B1. Introduction

(a) Principal Approaches to Writing Verified Software

(b) The Theorem Prover Agda

(c) Concept of a Type

(d) Rules and Judgements

(e) Dependent Judgements

(f) Dependent Types

(g) Examples of Dependent Types in Programming
Principal Approaches to Writing Verified Software

(i) **First a program is written.**
Then its correctness is verified.

Most common approach, when formal methods are applied.

- **Dr. Berger** is an expert in this area of research.
- Technology not yet developed.

Extract a program from it.
Prove that a solution for the problem exists.

- **Dr. Kullmann** is an expert on the theorem proving techniques used there.
- Requires advanced automated theorem proving technologies.
- Disadvantage: all or most considerations of the programmers are lost.
- Ordinary programming languages can be used.

- **Main advantage:**
- Ordinary programming languages can be used.
- Then its correctness is verified.
- (ii) **First a program is written.**

Most common approach, when formal methods are applied.

- **Dr. Berger** is an expert in this area of research.
- Technology not yet developed.

Prove that a solution for the problem exists.
Extract a program from it.

- **Dr. Kullmann** is an expert on the theorem proving techniques used there.
- Requires advanced automated theorem proving technologies.
- Disadvantage: all or most considerations of the programmers are lost.
- Ordinary programming languages can be used.

- **Main advantage:**
- Ordinary programming languages can be used.
- Then its correctness is verified.
Programs written in a language which allows to state properties of the program. Example: "This program sorts a list".

Advantages:
- Properties should be verified when compiling the program.

Disadvantages:
- In some cases parts of the program can even be found automatically.
- The information about properties needed might confuse the programmer.
- Programs will be very well documented.
- Programmer is forced to think very clearly.

Advantages:
- Proving and programming will be the same.

Effect:
- Might be too difficult for ordinary programmers.
- Still essentially an area of research. However, advanced tools exist already.
- Requires new programming languages.
- Requires new programming languages.

Disadvantages:
- Requires new programming languages.
- Might be too difficult for ordinary programmers.
- Still essentially an area of research. However, advanced tools exist already.
- Requires new programming languages.
(iv) Mixtures between (i), (iii).

- E.g. SPARK Ada.

In this lecture, we will follow the approach of (iii), based on dependent type theory.
There are several implementations of dependent type theory:

- NuPrl (Cornell, USA), the technically most advanced system.
- LEGO (Edinburgh), about to be replaced.
- Coq (INRIA, France), as well technically very advanced.
- The "ALF-family" (Gothenburg, Sweden) – has probably the clearest concepts.

ALFA, a graphical user interface for Agda, developed by Thomas Hallgren.
* Agda developed by Catarina Coquand.
* Half (Haskell ALF), developed by Thierry Coquand, Dan Synek.
* ALF (developed by Lena Magnusson)

In this module, we will use Agda, but ALFA can be used to create Agda code.

(b) The Theorem Prover Agda

(c) Anton Setzer 2003 (except for pictures)
This is close to the way proofs are carried out by hand.

Proofs for theorems.

In most theorem provers, one has to follow one or several goals, and derive proofs for them.

Develop proofs in it.

Half and Agda have an Emacs mode, which makes it quite convenient to

Half, Agda. All are written in Haskell.
The AlF-family has a different approach of successive refinement.

Proofs in Agda (Cont.)
Installation of Agda

Agda is installed in the Linux lab.

Follow the item "Getting Started with Agda" on the home page of this module.

The source code for the examples given in this lecture will be available from the course homepage.

See information from the course homepage.

Agda is most easily installed under Linux or other versions of Unix.

Agda is installed in the Linux lab.

Installation of Agda

It has been reported that it can be installed under CYGWIN, a UNIX emulation under Windows.

Please check whether the installation works.

(C) Anton Setzer 2003 (except for pictures)
Typed vs. untyped languages can be very difficult. To find such errors in untyped languages can be very difficult.

---

Examples of typed languages:
- Pascal
- C
- C++
- Java
- C#
- Haskell
- ML

Examples of untyped languages:
- Perl
- Python
- Visual Basic
- Lisp

---

Advantages of typed languages:
- Many errors are avoided, especially when using operations defined somewhere else.
- Greater freedom in programming.

Advantages of untyped languages:
- Greater flexibility in programming.

---

(c) Concept of a Type
In order to guarantee correctness of software, we make use of a much more refined type system. It will allow to specify any property of a program, which can be defined as a formula, as a type.
Types used in other languages:

- Scalar types: Booleans, integers, floating point numbers, characters, enumeration types.
- Simple compound types: Arrays, strings, record types, lists, sets.
- In functional programming additionally:
  - Function types, inductive data types (what can be defined using "data").
- In object-oriented programming (not relevant here): interfaces (and classes).

Types used in other languages:
Dependent versions of the above.

Inductive data types. More about this later.

- Product types (essentially records).
- Function types.

Types used in Dependent Type Theory.
The type theory will allow us to both write programs and prove their correctness. Correctness of programs is determined by their types. Therefore we derive that a program is of a type $A$, written $A : f$.

Example: For a sorting algorithm, we derive:

\[
\text{sort} : \text{NatList} \rightarrow \text{SortedList}.
\]
With all programs a type will be associated.

• Deriving the type of a program in Agda will look like programming.

Programs and their Types
Remark on the Syntax Used

In typetheory, oneusually usesas inPascal: single columns  for "has type".

In Haskell, "::" is used in lists, and "is used for "has type".

In order to be closer to Haskell and Cayenne, it was decided to use in Agda as well "::" (although lists don't play an important role there).

- usually use "::"
- except when referring to explicit Agda code (then "::" is used).

In this lecture we will
Remark on the Syntax Used (cont.)

- We often omit \( A \), writing simply \( \forall x : t \).
- We use usually \( \forall (x : A) \). \( t \) except when referring to Agda code.

\[ \forall (A : \text{type}) \rightarrow \text{A can usually be inferred by Agda automatically.} \]

- In Agda one writes \( n (x :: A) ! r \).

\[ A \text{ can usually be inferred by Agda automatically.} \]

\[ \forall (A : \text{type}) \rightarrow \text{A can write} \]

\[ \forall (A : \text{type}) \rightarrow \text{A can write} \]

Similarly for \( \exists \).
In functional programming, one creates a file containing expressions, sometimes together with their types. E.g. Haskell makes it easy to do this "by hand".

By using rules, dependent type theory is more subtle, and it is helpful to explain concepts.

The simple type theory of e.g. Haskell makes it easy to do this "by hand".

Now one can introduce $f : A \rightarrow B$.

E.g. one has defined before $f : V \rightarrow B, a : A$.

In functional programming, one creates a file containing expressions.

Rules
Consider the formation of \( f \) using \( a : A \):

\[
\begin{align*}
B : a & \quad f \\
A & \quad B \\
& \quad \leftarrow A : f
\end{align*}
\]

We write briefly for it:

\[ f : A \rightarrow B, a : A, \text{ then } B : a. \]

The steps taken before correspond to having a rule:

\[ \text{using } B : a \quad \text{and } a : A. \]

Consider the formation of \( f \):
So a rule is something like a template:

\[ f : A \rightarrow B, \quad \forall a \text{ in the previous example can be replaced by any expression.} \]

Rules might have zero premises, e.g.

\[ 0 : \mathbb{N} \]

In this case the “fraction line” will be omitted.

(0 is a natural number).

Rules (cont.)
We can use the above rule to create longer derivations:

\[ \frac{q \vdash \alpha \quad a \vdash B \quad \vdash \beta}{B : \alpha \leftarrow A \vdash \beta} \]

Example 1
\[ \frac{\text{f} : B}{A : \text{f} \leftarrow B \leftarrow \text{f} : C} \]

Then one with premises \( f \) of type \( f \) and \( \text{a} \) of type \( B \).

Example 1 (Cont.)
Derivation of \( a : A \), provided \( a : A \).

We assume a derivation of \( x : x \)\( \forall \) provided \( a : A \).

This uses 1 application of the previous rule.

\[
\begin{align*}
& \forall : a (x : (\forall : x)\forall) \\
\hline
& \forall : a \quad \forall \leftarrow \forall : x. (\forall : x)\forall
\end{align*}
\]

Critical Systems, CS-411, Lenterm 2003, Sec. BI-20
In Agda, these rules are implicit.

\[
\frac{B : a \ f}{A : a \ B} \quad \frac{V \, : \, a \ f}{B \leftarrow A : f}
\]

The rule

Assume we have introduced:

- Assume we have introduced:

Corresponds to the following:

Assume we have introduced:

- Assume we have introduced:

and want to solve the goal

\[
\{ i \ i \} : B
\]

- Goal: we don’t know yet what to fill in.

- Goal: we don’t know yet what to fill in.

Derivations in Agda

(C) Anton Setzer 2003 (except for pictures)
Derivations in Agda (Cont.)
However, one might not know what to fill in. One might guess it has something to do with $f$. So one inserts $f$ and uses menu "refine". The system shows $f :: \{i \mid i\} :: B$.

Now we can solve this goal by filling in $a$ and using refine: $f :: a :: B$.

We can ask for the type of the new goal $f :: \{i \mid i\} :: A$, and get:

We see exampleSimpleDerivation.agda, exampleSimpleDerivation2.agda, CriticalSystems, CS-411, Lentterm 2003, Sec. B1-23.
In ordinary functional programming, it is easy to determine the correctly formed types. Independent type theory the type structure is richer and more complicated. Proof steps are required to conclude that something is a type.

Therefore we have not only the judgement as in functional programming, but also a typing judgement A is a type, written:

\[ A : \text{Type} \]

\[ a : A \]

In dependent type theory the type structure is richer and more complicated. In ordinary functional programming, it is easy to determine the correctness of judgments.
Before deriving a: A we first have to show A: T"ype.
Equality Judgements

On a machine level, terms which reduce to the same, will be identified:

\[ r :: A, s :: A \]

They are identified as \( r = s \).

\[ r \text{ and } s \text{ should have equal types, say } A. \]

Therefore we need a Judgment, which states that \( r \) and \( s \) are equal.

When using rules, we need to be able to express that we can replace \( r \) by \( s \).

\[ \text{See exampleSimpleEquality.agda.} \]

In Agda this is done automatically, the user doesn’t see such equalities.

If one needs at some place \( r \), one can insert \( s \) instead of \( r \) and vice versa.

\[ \text{E.g. } s :: A, s = r \Rightarrow r \quad \text{will be identified.} \]

On a machine level, terms which reduce to the same, will be identified:

Equality Judgements

(\( \text{C}\) Anton Setzer 2003 (except for pictures))
Similarly, we will have equality between types, written as:

\[ \forall A = B : \text{Type} \]

- In simple type theory, there is only one way of writing a type.
- This is something novel in dependent type theory.

Equality Judgements (Cont.)
We have the following 4 types of judgements:

- \( A : \text{Type} \)
- \( a : A \) (\( a \) is of type \( A \))
- \( A = B : \text{Type} \) (\( A \) and \( B \) are equal types)
- \( a = b : A \) (\( a \) and \( b \) are equal elements of type \( A \))

In \texttt{Agda}, only \( A : \text{Type} \) and \( a : A \) are explicit.

In Dependent Type Theory

Summary of Judgements
Consider the judgment in functional programming:

\[(x : A) : x : A \rightarrow A\]

It turns out that the most suitable rule for deriving \(\land\)-expression has as premise in the above case:

\[\forall x : A, \forall \lambda x (\lambda x)\]

This requires that we have judgments which depend on assumptions about variables.

Critical Systems, CS-411, Lenterm 2003, Sec. B1

B1-29
\[ V : a \ f \iff V : a \ (\forall x : V) \iff V : f \]

For instance, if \( V : \text{Type} \), we can derive

In general this context can consist of several variables.

Here \( x : V \) is called context.

\[ V : x \iff V : x \]

If \( x : V \) then \( V : x \) is written as „If „

Dependent Judgements (Cont.)
Assume we have an operation for concatenating strings:

\[
\text{concat} : (x : \text{String}, y : \text{String}) \rightarrow \text{String}
\]

We want to define an operation double, which doubles a string:

\[
\text{double} : \text{String} \rightarrow \text{String}.
\]

- \text{double} (\text{"Hello"}) = \text{"HelloHello"}.

- \text{double} (\text{""}) = \text{""}.

- \text{double} (\text{"Hello"}) = \text{"HelloHello"}.

\text{Example}
Example (cont.)

Westartbyassuming $x: \text{String}$.

The first argument of $\text{concat}$ is a string, so we can apply $\text{concat}$ to $x$.

Now we can abstract from $x$:

$\text{concat} \in \text{String : String}$.

The right hand side is a function, and can be applied to $x$:

$\text{concat}$ is the function, which takes a string and concatenates $x$ in front of it.

$\text{concat} \in \text{String : String}$.

$\text{concat} \in \text{String : String}$.

We start by assuming $x: \text{String}$.
Currying/Uncurrying

We were using a *curried* version of `concat`, called the *curried* version of `concat`, that function takes one argument and returns the concatenation of two arguments.

- `concat` has two arguments.
- `concat` has two arguments.

If applied to one string, we obtain a function `concat` : `String → String`.

\[
\text{Called the *curried* version of } \text{concat.}
\]
This is the same mechanism as in Haskell.

- In Haskell, \((+) : \text{Int} \to \text{Int} \to \text{Int}\) can be applied to one argument:
  - \((+)\ 3\) is the function, which takes a number and returns the number incremented by 3.
  - Therefore \((+3)\ 5\) (which is \(((+)\ 3)\ 5\) and is usually written infix \(3+5\)) is 8.
  - \(\text{map}\ ((+)\ 3)\ [1,2,3]\) returns the application of \((+3)\) to each element of \([1,2,3]\), namely \([4,5,6]\).
Currying/Uncurrying (Cont.)

Note that \((f, \langle x, y \rangle)\) is the pair consisting of \(x\) and \(y\). Write \(\text{concat}(x, y)\) for the application. In Haskell this would be written as \((\text{String, String}) \rightarrow \text{String}\), and we would write \(\text{concat}(x, y)\) for the application. In dependent type theory it has type \((\text{String} \times \text{String}) \rightarrow \text{String}\), application is written like \(\text{concat}\langle x, y \rangle\).

So we when applying \(\text{concat}\) to two strings, we first form the pair of these strings. The uncurred version \(\text{concat}\) of \(\text{concat}\) takes as argument one pair of

\(\text{Currying/Uncurrying (Cont.)}\)
The completed derivation of double reads as follows, given in two pieces:

- Note that the rules we are using haven’t been introduced yet!

Example (Cont.)
In the previous derivation, we first expanded the context of concat by adding a not needed \( x : \text{String} \):

\[
\text{concat} : \text{String} \leftarrow \text{concat} : \text{String} \leftarrow \text{String} \leftarrow \text{String} : \text{concat} : \text{String}
\]

This was so that when applying the rule we have in both premises the same context:

\[
\text{String} : \text{concat} : x \leftarrow \text{concat} : x
\]

\[
\text{String} : \text{concat} : \text{String} \leftarrow \text{concat} : \text{String} \leftarrow \text{String} : x
\]

\[
\text{String} : \text{concat} : \text{String} \leftarrow \text{concat} : \text{String} \leftarrow \text{String} : x
\]
This is a bit pedantic, and often one omits this step and writes the top of the above derivation simply as:

\[ \text{String : } x \]  

Further writing \( a : A \) is the same as \( a : A \) (with empty context).

\( \iff \) If we have no context, we usually omit the \( \iff \).
In Agda, we have no explicit contexts, since we don't use rules. However, if we have the open judgment

\[ \{ i \ i \} = \]

\[ \forall :: \]

\[ (B :: x) \ f \]

Then we can make use of \( B :: x \) for refining the goal.

This context can be shown with a menu.

See exampleShowContext.agda.

ConteXts in Agda
Example: Derivation of double of double

\[
\begin{aligned}
\{i \mid \text{concat} :: \text{String} :: \text{String} :: (\text{String} :: x) \} &= \\
&\text{double}
\end{aligned}
\]

We can insert into the goal concat:

\[
\begin{aligned}
\{i \mid i\} &= \\
&\text{concat} :: \text{String} :: (\text{String} :: x) \text{ double}
\end{aligned}
\]

We start with:

See `exampleDoubleString.agda`.

(c) Anton Setzer 2003 (except for pictures)
Example: Derivation of double (Cont.)

We can check the context of each of these goals and find that both contain $\text{String} :: x$.

We check with Agda, that the two new goals require both type $\text{String}$.

\[
\{ i \ i \} \ { i \ i \} = \text{concat} :: \text{String} :: \text{String} :: x \\
\text{double} \ (\text{String} :: x) \quad \text{when using menu refine, we obtain:}
\]
Example: Derivation of double (cont.)

We are done.

\[
\text{double}(x::\text{String})::\text{String} = \text{concat } x\
\]

Doing the same with the second goal gives:

\[
\{ i \mid i \} \text{ double}(x::\text{String})::\text{String} = \text{concat } x\
\]

We insert \( x \) into the first goal and refine:
Assume we want to assign a type to a sorting function sort.

What is SortedList?

We assume some notion of NatList (list of natural numbers).

\[
\text{sort} : \text{NatList} \rightarrow \text{SortedList}.
\]

It will be •

A dependent type

\[
\text{sort} : \text{NatList} \rightarrow \text{SortedList}.
\]

We assume some notion of NatList (list of natural numbers).

What is SortedList?

An element of SortedList is a list which is sorted.

It is a pair \( \langle l, p \rangle \) s.t.

- It is a pair \( \langle l, p \rangle \) s.t.
- \( l \) is a NatList.
- \( p \) is a proof or verification that \( l \) is sorted.

\[
d \ast 
\]

\[
\ast 
\]

\[
\ast 
\]

\[
\ast 
\]

\[
\ast 
\]

\[
\ast 
\]
SortedLists

For the moment, ignore what is meant by "sorted" as a type.

Sorted(l) is a dependent type.

\( \text{Sorted}(l) \)

\( d : \text{Sorted}(l) \)

\( l : \text{NatList} \)

Elements of SortedList are pairs \( \langle d, l \rangle \) s.t.

\( \text{Sorted}(l) \) is a predicate expressed as a type.

Only important: Sorted depends on \( l \).

Sorted Lists
If \( l \) is sorted, then \( \text{Sorted}(l) \) will have an element.

If \( l \) is not sorted, \( \text{Sorted}(l) \) will be empty, it has no element.

- Then it is not possible to write a program which computes an element of
  \( \text{Sorted}(l) \).

- It is possible to write a program which computes an element of \( \text{Sorted}(l) \).

Sorted lists (cont.)
The Dependent Product

Then the pair \( d,p \) will be an element of

\[
\text{SortedList} := (\forall \mathbf{l} : \text{NatList}) \times \text{Sorted}(\mathbf{l})
\]

expresses:

\[
\text{SortedList} \text{ is the type of pairs of } \langle d,\mathbf{l} \rangle \text{ s.t.}: \text{Sorted}(\mathbf{l})
\]

called the dependent product

\[
(\forall \mathbf{l} : \text{NatList}) \times \text{Sorted}(\mathbf{l}) \leftarrow \text{sort : NatList} \Rightarrow \text{NatList}
\]

\[
\text{sort converts lists into sorted lists.}
\]
The Dependent Function Type

From a sorting function we know more:

- It takes a list and converts it into a sorted list with the same elements.

Assume a type (or predicate) \( \text{EqElements}(l, l') \) standing for

- \( l \) and \( l' \) have the same elements.

\( (\text{except for pictures}) \)
The dependent function type (Cont.)

- The result type depends on the arguments:
  - The type of sort is an instance of the dependent function type:
    - sort is a program, which takes a list and returns a sorted list with the same elements.
    - sort is a list, which is sorted and has the same elements.
    - sort is a list, which is sorted and has the same elements.

\[
((\lambda l.l)(\lambda l.\text{Elements}(l) \times \lambda l.\text{Sorted}(\text{NatList}:l) : \text{NatList}:l)) \leftarrow (\lambda l.\text{NatList}:l)
\]

A refined version of sort has type

The Dependent Function Type (Cont.)
Dependent types are often needed in programming. Some examples:

- In Java, a relatively big library of "collection classes" is available.
- In Java, type checking happens at run time rather than at compile time.
  - Elements of the list have then to be downcasted to their original type.
  - Elements of other types have to be upcasted to Object.
  - Instead, in Java only lists of elements of type Object are available.
  - Cannot be expressed in Java.
- However, this is a dependent type, depending on a type A.
- It would be nice to have "lists of type A".
- Provides implementations of lists, sets, hash tables etc.

Examples of Dependent Types in Programming
Example

Assume a class StudentEntry.

- If we have a list of StudentEntry, we can cast this element down to type StudentEntry.
- If it was originally a StudentEntry, we can obtain an element of Object.
- If we retrieve an element (e.g., the first element) of List<StudentEntry>, we cannot be determined at compile time, only at run time.
- However, whether we have an element of StudentEntry, we can cast this element down to type Object.

• If we have a list of StudentEntry, this element will first be converted (upcasted) to type Object.
What is needed is a weak form of dependent types, called **polymorphism**.

- Types might depend on other types but not on elements of types.
- In C++, this form of dependency is available (called **templates**).
  
  One writes for instance `List<A>` for lists of type `A`.

- In Java it might be available in the next release 1.5.
- In Haskell and ML it is available.

- E.g. `\ x . x : α → α`, i.e. `\ x . x` is of type `α → α` for every type `α`. 

**Polymorphism**
Examples of Dependent Types in Programming (Cont.)

Matrix multiplication is an operation, which takes three natural numbers, and has as result an $n \times k$-matrix, and an $m \times m$-matrix and an $m \times m$-matrix, and thus to be done at run-time. Checking that the dimensions are in accordance has to be done at run-time.

This means memory allocation has to be done at run time. Or by dynamically allocating arrays.

- Usually, this problem is solved by waste of memory: 
- Taking matrices which are big enough and restricting the operation to sub-matrices.

- The type of this function is a dependent type: The types of $n \times m$-matrices of $m \times m$-matrices depend on $n$, $m$, $k$.

Matrix multiplication is an operation, which takes three natural numbers, and has as result an $n \times k$-matrix, and $n \times m$-matrix and $m \times m$-matrix, and thus to be done at run-time. Checking that the dimensions are in accordance has to be done at run-time.

This means memory allocation has to be done at run time. Or by dynamically allocating arrays.

- Usually, this problem is solved by waste of memory: 
- Taking matrices which are big enough and restricting the operation to sub-matrices.
Let \( N \) be the type of natural numbers (i.e., \( 0, 1, \ldots \)). \( N \) will be introduced later.

Let \( \text{Mat}(n,m) \) be the type of \( n \times m \)-matrices.

\[
\begin{align*}
(\text{Mat}(n,m)) & \rightarrow (\text{Mat}(m,k)) \\
(\text{Mat}(u,m)) & \rightarrow (\text{Mat}(m,k)) \\
(\text{Mat}(n,u)) & \rightarrow (\text{Mat}(m,k)) \\
\end{align*}
\]

Then matrix multiplication has type

\[
(\text{Mat}(n,m)) \times (\text{Mat}(m,k)) 
\]

\( \rightarrow \) \( \text{Mat}(n,k) \)

\text{Type of Matrix Multiplication}
A shorter notation for this type is

\[
\begin{align*}
\text{Mat}(n, k) & \leftarrow \text{Mat}(m, k) \\
\text{Mat}(m, m) & \leftarrow \text{Mat}(n, m) \\
(n, m, k : \mathbb{N}) &
\end{align*}
\]
The type of $f$ depends on $n$ and $m$, an example of a dependent type.

- a function $f : \text{Bool}^n \rightarrow \text{Bool}^m$.
- $m$, the number of outputs,
- $n$, the number of inputs,
- $n$ inputs and $m$ outputs can be considered as a function $\text{Bool}^n \rightarrow \text{Bool}^m$.

A digital component (e.g. a logic gate) with $n$ inputs and $m$ outputs can be considered as a component consisting of

\[ \{n, \text{the number of inputs}, \{m, \text{the number of outputs}, \{a \text{function } f : \text{Bool}^n \rightarrow \text{Bool}^m. \]

In general such a component is a triple consisting of

\[ \{n, \text{the number of inputs}, \{m, \text{the number of outputs}, \{\text{a function } f : \text{Bool}^n \rightarrow \text{Bool}^m. \]

Digital Components.
Examples of Dependent Types in Programming (Cont.)

Predicates are dependent types. 

Aarne Ranta has used dependent types in linguistics:

- In a sentence like “The man goes home,” the predicate “goes” depends on whether the subject (“The man”) is singular or plural.
- He constructed grammars based on dependent types and used them for translating sentences between different languages.

See the types of sort above.

Predicates are dependent types.