B1.1

Examples of Dependent Types in Programming

(a) Examples of Dependent Types in Programming
(b) The Theorem Prover Agda

B1-1

Principal Approaches to Writing Verified Software (Cont.)

Critical Systems, CS 411, Lentterm 2003, Sec. B1

(c) Anton Setzer, 2003 (copyright in publicity)

B1-2

Principal Approaches to Writing Verified Software

(c) Anton Setzer, 2003 (copyright in publicity)

B1-3

(c) Anton Setzer, 2003 (copyright in publicity)
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 Alfa-family (Gothenburg, Sweden), has probably the clearest concepts.
- Alf (developed by Lena Magnisson)
- Half (developed by Thierry Coquand, Dan Synek).
- Agda (developed by Catarina Coquand).
- Alfa, a graphical user interface for Agda, developed by Thomas Hallgren.
- Theorem Prover Agda.

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

- Agda is most easily installed under Linux or other versions of Unix.
- Please check whether the installation works.
- Follow the item "Getting started with Agda" on the home page of this module.

Agda can be used to create Agda code. In this lecture, we will give examples from Agda.

Proofs in Agda

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

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

Half and Agda are written in Haskell.

Agda is installed in the Linux lab.

Please check whether the installation works.

Agda can be used to create Agda code. In this lecture, we will give examples from Agda.

Proofs in Agda (Cont.)

The Alfa-family has a different approach of successful refinement.

Proofs in Agda (Cont.)
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 typed languages:
Many errors are avoided, especially when using operations defined somewhere else.

Advantage of untyped languages:
Greater freedom in programming.

Types used in other languages:
- Simple compound types:
  Booleans, strings, records, types, sets,
- Scalar types:
  Integers, floating point numbers, characters, enumeration types.

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 (r, s), where r, s are integers.
- Inductive datatypes.
  More about this later.

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.

In object-oriented programming (not relevant here): interfaces (and classes).

In dependent-type programming (not relevant here): inductives and inductive data types.

In functional programming:
- Function types: Inductive data types (= what can be defined using „data“).
- Simple compound types:
  Booleans, integers, floating point numbers, characters, enumeration types.

In other languages, these concepts are typically used to define types.
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:

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

Example: for a sorting algorithm, we derive:

\[
\forall f : \text{sort} \Rightarrow \text{sortList}
\]

We often omit \( \forall \), writing simply \( A \).

Remark on the Syntax Used (cont.)

- In Haskell and Cayenne, it was decided to use in Agda as
  - In this lecture we will well
  - although lists don't play an important role there.
  - In order to be close to Haskell and Cayenne, it was decided to use in Agda as
  - In Haskell, \( \exists \) is used in lists, and \( \forall \) is used for this type.
  - In type theory, one usually uses \( \exists \) in Pascal and \( \forall \) for this type.
  - \( \exists \) is usually used in lists, and \( \forall \) is used for this type.
  - \( \forall \) is used in lists, and \( \exists \) is used for this type.

Remark on the Syntax Used

- We often omit \( \forall \), except when returning to explicit Agda code.
  - We use usually \( \forall \) except when returning to Agda code.
  - \( \forall \) can usually be inferred by Agda automatically.
  - In Agda one writes \( \forall x : A \, p(x) \).
  - \( \forall \) is

Programs and their Types

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

(p) Rules and Judgements

• Correctness of programs is determined by their types.

• Therefore we derive that a program \( p \) is of a type \( A \), written:

\[
\forall x : A, \, p(x)
\]
In functional programming, one creates a file containing expressions, sometimes together with their types. E.g. one has defined before \( f : A \to B \), \( a : A \). Now one can introduce \( fa : B \).

The simple type theory of \( f \). 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.

Rules might have zero premises. E.g. 

\[ N : 0 \]

The simple type theory of \( f \). Haskell makes it easy to do this "by hand".

So a rule is something like a template.
Example 1 (Cont.)

\[ f : \text{A} \rightarrow (\text{B} \rightarrow \text{C}) \]

\[ a : \text{A} \]

\[ b : \text{B} \]

\[ c : \text{C} \]

Then the non with premises \( f \) of type \( \text{B} \rightarrow \text{C} \) and \( b \) of type \( \text{B} \).

Example 2

Derivation of

\[ (x : \text{A}) : x : \text{A} \]

\[ a : \text{A} \]

\[ b : \text{B} \]

\[ c : \text{C} \]

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

Example 1 (Cont.)

Critical Systems, CS 411, Lent term 2003, Sec. B1

Derivations in Agda (Cont.)
However, one might not know what to fill in. One might guess that it has something to do with \( f \). So one inserts \( f \) and sees mean `typing`.

Support given by the system shows \( \forall \) and \( f \) is a `type`, written:

\[
\forall : \text{Type}
\]

But as well as a `typing` judgment \( \forall \) is a `type`, written:

\[
\forall : \text{Type}
\]

Therefore we have not only the judgment as in functional programming.

In dependent type theory the type structure is richer and more complicated.

Proof steps are required to conclude that sometimes is a `type`.

In ordinary functional programming it is easy to determine the correctly.

Judgements (Cont.)

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

Equality Judgements

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

E.g. \( s : \text{=}(\text{x} : A) \text{x} \) 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.

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.

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

E.g. \( s : \text{=}x(x') \text{x} \) and \( r \) will be identified.

See \( \text{exampleSimpleEquality.agda} \).
Equality Judgements (Cont.)

- Consider the judgement in functional programming:

\[ \lambda(x : A). x : A \]

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

\[ \text{if } x : A \text{ then } x : A. \]

- This requires that we have judgements, which depend on assumptions about variables.

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:

- \( A : \text{Type} \)
- \( a : A \)
- \( A = B : \text{Type} \)
- \( a = b : A \)

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

Critical Systems, Csalli, Lentterm 2003, Sec. B1
Example

Assume we have an operation for concatenating strings:
\[
\text{concat} : (x \in \text{String}; y \in \text{String}) \rightarrow \text{String}
\]

We want to define an operation double which doubles a string:
\[
\text{double} : \text{String} \rightarrow \text{String}
\]

\[
\text{double} \equiv \text{concat} \, \text{Hello} = \text{HelloHello}
\]

Now we can abstract from \(x\):
\[
\text{double} \equiv \text{concat} \, x = \text{Hello} \rightarrow \text{Hello} \rightarrow \text{String}
\]

We start by assuming \(x\):
\[
\text{double} : \text{String} \rightarrow \text{String}
\]

Currying/Uncurrying (Cont.)

This is the same mechanism as in Haskell.

Currying/Uncurrying

Called the curried version of \text{concat}.

Which is the function, which takes a string \(y\) and concatenates \(x\) in front of it.

The first argument of \text{concat} is a string, so we can apply \text{concat} to \(x\):

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

\[
\text{double} \equiv \text{concat} \, \text{Hello} = \text{HelloHello}
\]

We are using a \text{curried version of} \text{concat}.

Currying

\text{concat} has two arguments.

\text{concat} takes one argument \(y\) and returns the concatenation of \(x\) and \(y\).

If \text{double} is applied to one string we obtain a function \text{concat} \, \text{Hello} \rightarrow \text{String}.

\text{concat} \, \text{Hello} \rightarrow \text{String} is the same mechanism as in Haskell.

\text{concat} \, \text{Hello} \rightarrow \text{String} is the function, which takes a string \(y\) and concatenates \(x\) in front of it.
Currying/Uncurrying (Cont.)

The uncurried version `concat` takes as argument one pair of strings. So when applying `concat` to two strings, we first form the pair of these strings.

The uncurried version `concat` of course takes two arguments and not one pair of arguments.

---

Example (cont.)

In the previous derivation, we first expanded the context of `concat` by adding:

```
concat : String × String  → String
```

This was so that when applying the rule:

```
concat : String × String  → String
```

we have in both premises the same context:

```
String : x
```

Further writing `a : a` is the same as `a : a` (with empty context):

```
String : x
```

-- If we have no context, we usually omit the :

If we have no context, we usually omit the :

```
String : x
```

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

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

The complete derivation of `double` reads as follows, given in two pieces:

---

Example (cont.)

This is a bit pedantic, and one omits this step and writes the top of:

```
concat : String × String  → String
```

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

```
String : x
```

This was so that when applying the rule:

```
concat : String × String  → String
```

we have in both premises the same context:

```
String : x
```

This is a bit pedantic, and one omits this step and writes the top of:

```
concat : String × String  → String
```

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

```
String : x
```

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

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

The complete derivation of `double` reads as follows, given in two pieces:

---

Example (cont.)

This is a bit pedantic, and often one omits this step and writes the top of:

```
concat : String × String  → String
```

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

```
String : x
```

This was so that when applying the rule:

```
concat : String × String  → String
```

we have in both premises the same context:

```
String : x
```

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

```
String : x
```

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

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

This is a bit pedantic, and often one omits this step and writes the top of:

```
concat : String × String  → String
```

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

```
String : x
```

This was so that when applying the rule:

```
concat : String × String  → String
```

we have in both premises the same context:

```
String : x
```

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

```
String : x
```

This was so that when applying the rule:

```
concat : String × String  → String
```

we have in both premises the same context:

```
String : x
```

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

```
String : x
```

This was so that when applying the rule:

```
concat : String × String  → String
```

we have in both premises the same context:

```
String : x
```
In Agda, we have no explicit contexts, since we don't use rules. However, if we have the open judgment, we can make use of it for refining the goal.

This context can be shown with a menu.

\[
\{ i \} = \text{String} :: \text{String} :: \text{String} :: x :: f
\]

When using menu refine, we obtain:

\[
\{ i \} \{ i \} = \text{String} :: \text{String} :: \text{String} :: x :: f
\]

When using menu refine, we obtain:

\[
\{ i \} \{ i \} = \text{String} :: \text{String} :: \text{String} :: x :: f
\]

We can check the context of each of these goals and find that both contain:

\[
\{ i \} \{ i \} = \text{String} :: \text{String} :: \text{String} :: x :: f
\]

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

We start with:

\[
\{ i \} = \text{String} :: \text{String} :: \text{String} :: x :: f
\]

We can insert into the goal:

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

When we make use of \( f \) for refining the goal:

\[
\{ i \} \{ i \} = \text{String} :: \text{String} :: \text{String} :: x :: f
\]

We can check the context of each of these goals and find that both contain:

\[
\{ i \} \{ i \} = \text{String} :: \text{String} :: \text{String} :: x :: f
\]

We can insert into the first goal and refine:

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

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:

\[
\{ i \} \{ i \} = \text{String} :: \text{String} :: \text{String} :: x :: f
\]

We can insert into the first goal and refine:

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

We can check the context of each of these goals and find that both contain:

\[
\{ i \} \{ i \} = \text{String} :: \text{String} :: \text{String} :: x :: f
\]

We can insert into the first goal and refine:

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

We can check the context of each of these goals and find that both contain:

\[
\{ i \} \{ i \} = \text{String} :: \text{String} :: \text{String} :: x :: f
\]
Assume we want to assign a type to a sorting function \textit{sort}.

\textit{SortedList} is a dependent type:

\begin{itemize}
  \item \text{sortedList}\ (d) \ 
  \begin{array}{l}
    \begin{array}{l}
      \text{sort : NatList} \rightarrow \text{NatList} \ 
      \text{expresses:}
    \end{array} \\
    \item \text{sortedList is the type of pairs (l, p).}
  \end{array}
\end{itemize}

\textit{SortedList} is the type of pairs (l, p) such that:

\begin{itemize}
  \item \text{p is a proof or verification that l is sorted.}
  \item \text{l is a pair : \text{NatList}.}
  \item \text{An element of \text{SortedList} is a list which is sorted.}
  \item \text{What is \text{SortedList}?}
\end{itemize}

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

\begin{itemize}
  \item \text{sort : NatList} \rightarrow \text{SortedList.}
  \item \text{If \text{sort} \ will be \text{Dependent Types}}.
\end{itemize}

\begin{itemize}
  \item \text{Then the pair (l, p) will be an element of \text{SortedList}.}
\end{itemize}

\begin{itemize}
  \item \text{It is not possible to write a program which computes an element of \text{SortedList}.}
  \item \text{If \text{l is not sorted}, \text{SortedList} will be empty. If \text{l has no element.}
\end{itemize}

\begin{itemize}
  \item \text{If \text{l is sorted}, then \text{SortedList} will have an element.}
\end{itemize}

\textbf{The Dependent Product}:

\begin{itemize}
  \item \text{\text{sortedList} (cont.})
\end{itemize}
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.
- The type of sort is an instance of the dependent function type:

\[ \text{sort} : l, l_0 : \text{NatList} \rightarrow \text{NatList} \] with the same elements.


\[ \text{sort} \] is a program which takes a list and returns a sorted list with the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

A refined version of sort has type

\[ \text{sort} : l, l_0 : \text{NatList} \rightarrow \text{NatList} \]

The Dependent Function Type (cont.)

- The result type depends on the arguments:
- The type of sort is an instance of the dependent function type:

\[ \text{sort} : l, l_0 : \text{NatList} \rightarrow \text{NatList} \] with the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

A refined version of sort has type

\[ \text{sort} : l, l_0 : \text{NatList} \rightarrow \text{NatList} \] with the same elements.

The Dependent Function Type

- The result type depends on the arguments:
- The type of sort is an instance of the dependent function type:

\[ \text{sort} : l, l_0 : \text{NatList} \rightarrow \text{NatList} \] with the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

A refined version of sort has type

\[ \text{sort} : l, l_0 : \text{NatList} \rightarrow \text{NatList} \] with the same elements.

The Dependent Function Type

- The result type depends on the arguments:
- The type of sort is an instance of the dependent function type:

\[ \text{sort} : l, l_0 : \text{NatList} \rightarrow \text{NatList} \] with the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

A refined version of sort has type

\[ \text{sort} : l, l_0 : \text{NatList} \rightarrow \text{NatList} \] with the same elements.

The Dependent Function Type

- The result type depends on the arguments:
- The type of sort is an instance of the dependent function type:

\[ \text{sort} : l, l_0 : \text{NatList} \rightarrow \text{NatList} \] with the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

A refined version of sort has type

\[ \text{sort} : l, l_0 : \text{NatList} \rightarrow \text{NatList} \] with the same elements.

The Dependent Function Type

- The result type depends on the arguments:
- The type of sort is an instance of the dependent function type:

\[ \text{sort} : l, l_0 : \text{NatList} \rightarrow \text{NatList} \] with the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

A refined version of sort has type

\[ \text{sort} : l, l_0 : \text{NatList} \rightarrow \text{NatList} \] with the same elements.

The Dependent Function Type

- The result type depends on the arguments:
- The type of sort is an instance of the dependent function type:

\[ \text{sort} : l, l_0 : \text{NatList} \rightarrow \text{NatList} \] with the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

\[ \text{sort} \] is a list which is sorted and has the same elements.

A refined version of sort has type

\[ \text{sort} : l, l_0 : \text{NatList} \rightarrow \text{NatList} \] with the same elements.
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).

In Java, it might be available in the next release 1.5.

In Haskell and ML it is available.

In both solutions, 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.

- For C++, memory allocation always takes place at compile-time.
- For Haskell, memory allocation is deferred until runtime.
- The types of n x m-matrices, m x k-matrices, and n x k-matrices and thus the result in a x k-matrix.
- The type of this function is a dependent type. The types of n x m-matrices, m x k-matrices, and n x k-matrices depend on n, m, k.

The type of this operation is a dependent type:

\[(n \times m) \times (m \times k) \rightarrow (n \times k)\]

A shorter notation for this type is

\[\text{Matrix multiplication (cont.)}\]

Type of Matrix multiplication

Let \(N\) be the type of natural numbers.

Then matrix multiplication has type

\((N : n) \times (N : m) \rightarrow (N : n \times m)\)

Let \(N\) be the type of natural numbers.

\(\text{Matrix multiplication (cont.)}\)

Type of Matrix multiplication

Type of Matrix multiplication (cont.)
Examples of Dependent Types in Programming (Cont.)

A digital component (e.g. a logic gate) with \( n \) inputs and \( m \) outputs can be considered as a function of \( \text{Bool}^n \rightarrow \text{Bool}^m \).  
In general, such a component is a triple consisting of

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

An example of such a component is a logic gate with \( n \) inputs and \( m \) outputs can be:

- a digital component (e.g. a logic gate) with \( n \) inputs and \( m \) outputs can be.

Critical Systems, CS 411, Lent term 2003, Sec. B.

Predicates are dependent types.

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.
- ...