1. From Simple to Dependent Types

(a) Approaches to Verified Software

(b) The Theorem Prover Agda

(c) Concept of a Type

(d) Dependent Types

(e) Dependent Types in Programming

(a) Approaches to Verified Software

We consider 4 principal approaches towards writing verified software.

(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.

Approaches to Verified Software

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

- E.g. from a proof of the statement

  > For every list there exists a sorted list having the same elements

  one can extract a program, which computes from a list a sorted list having the same elements.
  The correctness is guaranteed.
- Technology not yet far developed.
- **Dr. Berger** and **Dr. Seisenberger** are experts in this area of research.

Approaches to Verified Software

(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.

Approaches to Verified Software

- **Disadvantages of (iii):**
  - 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**.

Approaches to Verified Software

(iv) **Mixtures** between (i), (iii).

- E.g. SPARK Ada.

In this lecture, we will follow the approach of (iii), based on dependent type theory.

(b) The Theorem Prover Agda

Implementations of Dependent Type Theory

- **NuPrl** (Cornell, USA), a technically very advanced system.
  - Uses so called “extensional type theory”.
- **Coq** (INRIA, France), as well technically very advanced.
- **LEGO** (Edinburgh), about to be replaced.
  - Coq and LEGO use so called “**impredicative type theory**”.
- **Epigram** (Nottingham, Durham). New system, not yet as far developed as the other systems.

Implem. of Dept. Type Theory

- The "**Alf**-family" (Gothenburg, Sweden) – has probably the clearest concepts.
  - **Alf** (developed by Lena Magnusson)
    - Proof engine written in SML.
  - **Half** (= Haskell Alf), developed by Thierry Coquand, Dan Synek.
    - Original goal was to rewrite Alf using Haskell.
  - **Agda** developed by Catarina Coquand.
  - **Agda2** developed by Ulf Norell.
  - **Alfa**, a graphical user interface for Agda, developed by Thomas Hallgren.
- "**Cayenne**" (Gothenburg, Sweden) is a dependently typed programming language (not intended as a theorem prover), with a syntax very similar to Agda.

Proofs in Agda

- In this module we will use Agda2.
- **Half, Agda** are written in **Haskell**.
- **Half** and **Agda** have an **Emacs/XEmacs 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**.

Proofs in Agda (Cont.)

- 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.

Thierry Coquand

The main theoretician behind Agda.

Catarina Coquand

The implementor of Agda. Wife of Thierry Coquand.
Now head of the
Dept. of Computer Science and Engineering,
Chalmers University of Technology, Gothenburg, Sweden.

(c) Concept of a Type

Typed vs. untyped languages

- **Examples of typed languages:**
  Pascal, C,C++,Java, C#, Haskell, ML, . . .
- **Examples of untyped languages:**
  Perl, Python, Visual Basic, Lisp, . . .

Advantages/Disadvantages

- **Advantage of untyped languages:**
  Greater **freedom** in programming.
- **Advantages 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.
  - Types are very **natural comments** to programs, which express the basic functionality of a program. Typed languages enforce this kind of comments, and therefore better documentation.

Concept of a Type (Cont.)

- 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**.

Need for Rich Type Structures

- In general programming, one wants a very **rich type structure**.
  - The richer the type structure, the
    - more data types one can define,
    - the more flexibility one has when writing programs, without loosing the advantages of types (preventing errors).
- In this lecture we will mainly focus on theorem proving.
- Greater flexibility touched at the end.

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, algebraic types (= what can be defined using “data”).
- In **object-oriented programming** (not relevant here): interfaces (and classes).

Types used in Dep. Type Theory

- **Function types.**
  Let $\mathrm{NatList}$ the type of lists of natural numbers.
  Then $\mathrm{NatList} \to \mathrm{NatList}$ is the type of functions mapping lists of natural numbers to lists of natural numbers.
  - E.g. sorting functions (without any correctness conditions) are elements of this type.

Types used in Dep. Type Theory

- **Products** (essentially records).
  $\mathrm{Int} \times \mathrm{Char}$ is the type of pairs $\langle r, s \rangle$, where r is an integer and s is a character.
  - E.g. $\langle 2, \text{'c'} \rangle : \mathrm{Int} \times \mathrm{Char}$.
  - In Haskell notation, products are written as follows:
    - Haskell notation for $A \times B$ is (A, B),
    - Haskell notation for $\langle a, b \rangle$ is (a, b).

    e.g. $(2, 3) :: (\mathrm{Int}, \mathrm{Int})$.
  - Agda notation will be that of a record type in other languages.

Types used in Dep. Type Theory

- **Algebraic types.**
  - Types denoted in Haskell by the keyword **data**.
- **Dependent versions** of the above.

(d) Dependent Types

- Assume we want to assign a type to a sorting function sort on lists of natural numbers.
- In most programming language, the type of it is essentially

$$\mathrm{sort} : \mathrm{NatList} \to \mathrm{NatList}$$

  for the type of lists of natural numbers $\mathrm{NatList}$.
- In dependent type theory, we can demand more correctness, namely that its type is

$$\mathrm{sort} : \mathrm{NatList} \to \mathrm{SortedList} \quad .$$

  - We assume some notion of $\mathrm{NatList}$ (list of natural numbers).

SortedList

- What is SortedList?
  - An element of SortedList is a list which is sorted.
  - It is a pair $\langle l, p \rangle$ s.t.
    - l is a NatList.
    - p is a proof or verification that l is sorted:
      - $p : \mathrm{Sorted}\ l$.
      - Note that the functional notation:
        We write

        $$\mathrm{Sorted}\ l$$

        instead of

        $$\mathrm{Sorted}(l)$$

        as in ordinary logic.

Sorted Lists

- For the moment, ignore what is meant by $\mathrm{Sorted}\ l$ as a type.
- Only important: $\mathrm{Sorted}\ l$ depends on l.
  - $\mathrm{Sorted}\ l$ is a predicate expressed as a type.
- Elements of SortedList are pairs $\langle l, p \rangle$ s.t.
  - $l : \mathrm{NatList}$.
  - $p : \mathrm{Sorted}\ l$.
- $\mathrm{Sorted}\ l$ is a dependent type.

Sorted Lists (Cont.)

- An element of $\mathrm{Sorted}\ l$ will be a **proof** that l is sorted.
- If l **is sorted**, then $\mathrm{Sorted}\ l$ will be provable, and therefore **will have an element.**
  - It is possible to write a program which computes an element of $\mathrm{Sorted}\ l$.
- If l is **not sorted**, then $\mathrm{Sorted}\ l$ will have no proof and it will therefore **no element**.
  - Then it is not possible to write a program which computes an element of $\mathrm{Sorted}\ l$.

The Dependent Product

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

$$\mathrm{SortedList} := (l : \mathrm{NatList}) \times (\mathrm{Sorted}\ l) \quad .$$

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

  called the **dependent product**
- $\mathrm{sort} : \mathrm{NatList} \to ((l : \mathrm{NatList}) \times (\mathrm{Sorted}\ l))$ expresses:
  - 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) $\mathrm{EqElements}\ l\ l'$ standing for
  - l and l' have the same elements.

The Dependent Function Type

- A refined version of sort has type

$$(l : \mathrm{NatList}) \to ((l' : \mathrm{NatList}) \times (\mathrm{Sorted}\ l') \times (\mathrm{EqElements}\ l\ l'))$$

- "$\mathrm{sort}\ l$ is a list, which is sorted and has the same elements".
- "sort is a program, which takes a list and returns a sorted list with the same elements."
- The type of sort is an instance of the **dependent function type**:
  - The result type depends on the arguments.

(e) Dependent Types in Programmir

Dependent types are often **needed in programming**, even if no verification is needed.
We give 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 old versions of Java (without templates).
  - Instead, in Java only **lists of elements of type Object** are available.

Dependent Types in Programming

- (Collection classes in Java, Cont.)
  - 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<A> for lists of type A.
- In **Java** they are available from Java 5 onwards (with major restrictions).
- In **C#**, it is available from C# 2 onwards (as well with major restrictions).

Polymorphism

- In **Haskell** and **ML**, it is available.
  - E.g. \ x→x :: $\alpha \Rightarrow \alpha \rightarrow \alpha$,
    which means that $\lambda x.x$ is of type $\alpha \rightarrow \alpha$ for every type α.

Dependent Types in Programming

- **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.

Matrix Multiplication

- 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**.

Type of Matrix Multiplication

- Let $\mathbb{N}$ be the type of natural numbers (i.e. 0, $1, \ldots,$; $\mathbb{N}$ will be introduced later).
- Let $\mathrm{Mat}\ n\ m$ be the type of $n \times m$-matrices. (Will be introduced later).
  Then matrix multiplication has type

$$
\begin{array}{l}
(n : \mathbb{N}) \\
\quad \rightarrow (m : \mathbb{N}) \\
\qquad \rightarrow (k : \mathbb{N}) \\
\qquad\quad \rightarrow \mathrm{Mat}\ n\ m \\
\qquad\qquad \rightarrow \mathrm{Mat}\ m\ k \\
\qquad\qquad\quad \rightarrow \mathrm{Mat}\ n\ k
\end{array}
$$

Type of Matrix Multiplication (Cont.)

- A shorter notation for this type is

$$
\begin{array}{l}
(n, m, k : \mathbb{N}) \\
\quad \rightarrow \mathrm{Mat}\ n\ m \\
\qquad \rightarrow \mathrm{Mat}\ m\ k \\
\qquad\quad \rightarrow \mathrm{Mat}\ n\ k
\end{array}
$$

Dependent Types in Programming

Digital Components.

- A digital component (e.g. a logic gate) with n inputs and m outputs can be considered as a function $\mathrm{Bool}^n \to \mathrm{Bool}^m$.
- This function type is a dependent type, since it depends on the parameters n and m.

Dependent Types in Programming

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

Dependent Types in Linguistics

- **Aarne Ranta** has used dependent types in **linguistics**:
- In a sentence like “The man goes home”, the **verb** (“goes”) depends on,
  - whether the **noun** (“the man”) is **singular** (then the predicate is “**goes** home”)
  - or **plural** (then the predicate is “**go** home”).
- Aarne Ranta constructed **grammars based on dependent types**, and used them for translating sentences between different languages automatically.

More Advanced Applications

- **Hongwei Xi** and **Robert Harper** have introduced a **dependently typed assembly language** (DTAL), which allows to guarantee that array bounds are not violated, without having to carry out run time checks.
- **Petri Mäenpää** and **Matti Tikkanen** have used dependent types for the **modelling of databases**.
- **Simon Thompson**, **Erik Poll** and **John Schackell** have proposed the use of dependent types in order to **integrate logic** (based on dependent types) into **computer algebra systems**.

  (From O. Solaja: *The verification of commonly used algorithms in dependent type theory*, 3rd year project, Swansea 2004, p. 4.)