists y.\, x = f\ y$

Together with the primitive inference rules of HOL, the axioms **A1**, **A2** and **A3** define (classical) higher order propositional and functional calculus. The additional axioms **A4** and **A5** are the axiom of choice and axiom of infinity.

**Inference Rules.** The style of proof used in Gordon's formulation of higher order logic is a form of natural deduction in which sequents are used to keep track of assumptions. A sequent is written $\varphi_1, \ldots, \varphi_n \vdash \psi$, where $\varphi_1, \ldots, \varphi_n$ is a sequence of boolean terms called the assumptions and $\psi$ is a boolean term called the conclusion.[5] This sequent notation can be read as the metalinguistic assertion that there exists a natural deduction proof of the conclusion $\psi$ from the assumptions in $\varphi_1, \ldots, \varphi_n$. When there are no assumptions, the notation $\vdash \psi$ is used. In this case, $\psi$ is a formal theorem of the logic.

The inference rules of HOL are without surprises. For example, one has the following rules introduction and elimination rules for implication.

$$\frac{\varphi_1, \ldots, \varphi_n, \psi \vdash \chi}{\varphi_1, \ldots, \varphi_n \vdash \psi\ \supset\ \chi} \qquad\qquad \frac{\varphi_1, \ldots, \varphi_n \vdash \psi\ \supset\ \chi \qquad \varphi_1, \ldots, \varphi_n \vdash \psi}{\varphi_1, \ldots, \varphi_n \vdash \chi}$$

Note that these rules do not explicitly mention contexts, as discussed above on in section 1. For a complete account of the HOL inference rules, both primitive and derived, see [5].

[5] In fact, not sequences but *sets* of assumptions are used in the formulations of HOL presented in [3, 5].

# 3 Dependent type theory

In this section we do not intend to give a complete and systematic description of dependent type theory (DTT)—for this, refer to [11] or [16]. Rather it is our aim to get across just the main ideas and features of such a type theory, especially the dependent product $\Pi$ and the dependent sum $\Sigma$. Our secondary aim is to point out some subtleties in the presentation of the calculus.

The language we use is rather informal. We shall use :: for inhabitation in DTT in order to distinguish it from the typing symbol : used in HOL.

## 3.1 Informal description

In HOL a term variable $x$ may occur in a term (e.g. in $x + 3 : num$) but not in a type. This is different in dependent type theory, where for example one can have a term $n$ that denotes a natural number and occurs in types. Some typical examples are

| | |
|---|---|
| $Nat[n]$ | the type of natural numbers less than $n$, i.e. $\{0, 1, \ldots, n-1\}$ |
| $List[n]$ | the type of lists of length $n$ |
| $Matrix[n, m]$ | the type of $n \times m$ matrices |

One can then have term judgements that involve these types, for example:

$$\begin{array}{rcl} n :: Nat, m :: Nat[n] & \vdash & \mathsf{suc}\, m :: Nat[\mathsf{suc}\, n] \\ n :: Nat, m :: Nat, z :: Matrix[n, m] & \vdash & \mathsf{firstrow}\, z :: List[n] \end{array}$$

where $\mathsf{suc}{:}num{\to}num$ is the successor function on natural numbers in HOL, and $\mathsf{firstrow}$ is a HOL function with the obvious meaning.

A first thing to note about DTT is that a dependent type makes sense only in a context which contains declarations of its free variables. For example one can have

$$n :: Nat \vdash Nat[n] :: DType \qquad\qquad n :: Nat[5] \vdash Nat[n] :: DType$$

where it is clear that the sets denoted by the type $Nat[n]$ in the judgement on the left can be quite different from the ones denoted by $Nat[n]$ in the judgement on the right.[6] Thus the most basic judgements of DTT have the form

$$\Gamma \vdash A :: DType$$

where $\Gamma$ is a context of the form $x_1 :: A_1, \ldots, x_n :: A_n$. This says that $A$ is a (dependent) type in context $\Gamma$. Alternatively one can write

$$x_1 :: A_1,\ x_2 :: A_2[x_1], \ldots,\ x_n :: A_n[x_1, \ldots, x_{n-1}] \vdash A[x_1, \ldots, x_n] :: DType$$

where the possible occurrences of variables are made explicit. The second form of judgement in dependent type theory is

$$\Gamma \vdash A = B :: DType$$

[6] Notice, by the way, that on the right-hand side $n :: Nat[5]$ is implicitly coerced to $n :: Nat$.

This says that $A$ and $B$ are equal dependent types, where it is therefore understood that $\Gamma \vdash A :: \mathit{DType}$ and $\Gamma \vdash B :: \mathit{DType}$. Notice that such a type equality judgement can have a computational content, since terms may occur in types. As a trivial example, consider

$$\vdash \mathit{Nat}[25] = \mathit{Nat}[\mathsf{square}\, 5] :: \mathit{DType}$$

Thirdly, for a dependent type $\Gamma \vdash A :: \mathit{DType}$ we can say that $a$ is a term of type $A$ by writing

$$\Gamma \vdash a :: A$$

where it is understood that all free variables of $a$ occur in $\Gamma$. Finally, one wants judgements about equality of terms. These take the form

$$\Gamma \vdash a = b :: A$$

for terms $\Gamma \vdash a :: A$ and $\Gamma \vdash b :: A$. As a typical example one can state the following in DTT.

$$n :: \mathit{Nat}[2] \vdash n^4 = n^5 :: \mathit{Nat}[2]$$

Notice how much information is contained in this judgement. Indeed, the facility for compact expression of complex facts is what makes DTT so attractive.

The above judgements are the four kinds of judgement distinguished by Martin-Löf [11].

**Products and sums.** The most often used type constructors in HOL are function space $\rightarrow$ and cartesian product $\times$. In DTT, these generalize to the dependent product $\Pi$ and dependent sum $\Sigma$. The formation rules for these types are

$$\frac{\Gamma, x :: A \vdash B :: \mathit{DType}}{\Gamma \vdash \Pi_{x::A}.\, B :: \mathit{DType}} \qquad \frac{\Gamma, x :: A \vdash B :: \mathit{DType}}{\Gamma \vdash \Sigma_{x::A}.\, B :: \mathit{DType}}$$

That is, given a dependent type $B$ (possibly) containing a variable of type $A$, then one can form the dependent product $\Pi_{x::A}.\, B$ and the dependent sum $\Sigma_{x::A}.\, B$. Denotationally one thinks about these in the following way.

$$\begin{aligned} [\![ \Pi_{x::A}.\, B ]\!] &= \left\{ \begin{array}{l} \text{the set of functions } f \text{ with domain } [\![ A ]\!], \text{such that for each } a \in [\![ A ]\!] \\ \text{one has } f(a) \in [\![ B[a/x] ]\!]; \end{array} \right. \\ [\![ \Sigma_{x::A}.\, B ]\!] &= \text{the set of `dependent' pairs } (a, b) \text{ where } a \in [\![ A ]\!] \text{ and } b \in [\![ B[a/x] ]\!]. \end{aligned}$$

Thus in case a variable $x$ does not occur in $B$ one has that $\Pi_{x::A}.\, B$ is $A \rightarrow B$ and that $\Sigma_{x::A}.\, B$ is $A \times B$. In this way $\Pi$ and $\Sigma$ generalize $\rightarrow$ and $\times$.

In line with the interpretations given above, one has the following introduction, elimination and conversion rules for $\Pi$ and $\Sigma$.

$$\frac{\Gamma, x :: A \vdash b :: B}{\Gamma \vdash \lambda x :: A.\, b :: \Pi_{x::A}.\, B}\ (\Pi\text{-I}) \qquad \frac{\Gamma \vdash c :: \Pi_{x::A}.\, B \qquad \Gamma \vdash a :: A}{\Gamma \vdash c\, a :: B[a/x]}\ (\Pi\text{-E})$$

$$\frac{\Gamma, x :: A \vdash b :: B \qquad \Gamma \vdash a :: A}{\Gamma \vdash (\lambda x :: A.\, b)\, a = b[a/x] :: B[a/x]} \qquad \frac{\Gamma \vdash c :: \Pi_{x::A}.\, B}{\Gamma \vdash \lambda x :: A.\, c\, x = c :: \Pi_{x::A}.\, B}$$

$$\frac{\Gamma \vdash a :: A \qquad \Gamma \vdash b :: B[a/x]}{\Gamma \vdash \langle a, b\rangle :: \Sigma_{x::A}.\, B}\ (\Sigma\text{-I})$$

$$\frac{\Gamma \vdash c :: \Sigma_{x::A}.\, B}{\Gamma \vdash \pi c :: A}\ (\Sigma\text{-E}_1) \qquad \frac{\Gamma \vdash c :: \Sigma_{x::A}.\, B}{\Gamma \vdash \pi' c :: B[\pi c/x]}\ (\Sigma\text{-E}_2)$$

$$\frac{\Gamma \vdash a :: A \qquad \Gamma \vdash b :: B[a/x]}{\Gamma \vdash \pi\langle a, b\rangle = a :: A} \qquad \frac{\Gamma \vdash a :: A \qquad \Gamma \vdash b :: B[a/x]}{\Gamma \vdash \pi'\langle a, b\rangle = b :: B[a/x]}$$

$$\frac{\Gamma \vdash c :: \Sigma_{x::A}.\, B}{\Gamma \vdash \langle \pi c, \pi' c\rangle = c :: \Sigma_{x::A}.\, B}$$

The $\Sigma$ types we are describing are the so-called 'strong sums', which come equipped with both projections. See [8] for more details about strong and weak sums.

### 3.2 Examples

Let's turn to some examples. The first one is taken from a paper by Hanna, Daeche, and Longley [7]. On first thought one may view a type *date* as $Nat \times Nat \times Nat$, where the first component represents the year, the second the month and the third the day. This can obviously be done in a far more precise way, since the number of months in a year does not exceed 12 and the number of days in a month does not come above 31. So a second try is $Nat \times Nat[12] \times Nat[31]$, which is already much better. But not every month has 31 days; even worse, the length of the month of February depends on the year. So the best representation is

$$date \quad = \quad \Sigma_{y::Nat}.\, \Sigma_{m::Nat[12]}.\, Nat[\mathsf{length\ of\ month}\ m\ \mathsf{in\ year}\ y]$$

where the term '$\mathsf{length\ of\ month}\ m\ \mathsf{in\ year}\ y$' is defined by cases in an obvious way. A typical term of type *date* is $\langle 1992, \langle 5, 16\rangle\rangle$.

This example nicely illustrates an important point made in [7]: by using dependent types one has a precise and concise typing discipline. This is especially convenient in hardware verification, where there are many kinds of values that are parameterized by size—for example bit-vectors, or integers mod $n$.

The second example comes from [11]. It shows that the axiom of choice (AC) holds in dependent type theory. Under the 'propositions-as-types' reading one views a type $A$ as a proposition and a term $a :: A$ as a proof of $A$. Then one further reads

$\Pi_{x::A}.\, B$ as universal quantification: for all $x$ in $A$ one has $B$

$\Sigma_{x::A}.\, B$ as existential quantification: there is an $x$ in $A$ for which $B$

Indeed a proof $c :: \Pi_{x::A}.\, B$ is then seen as a function which gives for each element $a$ of $A$ a proof $c\, a$ of $B[a/x]$. And a proof $c :: \Sigma_{x::A}.\, B$ consists of an element $\pi c$ of $A$ and a proof $\pi' c$ of $B[\pi c/x]$. This is the so-called 'Brouwer-Heyting-Kolmogorov' interpretation, see [17].

Thus the fact that (AC) holds in DTT amounts to the existence of a term $\mathsf{ac}$ such that

$$\vdash \mathsf{ac} :: (\Pi_{x::A}.\, \Sigma_{y::B}.\, C[x, y]) \to (\Sigma_{f::(\Pi_{x::A}.B)}.\, \Pi_{x::A}.\, C[x, f\, x])$$

We reason informally; suppose we are given a term $z :: \Pi_{x::A}.\, \Sigma_{y::B}.\, C[x, y]$. That is, suppose $z$ is a proof of 'for all $x$ in $A$ there is a $y$ in $B$ such that $C[x, y]$'. The aim is then to construct a function

$f$ which gives such a $y$ in $B$ for each $x$ in $A$. Notice that for every $x :: A$ one has $z\,x :: \Sigma_{y::B}.\, C[x, y]$ and thus $\pi(z\,x) :: B$ and $\pi'(z\,x) :: C[x, \pi(z\,x)]$. So if we take $f$ to be the term $\lambda x :: A.\, \pi(z\,x)$ of type $\Pi_{x::A}.\, B$, then $\pi'(z\,x) :: C[x, f\,x]$, which yields $\lambda x :: A.\, \pi'(z\,x) :: \Pi_{x::A}.\, C[x, f\,x]$. Hence we have

$$\mathsf{ac} \;=\; \lambda z :: (\Pi_{x::A}.\, \Sigma_{y::B}.\, C[x, y]).\, \langle \lambda x :: A.\, \pi(z\,x),\; \lambda x :: A.\, \pi'(z\,x)\rangle$$

Our third and final example is taken from [15]. The perspective is again 'propositions-as-types'. The aim of predicate logic is to formalize inferences like the following: 'All men are mortal; Socrates is a man; hence Socrates is mortal.' But not everything can be formulated in predicate logic. A famous elusive phrase is the so-called *donkey sentence*:

every man who owns a donkey beats it.

Try to formalize it! You'll find that the problem lies in 'it' referring back to the donkey. One might want to try

$$\forall d \in Donkey.\, \forall m \in Man.\, \mathsf{Owns}(m, d) \;\supset\; \mathsf{Beats}(m, d)$$

But the quantification here is over all donkeys instead of over men owning a donkey. This problem is solved in dependent type theory in the following elegant way.

$$\Pi_{m::Man}.\, \Pi_{x::(\Sigma_{d::Donkey}.\, \mathsf{Owns}(m,d))}.\, \mathsf{Beats}(m, \pi x)$$

### 3.3 Some formalities

The possibility in dependent type theory of term variables occurring in types has advantages when it comes to expressive power. A definite disadvantage is that it becomes rather cumbersome to formulate the calculus in a precise way. This is because one cannot first give the rules for types and then for terms; they depend on each other, and so one has to present them in a simultaneous induction.

A further point is that one needs certain predefined dependent types (like $Nat[n]$) to start with. Otherwise one doesn't get off the ground. Below we present the basic rules, mainly having to do with contexts. In these rules, $\mathcal{J}$ stands for one of the four judgements $A :: DType$, $a :: A$, $A = B :: DType$ and $a = b :: A$.

$$\textbf{start}\quad \frac{}{\Gamma \vdash A :: DType}\ \text{(for predefined } A \text{ and } \Gamma)$$

$$\textbf{projection}\quad \frac{\Gamma \vdash A :: DType}{\Gamma, x :: A \vdash x :: A}\ \text{(with } x \text{ a fresh variable)}$$

$$\textbf{exchange}\quad \frac{\Gamma, x :: A, y :: B, \Delta \vdash \mathcal{J}}{\Gamma, y :: B, x :: A, \Delta \vdash \mathcal{J}}\ \text{(if } x \text{ not free in } B)$$

$$\textbf{weakening}\quad \frac{\Gamma, \Delta \vdash \mathcal{J} \qquad \Gamma \vdash A :: DType}{\Gamma, x :: A, \Delta \vdash \mathcal{J}}\ \text{(with } x \text{ a fresh variable)}$$

$$\textbf{substitution}\quad \frac{\Gamma, x :: A, \Delta \vdash \mathcal{J} \qquad \Gamma \vdash a :: A}{\Gamma, \Delta[a/x] \vdash \mathcal{J}[a/x]}$$

We shall assume that we always have some start rules together with these basic context rules, in addition to whatever other rules are present in the theory.

# 4 Translating type dependency

The typical reaction of an experienced HOL user to the examples in section 3.2 is: 'But that can all be done in HOL; just use some suitable predicates to mimic dependent types'. The translation we are about to describe can be seen as a systematic elaboration of such a reaction.

## 4.1 Prejudgements

The table below gives a first description of the intended translation of dependent type theory into higher order logic.

| **DTT** | **becomes in HOL** |
|---|---|
| type, in isolation | predicate |
| declaration of variable $v$ | assumption of a predicate being true of $v$ |
| context | 'well-formed' list of assumptions |
| equality of types, in context | equivalence of predicates, with a list of assumptions |
| inhabitation judgement, in context | theorem, with a list of assumptions |
| equality of terms, in context | equality of terms, with a list of assumptions |

This table will be explained in more detail. The HOL predicates that will be used for the translation have the following form.

$$P : \sigma \to \mathit{bool}$$

Such a predicate $P$ need not be closed; it may contain free term variables, say $x_1, \ldots, x_n$. In case we want to have these explicit, we write $P[x_1, \ldots, x_n]$. Of course we could work with closed terms $P : (\sigma_1 \times \cdots \times \sigma_n) \to \sigma \to \mathit{bool}$ or simply with propositions $P[x_1, \ldots, x_n, y] : \mathit{bool}$, but the above form makes the most efficient use of the underlying HOL mechanism for handling variables.

Informally we'll say that $a :: P$ if $a$ is a HOL term of type $\sigma$ for which $P\,a$ is true. For example, one can take $\mathit{Nat}[n]$ to be the predicate $\lambda m.\ m < n$ of HOL type $\mathit{num} \to \mathit{bool}$. Then $m :: \mathit{Nat}[n]$ if and only if $m : \mathit{num}$ in HOL and $m < n$.

**Definition 4.1** *A* pseudo context *is a sequence* $x_1 :: P_1, \ldots, x_n :: P_n$ *where*

- $P_1, \ldots, P_n$ *are* HOL *predicates of the form* $P_i : \sigma_i \to \mathit{bool}$*;*
- $x_1, \ldots, x_n$ *are* HOL *variables with* $x_i : \sigma_i$*;*
- *the free variables of* $P_i$ *are among* $x_1, \ldots, x_{i-1}$*.* $P_1$*, in particular, is closed.*

Next we describe pseudo versions of the four forms of judgement in dependent type theory.

**Definition 4.2** *Let* $\Gamma = x_1 :: P_1, \ldots, x_n :: P_n$ *be a pseudo context.*

(i) $Q$ *is a* pseudo type *in context* $\Gamma$ *if* $Q$ *is of the form* $\tau \to \mathit{bool}$ *and all its free variables are among* $x_1, \ldots, x_n$*.*

(ii) $a$ *is a* pseudo term *of type* $Q$ *in context* $\Gamma$ *if* $Q$ *is a pseudo type in* $\Gamma$ *as in (i) and* $a$ *is a*

HOL *term of type* $\tau$ *with free variables among* $x_1, \ldots, x_n$ *for which there is a* HOL *theorem (with assumptions)*

$$P_1\, x_1,\ \ldots,\ P_n\, x_n \vdash Q\, a$$

(iii) $Q$ *and* $R$ *are* equal pseudo types *in context* $\Gamma$ *if both* $Q$ *and* $R$ *are pseudo types in context* $\Gamma$ *for which the following is a* HOL *theorem*

$$P_1\, x_1,\ \ldots,\ P_n\, x_n \vdash \forall y.\, Q\, y = R\, y$$

*where* $=$ *on booleans means logical equivalence in* HOL.

(iv) $a$ *and* $b$ *are* equal pseudo terms *of type* $Q$ *in context* $\Gamma$ *if both* $a$ *and* $b$ *are pseudo terms of type* $Q$ *in* $\Gamma$ *for which one has in* HOL

$$P_1\, x_1,\ \ldots,\ P_n\, x_n \vdash a = b$$

For example, with the formulation of $Nat[n]$ given above one has a 'pseudo term in context' $m :: Nat[10] \vdash m^2 :: Nat[82]$ and an 'equality of pseudo terms' $m :: Nat[2] \vdash m^4 = m^5 :: Nat[2]$. It is easy to work out the meanings of these 'pseudo judgements' from the above definition.

## 4.2 Validity of DTT rules

In order to prevent confusion about the calculus in which we are working, we shall from now on use $\vdash_{\mathrm{H}}$ for deducibility in HOL and $\vdash_{\mathrm{D}}$ for deducibility in DTT. It will be shown that all the DTT rules are valid under the interpretation described above. For a pseudo judgement $\mathcal{J}$ in context $\Gamma$, we shall write $\Gamma \vdash_{\mathrm{D}} \mathcal{J}$ if this is a judgement of DTT that can be derived from the underlying translation.

**Lemma 4.3** *Start rules are available. First of all, for any* HOL *type* $\sigma$, *define*

$$D(\sigma) \;=\; \lambda x.\, \mathsf{T} \;:\; \sigma \to bool$$

*where* $\mathsf{T} : bool$ *is the truth value 'true'. Then one has in the empty context* $\vdash_{\mathrm{D}} D(\sigma) :: DType$. *We'll write* $Nat = D(num)$ *and* $Bool = D(bool)$ *for the resulting 'dependent' naturals and booleans.*

*Here are some more primitive dependent types.*

$$\begin{array}{rcl} Nat[n] & = & \lambda m.\, m < n \;\;:\;\; num \to bool \\ List[n] & = & \lambda l.\, \mathsf{length}(l) = n \;\;:\;\; (num)list \to bool \\ Matrix[n,m] & = & \lambda z.\, \mathsf{length}(z) = n \;\wedge\; \forall i \le n.\, \mathsf{depth}(i,z) = m \;\;:\;\; ((num)list)list \to bool \end{array}$$

*where* $\mathsf{depth}(i,z)$ *is a* HOL *term which gives the length of the* $i$*-th (list) element of a list of lists* $z$. *We also have the following example judgements that involve these primitive types*

$$\begin{array}{rcl} n :: Nat & \vdash_{\mathrm{D}} & Nat[n] :: Dtype \\ n :: Nat & \vdash_{\mathrm{D}} & List[n] :: Dtype \\ n :: Nat, m :: Nat & \vdash_{\mathrm{D}} & Matrix[n,m] :: Dtype \end{array}$$

*Proof.* Obvious, by definition 4.2 (i). $\square$

**Lemma 4.4** *The context rules 'projection', 'exchange', 'weakening' and 'substitution' are all valid.*

*Proof.* Suppose that $\Gamma = x_1 :: P_1, \ldots, x_n :: P_n$ is a context with $\Gamma \vdash_{\mathrm{D}} Q :: \mathit{DType}$. Then $\Gamma, y :: Q \vdash_{\mathrm{D}} y :: Q$ translates to the sequent $P_1\, x_1, \ldots, P_n\, x_n, Q\, y \vdash_{\mathrm{H}} Q\, y$ being a theorem. The $P_i\, x_i$ are simply superfluous assumptions. In an equally simple way one has that the exchange rule involves changing the order of HOL assumptions and that weakening involves adding another HOL assumption. This can all be done without problems. Finally for the substitution rule, suppose we have $\Gamma \vdash_{\mathrm{D}} a :: Q$ and $\Gamma, y :: Q, \Delta \vdash_{\mathrm{D}} \mathcal{J}$. Then we have a HOL theorem $P_1\, x_1, \ldots P_n\, x_n \vdash_{\mathrm{H}} Q\, a$. Thus in the HOL interpretation of the judgement $\Gamma, y :: Q, \Delta \vdash_{\mathrm{D}} \mathcal{J}$ the assumption $Q\, y$ can be removed by first substituting $a$ for $y$ and then performing a cut on $Q\, a$. So one obtains $\Gamma, \Delta[a/x] \vdash_{\mathrm{D}} \mathcal{J}[a/x]$ as required. □

**Lemma 4.5** *The rules for the dependent product* $\Pi$ *are all valid.*

*Proof.* Let $\Gamma = x_1 :: P_1, \ldots, x_n :: P_n$ be a context provided with a type $\Gamma, y :: Q \vdash_{\mathrm{D}} R :: \mathit{DType}$, say with $Q$ of HOL type $\sigma \to \mathit{bool}$ and $R$ of HOL type $\tau \to \mathit{bool}$. We define a new predicate $\Pi_{y::Q}.\, R$ of HOL type $(\sigma \to \tau) \to \mathit{bool}$ by

$$\Pi_{y::Q}.\, R \quad = \quad \lambda f.\, \forall y{:}\sigma.\, Q\, y \;\supset\; R\,(f\, y)$$

Then $\Gamma \vdash_{\mathrm{D}} \Pi_{y::Q}.\, R :: \mathit{DType}$ because all the free variables of $\Pi_{y::Q}.\, R$ are among $x_1, \ldots, x_n$. Note that $y$ may, of course, occur in $R$.

Next we have to establish the validity of the introduction and elimination rules. Therefore assume we have $\Gamma, y :: Q \vdash_{\mathrm{D}} b :: R$. That is, assume that $b$ is a HOL term of type $\tau$ with free variables among $x_1, \ldots, x_n, y$ for which there is a HOL theorem $P_1\, x_1, \ldots, P_n\, x_n, Q\, y \vdash_{\mathrm{H}} R\, b$. Then we deduce in HOL,

$$\begin{array}{c}
P_1\, x_1, \ldots, P_n\, x_n, Q\, y \vdash_{\mathrm{H}} R\, b \\
\hline
P_1\, x_1, \ldots, P_n\, x_n, Q\, y \vdash_{\mathrm{H}} R((\lambda y.\, b)\, y) \\
\hline
P_1\, x_1, \ldots, P_n\, x_n \vdash_{\mathrm{H}} Q\, y \;\supset\; R((\lambda y.\, b)\, y) \\
\hline
P_1\, x_1, \ldots, P_n\, x_n \vdash_{\mathrm{H}} \forall y{:}\sigma.\;\, Q\, y \;\supset\; R((\lambda y.\, b)\, y) \\
\hline
P_1\, x_1, \ldots, P_n\, x_n \vdash_{\mathrm{H}} (\lambda f.\, \forall y{:}\sigma.\;\, Q\, y \;\supset\; R(f\, y))\, (\lambda y.\, b) \\
\hline
P_1\, x_1, \ldots, P_n\, x_n \vdash_{\mathrm{H}} (\Pi_{y::Q}.\, R)\, (\lambda y.\, b)
\end{array}$$

and thus by defining $\lambda y :: Q.\, b = \lambda y.\, b$ we obtain

$$\Gamma \vdash_{\mathrm{D}} \lambda y :: Q.\, b :: \Pi_{y::Q}.\, R$$

Hence abstraction in our translation of DTT is simply abstraction in HOL. Similarly for application: given $\Gamma \vdash_{\mathrm{D}} c :: \Pi_{y::Q}.\, R$ and $\Gamma \vdash_{\mathrm{D}} a :: Q$ we use the following deduction in HOL

$$\dfrac{\dfrac{\dfrac{P_1\, x_1, \ldots, P_n\, x_n \vdash_{\mathrm{H}} (\Pi_{y::Q}.\, R)\, c}{P_1\, x_1, \ldots, P_n\, x_n \vdash_{\mathrm{H}} \forall y{:}\sigma.\;\, Q\, y \;\supset\; R\,(c\, y)}}{P_1\, x_1, \ldots, P_n\, x_n \vdash_{\mathrm{H}} Q\, a \;\supset\; R[a/y]\,(c\, a)} \qquad P_1\, x_1, \ldots, P_n\, x_n \vdash_{\mathrm{H}} Q\, a}{P_1\, x_1, \ldots, P_n\, x_n \vdash_{\mathrm{H}} R[a/y]\,(c\, a)}$$

which yields that $\Gamma \vdash_{\mathrm{D}} c\, a :: R[a/y]$. The associated $\beta$ and $\eta$ rules are obviously valid. □

**Lemma 4.6** *The rules for the dependent sum* $\Sigma$ *are valid.*

*Proof.* Assume again we have a context $\Gamma = x_1 :: P_1, \ldots, x_n :: P_n$ and a type $\Gamma, y :: Q \models_{\mathrm{D}} R :: DType$ with $Q : \sigma \to bool$ and $R : \tau \to bool$. We define $\Gamma \models_{\mathrm{D}} \Sigma_{y::Q}.\, R :: DType$ by

$$\Sigma_{y::Q}.\, R \quad = \quad \lambda z.\, Q\,(\pi z) \;\wedge\; R[\pi z/y]\,(\pi' z) \quad : \quad (\sigma \times \tau) \to bool$$

If we have terms $\Gamma \models_{\mathrm{D}} a :: Q$ and $\Gamma \models_{\mathrm{D}} b :: R[a/y]$, then we deduce in HOL

$$\dfrac{\dfrac{P_1\, x_1, \ldots, P_n\, x_n \models_{\mathrm{H}} Q\, a \qquad\qquad P_1\, x_1, \ldots, P_n\, x_n \models_{\mathrm{H}} R[a/y]\, b}{P_1\, x_1, \ldots, P_n\, x_n \models_{\mathrm{H}} Q\, a \;\wedge\; R[a/y]\, b}}{P_1\, x_1, \ldots, P_n\, x_n \models_{\mathrm{H}} (\Sigma_{y::Q}.\, R)\,(\langle a, b\rangle)}$$

which yields $\Gamma \models_{\mathrm{D}} \langle a, b\rangle :: \Sigma_{y::Q}.\, R$, where $\langle -, - \rangle$ denotes the pairing from HOL. Similarly from $\Gamma \models_{\mathrm{D}} c :: \Sigma_{y::Q}.\, R$ one obtains $\Gamma \models_{\mathrm{D}} \pi c :: Q$ and $\Gamma \models_{\mathrm{D}} \pi' c :: R[\pi c/y]$. □

The identity types from [11] are also supported by our interpretation.

**Lemma 4.7** *The following rules for identity types are valid.*

$$\dfrac{\Gamma \models_{\mathrm{D}} a :: Q \qquad \Gamma \models_{\mathrm{D}} b :: Q}{\Gamma \models_{\mathrm{D}} \mathsf{Eq}_Q(a, b) :: DType} \qquad\qquad \dfrac{\Gamma \models_{\mathrm{D}} a = b :: Q}{\Gamma \models_{\mathrm{D}} * :: \mathsf{Eq}_Q\,(a, b)}$$

$$\dfrac{\Gamma \models_{\mathrm{D}} c :: \mathsf{Eq}_Q\,(a, b)}{\Gamma \models_{\mathrm{D}} a = b :: Q} \qquad\qquad \dfrac{\Gamma \models_{\mathrm{D}} c :: \mathsf{Eq}_Q\,(a, b)}{\Gamma \models_{\mathrm{D}} c = * :: \mathsf{Eq}_Q\,(a, b)}$$

*That is, $\mathsf{Eq}_Q(a, b)$ is inhabited if and only if $a = b$ and its only possible inhabitant is $*$.*

*Proof.* Define $\mathsf{Eq}_Q(a, b) = \lambda z.\, a = b \;:\; one \to bool$ with $z$ not in $a$ or $b$. □

**Theorem 4.8** *Dependent type theory can be translated into* HOL.

*Proof.* By the above series of lemmas. □

**Remark 4.9** The previous theorem deals with only the essential aspects of DTT. Here are some more details about the translation. One easily verifies that the following rules hold.

$$\dfrac{\Gamma \models_{\mathrm{D}} Q = R :: DType \qquad \Gamma \models_{\mathrm{D}} a :: Q}{\Gamma \models_{\mathrm{D}} a :: R} \qquad\qquad \dfrac{\Gamma \models_{\mathrm{D}} a :: Q \qquad \Gamma \models_{\mathrm{D}} a = b :: Q}{\Gamma \models_{\mathrm{D}} b :: Q}$$

We further mention that the translation yields a so-called 'extensional' version of dependent type theory, see section 11–5 of [17]. Rules like the following are valid.

$$\dfrac{\Gamma \models_{\mathrm{D}} Q = Q' :: DType \qquad \Gamma, y :: Q \models_{\mathrm{D}} R = R' :: DType}{\Gamma \models_{\mathrm{D}} \Pi_{y::Q}.\, R \;=\; \Pi_{y::Q'}.\, R' :: DType} \qquad \dfrac{\Gamma, x :: Q \models_{\mathrm{D}} b = b' :: R}{\Gamma \models_{\mathrm{D}} \lambda x.\, b = \lambda x.\, b' :: \Pi_{y::Q}.\, R}$$

Finally we'll have a closer look at the embedding $D : \mathrm{HOL} \to \mathrm{DTT}$ from lemma 4.3. We show that it trivially extends to terms and preserves $\times$ and $\to$.

**Theorem 4.10** (i) HOL *terms $M : \tau$ with free variables $x_1 : \sigma_1, \ldots, x_n : \sigma_n$ are the same as terms $x_1 :: D(\sigma_1), \ldots, x_n :: D(\sigma_n) \models_{\mathrm{D}} M :: D(\tau)$ in our translated* DTT.

(ii) *For* HOL *types $\sigma, \tau$ one has equalities of types*

$$\models_{\mathrm{D}} D(\sigma \times \tau) = D(\sigma) \times D(\tau) :: DType \qquad \textit{and} \qquad \models_{\mathrm{D}} D(\sigma \to \tau) = D(\sigma) \to D(\tau) :: DType$$

*Proof.* (i) Obvious; the predicates involved in $D(-)$ are always true and hence irrelevant.

(ii) By definition one has logical equivalences

$$\begin{array}{rcll} \models_{\mathrm{H}} (D(\sigma) \times D(\tau))\, z & = & (\Sigma_{y::D(\sigma)}.\, D(\tau))\, z & \text{with } y \text{ not in } D(\tau) \\ & = & D(\sigma)\, (\pi z) \;\wedge\; D(\tau)[\pi z/y]\, (\pi' z) & \\ & = & \top \wedge \top \;=\; \top \;=\; D(\sigma \times \tau)\, z. & \end{array}$$

Thus $\models_{\mathrm{D}} D(\sigma \times \tau) = D(\sigma) \times D(\tau) :: \mathit{DType}$ by definition 4.2 (iii). Similarly for $\to$ we have

$$\begin{array}{rcll} \models_{\mathrm{H}} (D(\sigma) \to D(\tau))\, f & = & (\Pi_{y::D(\sigma)}.\, D(\tau))\, f & \text{with } y \text{ not in } D(\tau) \\ & = & \forall y{:}\sigma.\;\; D(\sigma)\, y \;\supset\; D(\tau)\, (f\, y) & \\ & = & \forall y{:}\sigma.\;\; \top \supset \top \;=\; \top \;=\; D(\sigma \to \tau)\, f. & \end{array}$$

□

## 4.3 Translation of logic

Having seen the above translation of dependent type theory into HOL one asks how much of the logic of HOL can be used in the translated DTT. After all, by lemma 4.3 there is a dependent type $\mathit{Bool} = D(\mathit{bool})$. But note that if we go down this road in seeking an answer, we depart from a 'propositions-as-types' perspective, in that we will be using terms $\varphi :: \mathit{Bool}$ as logical formulas instead of types $A :: \mathit{DType}$ (as in the last two examples in section 3.2).

Recall that in a formulation with explicit contexts of term variables, the logic of HOL can be given in terms of sequents $\Gamma \mid \varphi_1, \ldots, \varphi_m \models_{\mathrm{H}} \psi$, where $\varphi_1, \ldots, \varphi_n, \psi$ are terms of type *bool* with all to their free term variables declared in the context $\Gamma$. We hope that this notation clearly conveys the essential point that with HOL one has (classical) logic in a simply typed ambience.

The same notation, only this time with the annotated turnstile $\models_{\mathrm{D}}$, will be used to describe the logic we find in our translated dependent type theory. This expresses the fact that we now have (classical) logic in a dependently typed environment. Thus we write

$$\Gamma \mid \varphi_1, \ldots, \varphi_m \models_{\mathrm{D}} \psi$$

where $\Gamma = x_1 :: P_1, \ldots, x_n :: P_n$ is a (translated) dependent context as before and $\varphi_1, \ldots, \varphi_m, \psi$ are terms of dependent type *Bool* in context $\Gamma$, if there is a HOL theorem

$$P_1\, x_1, \ldots, P_n\, x_n, \varphi_1, \ldots, \varphi_m \models_{\mathrm{H}} \psi$$

We shall call the resulting system *dependent logic*.

**Proposition 4.11** *In dependent logic one has universal and existential quantification over the values of dependent types. These are given by the following rules.*

$$\frac{\Gamma, y :: Q \mid \varphi_1, \ldots, \varphi_m \models_{\mathrm{D}} \psi}{\Gamma \mid \varphi_1, \ldots, \varphi_m \models_{\mathrm{D}} \forall y :: Q.\, \psi}\; (y \text{ not in } \varphi_1, \ldots, \varphi_m)$$

$$\frac{\Gamma \models_{\mathrm{D}} a :: Q \qquad \Gamma \mid \varphi_1, \ldots, \varphi_m \models_{\mathrm{D}} \forall y :: Q.\, \psi}{\Gamma \mid \varphi_1, \ldots, \varphi_m \models_{\mathrm{D}} \psi[a/y]} \qquad \frac{\Gamma \models_{\mathrm{D}} a :: Q \qquad \Gamma \mid \varphi_1, \ldots, \varphi_m \models_{\mathrm{D}} \psi[a/y]}{\Gamma \mid \varphi_1, \ldots, \varphi_m \models_{\mathrm{D}} \exists y :: Q.\, \psi}$$

$$\frac{\Gamma \mid \varphi_1, \ldots, \varphi_m \models_{\mathrm{D}} \exists y :: Q.\, \psi \qquad \Gamma, y :: Q \mid \varphi_1, \ldots, \varphi_m, \psi \models_{\mathrm{D}} \chi}{\Gamma \mid \varphi_1, \ldots, \varphi_m \models_{\mathrm{D}} \chi}\; (y \text{ not in } \varphi_1, \ldots, \varphi_m, \chi)$$

*Proof.* Suppose $Q$ is of HOL type $\tau \to bool$. Then take

$$\forall y :: Q.\, \psi \;=\; \forall y{:}\tau.\; Q\,y \;\supset\; \psi \qquad \text{and} \qquad \exists y :: Q.\, \psi \;=\; \exists y{:}\tau.\; Q\,y \;\wedge\; \psi$$

The above rules then follow from some easy deductions in the HOL logic. □

**Proposition 4.12** *Dependent logic has separation of the following form.*

$$\frac{\Gamma, y :: Q \vdash_{\mathrm{D}} \psi :: Bool}{\Gamma \vdash_{\mathrm{D}} \{y :: Q \mid \psi\} :: DType}\ (\text{sep}) \qquad \frac{\Gamma \vdash_{\mathrm{D}} a :: Q \qquad \Gamma \vdash_{\mathrm{D}} \psi[a/x]}{\Gamma \vdash_{\mathrm{D}} a :: \{y :: Q \mid \psi\}}$$

$$\frac{\Gamma, y :: Q \vdash_{\mathrm{D}} \psi :: Bool}{\Gamma, z :: \{y :: Q \mid \psi\} \vdash_{\mathrm{D}} z :: Q} \qquad \frac{\Gamma, y :: Q \mid \varphi_1, \ldots, \varphi_n, \psi \vdash_{\mathrm{D}} \chi}{\Gamma, z :: \{y :: Q \mid \psi\} \mid \varphi_1[z/y], \ldots, \varphi_n[z/y] \vdash_{\mathrm{D}} \chi[z/y]}$$

*Proof.* Take $\{y :: Q \mid \psi\} = \lambda y.\, Q\,y \;\wedge\; \psi$. □

**Example 4.13** (i) The above rule (sep) is quite useful for introducing new types. For example one can make $Nat[n]$ an abbreviation for the type introduced by

$$\frac{n :: Nat, m :: Nat \vdash_{\mathrm{D}} m < n :: Bool}{n :: Nat \vdash_{\mathrm{D}} \{m :: Nat \mid m < n\} :: Dtype}$$

and the interval $Int[m, n] = \{m, m{+}1, \ldots, n\}$ is defined by

$$\frac{n :: Nat, m :: Nat[n{+}1], k :: Nat[n{+}1] \vdash_{\mathrm{D}} m \leq k :: Bool}{n :: Nat, m :: Nat[n{+}1] \vdash_{\mathrm{D}} \{k :: Nat[n{+}1] \mid m \leq k\} :: DType}$$

(ii) There are dependent equality predicates $=_P$ for dependent types $P$ for which one has rules in both directions

$$\frac{\Gamma \vdash_{\mathrm{D}} a = b :: P}{\Gamma \mid \emptyset \vdash_{\mathrm{D}} a =_P b} \qquad \frac{\Gamma \mid \emptyset \vdash_{\mathrm{D}} a =_P b}{\Gamma \vdash_{\mathrm{D}} a = b :: P}$$

where, for clarity, the empty logical context is written explicitly. One simply defines $a =_P b$ to be '$P\,a \;\wedge\; P\,b \;\wedge\; a = b$'.

**Remark 4.14** The above dependent logic with sequents $\Gamma \mid \varphi_1, \ldots, \varphi_m \vdash_{\mathrm{D}} \psi$ corresponds to the internal language of a topos as described in detail in [14]. Because our language is based on the classical logic of HOL, we get the language of a *boolean* topos—that is, of a topos with classical logic.

## 4.4 Translation of some additional type constructors

In this section it will be shown how some extra features of HOL extend to the translated DTT. First, we mention that a consequence of theorem 4.10 is the following (expected) result.

**Proposition 4.15** *The type* $Nat = D(num)$ *is a natural numbers object in* DTT*, i.e. it comes with the following rules*

$$\frac{}{\vdash_{\mathrm{D}} 0 :: Nat} \qquad \frac{\Gamma \vdash_{\mathrm{D}} n :: Nat}{\Gamma \vdash_{\mathrm{D}} \mathsf{suc}\, n :: Nat}$$

$$\frac{\Gamma \models_{\mathrm{D}} a :: Q[0/z] \qquad \Gamma, x :: Nat, y :: Q[x/z] \models_{\mathrm{D}} b :: Q[\mathsf{suc}\, x/z]}{\Gamma, n :: Nat \models_{\mathrm{D}} \mathsf{R}_{x,y}(n, a, b) :: Q[n/z]}$$

*where $\mathsf{R}_{x,y}$ is a constant which binds the variables $x, y$ in $b$. It satisfies*

$$\begin{array}{rcl} \mathsf{R}_{x,y}(0, a, b) & = & a \\ \mathsf{R}_{x,y}(\mathsf{suc}\, n, a, b) & = & b[n/x, \mathsf{R}_{x,y}(n, a, b)/y] \end{array}$$

*These are the rules for dependent naturals as described in [11].*

*Proof.* Assume $\Gamma = x_1 :: P_1, \ldots, x_n :: P_n$ and $Q : \sigma \to bool$ where the variable $z :: Nat$ in $Q$ is one of the $x_i$. The validity of the two introduction rules follows from (i) in theorem 4.10. For the elimination rule we make use the following HOL theorem, see [3] or [12].

$$\models_{\mathrm{H}} \forall x f.\ \exists! g.\ (g\ 0 = x)\ \wedge\ \forall n.\ g(\mathsf{suc}\, n) = f\ (g\ n)\ n$$

Applying this to $a$ and $\lambda xy.b$, we obtain the existence of a term $g : num \to \sigma$ with

$$\models_{\mathrm{H}} g\,0 = a \qquad \text{and} \qquad \models_{\mathrm{H}} g(\mathsf{suc}\, n) = b[n/x,\, g\, n/y]$$

From the premisses

$$P_1\, x_1, \ldots, P_n\, x_n \models_{\mathrm{H}} Q[0/x]\, a \qquad \text{and} \qquad P_1\, x_1, \ldots, P_n\, x_n, Q[x/z]\, y \models_{\mathrm{H}} Q[\mathsf{suc}\, x/z]\, b$$

one obtains by an induction proof in HOL that $P_1\, x_1, \ldots, P_n\, x_n \models_{\mathrm{H}} Q[n/z]\,(g\, n)$. So if we put $\mathsf{R}_{x,y}(n, a, b) = g\, n$, then $\Gamma, n :: Nat \models_{\mathrm{D}} \mathsf{R}_{x,y}(n, a, b) :: Q[n/z]$. □

As discussed in [12], a coproduct (or sum) type $\sigma + \tau$ with the following introduction and elimination rules is definable in HOL.

$$\frac{M : \sigma}{\mathsf{inl}\, M : \sigma + \tau} \qquad \frac{N : \tau}{\mathsf{inr}\, N : \sigma + \tau} \qquad \frac{L : \sigma + \tau \qquad P[x] : \rho \qquad Q[y] : \rho}{\mathsf{case}_{x,y}(L, P, Q) : \rho}$$

where the variables $x : \sigma$ in $P$ and $y : \tau$ in $Q$ become bound in $\mathsf{case}_{x,y}(L, P, Q)$. The associated conversions are

$$\begin{array}{rcl} \mathsf{case}_{x,y}(\mathsf{inl}\, M, P, Q) & = & P[M/x] \\ \mathsf{case}_{x,y}(\mathsf{inr}\, N, P, Q) & = & Q[N/y] \\ \mathsf{case}_{x,y}(L, R[(\mathsf{inl}\, x)/z], R[(\mathsf{inr}\, y)/z]) & = & R[L/z] \end{array}$$

**Proposition 4.16** *Coproducts can be defined in (our version of)* DTT *with rules*

$$\frac{\Gamma \models_{\mathrm{D}} P :: DType \qquad \Gamma \models_{\mathrm{D}} Q :: DType}{\Gamma \models_{\mathrm{D}} P + Q :: DType}$$

$$\frac{\Gamma \models_{\mathrm{D}} a :: P}{\Gamma \models_{\mathrm{D}} \mathsf{inl}\, a :: P + Q} \qquad \frac{\Gamma \models_{\mathrm{D}} b :: Q}{\Gamma \models_{\mathrm{D}} \mathsf{inr}\, b :: P + Q}$$

$$\frac{\Gamma, x :: P \models_{\mathrm{D}} c :: R \qquad \Gamma, y :: Q \models_{\mathrm{D}} d :: R}{\Gamma, z :: P + Q \models_{\mathrm{D}} \mathsf{case}_{x,y}(z, c, d) :: R}\ \text{(with } x, y \text{ not in } R\text{)}$$

*Moreover, the embedding $D : \mathrm{HOL} \to \mathrm{DTT}$ preserves coproducts.*

*Proof.* Assume $\Gamma = x_1 :: P_1, \ldots, x_n :: P_n$ with $P : \sigma \to bool$ and $Q : \tau \to bool$. Define

$$P + Q \quad = \quad \lambda z.\, (\exists x{:}\sigma.\ z = \mathsf{inl}\, x \ \wedge\ P\, x) \ \vee\ (\exists y{:}\tau.\ z = \mathsf{inr}\, y \ \wedge\ Q\, y) \quad : \quad (\sigma + \tau) \to bool$$

The above introduction rules are then clearly valid. For the elimination rule we deduce in HOL from the first premisse

$$\dfrac{\dfrac{\dfrac{P_1\, x_1, \ldots, P_n\, x_n, P\, x \vdash_{\mathrm{H}} R\, c}{P_1\, x_1, \ldots, P_n\, x_n, (z = \mathsf{inl}\, x \ \wedge\ P\, x) \vdash_{\mathrm{H}} R\, c}}{P_1\, x_1, \ldots, P_n\, x_n, (z = \mathsf{inl}\, x \ \wedge\ P\, x) \vdash_{\mathrm{H}} R\, (\mathsf{case}_{x,y}(z, c, d))}}{P_1\, x_1, \ldots, P_n\, x_n, \exists x{:}\sigma.\ (z = \mathsf{inl}\, x \ \wedge\ P\, x) \vdash_{\mathrm{H}} R\, (\mathsf{case}_{x,y}(z, c, d))}$$

In a similar way one obtains from the second premisse a theorem

$$P_1\, x_1, \ldots, P_n\, x_n, \exists y{:}\tau.\ (z = \mathsf{inl}\, y \ \wedge\ Q\, y) \vdash_{\mathrm{H}} R\, (\mathsf{case}_{x,y}(z, c, d))$$

By unfolding the definition of $P + Q$ and using $\vee$-elimination in HOL one gets the theorem

$$P_1\, x_1, \ldots, P_n\, x_n, (P + Q)\, z \vdash_{\mathrm{H}} R\, (\mathsf{case}_{x,y}(z, c, d))$$

That is, one has in DTT that $\Gamma, z :: P + Q \vdash_{\mathrm{D}} \mathsf{case}_{x,y}(z, c, d) :: R$.

The embedding $D : \mathrm{HOL} \to \mathrm{DTT}$ preserves coproducts because for HOL types $\sigma, \tau$ one has

$$\begin{array}{rcl} \vdash_{\mathrm{H}} (D(\sigma) + D(\tau))\, z & = & (\exists x{:}\sigma.\ z = \mathsf{inl}\, x \ \wedge\ D(\sigma)\, x) \ \vee\ (\exists y{:}\tau.\ z = \mathsf{inr}\, y \ \wedge\ D(\tau)\, y) \\ & = & (\exists x{:}\sigma.\ z = \mathsf{inl}\, x) \ \vee\ (\exists y{:}\tau.\ z = \mathsf{inr}\, y) \\ & \overset{*}{=} & \mathsf{T} \ = \ D(\sigma + \tau)\, z. \end{array}$$

in which the equivalence $\overset{*}{=}$ holds by the following argument. For $z : \sigma + \tau$ abbreviate

$$R[z] \quad = \quad (\exists x{:}\sigma.\ z = \mathsf{inl}\, x) \ \vee\ (\exists y{:}\tau.\ z = \mathsf{inr}\, y)$$

Then, using the fact that the HOL types $\sigma$ and $\tau$ are non-empty we have $\vdash_{\mathrm{H}} R[(\mathsf{inl}\, x)/z] = \mathsf{T}$ and $\vdash_{\mathrm{H}} R[(\mathsf{inr}\, y)/z] = \mathsf{T}$. And thus

$$\begin{array}{rcl} R[z] & = & \mathsf{case}_{x,y}(z,\ R[(\mathsf{inl}\, x)/z],\ R[(\mathsf{inr}\, y)/z]) \\ & = & \mathsf{case}_{x,y}(z,\ \mathsf{T},\ \mathsf{T}) \\ & = & \mathsf{case}_{x,y}(z,\ \mathsf{T}[(\mathsf{inl}\, x)/z],\ \mathsf{T}[(\mathsf{inr}\, y)/z]) \\ & = & \mathsf{T}[z/z] \ = \ \mathsf{T}. \end{array}$$

□

Preliminary work indicates that all the type constructors definable using the HOL system's recursive types package [12] (e.g. the polymorphic list type $(\alpha)list$) can also be extended to DTT in a uniform and straightforward way. The details have, however, been left for future work.

## 5 Examples

In this section we describe three illustrative examples. The first one concerns the typical difference between programming in simple type theory and in dependent type theory. The second one involves the dependent logic described in section 4.3 and focusses on the use of the dependent and logical contexts. The third example gives some details of the proof of the pigeon hole principle in dependent logic.

Let's return to the first example in section 3.2. Consider the types of dates

$$\begin{aligned} date_1 &= Nat \times Nat \times Nat \\ date_2 &= \Sigma_{y::Nat}.\, \Sigma_{m::Nat[12]}.\, Nat[\textsf{length of month } m \textsf{ in year } y] \end{aligned}$$

together with functions

$$\textsf{dayadder}_i \;:\; Nat \times date_i \;\to\; date_i$$

which add a number of days to a given date (for $i = 1, 2$). The rather complicated precise form of such functions is not of much interest at this point. What we do want to emphasize is that the well-typedness of $\textsf{dayadder}_1$ is a trivial matter, whereas proving the well-typedness of $\textsf{dayadder}_2$ involves showing that it yields a triple whose components lie in the appropriate range (viz. in $Nat[12]$ and $Nat[\textsf{length of month } m \textsf{ in year } y]$). One of the main advantages of typing in general is that it provides partial specification. We conclude that functions that are well-typed in dependent type theory are more likely to be correct than functions that are well-typed in simple type theory.

Of course there is a price to pay; in DTT a type of a term must be provided by the programmer, together with a proof of well-typedness. This becomes particularly clear in our implementation, where such a proof is done in HOL. In contrast, type information in simple type theory can be inferred automatically.

Once it has been shown in our version of DTT that a term is well-typed, the typing is preserved even though the term itself is transformed by deductions in the underlying HOL logic (for example, by $\beta$-reduction). Thus there is a similarity—but on a different level—with functional languages whose programs are implemented as untyped combinators. Running a well-typed program means running it as a combinator term which is stripped of all type information. Similarly a well-typed DTT term can be run as a HOL term (i.e. term of the simply typed $\lambda$-calculus) in the underlying HOL logic.

The second example involves the logic of our dependent type theory. We introduce

$$\frac{\Gamma \vdash_{\mathrm{D}} P :: DType}{\Gamma \vdash_{\mathrm{D}} \mathcal{A}s(P) :: DType}$$

where $\mathcal{A}s(P)$ is an abbreviation defined by

$$\mathcal{A}s(P) \;=\; \Sigma_{m::P\to P\to P}.\, \Sigma_{u::P}.\, \Sigma_{v::P}.\, \Pi_{x::P}.\, \mathsf{Eq}_P(m\, u\, x, x) \;\wedge\; \mathsf{Eq}_P(m\, x\, v, x)$$

This type expresses the property that $P$ carries an applicative structure with left and right units. For an element $z :: \mathcal{A}s(P)$ we abbreviate

$$\mathsf{m}\, z = \pi\, z, \qquad \mathsf{lu}\, z = \pi\,(\pi'\, z), \qquad \text{and} \qquad \mathsf{ru}\, z = \pi\,(\pi'\,(\pi'\, z))$$

for the multiplication, left unit, and right unit involved. The following formula states that such an applicative structure $z :: \mathcal{A}s(P)$ is commutative.

$$\mathsf{comm}\, z \;=\; \forall x :: P.\, \forall y :: P.\, \mathsf{m}\, z\, x\, y =_P \mathsf{m}\, z\, y\, x$$

An easy result in our logic is the theorem

$$\Gamma, z :: \mathcal{A}s(P) \mid \mathsf{comm}\, z \vdash_{\mathrm{D}} \mathsf{lu}\, z =_P \mathsf{ru}\, z$$

which states a basic fact about such applicative structures.

This example illustrates the interaction between declarations in the *dependent* context of a sequent (the part before the separator '|') and the formulas in the *logical* context (the part after '|'). Similarly, one can define a type $\mathcal{G}(P)$ expressing that $P$ carries a group structure. If all the theorems one proves in dependent logic about a group $z :: \mathcal{G}(P)$ involve the logical assumption $\mathsf{comm}\, z$ stating that $z$ is commutative, then it is better to put this requirement into the dependent context by assuming $z :: \mathcal{AG}(P)$, where $\mathcal{AG}(P)$ means that $P$ carries an *abelian* group structure.

The third example is about the so-called pigeon hole principle. It says that if you distribute $n{+}2$ items over $n{+}1$ pigeon holes (for $n \geq 0$), then there is at least one pigeon hole containing two items. In a more mathematical formulation, it says that there is no injective function $Nat[n{+}2] \to Nat[n{+}1]$:

$$n :: Nat, f :: Nat[n{+}2] \to Nat[n{+}1] \mid \mathsf{injective}\, f \vdash_{\mathrm{D}} \mathsf{F}$$

where $\mathsf{F}$ denotes false. The principle can be proved by induction on $n$. We reason semi-informally in dependent logic and use the following lemma.

**Lemma 5.1** *For $n :: Nat$ define the function $g_n :: Nat[n{+}2] \to Nat[n{+}2] \to Nat[n{+}1]$ by*

$$g_n = \lambda m :: Nat[n{+}2].\, \lambda k :: Nat[n{+}2].\ \mathsf{if}\ \ k{<}m\ \ \mathsf{then}\ \ k\ \ \mathsf{elseif}\ \ k{>}m\ \ \mathsf{then}\ \ k{-}1\ \ \mathsf{else}\ \ 0\ \ \mathsf{fi}$$

*Then one has*

$$n :: Nat, m, k_1, k_2 :: Nat[n{+}2] \mid\ m \neq k_1, m \neq k_2 \vdash_{\mathrm{D}} g_n\, m\, k_1 = g_n\, m\, k_2\ \supset\ k_1 = k_2$$

*Proof.* Distinguish the cases

- $k_1 < m, k_2 < m$. Then $k_1 \;=\; g_n\, m\, k_1 \;=\; g_n\, m\, k_2 \;=\; k_2$.
- $k_1 > m, k_2 > m$. Then $k_1 \;=\; (g_n\, m\, k_1) + 1 \;=\; (g_n\, m\, k_2) + 1 \;=\; k_2$.
- $k_1 < m, k_2 > m$. This is impossible, since it yields $m \;>\; g_n\, m\, k_1 \;=\; g_n\, m\, k_2 \;\geq\; m$.
- $k_1 > m, k_2 < m$. As before. □

We now give the induction proof.
Case $n = 0$. Assume $f :: Nat[2] \to Nat[1]$. Then $f\, 0 = f\, 1 :: Nat[1]$, which, together with the assumption $\mathsf{injective}\, f$, yields the desired contradiction.
Case $n > 0$. Assume $f :: Nat[n{+}3] \to Nat[n{+}2]$ and put $m = f\,(n+2) :: Nat[n{+}2]$. Then by $\mathsf{injective}\, f$ one has $k :: Nat[n{+}2] \vdash_{\mathrm{D}} f\, k \neq m$. Define $f' = \lambda k :: Nat[n{+}2].\, g_n\, m\,(f\, k)$ where $g_n$ is the

function defined in the above lemma. We claim that $f' :: Nat[n{+}2] \to Nat[n{+}1]$ is injective; for $k_1, k_2 :: Nat[n{+}2]$ one derives using the above lemma

$$\dfrac{\dfrac{\dfrac{f'\,k_1 = f'\,k_2}{g_n\,m\,(f\,k_1) = g_n\,m\,(f\,k_2)} \qquad f\,k_1 \neq m \qquad f\,k_2 \neq m}{f\,k_1 = f\,k_2}}{k_1 = k_2}$$

Hence the induction hypothesis applied to $f'$ yields the required contradiction.

# 6 Implementation

The HOL system is based on the LCF approach to interactive theorem proving, originally due to Milner [6]. As in LCF, the system is based on the strongly-typed functional programming language ML.[7] Propositions and theorems of the logic are represented by ML abstract data types, and theorem-proving takes place by executing ML programs that operate on values of these data types. Because ML is a programming language, the user can write arbitrarily complex programs to implement proof strategies. Furthermore, because of the way the logic is represented in ML, such user-defined proof strategies are guaranteed to perform only valid logical inferences.

This approach is explained in more detail as follows. ML is a strongly-typed language; all expressions have types, and only consistently-typed expressions are syntactically well-formed. This type discipline is the basis for the soundness of proofs in HOL. The HOL system is built on top of ML by adding an abstract data type `thm`, values of which are theorems of higher order logic. The only predefined values of type `thm` are those which correspond to the five axioms of higher order logic listed in section 2. Furthermore, the only way of creating new values of type `thm` is by using certain built-in ML functions that take theorems as arguments and return theorems as results. Each of these corresponds to one of the primitive inference rules of the logic and returns only theorems that are deducible using this rule. Any value of type `thm` obtained in HOL must therefore be either an axiom or have been generated using the functions that represent the primitive inference rules of the logic—i.e. the only way to generate a theorem is to prove it.

In addition to the primitive inference rules, there are many derived inference rules available in the HOL system. These are ML procedures that perform commonly-used sequences of primitive inferences by calling the appropriate sequence of ML functions. Derived inference rules allow the user of HOL to do proofs in bigger steps, omitting explicit mention of all the necessary primitive inferences. The ML code for a derived rule can be arbitrarily complex, but it will never produce a theorem that does not follow by valid logical inference.

The LCF methodology just described is also the basis for the implementation of our translation of DTT into higher order logic. The judgements of DTT are represented by the values of certain abstract data types in ML, and these are defined so that the only admissible operations over them are ones that correspond to valid inferences of dependent type theory. The table below gives a sketch of the ML data types involved. The ML type `term` in the third column is a predefined HOL type whose elements are the well-typed terms of higher order logic.

[7] Not Standard ML, but an earlier version of the language; see [5].

<table>
<tr><th>Judgement</th><th>ML type</th><th>Representation</th></tr>
<tr><td>$\Gamma \vdash_{\mathrm{D}} A :: DType$</td><td><code>dtype</code></td><td><code>(term)list</code> $\times$ <code>term</code></td></tr>
<tr><td>$\Gamma \vdash_{\mathrm{D}} A = B :: DType$</td><td rowspan="3"><code>dthm</code></td><td><code>thm</code> $\times$ <code>dtype</code> $\times$ <code>dtype</code></td></tr>
<tr><td>$\Gamma \vdash_{\mathrm{D}} a :: A$</td><td><code>thm</code> $\times$ <code>dtype</code></td></tr>
<tr><td>$\Gamma \vdash_{\mathrm{D}} a = b :: A$</td><td><code>thm</code> $\times$ <code>dthm</code> $\times$ <code>dthm</code></td></tr>
</table>

The left column of this table lists the four forms of judgement in DTT, and the middle column shows the names of the ML abstract data types whose values represent these judgements. In ML, one defines an abstract data type by giving its values appropriate representations in an already existing data type. The third column of the table shows the representing types used in defining the abstract data types `dtype` and `dthm`. As was pointed out in section 3.3, the definitions of these two data types must be recursive, since the DTT rules for types and terms depend on each other.

As can be seen from this table, the HOL implementation of dependent type theory closely follows the interpretations of the four forms of DTT judgement given by definition 4.2. For example, a type-in-context $\Gamma \vdash_{\mathrm{D}} A :: Dtype$ (i.e. an element of `dtype`) is represented by a pair consisting of a list of terms representing the context $\Gamma$ together with a term representing the type $A$. This corresponds directly to the pseudo-context and predicate in the first clause of definition 4.2. Likewise, an inhabitation judgement $\Gamma \vdash_{\mathrm{D}} a :: A$ is represented by a type in context together with the appropriate HOL theorem (i.e. an element of `thm`).

Once the representation for an ML abstract data type has been specified, operations on the values of the abstract data type can then be defined in terms of corresponding operations on this representation. In ML, this is done in such a way that both the representation and the operations over it are hidden once the definition of an abstract type is completed, so that only the abstract values and operations are then available to the user. In the LCF approach to theorem proving, where abstract data types represent the judgements of a logic, these abstract operations correspond to primitive inference rules and are the only means of constructing judgements.

Our implementation of DTT also follows this general approach, with one important modification. In HOL there is a small fixed set of primitive inference rules, and so these can be taken as the only operations of the abstract type `thm` of HOL theorems. The set of inference rules for DTT, however, is rather open-ended; one often extends the system by postulating new rules for additional type constructors. We have therefore defined the operations on our abstract type `dthm` not to be inference rules, but rather mappings into the interpretation of DTT judgements in higher order logic. This allows new rules to be added just by programming new *derived* rules, rather than modifying the definition of the abstract type itself.[8]

For inhabitation judgements $\Gamma \vdash_{\mathrm{D}} a :: A$, for example, the definition of `dthm` provides a single operation, namely the ML function

```
INHAB : dtype → thm → dthm
```

This function is expected to be applied to an element of `dtype`

$$x_1 :: P_1, \ldots, x_n :: P_n \vdash_{\mathrm{D}} A :: Dtype$$

representing a dependent type in context and a corresponding HOL theorem

$$P_1\ x_1, \ldots, P_n\ x_n \vdash_{\mathrm{H}} A\ a$$

[8]This is in fact not quite true; see below.

where all the free variables in $a$ occur in $\{x_1, \ldots, x_n\}$. When applied to these values, the ML function `INHAB` produces an element of `dthm` that represents the inhabitation judgement

$$x_1 :: P_1, \ldots, x_n :: P_n \models_{\mathrm{D}} a :: A$$

This is just a direct implementation of clause (ii) of definition 4.2, which gives the interpretation of inhabitation judgements from DTT in the HOL logic. Similar mappings into `dthm` are defined for type equality judgements $\Gamma \models_{\mathrm{D}} A = B :: \mathit{Dtype}$ and term equality judgements $\Gamma \models_{\mathrm{D}} a = b :: A$. Note that a run-time error occurs if any of these functions is applied to inappropriate arguments; this ensures that `dthm` contains only judgements that are valid under our interpretation.

The implementation also provides functions that map DTT judgements (i.e. elements of `dthm`) to their interpretations. For example, for inhabitation judgements we have ML functions

`HOL : dthm` $\rightarrow$ `thm` and `DTYPE : dthm` $\rightarrow$ `dtype`

which extract from an inhabitation judgement its HOL theorem and dependent type components, respectively. Similar functions are provided for type equality and term equality judgements. These functions, together with the mappings into the interpretation explained above, allow us to program derived inference rules for DTT in the system. For example, the derivation given in section 4.2 for the dependent product rule

$$\frac{\Gamma, x :: A \models_{\mathrm{D}} b :: B}{\Gamma \models_{\mathrm{D}} \lambda x :: A.\, b :: \Pi_{x::A}.\, B} \ (\Pi\text{-I})$$

is implemented by an ML function

`PRODUCT_INTRO : dthm` $\rightarrow$ `dthm`

that extracts the HOL component of the judgement above the line, carries out the little HOL proof shown in the proof of lemma 4.5, and then injects the result back into the type `dthm` to get the judgement below the line.

The scheme described above applies only to the definition of the abstract type `dthm`, whose elements are the type equality, inhabitation, and term equality judgements of DTT. Judgements of the remaining form, namely dependent types in context, are represented by elements of the abstract type `dtype`. In contrast to the approach taken to defining `dthm`, the abstract type `dtype` is defined so that its operations form a set of 'primitive inference rules' for inferring judgements of the form $\Gamma \models_{\mathrm{D}} A :: \mathit{Dtype}$. In this case, we have chosen *not* to make available direct mappings into the interpretation, since otherwise any HOL predicate (in a suitable context) could become a dependent type. Instead, we wish `dtype` to contain only judgements that follow from explicitly-stated type formation rules together with the context rules in section 3.3.

Note that the construction of an element of `dthm` always requires an element of `dtype`, so by adopting the method just described we are also suitably restricting the `dthm` judgements we can generate to ones that involve only the dependent types we choose to make available. On the other hand, this scheme also means that we must extend the ML definition of `dtype` with new type formation rules whenever we wish to add a new type constructor; the rules cannot just be derived from existing ones. Furthermore, derived rules for `dthm` must carry out actual proofs of the required `dtype` judgements, in addition to doing the HOL proofs involved.

The ML types and functions described above form only the most primitive basis required for a theorem prover for DTT. To make the system practical, a very considerable infrastructure will have to be built on top of this basis—this still remains to be done. Future work will also involve an investigation of how best to mix reasoning in both DTT and HOL.

## References


[1] A. Church, 'A Formulation of the Simple Theory of Types', *The Journal of Symbolic Logic*, vol. 5 (1940), pp. 56–68.

[2] A. Felty, 'Encoding Dependent Types in an Intuitionistic Logic', in *Logical Frameworks*, edited by G. Huet and G. Plotkin (Cambridge University Press, 1991), pp. 215–251.

[3] M. Gordon, 'HOL: A Machine Oriented Formulation of Higher Order Logic', Technical Report 68, Computer Laboratory, University of Cambridge, revised version (July 1985).

[4] M. J. C. Gordon, 'HOL: A Proof Generating System for Higher-Order Logic', in *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P. A. Subrahmanyam, Kluwer International Series in Engineering and Computer Science (Kluwer, 1988), pp. 73–128.

[5] M. J. C. Gordon and T. F. Melham (eds.), *Introduction to HOL: A theorem proving environment for higher order logic*, forthcoming book (Cambridge University Press, 1993).

[6] M. J. Gordon, A. J. Milner, and C. P. Wadsworth, *Edinburgh LCF: A Mechanised Logic of Computation*, Lecture Notes in Computer Science, vol. 78 (Springer-Verlag, 1979).

[7] F. K. Hanna, N. Daeche, and M. Longley, 'Specification and Verification Using Dependent Types', *IEEE Transactions on Software Engineering*, vol. 16, no. 9 (September 1990), pp. 949–964.

[8] B. P. F. Jacobs, *Categorical Type Theory* (Ph.D. dissertation, University of Nijmgen, 1991).

[9] M. Leeser, 'Using Nuprl for the verification and synthesis of hardware', in *Mechanized Reasoning and Hardware Design: a Discussion Meeting held at the Royal Society, October 1991*, edited by C. A. R. Hoare and M. J. C. Gordon, Prentice Hall International Series in Computer Science (Prentice Hall, 1992), pp. 49–68.

[10] A. C. Leisenring, *Mathematical Logic and Hilbert's $\varepsilon$-Symbol*, University Mathematical Series (Macdonald & Co., 1969).

[11] P. Martin-Löf, *Intuitionistic Type Theory* (Bibliopolis, Naples, 1984).

[12] T. F. Melham, 'Automating Recursive Type Definitions in Higher Order Logic', in *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P. A. Subrahmanyam (Springer-Verlag, 1989), pp. 341–386.

[13] B. Nordström, K. Petersson, and J. M. Smith, *Programming in Martin-Löf's Type Theory: An Introduction*, International Series of Monographs on Computer Science 7 (Oxford University Press, 1990).

[14] W. Phoa, 'An introduction to fibrations, topos theory, the effective topos and modest sets', Technical Report ECS-LFCS-92-208, LFCS, Department of Computer Science, University of Edinburgh (April 1992).

[15] G. Sundholm, 'Proof Theory and Meaning', in *Alternatives in Classical Logic*, vol. 3 of *Handbook of Philosophical Logic*, edited by D. Gabbay and F. Guenthner, 4 vols. (D. Reidel, 1983–9), pp. 471–506.

[16] A. S. Troelstra, 'On the Syntax of Martin-Löf's Theories', *Theoretical Computer Science*, vol. 51, nos. 1–2 (1987), pp. 1–26.

[17] A. S. Troelstra and D. van Dalen, *Constructivism in Mathematics: An Introduction*, Studies in Logic and the Foundations of Mathematics, 2 vols. (North Holland, 1988).