Seminar Report

Constructive Type Theory and dependent types in Isabelle

Michael Färber

31 July 2012

Supervisor: Cezary Kaliszyk

Abstract

In this seminar report, we introduce the theorem prover Isabelle, Constructive Type Theory and dependent types. Furthermore, we present a dependent list type which we developed with the technologies mentioned above, and inspect the strengths and weaknesses of our approach.

Contents

1 Introduction

Isabelle is a generic theorem prover, which can be used to define and reason about different object-logics. We used it in conjunction with Constructive Type Theory (CTT), which is a type theory supporting dependent types. Dependent types are types which depend on values.

Consider the following C code as motivation for our work:

```
int main()
{
        int array[10];
        for (int i = 0; i <= 10; i++)
                array[i] = 0;
        return 0;
}
```

In this example, we show a programming error which is quite common — accessing array values outside their bounds. Such errors are not easy to detect at compile time, because the type of the array is just **int**∗ and does not carry any information about the *size* of the array, thereby making bounds determination difficult.

To shift the burden of determining array size from the compiler to the user, one can use a dependent list type, which stores the list length in the type. This makes it easy for the compiler to find out-of-bounds errors, because it has the length of each list at hand — in the type! On the other hand, proving the length of lists takes users additional time and requires experience.

We successfully implemented a dependent list type as motivated above in Isabelle using CTT, and wrote a number of proofs of list length to see how practicable our approach is. It turned out that the list type works well, but our proofs were unneccessarily big, which may be a result of missing tactics for CTT in Isabelle.

This document is structured as follows: In section §2, we describe the theorem prover Isabelle and give examples of its syntax. In section §3, we introduce Constructive Type Theory, which we'll later use together with Isabelle. To understand Constructive Type Theory, one needs to know about dependent types, which we explain in section §4. In section §5, we present our dependent list type in detail and describe its implementation as well as proofs about it. In section §6, we conclude by stating results of our work and giving an outlook to future work. We present the full proofs in the appendix.

1.1 Related work

While Isabelle is a logical framework allowing several object-logics to be implemented on top of it, there are theorem provers which allow only one object-logic to be used: For example, Coq [6] has a fixed logic, namely the Calculus of Constructions, which also allows dependent types to be implemented. On the other hand, other logical frameworks beside Isabelle supporting different object-logics exist, for example Twelf [8].

Operator	Description
$\Longrightarrow$	Implication
$\equiv$	Equality
$\bigwedge$	Universal quantification
$[\![p_1, p_2, \ldots, p_n]\!] \Longrightarrow q$	short-hand notation for $p_1 \Longrightarrow p_2 \Longrightarrow \ldots \Longrightarrow p_n \Longrightarrow q$

Table 1: Operators in Isabelle's meta-logic 'Pure'.

There exist several programming languages which support dependent types: For example, Agda [12] is a functional programming language with dependent types, which was developed by Ulf Norell. A similar language is Epigram [1], which was developed by Conor McBride and James McKinna. Dependent types are also used in the programming language F*, which is a programming language for distributed applications developed by Microsoft [16]. For an interesting comparison between some of the tools mentioned, see [18].

An alternative to verifying program correctness via type checking is static code analysis: Static code analysers try to verify properties in computer programs via various methods without having additional type information available. One programming language which was designed to support static code analysis is Spec# [5]. The language is a superset of C# and features constructs for e.g. preconditions, postconditions, and object invariants. These constructs are then used by Spec#'s static code verifier 'Boogie' in checking programs.

2 Isabelle

Isabelle is a theorem prover which was designed with the aim of supporting a variety of different logical systems. It is written in Standard ML and it features a hand-checked kernel, which assures the correct function of the theorem prover. To get an overview of Isabelle, [14] is a good introduction by Larry Paulson, who is one of the driving forces behind Isabelle. Also, the Isabelle tutorial and other documentation shipped with Isabelle are invaluable resources, although they mostly concentrate on Isabelle/HOL.

Different logical systems can be defined upon Isabelle's meta-logic 'Pure', which only contains a small number of logical operators; see table 1. The meta-logic is a fragment of intuitionistic higher-order logic, see [13].

Isabelle ships several logical systems by default (see figure 1); among these are:

- HOL: Higher-Order Logic, by far the most used logic in Isabelle.
- IFOL: Intuitionistic First-Order Logic. Forms the base of several other logics.
  - FOL: First-Order Logic. IFOL plus the classical axiom.
    - ZF: Zermelo-Fraenkel Set Theory. The second-most used logic in Isabelle after HOL.

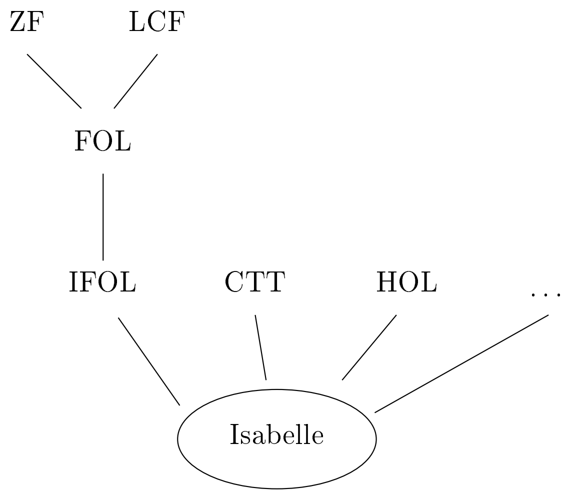

Figure 1: Logical systems in Isabelle.

  * LCF: Logic for Computable Functions. Historically relevant as logic of the theorem prover Edinburgh LCF from 1972, which motivated the development of the programming language ML. The successor of LCF, the theorem prover HOL, later motivated the development of Isabelle.

- CTT: Constructive Type Theory. We will mostly be concerned with it in this paper and discuss it below.

- ...

Forming object logics in Isabelle involves several tasks:

- Defining types and constants.

- Postulating axioms for these constants.

- Deriving lemmata from the axioms.

Consider an excerpt from the Isabelle version of IFOL as an example:

```
typedecl o

axiomatization
  False :: o and
  conj :: "[o, o] ⇒ o"  (infixr "&" 35) and
  disj :: "[o, o] ⇒ o"  (infixr "|" 30) and
  imp :: "[o, o] ⇒ o"  (infixr "⟶" 25)
where
  conjI: "⟦ P;  Q ⟧ ⟹ P&Q" and
  conjunct1: "P&Q ⟹ P" and
  conjunct2: "P&Q ⟹ Q" and

  disjI1: "P ⟹ P|Q" and
  disjI2: "Q ⟹ P|Q" and
  disjE: "⟦ P|Q;  P ⟹ R;  Q ⟹ R ⟧ ⟹ R" and

  impI: "(P ⟹ Q) ⟹ P⟶Q" and
  mp: "⟦ P⟶Q;  P ⟧ ⟹ Q" and

  FalseE: "False ⟹ P"
```

In this excerpt, there is a type o declared first, which is the type of propositions. Then there is a number of constants introduced, namely ‘False’, ‘conj’, ‘disj’ and ‘imp’. These constants each have a type; for example, the type of ‘False’ is just o. Similarly, the type of the other constants in this excerpt is $[o, o] \Rightarrow o$, meaning that they are functions which take a pair of propositions and return a proposition. Along with the type of the constants, short-hand notation is defined as well, to allow writing logical symbols with characters like ‘&’, ‘/’ and ‘⟶’.

In the second part of the excerpt, the axioms for the constants are introduced. For example, the axiom ‘conjI’ states that assuming P and Q hold, $P\&Q$ holds. Conversely, if we have $P\&Q$, then by ‘conjunct1’ and ‘conjunct2’ we can say that P as well as Q hold.

After postulating axioms, we derive lemmata from the given axioms to show certain properties of the logical system. For example, the following lemma (taken from IFOL) states that if $P\&Q$ holds and $[\![P;Q]\!] \Longrightarrow R$ holds as well, then R holds:

```
lemma conjE:
  assumes major: "P & Q"
    and r: "⟦ P; Q ⟧ ⟹ R"
  shows R
  apply (rule r)
   apply (rule major [THEN conjunct1])
  apply (rule major [THEN conjunct2])
  done
```

3 Constructive Type Theory

3.1 History

Constructive Type Theory, also called Intuitionistic Type Theory or Martin-Löf’s Type Theory, was devised by Swedish logician Per Martin-Löf in 1971 [9]. The first version of his type theory was impredicative, meaning that it defined an object by referring to itself. In particular, the theory contained a reflection principle

$$U \in U,$$

meaning that the universe U is an element of itself. However, this made the theory inconsistent, as famously shown by Girard’s paradox — see [15]. As a result, Martin-Löf developed new, consistent versions of his type theory, one of which is implemented in Isabelle. An excellent introduction to CTT is [11], and a related document by the same authors discusses CTT in greater length and detail [10].

3.2 Types and Instances

When implementing a logic like CTT in Isabelle, we have to think about the sorts of objects in our logic. For example, in IFOL we had only one sort, namely Church's type o, the type of propositions. In contrast, in CTT we have two sorts, namely:

- type (abbreviated t)
- instance (abbreviated i)

To explain these two sorts, consider natural numbers. In CTT, natural numbers are defined by first specifying that there exists a type representing the type of natural numbers. We express that by saying

$$N \; type,$$

meaning that N is a type.

Now we can describe the elements or *instances* of this type, namely:

$$\begin{gathered} 0 \in N \\ n \in N \Longrightarrow \mathrm{succ}(n) \in N, \end{gathered}$$

where succ is a function from instances to instances. $0 \in N$ means that the constant 0 is an instance of the type N, and $n \in N$ means that the variable n is an instance of N. That way we have defined our type of natural numbers by saying that N is a type, then specifying which elements are instances of that type. Of course, for a real natural number type, one requires more axioms than given here, but this is just to give you an intuition how types and instances work. You can also think of types as *sets*, which you can define by specifying which elements they contain.

3.3 Propositions as types

You might wonder at this point how propositions can be represented in CTT, when there does not seem to be a sort of propositions. Actually, there is a way to define propositions in CTT:

A proposition is identified with the type of its proofs.

Let's assume we want to prove an implication, namely $P \supset Q$. Then we need to find at least one function which, given any element of type P, always returns an element of type Q. If we can find such a function, then it will have the type $P \longrightarrow Q$.

Alternatively, let's assume we want to prove $P \wedge Q$. Then we need to find a pair which contains a proof p of P and a proof q of Q, so the pair would be $\langle p, q \rangle$. That pair would then be of the type $P \times Q$, the cartesian product of P and Q.

In fact, this way to interpret proofs is consistent with the Curry-Howard isomorphism and the Brouwer–Heyting–Kolmogorov interpretation, see [9, 17].

3.4 Functions

To describe functions in CTT, we first have to describe the kinds of equivalence there exist in CTT. Because we have two types available in CTT, it makes sense to define equality for both of these types, namely:

- Type equality: Can be expressed by writing $A = B$, meaning that two types A and B are equal.
- Instance equality: Can be expressed by writing $a = b \in A$, meaning that two instances are equal and they *both* are elements of type A.

Now we can describe functions in CTT. Functions in CTT can be defined with a Church-style lambda calculus, so an example would be

$$\lambda\!\!\lambda x \in N.\text{succ}(x),$$

which is the function returning the successor of a natural number. Note that at this point we wrote $\lambda\!\!\lambda$ instead of λ; this is to distinguish our object-logic lambda functions from meta-logic Curry-style lambda functions, which are provided by Isabelle.

Functions are *instances* in CTT, meaning that they have a type. So what type does the function above have? It is the type of functions from N to N, which is written as

$$N \longrightarrow N,$$

therefore

$$\lambda\!\!\lambda x \in N.\text{succ}(x) \in N \longrightarrow N.$$

Application works as following: Assuming we have an identity function for a type A

$$\lambda\!\!\lambda x \in A.x \in A \longrightarrow A,$$

we can apply it to an instance as following:

$$(\lambda\!\!\lambda x \in A.x)`a,$$

where ` is the application operator. By β-reduction, we can then conclude that

$$(\lambda\!\!\lambda x \in A.x)`a = a \in A,$$

provided that $a \in A$.

Function types like $N \longrightarrow N$ are just a special case of a more general function type, namely the dependent product, which has the general form

$$\Pi a \in A.B(a).$$

This is the type of functions which take instances a of type A and return instances of type $B(a)$, where B is a type *depending* on an instance a. In case of the function type $N \longrightarrow N$, we could write that as

$$\Pi n \in N.B(n),$$

where $B(n) = N$ for any n. Actually, B can be defined in that case as $\lambda n.N$, which is a meta-logic function returning the type N for any n.

At this point, it may be still unclear what a type depending on an instance might be. To remedy this, let's study dependent types.

$$\begin{tikzcd}
 & \lambda\omega \arrow[rr] & & \lambda C \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda P2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[uu, dashed] \arrow[rr, dashed] & & \lambda P\underline{\omega} \arrow[uu] \\
\lambda \rightarrow \arrow[uu] \arrow[ur, dashed] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

Figure 2: The λ-cube.

4 Dependent types

In typed lambda calculi, we have two different sorts: types and terms.[1] These two sorts can depend on each other in several ways, motivating dependent types. Good introductions to dependent types and to lambda calculus in general can be found in [3, 4].

The kinds of dependencies between types and terms can be shown with the λ-cube (see figure 2): At the corners of the λ-cube, different lambda calculi reside, each allowing different sets of dependencies between types and terms. The arrow $\longrightarrow$ stands for the inclusion relation, meaning that when one lambda calculus points to the other, the latter "inherits" all the functionality of the former. For example, λP allows all kinds of dependencies which $\lambda^{\rightarrow}$ supports, because $\lambda^{\rightarrow}$ points to λP.

Now to present a few lambda calculi in detail:

- $\lambda^{\rightarrow}$: The simply-typed lambda calculus allows only one kind of dependency, namely the dependency of *terms on terms*. This means that one can define functions in $\lambda^{\rightarrow}$ which take a term and return another term, for example
$$t(a) = fa.$$
In fact, all lambda calculi on the λ-cube support this dependency.

- $\lambda 2$: The polymorphic or second-order lambda calculus inherits the dependency of terms on terms from $\lambda^{\rightarrow}$ and adds another dependency, namely the dependency of *terms on types*. For example, one can define an identity function constructor
$$t(A) = \lambda x : A.x$$
which, given a type A, returns the identity function for this particular type A.

[1]"Terms" correspond to the "instances" in CTT.

- $\lambda\underline{\omega}$: This lambda calculus supports dependencies of *types on types*, allowing functions like
$$T(A) = A \longrightarrow A,$$
which takes a type A and returns the type of functions from A to A.

- λP: This lambda calculus supports dependencies of *types on terms*, allowing functions like
$$\begin{aligned} T(0) &= B, \\ T(n+1) &= A \longrightarrow T(n). \end{aligned}$$
This function $T(n)$ returns the type of functions which take n instances of type A and return one instance of type B. Note that in this example, A and B are fixed, because in λP it is not possible to define types depending on types.

In the four lambda calculi presented, we showed that four kinds of dependencies between types and terms exist:

- terms depending on terms,
- terms depending on types,
- types depending on types, and
- types depending on terms.

The least powerful lambda calculus on the λ-cube is $\lambda^{\rightarrow}$, because it only supports terms depending on terms, and the most powerful lambda calculus on the λ-cube is λC, which is Coquand and Huet's *Calculus of Constructions* [7] and supports all four kinds of dependencies between types and terms. CTT supports types depending on terms and terms depending on terms, so it resides in λP.

5 Dependent list type

5.1 Idea

In most programming languages, list types only depend on the type of elements in the list. In a potential implementation, one may use such a list type in the following way:

$$\begin{gathered} \mathrm{nil}(A) \in \mathrm{List}(A), \\ \mathrm{cons}(A)`a`\,\mathrm{nil}(A) \in \mathrm{List}(A). \end{gathered}$$

Here the empty list nil(A) only carries the type information that it is of type List(A), which is the type of lists containing elements of type A. The non-empty list cons(A)‘a‘ nil(A) (containing only the element $a \in A$) has exactly the same type as nil(A). Such a list type may be implemented in a lambda calculus like $\lambda\underline{\omega}$, because our list type only depends on another type.

However, we could think of a different list type which also includes the *list length* in the type. One might use such a list type as follows:

$$\mathrm{nil}(A) \in \mathrm{List}(A, 0),$$
$$\mathrm{cons}(A)`0`a`\,\mathrm{nil}(A) \in \mathrm{List}(A, 1).$$

This has the advantage that more information about lists is available at compile-time, which can prevent errors early because the type checker can check if certain operations are permitted. For example, we may demand that it should only be possible to retrieve the head of a non-empty list, by specifying that the head operation works on lists of non-zero length only. By encoding this in the type of the head operation, the type checker checks for every head operation if its input list has the correct type, which is only the case if it is non-empty, and throws an error at compile-time otherwise.

5.2 Implementation

At the beginning of the implementation of our dependent list type, we specify the type of our list type. Because our list type should depend on the type of elements and on the list length, List is a function which takes a tuple containing a type and an instance, and returns a type. Furthermore, we reason about this function in the object-logic as follows: If A is a type and $n \in N$, then $\mathrm{List}(A, n)$ is a type as well.

```
axiomatization
  List :: "[t,i]⇒t"
where
  list_type: "⟦A type; n ∈ N⟧ ⟹ List(A,n) type"
```

Having defined our list type, we can specify the elements of our list type. For this purpose, we define nil and cons. The nil function always represents an empty list for a given type A. The cons function for a type A takes a natural number n, an element of type A, and a List of elements of type A with length n. Given that, cons returns a List with a length by one greater than the input list length n. Note that giving the length of a list as a separate argument to the cons function is necessary, because otherwise we cannot accept lists as input arguments, due to the fact that one needs to give the list length in the list type. [2]

```
axiomatization
  nil  :: "t ⇒ i" and
  cons :: "t ⇒ i"
where
  nil_type: "A type ⟹ nil(A) ∈ List(A,0)" and
 cons_type: "A type ⟹ cons(A) ∈ (Π n ∈ N. A ⟶ List(A,n) ⟶
           List(A,succ(n)))"
```

[2]There exist theorem provers like Matita [2] which allow automatic derivation of some arguments from given types. Therefore such provers could make it obsolete to give the list length as a separate argument the way we did, because it is already present in the type of the list.

Having defined nil and cons, we now approach head and tail, here abbreviated as hd and tl. First we define the meta-logic type of hd and tl, then the object-logic type. The meta-logic types of hd and tl are the same, namely the type of functions which take a type and return an instance. On the other hand, the object-logic types of hd and tl differ: While both nil(A) and cons(A) take a natural number n and a list of type A with length succ(n), they return different objects – hd returns an object of type A, and tl one of type List(A, n).

The results of application to hd and tl are determined by the rules hd_appl and tl_appl — these rules correspond to iota-reduction. There we see that only the application of hd and tl to non-empty lists (constructed via cons) is defined.

```
axiomatization
  hd :: "t⇒i" and
  tl :: "t⇒i"
where
  hd_type: "A type ⟹
            hd(A) ∈ (Π n ∈ N. List(A,succ(n)) ⟶ A)" and
  hd_appl: "A type ⟹
            hd(A) ‘ succ(n) ‘ (cons(A) ‘ n ‘ h ‘ t) = h ∈ A" and

  tl_type: "A type ⟹
            tl(A) ∈ (Π n ∈ N. List(A,succ(n)) ⟶ List(A,n))" and
  tl_appl: "A type ⟹
            tl(A) ‘ succ(n) ‘ (cons(A) ‘ n ‘ h ‘ t) = t ∈ List(A,n)"
```

Now we define a list recursor function, which we call listrec. The output type B of listrec depends on the length n of the input list and on the input list itself. These dependencies can be used for example to construct a list with the same length as the input list.

listrec takes several parameters: First, it takes the type A of elements in the input list, and it takes a dependent type B, which gives the output type. Then, similar to hd and tl, it takes a list length n and a list of type List(A, n). Furthermore, listrec takes an element $bn \in B(0, \text{nil}(A))$, which will be the output for an input list of zero length. Also, listrec takes a parameter bc, which is a function taking the head and the tail of a list and an element of the type $B(n, l)$, which is the result of the application of bc respectively bn on the tail of the list, and returning an element of type $B(\text{succ}(n), \text{cons}(A)`n`h`t)$.

This function is inspired by the axioms NE, NEL, NC0 and NC_succ from the Isabelle CTT theory, which implement induction on natural numbers.

```
axiomatization
  listrec :: "t⇒(i⇒i⇒t)⇒i"
where
  listrec_type: "⟦A type; (Π n ∈ N. Π l ∈ List(A,n). B(n,l)) type⟧ ⟹
                 listrec(A,B) ∈ Π n ∈ N. Π l ∈ List(A,n).
                 (Π bn ∈ B(0,nil(A)).
                  Π bc ∈ (Π nc ∈ N. Π hc ∈ A. Π tc ∈ List(A,nc).
                   B(nc,tc) ⟶ B(succ(nc),cons(A)‘nc‘hc‘tc)). B(n,l))" and

  listrec_nappl: "⟦A type; (Π n ∈ N. Π l ∈ List(A,n). B(n,l)) type⟧ ⟹
                  listrec(A,B) ‘ 0 ‘ nil(A) ‘ bn ‘ bc = bn ∈ B(0,nil(A))"
                 and
```

```
listrec_cappl: "⟦A type; (Π n ∈ N. Π l ∈ List(A,n). B(n,l)) type⟧ ⟹
                listrec(A,B) ‘ succ(n) ‘ (cons(A) ‘ n ‘ h ‘ t) ‘
                bn ‘ bc =
                bc ‘ n ‘ h ‘ t ‘ (listrec(A,B) ‘ n ‘ t ‘ bn ‘ bc) ∈
                B(succ(n),cons(A) ‘ n ‘ h ‘ t)"
```

map is a function using listrec, taking an element of type List(A, n) and returning a List(B, n), which it receives by applying a function of type $A \longrightarrow B$ to each element of the input list, and concatenating the results to the output list.

definition
```
  map :: "t⇒t⇒i" where
  "map(A,B) ≡ λλn l f. listrec(A, λx y. List(B,x)) ‘ n ‘ l ‘ nil(B) ‘
   (λλn h t r. cons(B) ‘ n ‘ (f ‘ h) ‘ r)"
```

We prove that map, when applied to a List(A, n), produces a List(B, n), meaning it preserves the length of the input list. (See the appendix for the proof.)

lemma `"⟦n ∈ N; A type; B type; l ∈ List(A,n); f ∈ A ⟶ B⟧ ⟹`
```
       map(A,B) ‘ n ‘ l ‘ f ∈ List(B,n)"
```

To test listrec, we define a length function using listrec, and prove that its output really equals the length of the list. (Again, see the appendix for the proof.)

definition
```
  length :: "t⇒i" where
  "length(A) ≡ λλn l. listrec(A,λx y. N) ‘ n ‘ l ‘ 0 ‘
   (λλn h t r. succ(n))"
```

lemma `"⟦n ∈ N; A type; l ∈ List(A,n)⟧ ⟹ length(A) ‘ n ‘ l = n ∈ N"`

6 Conclusion

In the course of this seminar, we studied Constructive Type Theory (CTT) and its implementation in Isabelle. We did so by developing a dependent list type in CTT, thereby uncovering strengths and weaknesses of the logic.

Our dependent list type, encoding the list length in the type, works fine, but it has one flaw: Because CTT does not allow dependencies of types on types (which is present in $\lambda P\underline{\omega}$), we can't directly encode the type dependency of our list type in the object-logic (CTT), but we have to use Isabelle's meta-logic for that purpose. That is very unelegant and demands an implementation in a stronger logic, such as the *Calculus of Constructions*, for which a dependent list type (called *vector*) has already been implemented in Coq.

Apart from this, one of the greatest annoyances in proving properties of our list type was β-reduction — finding a reliable way to automate this would drastically shorten our proofs.

Learning Isabelle by starting with CTT turned out to be quite challenging, because most documentation and tutorials are concerned with HOL, which makes it hard to distinguish between the features Isabelle offers for all logics and those which are relevant to HOL only.

In the future, to aid the development of dependent types which also rely on types depending on types, it would be beneficial to establish more expressive object-logics in Isabelle, like the Calculus of Constructions or Pure Type Systems [3]. One might argue that establishing systems like the Calculus of Constructions in Isabelle would not be very helpful, as there already exists a dedicated theorem prover for this object-logic (Coq), but still, we might profit from implementing such an object-logic in a logical framework, because we may then use existing procedures of the logical framework and use results from one object-logic in another one.

Another thing one could do is to define new dependent data types, for example a dependent map type: This map type would save values of type B, which can be indexed via values of type A. To ensure that accessing values of type A not present in the map is detected as error, we could save a list of values of type A in the type of the map. Whenever one wants to access the map then, he would have to prove that the element he wants to access is contained in the values list. For this list, one could use a dependent list type as ours. Such a map type could be used as follows:

$$\begin{gathered}\mathrm{empty}(A, B) \in \mathrm{Map}(A, B, \mathrm{nil}(A)),\\ \mathrm{add}(A, B)`n`l`a`b`m \in \mathrm{Map}(A, B, \mathrm{cons}(A)`n`a`l).\end{gathered}$$

An interesting use case of dependent types is group theory: One can represent groups $(G, \bullet)$ as dependent types, with a type dependency on G and an instance dependency on the operation $\bullet$. Together with the group axioms, this allows one to reason about groups in general. Other mathematical structures, such as the type of n-dimensional vector spaces $\mathbb{R}^n$ can be expressed as dependent types as well, enabling one to prove mathematical properties of them. Research could also go into other dependent types, such as dependent pointers or functions with n arguments.

References

[1] T. Altenkirch, C. McBride, and J. McKinna. Why dependent types matter. Manuscript, available online, April 2005.

[2] A. Asperti, W. Ricciotti, C. S. Coen, and E. Tassi. The Matita interactive theorem prover. In *CADE*, volume 6803, 2011.

[3] H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[4] H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

[5] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceedings of the 2004 international conference*

on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, CASSIS'04, pages 49–69. Springer-Verlag, 2005.

[6] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[7] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2–3), 1988.

[8] H. Ganzinger, F. Pfenning, and C. Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Automated Deduction — CADE-16*, volume 1632 of *Lecture Notes in Computer Science*, pages 679–679. Springer, 1999.

[9] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[10] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's type theory.* Oxford University Press, 1990.

[11] B. Nordström, K. Petersson, and J. M. Smith. Martin-Löf's type theory. In *Handbook of Logic in Computer Science.* Oxford University Press, 2000.

[12] U. Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.

[13] L. C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, September 1989.

[14] L.C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[15] A. Stump. On Coquand's "An Analysis of Girard's Paradox", 1999.

[16] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 266–278. ACM, 2011.

[17] A.S. Troelstra. History of constructivism in the 20th century, 1991.

[18] F. Wiedijk. Comparing mathematical provers. In *Proceedings of the Second International Conference on Mathematical Knowledge Management*, MKM '03, pages 188–202. Springer-Verlag, 2003.

Appendix

In the appendix, we give the full proofs concerning our list type.

Recall the definition of map:

```
definition
  map :: "t⇒t⇒i" where
  "map(A,B) ≡ λλn l f. listrec(A, λx y. List(B,x)) ‘ n ‘ l ‘ nil(B) ‘
    (λλn h t r. cons(B) ‘ n ‘ (f ‘ h) ‘ r)"
```

We prove that map, when applied to a List(A, n), produces a List(B, n), meaning it preserves the length of the input list.

```
lemma "⟦n ∈ N; A type; B type; l ∈ List(A,n); f ∈ A ⟶ B⟧ ⟹
      map(A,B) ‘ n ‘ l ‘ f ∈ List(B,n)"
  apply (unfold map_def)
  apply (rule ProdE)
  defer
  apply assumption
  apply (rule ProdE)
  defer
  apply assumption
  apply (rule ProdE)
  defer
  apply assumption
  apply (rule ProdI)
  apply (rule NF)
  apply (rule ProdI)
  apply (rule list_type)
  apply assumption+
  apply (rule ProdI)
  apply (rule ProdF)
  apply assumption+
  apply (rule ProdE)+
  apply (rule listrec_type[of A "λx y. List(B, x)"])
  apply assumption
  apply (tactic {* typechk_tac [] *})
  apply (rule list_type)
  apply (tactic {* typechk_tac [] *})
  apply (rule list_type)
  apply (tactic {* typechk_tac [] *})
  apply (rule nil_type)
  apply assumption
  apply (rule list_type)
  apply (tactic {* typechk_tac [] *})
  apply (rule list_type)
  apply (tactic {* typechk_tac [] *})
  apply (rule cons_type)
  apply assumption
done
```

At this point, we noticed that β-reduction with more than one application (which we need at a later point) is not so trivial in CTT, therefore we made a small proof demonstrating β-reduction. It uses transitivity, substitution and reflexivity.

```
lemma "⟦A type; m ∈ A; n ∈ A⟧ ⟹ (λλx y. x) ‘ m ‘ n = m ∈ A"
apply (rule_tac b = "(λλy. m) ‘ n" in trans_elem)
apply (rule_tac a = "(λλx y. x) ‘ m" and c = "(λλy. m)" in subst_elemL)
apply (rule ProdC)
apply (tactic {* typechk_tac [] *})
apply (assumption)
apply (rule refl_elem)
apply (tactic {* typechk_tac [] *})
by (rule ProdC)
```

To test listrec, we define a length function using listrec, which is supposed to return the length of a list.

```
definition
  length :: "t⇒i" where
  "length(A) ≡ λλn l. listrec(A,λx y. N) ‘ n ‘ l ‘ 0 ‘
   (λλn h t r. succ(n))"
```

A small helper lemma helps us to determine the type of the application of arguments to listrec.

```
lemma listrec_appl: "⟦A type; (Π n ∈ N. Π l ∈ List(A,n). B(n,l)) type;
                     n ∈ N; l ∈ List(A,n);
                     bn ∈ B(0,nil(A));
                     bc ∈ Π nc ∈ N. Π hc ∈ A. Π tc ∈ List(A,nc).
                      Π rc ∈ B(nc,tc). B(succ(nc),cons(A)‘nc‘hc‘tc)⟧ ⟹
                    listrec(A,B) ‘ n ‘ l ‘ bn ‘ bc ∈ B(n,l)"
  apply (rule ProdE)+
  apply (rule listrec_type)
by assumption
```

Below is the proof that the length function really yields the desired output:

```
lemma "⟦n ∈ N; A type; l ∈ List(A,n)⟧ ⟹ length(A) ‘ n ‘ l = n ∈ N"
  apply (unfold length_def)
  apply (rule_tac b = "(λλl. listrec(A, λx y. N) ‘ n ‘ l ‘ 0 ‘
   (λλn h t r. succ(n))) ‘ l" in trans_elem)
  apply (rule_tac a = "(λλn l. listrec(A, λx y. N) ‘ n ‘ l ‘ 0 ‘
   (λλn h t r. succ(n))) ‘ n" and
   c = "(λλl. listrec(A, λx y. N) ‘ n ‘ l ‘ 0 ‘
   (λλn h t r. succ(n)))" in subst_elemL)
  prefer 2
  apply (rule refl_elem)
  apply (tactic {* typechk_tac [] *})
  apply (rule ProdC)
  apply assumption
  apply (tactic {* typechk_tac [] *})
  apply (rule list_type)
  apply (tactic {* typechk_tac [] *})
  apply (rule listrec_type)
  apply (tactic {* typechk_tac [] *})
  apply (rule list_type)
  apply (tactic {* typechk_tac [] *})
  apply (rule list_type)
apply (tactic {* typechk_tac [] *})

apply (rule_tac b = "listrec(A, λx y. N) ‘ n ‘ l ‘ 0 ‘
 (λλn h t r. succ(n))" in trans_elem)
apply (rule ProdC)
apply assumption
apply (tactic {* typechk_tac [] *})
apply (rule listrec_type)
apply (tactic {* typechk_tac [] *})
apply (rule list_type)
apply (tactic {* typechk_tac [] *})
apply (rule list_type)
apply (tactic {* typechk_tac [] *})

apply (rule EqE)
apply (rule_tac B = "λn l. Eq(N, listrec(A, λx y. N) ‘ n ‘ l ‘ 0 ‘
 (λλn h t r. succ(n)), n)" in listrec_appl)
apply assumption
apply (rule ProdF)
apply (rule NF)
apply (rule ProdF)
apply (rule list_type)
apply assumption+
apply (rule EqF)
apply (rule NF)
apply (rule listrec_appl)
apply (tactic {* typechk_tac [] *})
apply (rule list_type)
apply assumption+
apply (rule list_type)
apply assumption+
apply (rule EqI)
apply (rule listrec_nappl)
apply (tactic {* typechk_tac [] *})
apply (rule list_type)
apply assumption+
apply (rule ProdI)
apply (rule NF)
apply (rule ProdI)
apply assumption
apply (rule ProdI)
apply (rule list_type)
apply assumption+
apply (rule ProdI)
apply (rule EqF)
apply (rule NF)
apply (rule listrec_appl)
apply (tactic {* typechk_tac [] *})
apply (rule list_type)
apply assumption+
apply (rule list_type)
apply assumption+
apply (rule EqI)
apply (rule_tac b = "(λλn h t r. succ(n)) ‘ nc ‘ hc ‘ tc ‘
 (listrec(A,λx y. N) ‘ nc ‘ tc ‘ 0 ‘ (λλn h t r. succ(n)))" in trans_elem)
apply (rule listrec_cappl)
apply (tactic {* typechk_tac [] *})
apply (rule list_type)
apply assumption+

apply (rule_tac b = "(λλh t r. succ(nc)) ‘ hc ‘ tc ‘
 (listrec(A, λx y. N) ‘ nc ‘ tc ‘ 0 ‘ (λλn h t r. succ(n)))" in trans_elem)
apply (rule_tac a = "(λλn h t r. succ(n)) ‘ nc" and
 c = "(λλh t r. succ(nc))" in subst_elemL)
apply (rule ProdC)
apply (tactic {* typechk_tac [] *})
apply assumption
apply (rule_tac n = "nc" in list_type)
apply assumption
apply assumption
apply (rule NF)
apply (rule refl_elem)
apply (tactic {* typechk_tac [] *})
apply (rule listrec_type)
apply (tactic {* typechk_tac [] *})
apply (rule list_type)
apply assumption+
apply (rule list_type)
apply assumption+

apply (rule_tac b = "(λλt r. succ(nc)) ‘ tc ‘
 (listrec(A, λx y. N) ‘ nc ‘ tc ‘ 0 ‘
  (λλn h t r. succ(n)))" in trans_elem)
apply (rule_tac a = "(λλh t r. succ(nc)) ‘ hc" and
 c = "(λλt r. succ(nc))" in subst_elemL)
apply (rule ProdC)
apply assumption
apply (rule_tac A = "List(A,nc)" in ProdI)
apply (rule list_type)
apply assumption+
apply (rule_tac A = "N" in ProdI)
apply (tactic {* typechk_tac [] *})
apply (rule refl_elem)
apply (tactic {* typechk_tac [] *})
apply (rule listrec_type)
apply (tactic {* typechk_tac [] *})
apply (rule list_type)
apply assumption+
apply (rule list_type)
apply assumption+

apply (rule_tac b = "(λλr. succ(nc)) ‘
 (listrec(A, λx y. N) ‘ nc ‘ tc ‘ 0 ‘
  (λλn h t r. succ(n)))" in trans_elem)
apply (rule_tac a = "(λλt r. succ(nc)) ‘ tc" and
 c = "(λλr. succ(nc))" in subst_elemL)
```

```
  apply (rule ProdC)
  apply assumption
  apply (rule_tac A = "N" in ProdI)
  apply (tactic {* typechk_tac [] *})
  apply (rule refl_elem)
  apply (tactic {* typechk_tac [] *})
  apply (rule listrec_type)
  apply (tactic {* typechk_tac [] *})
  apply (rule list_type)
  apply assumption+
  apply (rule list_type)
  apply assumption+

  apply (rule ProdC)
  apply (rule ProdE)+
  apply (rule listrec_type)
  apply (tactic {* typechk_tac [] *})
  apply (rule list_type)
  apply assumption+
  apply (rule list_type)
  apply assumption+
done
```