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

- Most common approach, when formal methods are applied.
- Main advantage:
  Ordinary programming languages can be used.
- Disadvantage: all or most considerations of the programmers are lost.
- Requires advanced automated theorem proving technologies.
- **Dr. Kullmann** is an expert on the theorem proving techniques used there.

(ii) **Prove that a solution for the problem exists.**  
Extract a program from it.

- Technology not yet far developed.
- **Dr. Berger** is an expert in this area of research.
(iii) **Programs written in a language which allows to state properties of the program.**

Example: “This program sorts a list”.

Properties should be verified when compiling the program

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

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

- **Effect:**
  * Proving and programming will be the same.
(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.
- Coq (INRIA, France), as well technically very advanced.
- LEGO (Edinburgh), about to be replaced.
- The “Alf-family” (Gothenburg, Sweden) – has probably the clearest concepts.
  * Alf (developed by Lena Magnusson)
  * Half (= Haskell Alf), developed by Thierry Coquand, Dan Synek.
  * Agda developed by Catarina Coquand.
  * Alfa, a graphical user interface for Agda, developed by Thomas Hallgren.

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

- Half, Agda, Alf are written in Haskell.

- Half and Agda have an Emacs mode, which makes it quite convenient to develop proofs in it.

- In most theorem provers, one has to follow one or several goals, and derive proofs for them. This is close to the way, proofs are carried out by hand.
The Alf-family has a different approach of successive refinement.

- One starts writing the proof code similarly to writing functional programs.
- What cannot be done without machine assistance can be left open in the form of holes (goals).
- Now one can successively, assisted by the system, fill in those goals.
- Therefore proof/program development in the Alf family is very close to ordinary programming.
Installation of Agda

- Agda is installed in the Linux lab.
  - Follow the item “Getting started with Agda” on the home page of this module.
  - Please check whether the installation works.

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

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

- See information from the course home page.

- The source code for the examples given in this lecture will be available from the course home page.
Typed vs. untyped languages

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

- **Examples of untyped languages:**
  Perl, Python, Visual Basic, Lisp.

- **Advantage of untyped languages:**
  Greater freedom in programming.

- **Advantage of typed languages:**
  Many errors are avoided, especially when using operations defined somewhere else.
  To find such errors in untyped languages can be very difficult.
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 Dependent Type Theory

- **Function types.**
  Int → Int is the type of functions mapping integers to integers.

- **Products** (essentially records).
  Int × Int is the type of pairs \( \langle r, s \rangle \), where \( r, s \) are integers.
  - E.g. \( \langle 2, 3 \rangle : \text{Int} \times \text{Int} \).
  - Haskell notation
    * for \( A \times B \) is \((A, B)\),
    * for \( \langle a, b \rangle \) is \((a, b)\).
    e.g. \( (2, 3) :: (\text{Int}, \text{Int}) \).

- **Inductive data types.** More about this later.

- **Dependent versions** of the above.
(d) Rules and Judgements

- 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 $p$ is of a type $A$, written
    \[ f : A \]
  - Example: for a sorting algorithm, we derive:
    \[ \text{sort} : \text{NatList} \rightarrow \text{SortedList} \]
Programs and their Types

- With all programs a type will be associated.
- Deriving the type of a program in Agda will look like programming.
Remark on the Syntax Used

- In type theory, one usually uses as in Pascal single colons “:” for “has type”.

- In Haskell “.” is used in lists, and “::” is used for “has type”.

- In order to be close to Haskell and Cayenne, it was decided to use in Agda as well “::” (although lists don’t play an important rôle there).

- In this lecture we will
  - usually use “:”,
  - except when referring to explicit Agda code (then “::” is used).
Remark on the Syntax Used (Cont.)

• Similarly for $\lambda$:
  
  – In type theory one uses $\lambda x. r$.
  – If one is more explicit about the type of $x$, one can write
    \[
    \lambda(x : A).r
    \]
    
  – In Agda one writes $\setof{(x :: A) \rightarrow r}$.
    * $A$ can usually be inferred by Agda automatically.
  – We use usually $\lambda(x : A).t$ except when referring to Agda code.
  – We often omit $A$, writing simply $\lambda x.t$.  

Critical Systems, CS_411, Lentterm 2003, Sec. B1
In functional programming, one creates a file containing expressions, sometimes together with their types. E.g. one has defined before $f : A \rightarrow B$, $a : A$. Now one can introduce $f\ a : B$.

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

Dependent type theory is more subtle, and it is helpful to explain concepts by using rules.
• Consider the formation of

\[ \text{\( f \ a : B \)} \]

using \( f : A \to B \) and \( a : A \).

• The steps taken before correspond to having a rule:
  If \( f : A \to B, \ a : A \), then \( f \ a : B \).
  We write briefly for it

\[
\begin{array}{c}
\text{\( f : A \to B \)} \quad \text{\( a : A \)} \\
\hline
\text{\( f \ a : B \)} \\
\end{array}
\]
• So a rule is something like a template: 
  $f, A, B, a$ in the previous example can be replaced by any expression.

• Rules might have zero premises, e.g.

  \[ 0 : \mathbb{N} \]

  (0 is a natural number).
  
  – In this case the “fraction line” will be omitted.
Example 1

- We can use the above rule to create longer derivations:

\[
\begin{align*}
  f &: \ A \to (B \to C) & a &: \ A \\
  fa &: B \to C & b &: B \\
  fab &: C
\end{align*}
\]

- We will abbreviate $A \to (B \to C)$ by $A \to B \to C$.

- The above uses 2 applications of the previous rule.
  - First one with premises $f$ of type $A \to B \to C$ and $a$ of type $A$
    - $f$ in the rule is $f$,
    - $A$ in the rule is $A$,
    - $B$ in the rule is $B \to C$,
    - $a$ in the rule is $a$. 
Example 1 (Cont.)

\[ f : A \rightarrow (B \rightarrow C) \quad a : A \]

\[ \frac{f a : B \rightarrow C}{f a b : C} \quad b : B \]

- Then one with premises \( f a \) of type \( B \rightarrow C \) and \( b \) of type \( B \).
  
  - \( f \) in the rule is \( f a \),
  - \( A \) in the rule is \( B \),
  - \( B \) in the rule is \( C \),
  - \( a \) in the rule is \( b \).
Example 2

- Derivation of $(\lambda(x : A).x) \ a : A$, provided $a : A$. (We assume a derivation of $\lambda(x : A).x : A \to A$).

$$
\frac{
\lambda(x : A).x : A \to A \quad a : A
}{\left(\lambda(x : A).x\right) a : A}
$$

- This uses 1 application of the previous rule.
  - $f$ in the rule is $\lambda(x : A).x$,
  - $A$ in the rule is $A$,
  - $B$ in the rule is $B$,
  - $a$ in the rule is $a$. 
Derivations in Agda

- In Agda, these rules are implicit.

- The rule

\[
\frac{f : A \to B \quad a : A}{f \ a : B}
\]

corresponds to the following:

- Assume we have introduced:
  - \( f :: A \to B, a :: A \).
  
  and want to solve the goal
  
  - \( \{! !\} :: B \).
  - Goal \( \{! !\} \) means: we don’t know yet what to fill in.
Derivations in Agda (Cont.)

- Then we can fill this goal by typing in $f \ a$:

- $\{!f \ a!\} :: B$

- If we then choose menu “refine”, the system shows:

- $f \ a :: B$. 
Support given by the System

• 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 $f \{! !\} :: B$.

• We can ask for the type of the new goal $\{! !\}$, and get: $\{! !\} :: A$

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

• See exampleSimpleDerivation.agda, exampleSimpleDerivation2.agda.
Judgements

- In ordinary functional programming, it is easy to determine the correctly formed types. In dependent 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

\[ a : A \]

but as well a typing judgement \( A \) is a type, written:

\[ A : \text{Type} \]
Before deriving \( a : A \) we first have to show \( A : \text{Type} \).
Equality Judgements

- On a machine level, terms, which reduce to the same, will be identified:
  E.g. $s := (\lambda(x : A).x)\ r$ and $r$ will be identified.

- If one needs at some place $r$, one can insert $s$ instead of $r$ and vice versa.

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

- When using rules, we need to be able to express that we can replace $r$ by $s$.

- Therefore we need a judgment, which states that $r$ and $s$ are equal.
  - $r$ and $s$ should have equal types, say $A$.
  - The judgement is written as $r = s : A$. 
Equality Judgements (Cont.)

- Similarly, we will have equality between types, written as

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

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

We have the following 4 types of judgements:

- $A : \text{Type}$  
  “$A$ is a 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 Agda, only $A : \text{Type}$ and $a : A$ are explicit.
Consider the judgement in functional programming:

$$\lambda(x : A).x : A \rightarrow A$$

It turns out that the most suitable rule for deriving $\lambda$-expression has as premise in the above case: If $x : A$ then $x : A$.

This requires that we have judgements, which depend on assumptions about variables.
Dependent Judgements (Cont.)

- "If $x : A$ then $x : A$ is written as

\[ x : A \Rightarrow x : A \]

Here $x : A$ is called **context**.

- In general this context can consist of several variables.
  - For instance, if $A : \text{Type}$, we can derive

\[ f : A \rightarrow A, a : A \Rightarrow f a : A \]
• 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(“Hello”) = “HelloHello”}.
  - \text{double: String \rightarrow String}.
• We start by assuming $x : \text{String}$:
  
  $x : \text{String} \Rightarrow x : \text{String}$.

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

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

• The right hand side is a function, and can be applied to $x$:
  
  $x : \text{String} \Rightarrow \text{concat } x x : \text{String}$.

• Now we can abstract from $x$:
  
  $\text{double} := \lambda(x : \text{String}) \rightarrow \text{concat } x x : \text{String} \rightarrow \text{String}$.
Currying/Uncurrying

- We were using a **curryed** version of `concat`:
  - `concat` has two arguments. If applied to one string, we obtain a function `concat x : String → String`. That function takes one argument `y` and returns the concatenation of `x` and `y`.
  - Called the **curryed** version of `concat`.
This is the same mechanism as in Haskell

In Haskell, \((+)\) : \text{Int} \rightarrow \text{Int} \rightarrow \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.
- map \(((+\) 3 \([1, 2, 3]\) returns the application of \((+\) 3 to each element of \([1, 2, 3]\), namely \([4, 5, 6]\).
Currying/Uncurrying (Cont.)

- The **uncurryied** version `concat'` of `concat` takes as argument one pair of strings.

- So we when applying `concat'` to two strings, we **first form the pair** of these two arguments, and then apply the function to it.

- In dependent type theory it has type `(String \times String) \to String`, application is written like `concat \langle x, y \rangle`.

- In Haskell this would be written as `(String, String) \to String` and we would write `concat(x, y)` for the application.
  - Note that `(x, y)` is the **pair** consisting of `x` and `y`. 

Critical Systems, CS_411, Lentterm 2003, Sec. B1
The complete derivation of double reads as follows, given in two pieces:

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

\[
\begin{align*}
&\text{concat} : \text{String} \to \text{String} \to \text{String} \\
&x : \text{String} \Rightarrow x : \text{String} \quad x : \text{String} \Rightarrow \text{concat} : \text{String} \to \text{String} \to \text{String} \\
&x : \text{String} \Rightarrow \text{concat} \ x : \text{String} \to \text{String} \\
&x : \text{String} \Rightarrow x : \text{String} \quad x : \text{String} \Rightarrow \text{concat} \ x : \text{String} \to \text{String} \\
&x : \text{String} \Rightarrow \text{concat} \ x \ x : \text{String} \\
&\lambda(x : \text{String}) \to \text{concat} \ x \ x : \text{String} \to \text{String}
\end{align*}
\]
Example (Cont.)

- In the previous derivation, we first expanded the context of `concat` by adding a not needed $x : \text{String}$:

\[
\begin{align*}
\text{concat} : \text{String} \rightarrow \text{String} \rightarrow \text{String} \\
\frac{x : \text{String} \Rightarrow \text{concat} : \text{String} \rightarrow \text{String} \rightarrow \text{String}}{}
\end{align*}
\]

- This was so that when applying the rule

\[
\begin{align*}
x : \text{String} \Rightarrow & \quad x : \text{String} \Rightarrow x : \text{String} \\
& \Rightarrow x : \text{String} \Rightarrow \text{concat} : \text{String} \rightarrow \text{String} \rightarrow \text{String} \\
& \Rightarrow x : \text{String} \Rightarrow \text{concat} \ x : \text{String}
\end{align*}
\]

we have in both premises the same context.
• This is a bit pedantic, and often one omits this step and writes the top of the above derivation simply as:

\[
x : \text{String} \Rightarrow x : \text{String} \quad \text{concat} : \text{String} \rightarrow \text{String}
\]

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

• Further writing \( a : A \) is the same as \( \Rightarrow a : A \) (with empty context).
  
  – If we have no context, we usually omit the \( \Rightarrow \).
In Agda, we have no explicit contexts, since we don't use rules. However, if we have the open judgement

\[ f \ (x :: B) :: A = \{! !\} \]

Then we can make use of \( x :: B \) for refining the goal. This context can be shown with a menu.

See exampleShowContext.agda.
Example: Derivation of double

(See exampleDoubleString.agda).

- We start with

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

- We can insert into the goal concat:

  \[
  \text{double} \ (x :: \text{String}) \\
  :: \text{String} \\
  = \{!\text{concat}!\} \]
Example: Derivation of double (Cont.)

- When using menu refine, we obtain:

\[
\begin{align*}
\text{double} \quad (x :: \text{String}) \\
:: \text{String} \\
= \text{concat} \quad \{! \} \quad \{! \}
\end{align*}
\]

- We check with Agda, that the two new goals require both type String.

- We can check the context of each of these goals and find that both contain \( x :: \text{String} \).
Example: Derivation of double (Cont.)

• We insert $x$ into the first goal and refine:

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

• Doing the same with the second goal gives:

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

• We are done.
• Assume we want to assign a type to a sorting function \texttt{sort}.

• It will be

\begin{align*}
\text{sort} : \text{NatList} \rightarrow \text{SortedList}.
\end{align*}

• We assume some notion of \texttt{NatList} (list of natural numbers).

• What is \texttt{SortedList}?

  – An element of \texttt{SortedList} is a list which is sorted.
  – It is a pair $\langle l, p \rangle$ s.t.
    * $l$ is a \texttt{NatList}.
    * $p$ is a proof or verification that $l$ is sorted
For the moment, ignore what is meant by “sorted” as a type.

Only important: Sorted depends on $l$.

- Sorted($l$) is a predicate expressed as a type.

Elements of SortedList are pairs $\langle l, p \rangle$ s.t.

- $l : \text{NatList}$.
- $p : \text{Sorted}(l)$.

Sorted($l$) is a dependent type.
Sorted Lists (Cont.)

- If \( l \) is sorted, then \( \text{Sorted}(l) \) will have an element.
  - It is possible to write a program which computes an element of \( \text{Sorted}(l) \).

- 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 \( \text{Sorted}(l) \).
The Dependent Product

- Then the pair \( \langle l, p \rangle \) will be an element of

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

- \text{SortedList} is the type of pairs \( \langle l, p \rangle \) s.t.
  - \( l : \text{NatList} \),
  - \( p : \text{Sorted}(l) \).

  called the \textbf{dependent product}

- \text{sort} : \text{NatList} \rightarrow ((l : \text{NatList}) \times \text{Sorted}(l)) expresses:
  - \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.
The Dependent Function Type (Cont.)

- A refined version of \texttt{sort} has type

\[(l : \text{NatList}) \rightarrow ((l' : \text{NatList}) \times \text{Sorted}(l') \times \text{EqElements}(l, l'))\]

- “\texttt{sort}(l) is a list, which is sorted and has the same elements”.

- “\texttt{sort} is a program, which takes a list and returns a sorted list with the same elements.”

- The type of \texttt{sort} is an instance of the \texttt{dependent function type}:
  - The result type depends on the arguments.
Dependent types are often needed in programming. Some examples:

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

- Assume a class StudentEntry.

- If we have a list listOfStudentEntries, and add to it an element studentEntry of type StudentEntry, this element will first be converted (upcasted) to type Object.

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

- 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\(<\text{A}\>\) 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.  \(\lambda x. x :: \alpha \Rightarrow \alpha \rightarrow \alpha\),
    i.e. \(\lambda x. x\) is of type \(\alpha \rightarrow \alpha\) for every type \(\alpha\).
Matrix multiplication is an operation, which takes three natural numbers \(n, m, k\), an \(n \times m\)-matrix and an \(m \times k\)-matrix, and has as result an \(n \times k\)-matrix.

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

- Usually, this problem is solved by
  - taking matrices which are big enough and restricting the operation to \(n \times m, m \times k\) and \(n \times k\) sub-matrices,
    - waste of memory
  - or by dynamically allocating arrays.
    - This means memory allocation has to be done at run time.

  - In both solutions, checking that the dimensions are in accordance has to be done at run-time.
Let \( \mathbb{N} \) be the type of natural numbers (i.e. \( 0, 1, \ldots ; \mathbb{N} \) will be introduced later).

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

Then matrix multiplication has type

\[
(n : \mathbb{N}) \to (m : \mathbb{N}) \to (k : \mathbb{N}) \to \text{Mat}(n, m) \to \text{Mat}(m, k) \to \text{Mat}(n, k)
\]
A shorter notation for this type is

\[(n, m, k : \mathbb{N})\]
\[\rightarrow \text{Mat}(n, m)\]
\[\rightarrow \text{Mat}(m, k)\]
\[\rightarrow \text{Mat}(n, k)\]
Digital Components.

- A digital component (e.g. a logic gate) with \( n \) inputs and \( m \) outputs can be considered as a function \( \text{Bool}^n \rightarrow \text{Bool}^m \).

- In general such a component is a triple consisting of
  - \( n \), the number of inputs,
  - \( m \), the number of outputs,
  - a function \( f : \text{Bool}^n \rightarrow \text{Bool}^m \).

- The type of \( f \) depends on \( n \) and \( m \), an example of a dependent type.
Examples of Dependent Types in Programming (Cont.)

- **Predicates** are dependent types.
  - See the types of *sort* above.

- **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.