Integrated Verification in Type Theory (Lecture Notes)

Thorsten Altenkirch
Ludwig-Maximilian-University, Munich, Germany
email: `alti@informatik.uni-muenchen.de`

September 26, 1996

Contents

1 Introduction

These notes present some simple examples of mechanic program verification in Type Theory. The examples have been developed using the ALF proof editor [1].

Type Theory is a basic formalism which can be viewed as a constructive alternative to set theory. The important ideas were developed by the Swedish logician Per Martin-Löf in the early 70's [ML75]. Due to its constructive character Type Theory can be viewed as a pure functional programming language whose type system is so powerful that specifications can be expressed as types. Verifying correctness is then reduced to type checking.

We shall present Type Theory based on this (modern) point of view. Intuitionistic logic can be encoded inside the theory using the idea of propositions as sets (and proofs as programs). We emphasise in particular the idea of integrated verification, i.e. where the program and the correctness proof are considered as a unit.

There are a number of implementations of Type Theory: One of the earliest is the *NuPRL system* which was developed by Constable and his group at Cornell [Con86]. Different to most of the other implementations NuPRL is based on *extensional Type Theory*, which has an undecidable type checking problem.

During two ESPRIT projects several systems based on Type Theory were developed in Europe: The LEGO system [LP92] in Edinburgh and the Coq system at INRIA are both based on the Calculus of Constructions by Coquand and Huet [CH86], which is an impredicative extension of Martin-Löf's Type Theory. Several generations of the ALF system were (and are) developed at Chalmers in Göteborg. This list is by no means complete and there are systems like Isabelle [PN90] which is a meta logic and can be (and has been) used to encode Type Theory.

There are a number of good introductions to Type Theory available now: lecture notes by Martin-Löf were once published as a book [ML84] which is, alas, not available anymore. A very nice tutorial on Type Theory with a similar spirit as ours has been published by Backhouse et al [BCMS89]. The book by Nordström et al [NPS90] also emphasises the programming view of Type Theory, one of the main differences to our exposition is that they do not use pattern matching. A more recent alternative to Nordström et al is Simon Thompson's book [Tho91]. Zhaohui Luo's book [Luo94] concentrates more on the meta theoretic investigation of UTT, which is pretty close to the language used in the LEGO system.

The material presented here is the background to some lectures I gave: First in February '95 as a postgraduate course at Chalmers University of Technology, then in October '95 as a one week seminar at the University of Kent and during summer term '96 as a course at the Ludwig-Maximilian-University in München. I prepare these notes especially for a course at the summer school ESSLLI '96 which is taking place in August '96 in Prague.

Acknowledgements

I would like to thank all the participants of the earlier version of this course for their interest and critical comments; Lena Magnusson and Johan Nordländer for their support with the ALF system; Martin Hofman for comments and Martin Weichert for comments and for proof reading these notes.

[1]Developed by Lena Magnusson (*proof engine*) and Johan Nordländer (*user interface*) at Chalmers University of Technology. See [MN94, Mag95] for some background on the ALF proof engine. A short and informal user's guide is available via ftp or WWW [AGNvS94]

2 Type Theory as a Programming Language

We shall introduce the basic concepts of Type Theory by example. Later we will be more precise. The Type Theory used here is the theory implemented in the ALF system and all the examples were developed with the system.

2.1 *Hello World* in Type Theory

We view Type Theory as a functional programming language with a powerful type system. One important difference to languages like SML or Haskell is that we only allow total (i.e. terminating programs). This is important if we want to use Type Theory as a programming logic.

A basic type is the type of sets **Set**. To every $A \in \mathbf{Set}$ we associate elements $a \in A$. New set constants or *connectives* are introduced by giving their constructors. Here is the definition of the of natural numbers Nat and the connective for lists List[2]:

```
Nat ∈ Set
    0 ∈ Nat
    s ∈ (Nat) Nat

List ∈ (A ∈ Set) Set
    nil ∈ (↓A ∈ Set) List(A)
    cons ∈ (↓A ∈ Set; a ∈ A; l ∈ List(A)) List(A)
```

The elements of Nat are well founded trees which can be built from the constructors, i.e. $0 \in \mathrm{Nat}, \mathrm{s}(0) \in \mathrm{Nat}, \ldots$. The connective List is itself a function, i.e. given a set A, $\mathrm{List}(A)$ is the set of lists which can be constructed using the constructors cons and nil: $\mathrm{nil}(A), \mathrm{cons}(A, a, \mathrm{nil}(A)), \mathrm{cons}(A, a, \mathrm{cons}(A, b, \mathrm{nil}(A))), \ldots$ where $a, b, \ldots \in A$.

We have already used *function types* in the example above, i.e. the type of s is (Nat)Nat which is denoted as $\mathrm{Nat} \to \mathrm{Nat}$ in more conventional notation (and in other implementations of Type Theory). By $(A; B)C$ etc. we denote curried function types, i.e. written as $A \to (B \to C)$ in conventional notation. The type of nil, $(A \in \mathbf{Set})\mathrm{List}(A)$, is an example of a *dependent function type*, in conventional notation this can be written as $\Pi A \in \mathbf{Set}.\mathrm{List}(A)$.

In conventional typed functional programming languages the constants nil and cons are polymorphic, i.e. they have infinitely many types. Here this is not the case, i.e. they are *monomorphic*. This is achieved by considering the (name of the) set A as an explicit parameter. This is the first use of *dependent types* we encounter.

We want now to define new functions over the sets we have introduced. Simple examples are addition of natural numbers add, and the second order function map which applies a function to every element of a list. We introduce new constants add, map and define them by *pattern matching*:

```
add ∈ (Nat; Nat) Nat
    add(0, h1) ≡ h1
    add(s(h2), h1) ≡ s(add(h2, h1))

map ∈ (↓A, ↓B ∈ Set; (A) B; List(A)) List(B)
    map(A, B, h, nil(_)) ≡ nil(B)
    map(A, B, h, cons(_, a, l)) ≡ cons(B, h(a), map(A, B, h, l))
```

map, *add* are called *non-canonical* or *implicit* constants. They do not introduce new elements in the sets defined so far because any application of add or map can be reduced to an expression constructed from canonical constants.

Not every definition from functional programming is legal in Type Theory: we require that the patterns are complete[3] and that the recursion is structural, i.e. a recursive call has to use a subexpression of the pattern. These restrictions are important to achieve the goals that non-canonical constants do not introduce new

[2] We ignore the ↓ for the moment.
[3] SML and Haskell give warnings for incomplete patterns.

elements in (first order) sets and that the reduction to canonical expressions terminates.

Another form of constants are *explicit constants* which can be viewed as implicit constants defined by a trivial pattern. An example is the following definition of the doubling function:

```
dbl ∈ (Nat) Nat
   dbl ≡ [h]add(h, h)
```

Here $[x]\ldots$ denotes λ-abstraction. λ-abstractions are also useful when defining *local* functions as in $\mathrm{map}([x]\mathrm{s}(\mathrm{s}(x)))l$ which increases every element of l by two.

Some implementations of Type Theory (e.g. LEGO, Coq) require to annotate λ-abstractions by the type of the domain, i.e. we would have to write $[x \in \mathrm{Nat}]\mathrm{add}(x, x)$. However, this is not necessary since we require that every definitions is typed and we expect that definitions do not contain β-redices (i.e. subterms of the form $([x]M)(N)$).

2.2 Hiding and argument synthesis

The syntax we have introduced seems very clumsy and explicit, e.g. consider the expression :

$$\mathrm{map}(\mathrm{Nat}, \mathrm{Nat}, \mathrm{s}, \mathrm{cons}(\mathrm{Nat}, 0, \mathrm{nil}(\mathrm{Nat}))$$

The explicit set parameter Nat clutters up the expression without giving any new information, because we can infer them from the types of s and 0.

Indeed, we can write

$$\mathrm{map}(?, ?, \mathrm{s}, \mathrm{cons}(?, 0, \mathrm{nil}(?))$$

and all the ?'s can be inferred. This mechanism is actually implemented in the ALF system.

The expression still looks too complicated, the ?'s don't convey any information either. Now we come back to the $\downarrow$'s we have ignored so far: They indicate that the marked arguments can be hidden, i.e. not printed. Using this the expression becomes:

$$\mathrm{map}(\mathrm{s}, \mathrm{cons}(0, \mathrm{nil}))$$

which is pretty close to the presentation in a polymorphic functional language.

The important point is that we don't have to make our basic language more complicated (e.g. by introducing polymorphism) to make the syntax legible and easy to write. We don't need type inference or the decidability of type inference as a basic property of our language. And the concept of argument synthesis turns out to be more general then type inference because we can also infer other parameters and not just the one which have type **Set** — this is often the case for the parameters to proofs.

Hiding is important to make complicated expressions readable. However, we can also hide too much, some important information can be lost. Hence, it is important for an implementation to make hidden arguments accessible if the user wants to see them.

2.3 Using dependent types in programming

In functional programming we will often introduce partial functions like $\mathrm{hd} \in (A \in \mathbf{Set}; \mathrm{List}(A))A$ which is defined by $\mathrm{hd}(A, \mathit{cons}(A, a, as)) \equiv a$. This is not a legal definition in Type Theory because the pattern is not complete, it does not cover the case of an empty list. In a functional programming evaluating hd(nil) will trigger an exception, i.e. a run-time error.

Using dependent types we can replace partial functions like hd by total ones and avoid the need for exceptions and run time errors. Moreover, since we have invested more effort at compile time, some programs can be, in principle[4], compiled to more efficient code. This is reminiscent to the gain in efficiency when moving from untyped programs (like LISP) to typed programs (like SML). Note that also in this case runtime errors are avoided at the same time.

We replace $\mathrm{List}(A) \in \mathbf{Set}$ by a refinement which we call vectors $\mathrm{Vec}(A) \in (\mathrm{Nat})\mathbf{Set}$. $\mathrm{Vec}(A)$ is a family of sets indexed by natural numbers and the idea is that $\mathrm{Vec}(A, n)$ is the set of lists of length n.

$\mathrm{Vec} \in (A \in \mathbf{Set};\, n \in \mathrm{Nat})\, \mathbf{Set}$
    $\mathrm{nil}_{\mathrm{Vec}} \in (\downarrow A \in \mathbf{Set})\, \mathrm{Vec}(A, 0)$
    $\mathrm{cons}_{\mathrm{Vec}} \in (\downarrow A \in \mathbf{Set};\, \downarrow n \in \mathrm{Nat};\, A;\, \mathrm{Vec}(A, n))\, \mathrm{Vec}(A, \mathrm{s}(n))$

$\mathrm{Vec}(A, n)$ are sets of expressions which can be built from $\mathrm{nil}_{\mathrm{Vec}}$ and $\mathrm{cons}_{\mathrm{Vec}}$ with the appropriate types, i.e.[5]

$$\begin{array}{rcl}
\mathrm{Vec}(A, 0) & = & \{\mathrm{nil}_{\mathrm{Vec}}\} \\
\mathrm{Vec}(A, \mathrm{s}(0)) & = & \{\mathrm{cons}_{\mathrm{Vec}}(a, \mathrm{nil}_{\mathrm{Vec}}) \mid a \in A\} \\
\mathrm{Vec}(A, \mathrm{s}(\mathrm{s}(0))) & = & \{\mathrm{cons}_{\mathrm{Vec}}(a_1, \mathrm{cons}_{\mathrm{Vec}}(a_2, \mathrm{nil}_{\mathrm{Vec}} \mid a_1, a_2 \in A\}) \\
\ldots & & \ldots
\end{array}$$

Now it is straightforward to define a total counterpart to hd:

$\mathrm{hd}_{\mathrm{Vec}} \in (A \in \mathbf{Set};\, n \in \mathrm{Nat};\, \mathrm{Vec}(A, \mathrm{s}(n)))\, A$
    $\mathrm{hd}_{\mathrm{Vec}}(A, n, \mathrm{cons}_{\mathrm{Vec}}(h_1, h_2)) \equiv h_1$

The pattern is complete because $\mathrm{nil}_{\mathrm{Vec}}$ is never of type $\mathrm{Vec}(A, s(n))$. The implementation of ALF supports the automatic generation of complete sets of patterns like this.

The function map can be refined to a function $\mathrm{map}_{\mathrm{Vec}}$ whose type indicates that it does not change the length of the list.

$\mathrm{map}_{\mathrm{Vec}} \in (\downarrow A, \downarrow B \in \mathbf{Set};\, \downarrow n \in \mathrm{Nat};\, (A)\, B;\, \mathrm{Vec}(A, n))\, \mathrm{Vec}(B, n)$
    $\mathrm{map}_{\mathrm{Vec}}(A, B, \_, h, \mathrm{nil}_{\mathrm{Vec}}(\_)) \equiv \mathrm{nil}_{\mathrm{Vec}}(B)$
    $\mathrm{map}_{\mathrm{Vec}}(A, B, \_, h, \mathrm{cons}_{\mathrm{Vec}}(\_, n_1, h_2, h_3)) \equiv$
        $\mathrm{cons}_{\mathrm{Vec}}(B, n_1, h(h_2), \mathrm{map}_{\mathrm{Vec}}(A, B, n_1, h, h_3))$

Another example is the function zip which applies a function componentwise to two lists constructing a new list (hence the name). A natural requirement for zip is that both lists have the same length, which can be easily expressed using dependent types and vectors.

$\mathrm{zip} \in (f \in (A; B)\, C;\, \mathrm{Vec}(A, n);\, \mathrm{Vec}(B, n))\, \mathrm{Vec}(C, n)$
    $\mathrm{zip}(f, \mathrm{nil}_{\mathrm{Vec}}, \mathrm{nil}_{\mathrm{Vec}}) \equiv \mathrm{nil}_{\mathrm{Vec}}$
    $\mathrm{zip}(f, \mathrm{cons}_{\mathrm{Vec}}(h_2, h_3), \mathrm{cons}_{\mathrm{Vec}}(h, h_4)) \equiv \mathrm{cons}_{\mathrm{Vec}}(f(h_2, h), \mathrm{zip}(f, h_3, h_4))$

An interesting consequence of the definition of zip is that we never have to analyse the second argument because it is determined by the form of the first one. This insight can be exploited to generate more efficient but still safe code.[6]

We may view $\mathrm{Vec}(A, n)$ as the type of arrays of a certain size. How can we define the access function for arrays without introducing partiality?

We introduce the family $\mathrm{Fin} \in (\mathrm{Nat})\mathbf{Set}$ which is a canonical representation of finite sets of size n and hence isomorphic to the set of natural numbers less than n.

$\mathrm{Fin} \in (\mathrm{Nat})\, \mathbf{Set}$
    $0_{\mathrm{Fin}} \in \mathrm{Fin}(\mathrm{s}(n))$
    $\mathrm{s}_{\mathrm{Fin}} \in (\mathrm{Fin}(n))\, \mathrm{Fin}(\mathrm{s}(n))$

[4]To my knowledge this idea has not yet been used in the implementation of a compiler.

[5]We omit hidden arguments here.

[6]This gain seems to be more then compensated by the additional argument of type Nat. However, we should be able to tell the compiler that this argument is never needed at runtime but only for type checking at compile time.

$\mathrm{Fin}(n)$ represents precisely the *address space* of $\mathrm{Vec}(A, n)$ and indeed we can implement a total version of nth as follows:

```
nth ∈ (Fin(n); Vec(A, n)) A
    nth(0_Fin, cons_Vec(h, h2)) ≡ h
    nth(s_Fin(h2), cons_Vec(h, h3)) ≡ nth(h2, h3)
```

Due to the use of argument synthesis and hiding the function definition above resembles the definition of the partial function nth for lists.

We summarise that dependent types can be used to avoid unsafe access to arrays without having to introduce a runtime overhead.

Exercise 1 *Write a library of vector-processing functions, with functions like:*

$$\begin{aligned}
\mathrm{append}_{\mathrm{Vec}} &\in (A \in \mathbf{Set}; m, n \in \mathrm{Nat}; \mathrm{Vec}(A, m); \mathrm{Vec}(A, n))\mathrm{Vec}(A, \mathrm{add}(m, n))\\
\mathrm{insert}_{\mathrm{Vec}} &\in (A \in \mathbf{Set}; n \in \mathrm{Nat}; \mathrm{Vec}(A, n); i \in \mathrm{Fin}(\mathrm{s}(n)); A)\mathrm{Vec}(A, \mathrm{s}(n))\\
\mathrm{delete}_{\mathrm{Vec}} &\in (A \in \mathbf{Set}; n \in \mathrm{Nat}; \mathrm{Vec}(A, s(n)); i \in \mathrm{Fin}(\mathrm{s}(n)))\mathrm{Vec}(A, n)
\end{aligned}$$

where append *is the obvious generalisation of append for lists and* insert *inserts an element at a specified position and* delete *deletes one.*

Exercise 2 *Define* $\mathrm{pow}(n) = 2^n$ *by repeated addition and define a function*

$$\mathrm{val} \in (n \in \mathrm{Nat}; x \in \mathrm{Vec}(n, \mathit{Bool}))\mathrm{Fin}(\mathrm{pow}(n))$$

which calculates the value of binary number represented as a (reversed) list of bits ($=$ Bool*).*

2.4 Higher-order sets

All the sets we have defined so far are first-order, i.e. the domains of the constructors do not include function types. However, we have already defined some second order functions, i.e. map and $\mathrm{map}_{\mathrm{Vec}}$.[7]

In the ALF's Type Theory all inhabitants of the type **Set** are defined by inductive definitions. In particular sets are not automatically closed under function types, it is not the case that $(A)B \in \mathbf{Set}$ even if $A, B \in \mathbf{Set}$. $(A)B$ is syntactically a type and never a set.

Indeed, in the moment we are not able to define the type of lists of functions $\mathrm{List}((A)B)$ because we can apply List only to a set not to a type.

However, the problem is not so serious, because we can represent the function set $\mathrm{Fun}(A, B)$ as follows :

```
Fun ∈ (A, B ∈ Set) Set
    fun ∈ ((A) B) Fun(A, B)
```

An important operation on function sets is application, which can be defined by pattern matching :

```
app ∈ (Fun(A, B); A) B
    app(fun(h2), h1) ≡ h2(h1)
```

One may ask why we do things in such a complicated way and do not just consider sets closed under $(A)B$. This is a possible choice but it has some disadvantages:

1. We lose the property that all sets are defined inductively (and hence we can always do pattern matching over a set).

2. In our system every constant has a fixed arity, i.e. the number of arguments it can be applied to. If we close sets under function types, a constant like $[A, x]x \in (A \in \mathbf{Set}; x \in A)A$ has an arbitrary number of arguments.

[7]The order of a function type $\mathrm{ord}((A)B)$ is defined as the $\max\{\mathrm{ord}(A) + 1, \mathrm{ord}(B)\}$. Hence *currying* $(A; B)C$ etc. does not increase the order.

3. We may want to consider a subsystem where all sets are first order. This can be easily achieved by restricting ourselves to sets whose constructors are first order.

The connective Fun introduces non-dependent function sets. In a similar way we can define dependent function spaces or Π-sets :

```
Π ∈ (A ∈ Set; B ∈ (A) Set) Set
    λ ∈ (f ∈ (a ∈ A) B(a)) Π(A, B)
app' ∈ (Π(A, B); a ∈ A) B(a)
    app'(λ(f), a) ≡ f(a)
```

Using higher order constructors we can introduce sets whose elements correspond to infinitary branching trees. An interesting application is the following notation system for countable ordinals :

```
Ord ∈ Set
    0' ∈ Ord
    s' ∈ (Ord) Ord
    lim ∈ ((Nat) Ord) Ord
```

Ordinal notations are generated inductively from $0'$, s' and lim. The latter allows us to construct limes ordinals over countably infinite sets of ordinals which are represented as a function of type (Nat)Ord. There is an obvious embedding from natural numbers to ordinals

```
nat2ord ∈ (Nat) Ord
    nat2ord(0) ≡ 0'
    nat2ord(s(h1)) ≡ s'(nat2ord(h1))
```

which we can use to define a representation for the smallest infinite ordinal ω :

```
ω ∈ Ord
    ω ≡ lim(nat2ord)
```

Using pattern matching we can implement ordinal addition:

```
addOrd ∈ (Ord; Ord) Ord
    addOrd(h, 0') ≡ h
    addOrd(h, s'(h2)) ≡ s'(addOrd(h, h2))
    addOrd(h, lim(h2)) ≡ lim([h1]addOrd(h, h2(h1)))
```

Exercise 3 *Implement multiplication and exponentiation for ordinal notations and define the ordinal ϵ_0 (the limes of ω towers). Implement the slow growing hierarchy $G \in (\mathrm{Ord};\mathrm{Nat})\mathrm{Nat}$ which is defined as follows:*

$$\begin{array}{rcl} G(\alpha, n) & = & 0 \\ G(\alpha+1, n) & = & G(\alpha, n)+1 \\ G(\lim(\lambda), n) & = & G(\lambda(n), n) \end{array}$$

This shows that we can define $G(\epsilon_0)$, which is not definable in Peano arithmetic.

The inhabitants of higher order sets like Ord can still be considered as well founded (but possibly infinitely branching) trees. This view is no longer possible when we use the set we are defining in a negative position, as in :

```
D ∈ Set
    lam ∈ ((D) D) D
```

Using D we can encode untyped λ-terms which destroys important properties of the system like decidability of equality and type checking.

Double negative definitions like

```
R ∈ (Set) Set
    r ∈ (A ∈ Set; ((R(A))A) A) R(A)
```

seem to be harmless as far as the essential properties are concerned. However, they are incompatible with our view that elements of sets are well-founded trees.

We conclude that we will only allow strictly positive definitions of sets.

3 Logic for free

So far we have just introduced programming language constructs. It seems that we have to extend the language to be able to express specifications and correctness proofs. Surprisingly, this is not the case, the logic is already there!

3.1 Propositional logic

The connectives of classical logic are defined as truth functions. I.e. to see that

$$((A \wedge B) \vee C) \to ((A \vee C) \wedge (B \vee C)) \tag{1}$$

is a tautology, we just have to check that all truth assignments to the propositional variables A, B, C evaluate to true. This is the well known idea of truth tables.

In intuitionistic logic we can define truth by provability. I.e. we explain how to construct proofs of compound propositions from proofs of components. This corresponds to the Brouwer-Heyting-Kreisel (BHK) semantics of intuitionistic logic:

conjunction A proof of $A \wedge B$ is a pair of proofs for A and B.

disjunction A proof $A \vee B$ is a bit b and

- a proof for A if $b = 0$,
- a proof for B if $b = 1$.

implication A proof for $A \to B$ is a procedure which computes proofs of B from proofs of A.

false There is no proof of False.

In Type Theory we go one step further and *identify a proposition with the set of its proofs.* The basic propositional connectives are now simply the Cartesian product $A \times B$ for conjunction, disjoint union $A + B$ for disjunction, the empty set $\emptyset$ for falsum and function space for implication.

Here are the definition of the propositional connectives in ALF (we repeat the function sets here) together with some useful implicit constants[8]:

```
And ∈ ($A, B$ ∈ Set) Set
    and ∈ ($A$; $B$) And($A, B$)
fst ∈ (And($A, B$)) $A$
    fst(and($h_1, h_2$)) ≡ $h_1$
snd ∈ (And($A, B$)) $B$
    snd(and($h_1, h_2$)) ≡ $h_2$
Or ∈ ($A, B$ ∈ Set) Set
    in$_1$ ∈ ($A$) Or($A, B$)
    in$_2$ ∈ ($B$) Or($A, B$)
or_case ∈ (($A$) $C$; ($B$) $C$; Or($A, B$)) $C$
    or_case($h, h_1$, in$_1$($h_3$)) ≡ $h(h_3)$
    or_case($h, h_1$, in$_2$($h_3$)) ≡ $h_1(h_3)$
Fun ∈ ($A, B$ ∈ Set) Set
    fun ∈ (($A$) $B$) Fun($A, B$)
app ∈ (Fun($A, B$); $A$) $B$
    app(fun($h_2$), $h_1$) ≡ $h_2(h_1)$
False ∈ Set
bot ∈ (False) $A$
```

Notation 1 *In the text we shall use the ordinary symbols* $\wedge, \vee, \to$ *to denote* And, Or, Fun. *Alternatively we will also use* $\times$ *and* $+$ *for* And *and* Or. *Alas the interface to the ALF editor is not yet so flexible (e.g. it doesn't allow infix notation).*

[8]Note that we haven't left out the definition of False but False is defined as a set with no constructors

Based on this we can now construct a proof of (1) using pattern matching (and an auxiliary function taut1) :

```
taut1 ∈ (A, B, C ∈ Set; Or(And(A, B), C)) And(Or(A, C), Or(B, C))
    taut1(A, B, C, in₁(and(h, h₂))) ≡ and(in₁(h), in₁(h₂))
    taut1(A, B, C, in₂(h₁)) ≡ and(in₂(h₁), in₂(h₁))
taut ∈ (A, B, C ∈ Set) Fun(Or(And(A, B), C), And(Or(A, C), Or(B, C)))
    taut ≡ [A, B, C]fun(taut1(A, B, C))
```

Note that the idea of the classical and the type theoretic proof are quite different. We do not consider possible truth assignments but we construct a concrete object, in fact a program, which gives direct evidence for the truth of the proposition.

Slogan : Theorem proving in Type Theory is the same as writing a (functional) program.

As in intuitionistic logic the principle of excluded middle $A \vee \neg A$ is not provable in Type Theory. The reason is that we cannot write a program of type $(A \in$ **Set**$)\mathrm{Or}(A, \mathrm{Fun}(A, \mathrm{False}))$. Such a program would decide for every set whether it is empty or not. Since we can encode arbitrary propositions in Type Theory such a program would magically decide the truth of any (unproven) propositions like Goldbach's hypothesis that there exists an infinite number of twin prime numbers.

Another principle which is unprovable in intuitionistic logic is $\neg\neg A \to A$. Indeed this is equivalent to the excluded middle. As rule of thumb we can say that proofs of negated formulas carry less information than positive (unnegated formulas). This also affects the *de Morgan Laws*: $(\neg A \wedge \neg B) \to \neg(A \vee B)$ is provable but $\neg(A \vee B) \to (\neg A \wedge \neg B)$ is not. As a consequence intuitionistic logic differs from its classical counterpart that we cannot define the connectives in terms of each other (i.e. in classical logic all connectives can be defined from $\vee, \neg$).

Exercise 4 *Encode and prove the following proposition in Type Theory :*

1. $((A \wedge B) \to C) \iff (A \to (B \to C))$
2. $((A \vee B) \to C) \iff ((A \to C) \wedge (B \to C))$
3. $(\neg A \wedge \neg B) \to \neg(A \vee B)$
   Can these be generalised to $\iff$ *?*
4. $\neg(A \iff \neg A)$

Exercise 5 *Show formally that the principles* $\neg\neg A \to A$ *and* $A \vee \neg A$ *are equivalent.*

3.2 Predicate logic

How do we interpret predicates like *Prime*? For every natural number we obtain a proposition $\mathrm{Prime}(n)$ which we consider as a set of proofs. Hence, a predicate is a family of sets $\mathrm{Prime} \in (\mathrm{Nat})$**Set** i.e. it has the same type as $\mathrm{Vec}(A)$ or Fin. However, the latter are uninteresting from a logical point of view[9].

In defining the basic connectives we follow the semantic intuition introduced in the last section :

Universal quantification Given a predicate $B \in (A)Set$ over $A \in$ **Set**, we say that B is universally true, if we can define a function which assigns to every $a \in A$ a proof in $B(a)$. This precisely corresponds to the dependent function space we have introduced earlier.

```
Π ∈ (A ∈ Set; B ∈ (A)Set) Set
    λ ∈ (f ∈ (a ∈ A) B(a)) Π(A, B)
app' ∈ (Π(A, B); a ∈ A) B(a)
    app'(λ(f), a) ≡ f(a)
```

[9] $\mathrm{Vec}(A)$ is always true, because it always has a proof $\mathrm{nil}_{\mathrm{Vec}}$ and Fin is true for $n > 0$ because it always has a proof 0_{Fin}.

Existential quantification Given $A \in \mathbf{Set}, B \in (A)\mathbf{Set}$ as above, we say that B is existentially true if we have an $a \in A$ and a proof $b \in B(a)$. This type is called a Σ-type and can be defined as follows:

```
Σ ∈ (A ∈ Set; B ∈ (A) Set) Set
   pair ∈ (a ∈ A; b ∈ B(a)) Σ(A, B)
π₁ ∈ (Σ(A, B)) A
   π₁(pair(a, b₁)) ≡ a
π₂ ∈ (p ∈ Σ(A, B)) B(π₁(p))
   π₂(pair(a, b₁)) ≡ b₁
```

Notation 2 *We will use* $\Pi x \in A.B, \Sigma x \in A.B$ *or alternatively* $\forall x \in A.B, \exists x \in A.B$ *to denote* $\Pi(A, [x]B), \Sigma(A, [x]B)$.

Σ-types are called the type of dependent pairs because they generalise the type of pairs And in a way similar to the way dependent function space Π generalises Fun.

There are also computational applications for Σ-types. Just to mention a simple one: we can recover the type of lists from vectors by defining $\mathrm{List}'(A) \equiv \Sigma n \in \mathrm{Nat}.\mathrm{Vec}(A, n)$, i.e. the infinite, disjoint union of $\mathrm{Vec}(A, n)$ over all $n \in \mathrm{Nat}$.

Again we cannot define the connectives in term of each other, i.e. although we can prove $(\exists a \in A.\neg B(a)) \to \neg(\forall a \in A.B(a))$ we cannot show the contraposition $\neg(\forall a \in A.\neg(B(a)) \to (\exists a \in A.B(a))$.

By the axiom of choice we mean here that for every relation $R \in (A; B)\mathbf{Set}$ such that we can show $\forall x \in A.\exists y \in B.R(a, b)$ we can actually construct a function $f \in (A)B$ such that $\forall x \in A.R(a, f(a))$. In classical set theory this axiom is considered as an additional assumption which has a number of (sometimes counterintuitive) consequences. The situation in intuitionistic logic and in Type Theory is different, because the proof of the premise already contains such a *choice function*. Indeed, we can prove the axiom of choice by pattern matching:

```
ac ∈ (A, B ∈ Set;
        R ∈ (A; B) Set;
        Π(A, [h]Σ(B, [h₁]R(h, h₁)))
      ) Σ(Fun(A, B), [h]Π(A, [h₁]R(h₁, app(h, h₁))))
   ac(A, B, R, λ(f)) ≡ pair(fun([h]π₁(f(h))), λ([a]π₂(f(a))))
```

Exercise 6 *Prove the following propositions in Type Theory:*

1. $((\exists x \in X.A(x)) \to B) \iff (\forall x \in X.A(x) \to B)$
2. $(\exists x : X.A(x)) \to (\neg(\forall x \in X.\neg A(x)))$

3.3 Equality

So far we have not introduced any (interesting) basic predicates or relations. We will use the mechanism for defining new sets to define new predicates — our first example is equality.

Equality for a set A is a predicate over pairs of A i.e. it will have the type $(A; A)\mathbf{Set}$. As before we will represent a polymorphic operation by explicit parametrisation, hence equality should have the type $(A \in \mathbf{Set}; A; A)\mathbf{Set}$.

The definition of equality in Type Theory is very simple: the only proof of an equality is reflexivity, which becomes the constructor id for equality proofs.

```
Id ∈ (↓A ∈ Set; a, b ∈ A) Set
   id ∈ (↓A ∈ Set; ↓x ∈ A) Id(A, x, x)
```

Let us prove that Id is an equivalence relation. Since reflexivity is the constructor, it remains to show symmetry and transitivity. It turns out that this is very easy using the idea of pattern matching in ALF:

```
symId ∈ (↓A ∈ Set; ↓a, ↓b ∈ A; Id(A, a, b)) Id(A, b, a)
   symId(A, _, b, id(_, _)) ≡ id(A, b)
transId ∈ (↓A ∈ Set; ↓a, ↓b, ↓c ∈ A; Id(A, a, b); Id(A, b, c)) Id(A, a, c)
   transId(A, _, _, c, id(_, _), id(_, _)) ≡ id(A, c)
```

How can we understand the proof of symmetry symId? Since the only constructor of $\mathrm{Id}(A, a, b)$ is $\mathrm{id}(a)$, hence the pattern is complete. Moreover since $\mathrm{id}(\ a)$ has the type $\mathrm{Id}(A, a, a)$ we know that $a \equiv b$[10]. Applying this constraint to the goal $\mathrm{Id}(A, b, a)$ we see that we only have to prove $\mathrm{Id}(A, b, b)$ which is easy just using $\mathrm{id}(b)$. The proof of transitivity can be understood in an analogous way.

A basic property we would expect from equality is that equal arguments can be substituted inside predicates and that equality is a congruence for every function definable in the system. Actually, the second property is easily derivable from the first but since both have very simple proofs using pattern matching we prove them independently :

```
respId ∈ (↓A, ↓B ∈ Set;
          ↓a, ↓a’ ∈ A;
          f ∈ (A) B;
          Id(A, a, a’)
         ) Id(B, f(a), f(a’))
   respId(A, B, _, a’, f, id(_, _)) ≡ id(B, f(a’))
substId ∈ (↓A ∈ Set; ↓a, ↓b ∈ A; Id(A, a, b); C ∈ (A) Set; C(a)) C(b)
   substId(A, _, b, id(_, _), C, h1) ≡ h1
```

Since we have already defined the set of natural numbers Nat we want to prove the axioms of arithmetic dealing with equality, i.e. that no successor is equal to 0 and that if the successors of two numbers are equal then already the numbers we started with are equal, i.e. that s is an injection. Again, we apply pattern matching :

```
injS ∈ (↓m, ↓n ∈ Nat; Id(Nat, s(m), s(n))) Id(Nat, m, n)
   injS(_, n, id(_, _)) ≡ id(Nat, n)
notId_0_S ∈ (↓i ∈ Nat; Id(Nat, 0, s(i))) False
```

The proof of injectivity uses the fact that from the constraint $\mathrm{s}(x) \equiv \mathrm{s}(y)$ we can derive the constraint $x \equiv y$. This follows from the principle that all expressions of type Nat can be built from the constructors. Similarly, we know that $0 \not\equiv \mathrm{s}(x)$ and hence we can generate an empty pattern in the proof of the no-confusion property[11].

Notation 3 *We denote* $\mathrm{Id}(A, x, y)$ *by* $x =_A y$ *or if the set* A *is clear from the context just* $x = y$.

Id can also be applied to higher-order sets like Fun, Π or Ord, However, this is not very useful since the *principle of extensionality*

$$(A, B \in \mathbf{Set}; f, g \in A \to B; (a \in A) f(a) = g(a)) f = g$$

is not provable. This is a shortcoming of *intensional Type Theory* as we use it[12].

3.4 Induction

The principle of induction is expressible in Type Theory since we allow sets and also families of sets (which represent predicates) as parameters. In fact induction can be proven using pattern matching:

[10]read a is definitionally equal to b see section 4 for the definition.

[11]There is a bug in the current implementation of ALF which makes it impossible to define empty patterns at top level. This can be avoided by using a case-expression:

```
notId_0_S ∈ (↓i ∈ Nat; Id(Nat, 0, s(i))) False
   notId_0_S(i, h) ≡ case h ∈ Id(Nat, 0, s(i)) of
                     end
```

[12]Martin Hofmann’s PhD thesis [Hof95a] investigates this problem and possible solutions.

```
indNat ∈ (P ∈ (Nat) Set;
           P(0);
           (n ∈ Nat; P(n)) P(s(n));
           n ∈ Nat
          ) P(n)
    indNat(P, h, h1, 0) ≡ h
    indNat(P, h, h1, s(h2)) ≡ h1(h2, indNat(P, h, h1, h2))
```

Computationally, the induction principle corresponds to the scheme of primitive recursion for dependent types. In conventional presentations of Type Theory this induction principle is the only way to define functions (or proofs) with domain Nat. Pattern matching arose as a generalisation of this scheme.

In the conventional approach we have to generate an induction principle for every type we introduce, e.g. for List and Fin

```
indList ∈ (A ∈ Set;
           P ∈ (List(A)) Set;
           P(nil(A));
           (a ∈ A; l ∈ List(A); P(l)) P(cons(A, a, l));
           l ∈ List(A)
          ) P(l)
    indList(A, P, h, h1, nil(_)) ≡ h
    indList(A, P, h, h1, cons(_, a, l1)) ≡ h1(a, l1, indList(A, P, h, h1, l1))

indFin ∈ (P ∈ (n ∈ Nat; Fin(n)) Set;
          (n ∈ Nat) P(s(n), 0_Fin(n));
          (n ∈ Nat; i ∈ Fin(n); P(n, i)) P(s(n), s_Fin(n, i));
          n ∈ Nat;
          i ∈ Fin(n)
         ) P(n, i)
    indFin(P, h, h1, _, 0_Fin(n1)) ≡ h(n1)
    indFin(P, h, h1, _, s_Fin(n1, h2)) ≡ h1(n1, h2, indFin(P, h, h1, n1, h2))
```

Usually, proofs by pattern matching can be translated into proofs using only the induction constants. However, in many examples these proofs are much more complicated than the original proofs. Another shortcoming is that they lead to very artificial programs using only primitive recursion. As an example consider the definition of max using pattern matching

```
max ∈ (Nat; Nat) Nat
    max(0, h1) ≡ h1
    max(s(h2), 0) ≡ s(h2)
    max(s(h2), s(h)) ≡ s(max(h2, h))
```

This can be translated into a program only using indNat using higher order functions:

```
max’ ∈ (Nat; Nat) Nat
    max’ ≡
        [h, h1]
            app(indNat(
                    [h2]Fun(Nat, Nat),
                    fun([h2]h2),
                    [n, h2]fun(indNat([h3]Nat, s(n), [n’, h3]s(app(h2, n’)))),
                    h),
                h1)
```

Obviously, the second version is much harder to read and it uses higher order functions.

However, pattern matching is in general not conservative over induction. A counterexample can be derived for the case of Id. Here the induction principle is the following:[13]

[13]The constant indId has also been called J or idpeel

$$
\begin{array}{l}
\mathrm{indId} \in (A \in \mathbf{Set};\\
\qquad a, b \in A;\\
\qquad P \in (a_1, b_1 \in A; \mathrm{Id}(A, a_1, b_1))\,\mathbf{Set};\\
\qquad (a_1 \in A)\, P(a_1, a_1, \mathrm{id}(A, a_1));\\
\qquad p \in \mathrm{Id}(A, a, b)\\
\qquad)\, P(a, b, p)\\
\quad \mathrm{indId}(A, \_, b, P, h, \mathrm{id}(\_, \_)) \equiv h(b)
\end{array}
$$

Using pattern matching we can derive the principle of uniqueness of identity proofs which is just the intuitive property that any two proofs of an identity (with the same type) have to be equal (because there is only one: id.)

$$
\begin{array}{l}
\mathrm{idUnique} \in (A \in \mathbf{Set};\ a, b \in A;\ p, q \in \mathrm{Id}(A, a, b))\ \mathrm{Id}(\mathrm{Id}(A, a, b), p, q)\\
\quad \mathrm{idUnique}(A, \_, b, \mathrm{id}(\_, \_), \mathrm{id}(\_, \_)) \equiv \mathrm{id}(\mathrm{Id}(A, b, b), \mathrm{id}(A, b))
\end{array}
$$

It is a bit surprising that any attempt to derive this principle from indId fails. It has been shown by Hofmann and Streicher [HS94] that idUnique is indeed independent from the conventional formulation of Type Theory[14]. It is interesting to note that concrete instances of idUnique like idUnique(Nat) are derivable. It has been conjectured that idUnique(Fun(Nat, Nat)) is not derivable.

However, this shortcoming can be easily fixed by adding a second induction principle to the theory:

$$
\begin{array}{l}
\mathrm{indId'} \in (A \in \mathbf{Set};\\
\qquad a \in A;\\
\qquad P \in (a_1 \in A; \mathrm{Id}(A, a_1, a_1))\,\mathbf{Set};\\
\qquad (a_1 \in A)\, P(a_1, \mathrm{id}(A, a_1));\\
\qquad p \in \mathrm{Id}(A, a, a)\\
\qquad)\, P(a, p)\\
\quad \mathrm{indId'}(A, a, P, h, \mathrm{id}(\_, \_)) \equiv h(a)
\end{array}
$$

We conjecture that pattern matching is conservative over the theory with both induction constants.

Exercise 7 *Derive an induction principle for* Vec.

Exercise 8 *Derive that* $\mathrm{add} \in (\mathrm{Nat}; \mathrm{Nat})\mathrm{Nat}$ *is commutative and associative using either* indNat *or pattern matching. Define* $\mathrm{mult} \in (\mathrm{Nat}; \mathrm{Nat})\mathrm{Nat}$ *and show that* $(\mathrm{add}, 0, \mathrm{mult}, 1)$ *is a ring.*

Exercise 9 *Show that* $\mathrm{indId}'(\mathrm{Nat})$ *can be derived from* indId *and* indNat.

3.5 Inductively defined relations

Often relations are defined inductively, i.e. as the least relation satisfying a certain property. In many cases (where the property is given by generalised Horn clauses) we can translate this into an inductively defined family. Here the constructors are the basic rules defining the type and the expressions correspond to derivation trees. The proofs which implicitly use induction over the structure of derivations are very often much simpler and shorter than conventional proofs using induction over natural numbers.

As an example consider the following definition of the relation $\leq$ on natural numbers:

$$
\begin{array}{l}
\mathrm{Le} \in (\mathrm{Nat};\ \mathrm{Nat})\ \mathbf{Set}\\
\quad \mathrm{le}_0 \in (n \in \mathrm{Nat})\ \mathrm{Le}(0, n)\\
\quad \mathrm{le}_s \in (m, n \in \mathrm{Nat};\ \mathrm{Le}(m, n))\ \mathrm{Le}(\mathrm{s}(m), \mathrm{s}(n))
\end{array}
$$

We want to show that Le is a preorder, i.e. reflexive and transitive. Reflexivity can be shown by a pattern matching proof recurring over Nat which corresponds to a normal induction. However, transitivity is best shown by pattern matching over the proof arguments, which corresponds to an induction over derivations:

[14] The proof uses a model of Type Theory where sets are interpreted as *groupoids*, i.e. categories where all arrows are isomorphisms.

```
transLe ∈ (i, j, k ∈ Nat; Le(i, j); Le(j, k)) Le(i, k)
    transLe(_, j, k, le_0(_), h_2) ≡ le_0(k)
    transLe(_, _, _, le_s(m, n, h_3), le_s(_, n_1, h)) ≡
        le_s(m, n_1, transLe(m, n, n_1, h_3, h))
```

It is interesting that two cases are sufficient. We see this as follows: the first proof argument is covered by the two possible cases $\mathrm{le}_0, \mathrm{le}_\mathrm{s}$. In the second cases the parameter j is equal to a successor and hence the second proof parameter cannot possibly be le_0. Hence the pattern is complete.

Exercise 10 *Define the type of binary trees* BT *as follows:*

```
BT ∈ Set
    leaf ∈ BT
    span ∈ (BT; BT) BT
```

Define a relation $\mathrm{LeBT} \in (\mathrm{BT};\mathrm{BT})\mathbf{Set}$ *such that* $\mathrm{LeBT}(b, b')$ *is provable iff* b *is a subtree of* b'. *Show that* LeBT *is a preorder (i.e. reflexive and transitive).*

4 ALF's Type Theory

In the last two sections Type Theory was introduced informally. In this section we shall have a closer look at the system.

4.1 Judgements of Type Theory

Our presentation of Type Theory is based on four basic judgements :

1. σ **is a type**.
2. $M \in \sigma$
   M is an expression of type σ.
3. $\sigma \equiv \tau$
   The types σ and τ are definitionally equal.
4. $M \equiv N \in \sigma$
   The expressions M, N of type σ are definitionally equal.

We are going to introduce a syntax for expressions and types together with the definition of the judgements.

4.2 Conventions

The following conventions are useful to deal with bound variables and substitutions. We use them informally, if we want to be fully precise it would be a good idea to use *nameless dummies* as introduced by de Bruijn [dB72].

Substitution By $M[x := N]$ or $\sigma[x := N]$ we denote the substitution of the free variable x by N. We only allow this operation if all the parameters and the result are well typed. By free variable we mean a variable which is not bound by $[x]-$ or by $(x \in -)-$.

α-alpha conversion Bound variables may be consistently renamed, i.e.

$$\begin{array}{rcll} [x]M & \equiv & [y](M[x := y]) & \text{if } y \text{ is not free in } M. \\ (x \in \sigma)\tau & \equiv & (y \in \sigma)(\tau[x := y]) & \text{if } y \text{ is not free in } \tau. \end{array}$$

4.3 Bureaucratic rules

- We require that $- \equiv -$ and $- \equiv - \in \sigma$ are equivalence relations (i.e. reflexive, symmetric and transitive).
- $- \equiv -$ and $- \equiv - \in \sigma$ are congruences with respect to all operations we introduce.
- We require that $M \in -$ and $M \equiv N \in -$ are closed under $\equiv$, that is $\dfrac{M \in \sigma \qquad \sigma \equiv \sigma'}{M \in \sigma'}$ $\dfrac{M \equiv M' \in \sigma \qquad \sigma \equiv \sigma'}{M \equiv M' \in \sigma'}$

4.4 Π-types

Π-formation $$\frac{\sigma \textbf{ is a type}. \quad \tau \textbf{ is a type} \text{ under the assumption } x \in \sigma}{(x \in \sigma)\tau \textbf{ is a type}.}$$

Π-introduction $$\frac{(x \in \sigma)\tau \textbf{ is a type}. \quad M \in \tau \text{ under the assumption } x \in \sigma}{[x]M \in (x \in \sigma)\tau}$$

Note that not all elements of Π-types are introduced by λ-abstraction, they can also be introduced by implicit or explicit constants.

Π-elimination $$\frac{M \in (x \in \sigma)\tau \qquad N \in \sigma}{M(N) \in \tau[x := N]}$$

Π-computation

$$(\beta) \quad \frac{[x]M \in (x : \sigma)\tau \qquad N \in \sigma}{([x]M)(N) \equiv M[x := N] \in \tau[x := N]}$$

$$(\eta) \quad \frac{M \in (x : \sigma)\tau \quad x \text{ is not free in } M.}{[x]M(x) \equiv M \in (x : \sigma)\tau}$$

Abbreviations

$$\begin{array}{rcl}
(x_1 \in \sigma_1; x_2 \in \sigma_2 \ldots; x_n \in \sigma_n)\tau & \equiv & (x_1 \in \sigma_1)(x_2 \in \sigma_2)\ldots(x_n \in \sigma_n)\tau \\
(\ldots; x \in \sigma; y \in \sigma \ldots)\tau & \equiv & (\ldots; x, y \in \sigma \ldots)\tau \\
[x_1, x_2, \ldots x_n]M & \equiv & [x1][x2]\ldots[x_n]M \\
M(N_1, N_2, \ldots, N_n) & \equiv & M(N_1)(N_2)\ldots(N_n) \\
(\sigma)\tau & \equiv & (x \in \sigma)\tau \\
 & & x \text{ not free in } \sigma
\end{array}$$

4.5 The type of sets

We only have two rules dealing with the type of all sets.

- **Set is a type**.
- $\dfrac{A \in \textbf{Set}}{A \textbf{ is a type}.}$

4.6 Inductively defined sets

We present a general scheme for inductively defined sets and allow the introduction of new elimination constants by pattern matching. This approach is based on the work by Peter Dybjer on schematic definitions of sets [Dyb91] and on the work by Thierry Coquand on pattern matching with dependent types [Coq92]. Thus our system differs from more traditional presentation of Type Theory which use a fixed collection of sets and constants. There are also presentation which we may call semi-traditional because they allow schematic definitions of sets but introduce a fixed elimination constant with every set.

We present the schematic rules by first presenting a general but inconsistent scheme and then stating some provisos which recover consistency.

First some notation: Every type has the form

$$(x_1 \in \sigma_1; \ldots; x_n \in \sigma_n)\tau$$

which we abbreviate by

$$(\vec{x} \in \vec{\sigma})\tau$$

Here $\vec{x} = x_1, x_2, \ldots, x_n$ is the sequence of variables and $\vec{\sigma} = \sigma_1, \sigma_2, \ldots, \sigma_n$ is the sequence of types. n may be 0, that is the sequence is empty.

We call a type *large* if $\tau = \mathbf{Set}$ and *small* otherwise.

4.6.1 Specification of sets

A collection of sets defined by mutual induction is given by :

1. A finite sequence of (free) names for the sets $A_1, A_2, \ldots, A_n$.
2. For every $1 \leq i \leq n$ an arity, that is a type of the form :

$$(\vec{x} \in \vec{\sigma}_i)(\vec{y} \in \vec{\tau}_i)\mathbf{Set}$$

such that all σ_{ji} are large and all τ_{ki} are small.

3. A finite sequence of (free) constructor names $c_1, c_2, \ldots, c_m$.
4. For every $1 \leq j \leq m$ a type of the form :

$$(\vec{x} \in \vec{\sigma}_i)(\vec{z} \in \vec{\rho}_j)A_i(\vec{M}_j)$$

This type is subject to the proviso of *positivity* which we define below.

We say that a set name A appears *strictly positive* in a type ρ, iff

1. A does not appear in ρ.
2. ρ has the form

$$(\vec{x} \in \vec{\sigma})A(\vec{M})$$

and A does not appear at all in $\vec{\sigma}$.

We require that all A_i have to appear only strictly positive in ρ_{ij}. This gives us the following typing judgements:

A-formation For every $1 \leq i \leq n$:

$$A_i \in (\vec{x} \in \vec{\sigma}_i)(\vec{y} \in \vec{\tau}_i)\mathbf{Set}$$

A-introduction For $1 \leq j \leq m$

$$c_j \in (\vec{x} \in \vec{\sigma}_i)(\vec{y} \in \vec{\rho}_j)A_i(\vec{M}_j)$$

4.6.2 Pattern matching

We first have to introduce some preliminaries.

We define *constructor expressions*:

1. A variable x is a constructor expression.
2. If all $M_i \in \vec{M}$ are constructor expressions and c is a constructor, then $c(\vec{M})$ is a constructor expression.

We say that a expression M is an instance of M' iff

$$M' \equiv M[\vec{x} := \vec{N}]$$

where all N_i are constructor expressions.

We define the relations *structurally smaller* $M \leq N$ and *strictly structurally smaller* $M < N$ by the following rules:

1. $x(\vec{M}) \leq x$.
2. If $M_i \leq N_i$ then $c(\vec{M}) \leq c(\vec{N})$.
3. If $M < N$ then $M \leq N$.
4. If $M \leq N_i$ then $M < c(\vec{N})$.

where c is a constructor.

We consider extensions of the subterm ordering on tuples with respect to a total ordering r of a subsequence of indices $i_1 < i_2 < \ldots < i_n$. We say that $\vec{M} <_r \vec{N}$ if $M_i < N_i$ and $M_k \equiv N_k$ for all $k < i$.

The specification of a collection of mutually inductive defined implicit constants is given by

1. A sequence of (free) names $d_1, \ldots, d_n$.
2. For every $1 \leq i \leq n$ a type
$$(\vec{x} \in \vec{\sigma}_i)\tau$$
3. A system of (well typed) equations of the form
$$d_i(\vec{N}) \equiv M \in \rho$$
where $\vec{N}$ are constructor expressions and the free variables of M are contained in the free variables of $\vec{N}$ and which are subject to the provisos stated below.

Provisos:

Completeness For every tuple of closed constructor expressions $\vec{L} \in \vec{\sigma}_i$ there is one and only one equation $d_i(\vec{N}) \equiv -$ such that $\vec{L}$ is an instance of $\vec{N}$.

Structural recursion For every $1 \leq i \leq n$ there is an ordering of the argument indices r such that when $d_i(\vec{N}) \equiv M$, where $\vec{N}$ is a constructor expression, is derivable from the equations and $d(\vec{N'})$ appears in M then $\vec{N'} <_r \vec{N}$.

Given the provisos hold we introduce the rules:

***d*-Elimination** For every $1 \leq i \leq n$:

$$d_i \in (\vec{x} \in \vec{\sigma}_i)\tau$$

***d*-Elimination** For all equations :

$$d_i(\vec{N}) \equiv M \in \rho$$

4.6.3 Case expressions

ALF also allows a limited form of case-expressions which correspond to local elimination constants. The syntax is

$$\text{case } M \in \sigma \text{ of } \begin{array}{l} N_1 \Rightarrow M_1 \\ \dots \\ N_n \Rightarrow M_n \end{array}$$

We view case expressions just as a shorthand for the introduction of a new implicit constant.

4.6.4 Universes

We allow the mutual definition of new set formers and implicit constants (as suggested in [Dyb92]) it becomes possible to define internal universes, e.g.

```
T ∈ (U) Set
    T(nat) ≡ Nat
    T(pi(a, b)) ≡ Π(T(a), [h]T(b(h)))
U ∈ Set
    nat ∈ U
    pi ∈ (a ∈ U; b ∈ (T(a)) U) U
```

Here U contains names for types and T is a function which assigns meaning to names. When we want to introduce Π-types it is necessary that U and T are defined at the same time. Note that both U and T fullfill the provisos stated earlier.

4.7 Properties

We claim (without proof) that the following properties hold:

Proposition 1 *The reduction relation we obtain by directing the equations is (strongly) normalising for all types σ and all $M \in \sigma$.*

Corollary 2 *The properties*

- σ **is a type**,
- $M \in \sigma$,
- *For types σ, τ whether $\sigma \equiv \tau$,*
- *For $M, N \in \sigma$ whether $M \equiv N \in \sigma$*

are decidable.

5 Internal verification

In this section we will consider some very simple examples of program verification. In a conventional approach to verification verifying a program would mean to proof a property of a program. We call this *external verification.* In Type Theory where we identify sets and propositions it is natural to consider program and correctness proof as a unit. The relevant properties of the program are here expressed by its type, which plays the role of a specification. We call this approach *internal verification.*

What is the role verification can play in the software development process? We believe that is highly unlikely that the development process can start with a formal and detailed specification of the problem. In practice we often start with an informal specification and a prototype. During the development process we strive towards a better program with a more detailed, possibly formal, specification. We try to reflect this by the way we present the examples, often we will start with a prototypic program and a vague idea what the program is supposed to do. We reimplement the program such that the specification is expressed by the type and the agreement between program and specification can be checked by a type checker.

5.1 Decidability of Le

We have already studied the relation Le in section 3.5. So far, we have considered Le from a declarative point of view, but it is easy to see that Le is decidable, i.e. we can write a functional program with boolean results :

```
Bool ∈ Set
    true ∈ Bool
    false ∈ Bool

le ∈ (Nat; Nat) Bool
    le(0, h1) ≡ true
    le(s(h2), 0) ≡ false
    le(s(h2), s(h)) ≡ le(h2, h)
```

Externally, the property of le we are interested in is that le(m, n) returns true if and only iff Le(m, n) holds. From this it follows that le returns false iff Le$(m, n) \to$ False holds

This external specification of the correctness of le can be expressed as follows :

```
LeSpec ∈ ((Nat; Nat) Bool) Set
    leSpec ∈ (le ∈ (Nat; Nat) Bool;
              (i, j ∈ Nat; Le(i, j)) Id(le(i, j), true);
              (i, j ∈ Nat; Id(le(i, j), true)) Le(i, j)
             ) LeSpec(le)
```

The external verification has the result that LeSpec(le) holds:

```
leLem1 ∈ (i, j ∈ Nat; Le(i, j)) Id(le(i, j), true)
    leLem1(_, j, le0(_)) ≡ id
    leLem1(_, _, leS(m, n, h1)) ≡ leLem1(m, n, h1)
true_neq_false ∈ (Id(true, false)) False
    true_neq_false(h) ≡ case h ∈ Id(true, false) of
                        end
leLem2 ∈ (i, j ∈ Nat; Id(le(i, j), true)) Le(i, j)
    leLem2(0, j, h) ≡ le0(j)
    leLem2(s(h1), 0, h) ≡ bot(true_neq_false(symId(h)))
    leLem2(s(h1), s(h2), h) ≡ leS(h1, h2, leLem2(h1, h2, h))
leOk ∈ LeSpec(le)
    leOk ≡ leSpec(le, leLem1, leLem2)
```

How would an internal verification of le look like? We can express the type of functions which decide Le directly. First we introduce the connective Dec $\in$ $(\mathbf{Set})\mathbf{Set}$. Dec$(P)$ means that we have either an element of P or we can show that P is empty, i.e. P is *decidable*. In conventional notation Dec$(P) = P \vee \neg P$. We may view Dec as a refinement of Bool.

```
Dec ∈ (Set) Set
    yes ∈ (P) Dec(P)
    no ∈ ((P) False) Dec(P)
```

Our goal is now to refine le to a function le$' \in (m, n \in$ Nat)Dec(Le(m, n). That is, the result of le$'$ is not just a boolean value, but it is a *proof* of either Le(m, n) or of (Le(m, n))False. The correctness of this function is expressed by its type, no further external verification is necessary.

```
lem1 ∈ (m, n ∈ Nat; Le(s(m), s(n))) Le(m, n)
    lem1(m, n, leS(_, _, h1)) ≡ h1
lem2 ∈ (n ∈ Nat; Le(s(n), 0)) False
    lem2(n, h) ≡ case h ∈ Le(s(n), 0) of
                 end
le' ∈ (m, n ∈ Nat) Dec(Le(m, n))
    le'(0, n) ≡ yes(le0(n))
    le'(s(h), 0) ≡ no(lem2(h))
    le'(s(h), s(h1)) ≡ case le'(h, h1) ∈ Dec(Le(h, h1)) of
                            yes(h2) ⇒ yes(leS(h, h1, h2))
                            no(h2) ⇒ no([h3]h2(lem1(h, h1, h3)))
                       end
```

If we compare le with le$'$ we note that le$'$ has essentially the same structure as le. true is replaced by yes$(-)$ and false by no$(-)$. The argument to yes or no represents a witness for the choice.

However, there is one point where le$'$ diverges from le, in the last case le$'$ does a case analysis on the recursive result whereas le just returns the recursive result. This case analysis preserves the decision yes or no but changes the witness.

In the case of $\mathrm{Dec}(\mathrm{Le}(m,n))$ the witnesses have no computational relevance. This is not always the case as we will see later. In the case of le the case analysis represents an unnecessary computation, which could be optimised once we inform the compiler that we are not interested in the witnesses. That is the existence of the witnesses is important at compile time (to see the correctness) but irrelevant at run time.

Exercise 11 *Show the decidability of* LeBT *from exercise 10, i.e. find an inhabitant of* $(b, b' \in \mathrm{BT})\mathrm{LeBT}(b, b')$.

Exercise 12 *Define a predicate* $\mathrm{Seg} \in (A \in \mathbf{Set}; List(A); List(A))\mathbf{Set}$ *such that* $\mathrm{Seg}(l, l')$ *is inhabited iff* l *is a segment (i.e. a consecutive sublist) of* l'. *Show that* Seg *is decidable, if the equality on* A *is decidable.*

Hint: *Define an auxiliary predicate* $\mathrm{ISeg} \in (A \in \mathbf{Set}; List(A); List(A))\mathbf{Set}$ *such that* $\mathrm{ISeg}(l, l')$ *holds iff* l *is a sublist of* l' *and establish decidability of* ISeg *first.*

5.2 Division

As a second simple example we consider division of natural numbers. That is given two natural numbers m, n we want to calculate the pair of quotient q and remainder r such that $qn + r = m$ and $r < n$. To avoid the exceptional case that $n = 0$ we solve the problem for $m, n+1$. A prototypical implementation just implements counting modulo n:

```
div ∈ (Nat; Nat) And(Nat, Nat)
    div(0, h1) ≡ and(0, 0)
    div(s(h2), h1) ≡ case div(h2, h1) ∈ And(Nat, Nat) of
                          and(h, h3) ⇒ case le(h1, s(h3)) ∈ Bool of
                                            true ⇒ and(h, s(h3))
                                            false ⇒ and(s(h), h3)
                                        end
                      end
```

A specification of the algorithm is that for $m, n+1$ we look for a pair of numbers q, r such that $q(n+1) + r = m$ and $r \leq n$. We specify this in ALF by using an inductive type with a single constructor which can be considered as a named version of the Σ-type:

```
DivSpec ∈ (m, n ∈ Nat) Set
    divSpec ∈ (q, r ∈ Nat; Id(m, add(r, mult(q, s(n)))); Le(r, n)) DivSpec(m, s(n))
```

Our goal is now to define a function with the type $(m, n \in \mathrm{Nat})\mathrm{DivSpec}(m, \mathrm{s}(n))$. The structure of this construction will follow the algorithm div.

When we try to construct a solution we realize that it is not decidability of Le we need but trichotomy. The idea is that we derive $r = n$ from $r \leq n$ and $n \leq r$.

We show trichotomy for the alternative version of Le which introduces Le and Lt by a mutual inductive definition.

```
Le ∈ (Nat; Nat) Set
    rLe ∈ (n ∈ Nat) Le(n, n)
    lt2le ∈ (Lt(m, n)) Le(m, n)
Lt ∈ (Nat; Nat) Set
    ltS ∈ (Le(m, n)) Lt(m, s(n))
```

```
trichLt ∈ (m, n ∈ Nat) Or(Lt(m, n), Le(n, m))
    trichLt(m, 0) ≡ in_2(le0(m))
    trichLt(0, s(h)) ≡ in_1(ltS(le0(h)))
    trichLt(s(h_1), s(h)) ≡ case trichLt(h_1, h) ∈ Or(Lt(h_1, h), Le(h, h_1)) of
                                 in_1(h_2) ⇒ in_1(ltS'(h_2))
                                 in_2(h_2) ⇒ in_2(leS(h_2))
                             end
```

Another lemma shows that $r \leq n$ and $n \leq r$ implies $r = n$ which again needs that $n \not< n$.

```
lem1 ∈ (n ∈ Nat; Lt(n, n)) False
    lem1(n, h) ≡ accNonRefl(Nat, Lt, n, accLt(n), h)
lem2 ∈ (n, r ∈ Nat; Le(n, r); Le(r, n)) Id(Nat, n, r)
    lem2(_, r, rLe(_), rLe(_)) ≡ id(Nat, r)
    lem2(_, r, rLe(_), lt2le(_, _, h)) ≡ bot(Id(Nat, r, r), lem1(r, h))
    lem2(n, r, lt2le(_, _, h_2), h_1) ≡ bot(Id(Nat, n, r), lem1(n, transLtLe(n, r, n, h_2, h_1)))
```

Using these components it is easy to derive div′. We need simple arithmetic reasoning which use some fundamental properties of equality discussed earlier.

```
div' ∈ (m, n ∈ Nat) DivSpec(m, s(n))
    div'(0, n) ≡ divSpec(0, 0, id, le0(n))
    div'(s(h), n) ≡
        case div'(h, n) ∈ DivSpec(h, s(n)) of
            divSpec(q, r, h_1, h_2) ⇒
                case trichLt(n, s(r)) ∈ Or(Lt(n, s(r)), Le(s(r), n)) of
                    in_1(ltS(h_4)) ⇒
                        divSpec(s(q),
                                0,
                                idS(transId(h_1,
                                            respId([h_3]add(h_3, mult(q, s(n)))),
                                                   lem2(r, n, h_2, h_4)))),
                                le0(n))
                    in_2(h_3) ⇒ divSpec(q, s(r), respId(s, h_1), h_3)
                end
        end
```

6 Example: insertion sort

Sorting programs are a well understood class of algorithms. We shall consider here one of the simplest sorting algorithms – insertion sort – and show how to derive an internally verified version of insertion sort. Here is a prototype of insertion sort:

```
insert ∈ (r ∈ (A; A) Bool; a ∈ A; as ∈ List(A)) List(A)
    insert(r, a, nil) ≡ cons(a, nil)
    insert(r, a, cons(a_1, l)) ≡ case r(a, a_1) ∈ Bool of
                                      false ⇒ cons(a_1, insert(r, a, l))
                                      true ⇒ cons(a, cons(a_1, l))
                                  end
sort ∈ (r ∈ (A; A) Bool; as ∈ List(A)) List(A)
    sort(r, nil) ≡ nil
    sort(r, cons(a, l)) ≡ insert(r, a, sort(r, l))
```

We can now use sort(le) ∈ (List(Nat))List(Nat) to sort lists of natural numbers in ascending order. Note that sort and insert are structurally recursive. The more efficient sorting algorithms like *merge sort* and *quick sort* do not have this property. We shall present a solution to the problem in the next section.

Here is an informal specification of the sorting problem: Given a list $l \in \mathrm{List}(A)$ and a relation $R \in (A; A)\mathbf{Set}$ we are looking for a new list $l' \in \mathrm{List}(A)$ which *is* *sorted wrt* R and which is a *permutation* of l.

It is possible to diverge already here: we have seen specifications of sorting where it is sufficient that the result list has the same members as l. Hence deletion and copying would be allowed.

To specify sorting more formally we have to decide what we mean by *sorted wrt* R and we have to specify *permutation.* Moreover we have to analyse which properties of R we need. A first guess would be decidability but it turns out that this is not the case.

There are a number of possible choices at this point, resulting in different verifications. This illustrates our point that a good specification is the result of the development process and not its starting point.

6.1 Specifying Sorted

There seem at least two possibilities to specify Sorted:

1. A list is sorted iff every element is R-related to all subsequent elements.
2. A list is sorted if all consecutive elements are R-related.

The first alternative requires R to be transitive (otherwise we can't sort), whereas the second one doesn't. The standard sorting algorithms seem to work when R is not transitive, hence 2. seems to be a better choice[15].

We define Sorted by two inductive relations:

```
Sorted ∈ ((A; A) Set; List(A)) Set
    sorted_nil ∈ Sorted(R, nil)
    sorted_cons ∈ (Sorted(R, bs);
                    SmallerList(R, a, bs)
                   ) Sorted(R, cons(a, bs))
SmallerList ∈ ((A; A) Set; a ∈ A; List(A)) Set
    sml_nil ∈ (a ∈ A) SmallerList(R, a, nil)
    sml_cons ∈ (R(a, b)) SmallerList(R, a, cons(b, bs))
```

6.2 Specifying Perm

A simple way to define that one list is a permutation of another is to say that the counts of all elements are the same. I.e. if we had a function count $\in (a \in A; l \in \mathrm{List}(A))\mathrm{Nat}$ which counts the number of occurrences of a in l we can define $\mathrm{Perm}(l, l')$ by $(a \in A)\mathrm{count}(a, l) = \mathrm{count}(a, l')$. However to define count we need that $- = -$ is decidable which is not the case in general. Moreover we could imagine that we want to sort objects whose equality is not decidable — an example would be to sort functions of type (Nat)Nat w.r.t. their value at 0.

Exercise 13 *Why is the following proposal not a solution? We define a relation* $\mathrm{Count} \in (A \in \mathbf{Set}; A; \mathrm{List}(A); \mathrm{Nat})\mathbf{Set}$ *inductively:*

```
Count ∈ (a ∈ A; l ∈ List(A); n ∈ Nat) Set
    count_nil ∈ (a ∈ A) Count(a, nil, 0)
    count_s ∈ (Count(a, xs, n)) Count(a, cons(a, xs), s(n))
    count_same ∈ ((Id(a, b)) False;
                   Count(a, xs, n)
                  ) Count(a, cons(b, xs), s(n))
```

Now we can define $\mathrm{Perm}_{\mathrm{count}}$

```
Perm_count ∈ (List(A); List(A)) Set
    perm_count ∈ ((a ∈ A; n ∈ Nat) Iff(Count(a, xs, n), Count(a, ys, n))
                 ) Perm_count(xs, ys)
```

[15]We have used 1. in the past. Sometimes it is easier to verify a program than to come up with a good specification.

We choose the following definition of Perm which avoids this problem: First we define Adjoin $\in (A; \text{List}(A); \textit{List}(A))\mathbf{Set}$ which means that $\text{Adjoin}(a, xs, axs)$ holds iff axs is obtained by inserting a in xs at an arbitrary position:

```
Adjoin ∈ (A; List(A); List(A)) Set
    ad0 ∈ (a ∈ A; xs ∈ List(A)) Adjoin(a, xs, cons(a, xs))
    ad1 ∈ (Adjoin(a, xs, axs)) Adjoin(a, cons(b, xs), cons(b, axs))
```

Now it is easy to define Perm by subsequent Adjoins:

```
Perm ∈ (List(A); List(A)) Set
    perm0 ∈ Perm(nil, nil)
    perm1 ∈ (Perm(xs, ys);
               Adjoin(a, xs, axs);
               Adjoin(a, ys, ays)
              ) Perm(axs, ays)
```

It is easy to show that Perm is reflexive and symmetric but the proof of transitivity is quite hard[16]. Another difficult property is to verify the inverse of perm_1:

$$(\text{Perm}(xs, ys); \textit{Adjoin}(a, xs', xs); \textit{Adjoin}(a, ys', ys))\text{Perm}(xs', ys')$$

which is useful when showing that Perm is decidable if $- = -$ is decidable.

Exercise 14

1. *Show that* Perm *is reflexive, symmetric (easy) and transitive (hard).*
2. *Show that* Perm *is decidable if the equality for the set A is decidable (hard).*
3. *Find a better definition of* Perm *which makes it easier to verify the properties above.*

6.3 The type of sorting programs

The type of solutions to a sorting problem can be specified as a family of sets indexed over lists:

```
SortSpec ∈ (R ∈ (A; A) Set; l ∈ List(A)) Set
    sortSpec ∈ (l' ∈ List(A); Perm(l, l'); Sorted(R, l')) SortSpec(R, l)
```

Again, this can be viewed as a named instance of a Σ-type.

However, we will hardly find an inhabitant of the type

$$(R \in (A; A)\mathbf{Set}; l \in \text{List}(A))\text{SortSpec}(R, l)$$

because we know nothing about the relation R (in the worst case it could be always false which makes it impossible to sort any list with more than one element).

Two sort a two element list $[a, b]$ we need that either $R(a, b)$ or $R(b, a)$ holds. We call this property *connected* and define:

```
Con ∈ (R ∈ (A; A) Set) Set
    con ∈ ((a, a' ∈ A) Or(R(a, a'), R(a', a))) Con(R)
```

We can use proofs of connectedness to decide how to to order the elements, i.e. this proof takes over the role of the parameter le in the prototype. Moreover, we don't need decidability of R at all. Indeed, it seems that all sorting programs can be refined to inhabitants of

$$(R \in (A; A)\mathbf{Set}; r \in \text{Con}(R); l \in \text{List}(A))\text{SortSpec}(R, l)$$

[16]Luckily, transitivity is not needed for *insertion sort*. However, I used it in the verification of *quick sort* and *merge sort*.

6.4 Insertion sort verified

Following the structure of the prototype we can now develop insertion sort. As a first step we need to give a specification of insert. Clearly insert adjoins an element to a sorted list thereby preserving sortedness. We define

```
InsSpec ∈ (R ∈ (A; A) Set; a ∈ A; l ∈ List(A)) Set
    insSpec ∈ (l' ∈ List(A);
                Adjoin(a, l, l');
                Sorted(R, l')
               ) InsSpec(R, a, l)
```

We can refine insert to a function of type

$$(\mathrm{Con}(R); a \in A; l \in \mathrm{List}(A); \mathrm{Sorted}(l))\mathrm{InsSpec}(R, a, l)$$

We could explain the verified version of insert by reverse engineering, i.e. by rediscovering a verbal proof form the formal one. However, this is not how it was derived, i.e. by an interaction with the ALF system.

```
insert ∈ (r ∈ Con(R); a ∈ A; l ∈ List(A); Sorted(R, l)) InsSpec(R, a, l)
    insert(r, a, nil, h) ≡
        insSpec(cons(a, nil), ad0(a, nil), sorted_cons(h, sml_nil(a)))
    insert(con($h_1$), a, cons($a_1$, $l_1$), sorted_cons($h_2$, $h_3$)) ≡
        case $h_1$(a, $a_1$) ∈ Or(R(a, $a_1$), R($a_1$, a)) of
            in$_1$(h) ⇒
                insSpec(cons(a, cons($a_1$, $l_1$)),
                        ad0(a, cons($a_1$, $l_1$)),
                        sorted_cons(sorted_cons($h_2$, $h_3$), sml_cons(h)))
            in$_2$(h) ⇒
                case insert(con($h_1$), a, $l_1$, $h_2$) ∈ InsSpec(R, a, $l_1$) of
                    insSpec(_, ad0(_, _), $h_5$) ⇒
                        insSpec(cons($a_1$, cons(a, $l_1$)),
                                ad1(ad0(a, $l_1$)),
                                sorted_cons($h_5$, sml_cons(h)))
                    insSpec(_, ad1($h_6$), $h_5$) ⇒
                        case $h_3$ ∈ SmallerList(R, $a_1$, cons(b, xs)) of
                            sml_cons($h_4$) ⇒
                                insSpec(cons($a_1$, cons(b, axs)),
                                        ad1(ad1($h_6$)),
                                        sorted_cons($h_5$, sml_cons($h_4$)))
                        end
                end
        end
```

Having done the main job of verifying insert, sort is now relatively easy:

```
sort ∈ (r ∈ Con(R); l ∈ List(A)) SortSpec(R, l)
    sort(r, nil) ≡ sortSpec(nil, perm0, sorted_nil)
    sort(r, cons(a, $l_1$)) ≡
        case sort(r, $l_1$) ∈ SortSpec(R, $l_1$) of
            sortSpec(l', h, $h_1$) ⇒ case insert(r, a, l', $h_1$) ∈ InsSpec(R, a, l') of
                                       insSpec(l'$_1$, $h_2$, $h_3$) ⇒ sortSpec(l'$_1$, perm1(h, ad0(a, $l_1$), $h_2$),    )
                                     end
        end
```

7 General recursion

We only consider programs which are structurally recursive. This is already quite a powerful class and goes beyond the usual scheme of primitive recursion. An example is the Ackermann function which is not primitive recursive but structurally recursive:

```
ack ∈ (Nat; Nat) Nat
    ack(0, $h_1$) ≡ s(0)
    ack(s(0), 0) ≡ s(s(0))
    ack(s(s(h)), 0) ≡ s(s(s(s(h))))
    ack(s($h_2$), s(h)) ≡ ack(ack($h_2$, s(h)), h)
```

This is an instance where we have to use a lexicographic extension of the structural ordering, i.e. the second argument has a higher weight than the first. Using higher order datatypes we can implement even faster growing functions, i.e. we have already seen that ALF's Type Theory is stronger than arithmetic.

However, there are many programs which don't fit into the scheme of structural recursion. We have already mentioned *merge sort* and *quick sort*.

7.1 Euclid's algorithm

As a very simple example we consider Euclid's algorithm for the calculation of the greatest common divisor.

```
sub ∈ (Nat; Nat) Nat
    sub(h, 0) ≡ h
    sub(0, s(h2)) ≡ 0
    sub(s(h1), s(h2)) ≡ sub(h1, h2)
gcd ∈ (Nat; Nat) Nat
    gcd(0, h1) ≡ h1
    gcd(s(h2), 0) ≡ s(h2)
    gcd(s(h2), s(h)) ≡ case le(h2, h) ∈ Bool of
                           true ⇒ gcd(sub(h2, h), s(h))
                           false ⇒ gcd(s(h2), sub(h, h2))
                       end
```

This function is not structural recursive in either argument, because it uses subtraction when calling itself recursively. Obviously, the size of the arguments gets always reduced and gcd is terminating but it is not covered by the simple scheme of structural recursion.

We can invent more elaborate recursion schemes to cover examples like gcd. However we will go another way and show that we can always reduce terminating recursion to structural recursion.

In the case of gcd we can do this by introducing a new argument: i.e. we define $\mathrm{gcd}_1(m,n,x) = \mathrm{gcd}(m,n)$ if $x > m{+}n$. We can define the new function by structural recursion. To make this explicit we introduce a fourth argument which is the proof that $x > m+n$, hence gcd_2 has the type $(m,n,x \in \mathrm{Nat}; \mathrm{Lt}(\mathrm{add}(m,n),x))\mathrm{Nat}$. gcd_2 can be defined with the same clauses as gcd but is structural recursive:

```
transLeLt ∈ (Le(i, j); Lt(j, k)) Lt(i, k)
    transLeLt(rLe(_), h1) ≡ h1
    transLeLt(lt2le(h2), h1) ≡ transLt(h2, h1)

leAdd ∈ (x, y, y' ∈ Nat; Le(y, y')) Le(add(x, y), add(x, y'))
    leAdd(0, y, y', h) ≡ h
    leAdd(s(h1), y, y', h) ≡ leS(leAdd(h1, y, y', h))
leAdd' ∈ (x, x', y ∈ Nat; Le(x, x')) Le(add(x, y), add(x', y))
    leAdd'(_, x', y, rLe(_)) ≡ rLe(add(x', y))
    leAdd'(x, _, y, lt2le(ltS(h))) ≡ lt2le(ltS(leAdd'(x, n, y, h)))

sub ∈ (m, n ∈ Nat) Σ(Nat, [h]Le(h, m))
    sub(m, 0) ≡ pair(m, rLe(m))
    sub(0, s(h)) ≡ pair(0, le0(0))
    sub(s(h1), s(h)) ≡ case sub(h1, h) ∈ Σ(Nat, [h']Le(h', h1)) of
                           pair(a, b) ⇒ pair(a, lt2le(ltS(b)))
                       end
```

```
gcd_2 ∈ (m, n, x ∈ Nat; Lt(add(m, n), x)) Nat
    gcd_2(0, n, s(h_1), h) ≡ n
    gcd_2(s(h_2), 0, s(h_1), h) ≡ s(h_2)
    gcd_2(s(h_2), s(h_3), s(h_1), h) ≡
        case le(h_2, h_3) ∈ Bool of
            false ⇒
                case sub(h_3, h_2) ∈ Σ(Nat, [h']Le(h', h_3)) of
                    pair(a_1, b_1) ⇒
                        gcd_2(s(h_2),
                              a_1,
                              h_1,
                              transLeLt(leAdd(s(h_2), a_1, h_3, b_1),
                                        ltS'_inv(substId(adds_lem(s(h_2), h_3),
                                                         [h_4]Lt(h_4, s(h_1)),
                                                         h))))
                end
            true ⇒
                case sub(h_2, h_3) ∈ Σ(Nat, [h']Le(h', h_2)) of
                    pair(a, b) ⇒
                        gcd_2(a,
                              s(h_3),
                              h_1,
                              transLeLt(leAdd'(a, h_2, s(h_3), b), ltS'_inv(h)))
                end
        end
```

Exercise 15 *Extend* gcd_2 *to a verified version which shows that the function indeed calculates the* greatest common divisor.

7.2 Well-founded recursion

The idea of presenting general recursion in Type Theory is to introduce additional arguments which are structurally reduced. In this section we consider a general construction[17]. which shows that we can represent all programs this way which can be shown to terminate using well-founded recursion. The additional argument which is structurally reduced is the proof of well-foundedness.

We start with a general fixpoint combinator which can be defined using general recursion:

```
fix ∈ (A, B ∈ Set; (A; (A)B) B; A) B
    fix(A, B, h, h_1) ≡ h(h_1, fix(A, B, h))
```

We can argue as follows: whenever $\mathrm{fix}(f, x)$ terminates there must be a well founded tree of recursive calls of f. This tree gives us an well founded ordering such that f is only called at smaller arguments recursively.

We can encode this internally. I.e. given a relation $R \in (A; A)\mathbf{Set}$ we define the well-founded domain of R: $\mathrm{Acc}(R) \in (A)\mathbf{Set}$, i.e. the set of $x \in A$ such that all trees which start with x and have edges in R have finite depth:

```
Acc ∈ ((A; A) Set; A) Set
    acc ∈ (a ∈ A; (b ∈ A; R(b, a)) Acc(R, b)) Acc(R, a)
```

This is a higher order inductive definition. The idea is that if for a given$x$ all y such that $R(y, x)$ are already in $\mathrm{Acc}(R)$ then also x is in $\mathrm{Acc}(R)$. This can be used directly for a definition of $\mathrm{Acc}(R)$. Note that all$r$-normal forms, i.e. all x such that no $R(y, x)$ exists are in $\mathrm{Acc}(R)$ because the precondition is vacuously true.

We can now implement a refined version of fix which realizes general well-founded recursion via structural recursion over the proof of $\mathrm{Acc}(R)$

[17]Due to Paulson [Pau86] and Nordström [Nor88]

```
fix' ∈ (A, B ∈ Set;
        R ∈ (A; A) Set;
        (a ∈ A; (b∈ A; R(b, a)) B) B;
        a ∈ A;
        Acc(R, a)
       ) B
    fix'(A, B, R, h, a, acc(_, h2)) ≡ h(a, [b, h1]fix'(A, B, R, h, b, h2(b, h1)))
```

A source for well-founded recursion are the internal versions of the structural orderings on inductively defined types. In the case of Nat this is Lt. Here is the proof that Acc(Lt).

```
accLt_aux ∈ (i, j ∈ Nat; Acc(Lt, i); Lt(j, s(i))) Acc(Lt, j)
    accLt_aux(i, _, h, ltS(rLe(_))) ≡ h
    accLt_aux(i, j, acc(_, h2), ltS(lt2le(h1))) ≡ h2(j, h1)
accLt ∈ (n ∈ Nat) Acc(Lt, n)
    accLt(0) ≡ acc(0, [b, h]bot(not_lt_0(h)))
    accLt(s(h)) ≡ acc(s(h), [b, h1]accLt_aux(h, b, accLt(h), h1))
```

Exercise 16 *Define* LtBT $\in$ (BT; BT)**Set** *such that* LtBT(b, b') *is provable iff* b *is a proper subtree of* b' *(i.e.* $b \neq b'$*). Show that* LtBT *is well-founded, i.e. that* $(b \in \mathrm{BT})\mathrm{Acc}(\mathrm{LtBT}, b)$ *is inhabited.*

Note: LeBT *(exercise 10) and* LtBT *can be defined by one mutually inductive definition.*

Exercise 17 *Given relations* $R \in (A; A)\mathbf{Set}, S \in (B; B)Set$ *we can define their lexical product by* $RS \in (A \times B; A \times B)$ *by* $RS((a, b), (a', b')) \iff R(a, a') \vee a = a' \wedge S(b, b')$*. Derive that* $(a \in A)\mathrm{Acc}(R, a)$ *and* $(b \in B)\mathrm{Acc}(R, b)$ *entails* $(ab \in A \times B)Acc(RS, ab)$*.*

7.3 Merge sort

We shall sketch how the ideas of the previous section can be applied to implement *merge sort* using only structural recursion.

Below is the prototype for merge sort:

```
split ∈ (A ∈ Set; List(A)) And(List(A), List(A))
    split(A, nil) ≡ and(nil, nil)
    split(A, cons(a, nil)) ≡ and(cons(a, nil), nil)
    split(A, cons(a, cons(a1, l1))) ≡
        case split(A, l1) ∈ And(List(A), List(A)) of
            and(h, h1) ⇒ and(cons(a, h), cons(a1, h1))
        end
merge ∈ (A ∈ Set; le ∈ (A; A) Bool; List(A); List(A)) List(A)
    merge(A, le, nil, h1) ≡ h1
    merge(A, le, cons(a, l), nil) ≡ cons(a, l)
    merge(A, le, cons(a, l), cons(a1, l1)) ≡
        case le(a, a1) ∈ Bool of
            false ⇒ cons(a1, merge(A, le, cons(a, l), l1))
            true ⇒ cons(a, merge(A, le, l, cons(a1, l1)))
        end
sort ∈ (A ∈ Set; le ∈ (A; A) Bool; List(A)) List(A)
    sort(A, le, nil) ≡ nil
    sort(A, le, cons(a, nil)) ≡ cons(a, nil)
    sort(A, le, cons(a, cons(a1, l1))) ≡
        case split(A, l1) ∈ And(List(A), List(A)) of
            and(h, h1) ⇒ merge(A, le, sort(A, le, h), sort(A, le, h1))
        end
```

We realize that sort is not structurally recursive because it calls itself recursively on the result of split. However, we can see that split produces always shorter lists if the length of the input is at least 2 and hence we should be able to use the results of the previous section (fix$'$,accLt) to implement sort only using structural recursion.

We partially specify split, i.e. we only express the fact that it will produce shorter lists if the input is not of the form nil or cons(a, nil) :

```
SplitSpec ∈ (List($A$)) Set
    splitSpecNil ∈ SplitSpec(nil)
    splitSpecSgl ∈ ($a$ ∈ $A$) SplitSpec(cons($a$, nil))
    splitSpec ∈ ($l, l_1, l_2$ ∈ List($A$);
                 Lt(length($l_1$), length($l$));
                 Lt(length($l_2$), length($l$))
                ) SplitSpec($l$)
```

We leave it as an exercise to reimplement split with the following type:

```
split' ∈ ($l$ ∈ List($A$)) SplitSpec($l$)
```

We define the relation Shorter on lists, and show that $\mathrm{Acc}(\mathrm{Lt}, \mathrm{length}(l))$ implies $\mathrm{Acc}(\mathrm{Shorter}, l)$:

```
Shorter ∈ (List($A$); List($A$)) Set
    Shorter ≡ [$h, h_1$]Lt(length($h$), length($h_1$))
accShorter ∈ (Acc(Lt, length($l$))) Acc(Shorter, $l$)
    accShorter(acc(_, $h_1$)) ≡ acc($l$, [$b, h$]accShorter($h_1$(length($b$), $h$)))
```

We can now reimplement sort by first define a functional which refelcts the fact that sort only recurrs on shorter lists.

```
sortRec ∈ ($le$ ∈ ($A$; $A$) Bool;
           $l$ ∈ List($A$);
           ($l'$ ∈ List($A$); Shorter($l'$, $l$)) List($A$)
          ) List($A$)
    sortRec($le$, nil, $h$) ≡ nil
    sortRec($le$, cons($a$, nil), $h$) ≡ cons($a$, nil)
    sortRec($le$, cons($a$, cons($a_1$, $l$)), $h$) ≡
        case split'(cons($a$, cons($a_1$, $l$))) ∈ SplitSpec(cons($a$, cons($a_1$, $l$))) of
            splitSpec(_, $l_2$, $l_3$, $h_1$, $h_2$) ⇒ merge($A$, $le$, $h$($l_2$, $h_1$), $h$($l_3$, $h_2$))
        end
sort' ∈ ($A$ ∈ Set; $le$ ∈ ($A$; $A$) Bool; List($A$)) List($A$)
    sort'($A$, $le$, $h$) ≡
        fix'(List($A$),
            List($A$),
            Shorter,
            sortRec($le$),
            $h$,
            accShorter(accLt(length($h$))))
```

Exercise 18 *Implement* $\mathrm{split}' \in (A \in \mathbf{Set}; l \in \mathrm{List}(A))\mathit{SplitSpec}(A, l)$.

Exercise 19 *Derive a verified version of merge sort, i.e. refine* sort' *such that it has the type of sorting programs:*

$$(R \in (A; A)\mathbf{Set}; r \in \mathrm{Con}(R); l \in \mathrm{List}(A))\mathrm{SortSpec}(R, l)$$

Hints:

1. *You have to implement a dependent version of* fix'.
2. *You need to show that* Perm *is transitive.*

8 Concluding remarks

In the current version of these notes I have left out two subjects which provide a good extension of the material presented here:

Type Theory as a Meta theory The Type Theory used here is very well suited to prove meta theoretic results of logics or λ-calculi. Examples are normalisation proofs [Coq94a, Alt93] or completeness results (such as the completeness of Kripke models for minimal intuitionistic logic).

Infinite objects Thierry Coquand gave recently a very elegant proposal [Coq94b] how to integrate infinite structures like streams into Type Theory.

I hope that these notes give some ideas how to use Type Theory for program verification. It should be possible to understand the concepts presented here without needing a lot of background in typed λ calculus or the meta theory of systems with dependent types. It can be used like a programming language where most users are also not aware of the subtleties of the underlying theory.

Having said this, Type Theory and typed λ-calculus are certainly a fascinating subject of study. I find it hard to recommend a text book which gives a good introduction into this subject. Some of the books mentioned earlier certainly give some background. Henk Barendregt's handbook article [Bar92] gives a good overview but concentrates for my taste too much on *pure type systems*. If one is interested in non-dependent typed λ-calculi Girard et al's book is certainly worthwhile reading [GLT89]. Roy Crole's book [Cro93] approaches the subject from the viewpoint of Category Theory but is generally self contained. Thomas Streicher [Str91] investigates the Calculus of Constructions also using Category Theory but seems to be a bit heavy going for the uninitiated. A good general reference and introduction for categorical models are Martin Hofmann's lecture notes [Hof95b].

On a more practical level it is certainly exciting to use one of the tools mentioned to play with Type Theory. The ALF system on which these notes are based is available by ftp from `file:ftp.cs.chalmers.se:pub/provers/walf`. The compiled version there is only for SUNs under Solaris. If you are interested to install ALF on other systems you should contact me by email.

There are new versions of ALF coming up with entirely different user interfaces and an entirely different proof engine developed by Thierry Coquand. The new versions should be even closer to functional programming languages and will in particular support flexible case- and let-expressions.

References

[AGNvS94] Thorsten Altenkirch, Veronica Gaspes, Bengt Nordström, and Björn von Sydow. *A User's Guide to ALF.* Chalmers University of Technology, Sweden, May 1994. Available on the WWW `file://ftp.cs.chalmers.se/pub/users/alti/alf.ps.Z`.

[Alt93] Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications*, LNCS 664, 1993.

[Bar92] Henk P. Barendregt. Lambda calculi with types. In Samson Abramsky, Dov Gabbay, and Tom S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, chapter 2.2. Oxford University Press, 1992.

[BCMS89] Roland Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1:19–84, 1989.

[CH86] Thierry Coquand and Gérard Huet. The Calculus of Constructions. Technical Report 530, INRIA, Centre de Rocquencourt, 1986.

[Con86] R. L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Coq92] Thierry Coquand. Pattern matching with dependent types. In *Proceeding from the logical framework workshop at Båstad*, June 1992.

[Coq94a] Catarina Coquand. From Semantics to Rules: a Machine Assisted Analysis. In Börger, Gurevich, and Meinke, editors, *CSL'93*, pages 91–105. Springer-Verlag, LNCS 832, 1994.

[Coq94b] Thierry Coquand. Infinite Objects in Type Theory. In *Types for Proofs and Programs*, LNCS, pages 62–78, Nijmegen, 1994. Springer-Verlag.

[Cro93] Roy L. Crole. *Categories for Types.* Cambridge University Press, 1993.

[dB72] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[Dyb91] Peter Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.

[Dyb92] Peter Dybjer. Universes and a general notion of simultaneous inductive-recursive definition in type theory. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, 1992.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[Hof95a] Martin Hofmann. *Extensional concepts in intensional type theory.* PhD thesis, University of Edinburgh, 1995.

[Hof95b] Martin Hofmann. Syntax and semantics of dependent types, 1995. to appear.

[HS94] Martin Hofmann and Thomas Streicher. A gropupoid model refutes uniqueness of identity proofs. In *Proceedings of LICS 94*, 1994.

[LP92] Z. Luo and R. Pollack. LEGO Proof Development System: User's Manual. Technical report, LFCS Technical Report ECS-LFCS-92-211, 1992.

[Luo94] Z. Luo. *Computation and Reasoning: A Type Theory for Computer Science.* International Series of Monographs on Computer Science. Oxford University Press, 1994.

[Mag95] Lena Magnusson. *An Implementation of ALF — a Proof Editor based on Martin-Löf's Type Theory wit Explicit Substitution.* PhD thesis, Chalmers University of Technology and Göteborgs University, 1995.

[ML75] Per Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium 1973*, pages 73–118, Amsterdam, 1975. North-Holland Publishing Company.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Napoli, 1984.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs*, LNCS, pages 213–237, Nijmegen, 1994. Springer-Verlag.

[Nor88] Bengt Nordström. Terminating General Recursion. *BIT*, 28(3):605–619, October 1988.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.

[Pau86] Lawrence C. Paulson. Constructing Recursion Operators in Intuitionistic Type Theory. *Journal of Symbolic Computation*, 2:325–355, 1986.

[PN90] Lawrence C. Paulson and Tobias Nipkow. Isabelle tutorial and user's manual. Technical report 189, Universtiy of Cambridge Computer Laboratory, Cambridge, January 1990.

[Str91] Thomas Streicher. *Semantics of Type Theory.* Birkhäuser, 1991.

[Tho91] Simon Thompson. *Type Theory and Functional Programming.* Addison Wesley, 1991.