Dependent Types at Work

LerNet Summer School
Piriápolis, Uruguay

Ana Bove and Peter Dybjer

Chalmers University of Technology, Göteborg, Sweden

25 February - 1 March 2008
What are dependent types?

Types that *depend* on elements of other types. Examples:

- $A^n$ – the type of $n$-dimensional *arrays (vectors)*
- $A^{m \times n}$ – the type of $m \times n$-*matrices*
- The type of *height-balanced* trees of height $n$
- The type of *node-balanced* trees of size $n$
Types that *depend* on elements of other types. Examples:

- $A^n$ – the type of $n$-dimensional arrays (vectors)
- $A^{m \times n}$ – the type of $m \times n$-matrices
- The type of *height-balanced* trees of height $n$
- The type of *node-balanced* trees of size $n$
- The type of *AVL* trees of size $n$ with elements ranging between $\text{min}$ and $\text{max}$
- Etc, etc

All these depend on a number $n$. Can types depend on elements of other types?
What are dependent types?

Types that depend on elements of other types. Examples:

- $A^n$ – the type of $n$-dimensional arrays (vectors)
- $A^{m \times n}$ – the type of $m \times n$-matrices
- The type of height-balanced trees of height $n$
- The type of node-balanced trees of size $n$
- The type of AVL trees of size $n$ with elements ranging between $min$ and $max$
- Etc, etc

All these depend on a number $n$. Can types depend on elements of other types? Yes, for example

- the type of terms of a given (object-language) type
Terminology

Synonyms:

- $B[x]$ is a type which \textit{depends} on $x : A$
- $B[x]$ is a \textit{family} of types which is \textit{indexed} by $x : A$
Is the type of *polymorphic* lists \([a]\) (Haskell notation) a dependent type? After all it depends on \(a\).
Is the type of *polymorphic* lists \([a]\) (Haskell notation) a dependent type? After all it depends on \(a\).

- No, \(a\) is a type – not an element of a type!
Parametrized types

Is the type of *polymorphic* lists \([a]\) (Haskell notation) a dependent type? After all it depends on \(a\).

- No, \(a\) is a type – not an element of a type!

Can’t we have a type of types?
Is the type of *polymorphic* lists \([a]\) (Haskell notation) a dependent type? After all it depends on \(a\).

- No, \(a\) is a type – not an element of a type!

Can’t we have a type of types?

- We can’t have a type of *all* types, because this will lead to inconsistency (Girard’s paradox).
- But we can have a type of *small* types. This type will be called \(\text{Set}\). You will see this later.
When were dependent types born?

Not in 1908. Russell's theory of types
Not in 1940. Church's simply typed lambda calculus
Not in 1975. Milner's simply typed functional programming language ML.

In 1957? FORTRAN's arrays of dimension $n$
In 1968? Howard introduced dependent types to extend Curry's correspondence between propositions and types
In 1968? De Bruijn used dependent types for formalizing mathematics in AUTOMATH
When were dependent types born?

- Not in 1908. Russell’s theory of types
When were dependent types born?

- Not in 1908. Russell’s theory of types
- Not in 1940. Church’s simply typed lambda calculus
When were dependent types born?

- Not in 1908. Russell’s theory of types
- Not in 1940. Church’s simply typed lambda calculus
- Not in 1975. Milner’s simply typed functional programming language ML.
When were dependent types born?

- Not in 1908. Russell’s theory of types
- Not in 1940. Church’s simply typed lambda calculus
- Not in 1975. Milner’s simply typed functional programming language ML.
- In 1957? FORTRAN’s arrays of dimension $n$
When were dependent types born?

- Not in 1908. Russell’s theory of types
- Not in 1940. Church’s simply typed lambda calculus
- Not in 1975. Milner’s simply typed functional programming language ML.

- In 1957? FORTRAN’s arrays of dimension $n$
- In 1968? Howard introduced dependent types to extend Curry’s correspondence between propositions and types
When were dependent types born?

- Not in 1908. Russell’s theory of types
- Not in 1940. Church’s simply typed lambda calculus
- Not in 1975. Milner’s simply typed functional programming language ML.

- In 1957? FORTRAN’s arrays of dimension $n$
- In 1968? Howard introduced dependent types to extend Curry’s correspondence between propositions and types
- In 1968? De Bruijn used dependent types for formalizing mathematics in AUTOMATH
Curry’s observation

Curry observed (already in 1930s) the correspondence between the axioms

\[ P \supset Q \supset P \]
\[ (P \supset Q \supset R) \supset (P \supset Q) \supset P \supset R \]

and the types of the combinators

\[ K : A \rightarrow B \rightarrow A \]
\[ S : (A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow B. \]

\( K \) is a “witness” (also “proof object”) to the truth of \( P \supset Q \supset P \). Similarly, for \( S \).

Also correspondence between \textit{modus ponens}

\[
\begin{array}{c}
P \supset Q \\
\hline
\hline
P \\
\end{array}
\]

\( \frac{P \supset Q}{P} \)

and typing rule for \textit{application}

\[
\begin{array}{c}
f : A \rightarrow B \\
\hline
a : A
\end{array}
\]

\[
f \ a : B
\]
Curry’s correspondence for propositional logic

One-to-one correspondence between combinatory terms and proofs in this implicational logic, can be extended to other connectives:

- $A \times B \text{ vs } P \& Q$
- $A + B \text{ vs } P \lor Q$
- $1 \text{ vs } \top$
- $\emptyset \text{ vs } \bot$

Also reduction of terms corresponds to simplification of proofs.

But you get intuitionistic logic: there is no (polymorphic) term of type

$$A + (A \rightarrow \emptyset)$$

corresponds to a (general) proof of

$$P \lor \neg P$$
Howard introduced Dependent types $A[x_1, \ldots, x_n]$, where $x_1, \ldots, x_n : D$.

- Cartesian product of a family of types $\prod_{x : D} A$
- Disjoint union of a family of types $\sum_{x : D} A$
- Identity types $\text{Id}(a, b)$ (containing “witnesses” or “proof objects”)

How to extend Curry’s idea to predicate logic? What corresponds to

- Predicates $P[x_1, \ldots, x_n]$?
- Universal quantification $\forall x. P$?
- Existential quantification $\exists x. P$?
- Identity $a = b$?
Dependent types are born (Howard)

How to extend Curry's idea to predicate logic? What corresponds to

- Predicates $P[x_1, \ldots, x_n]$?
- Universal quantification $\forall x. P$?
- Existential quantification $\exists x. P$?
- Identity $a = b$?

Howard introduced

- Dependent types $A[x_1, \ldots, x_n]$, where $x_1, \ldots, x_n : D$?
- Cartesian product of a family of types $\prod_{x:D} A$
- Disjoint union of a family of types $\sum_{x:D} A$
- Identity types $\text{Id}(a, b)$ (containing "witnesses" or "proof objects")
Can dependent types express any conceivable property (invariant)?
Can dependent types express any conceivable property (invariant)?

Yes and no.
From Curry-Howard to Agda

- Howard 1968 wrote up a tentative theory of dependent types for predicate logic with equality (in a letter)
- Scott 1970 published a somewhat more definite and powerful theory of "Constructive Validity"
- Martin-Löf 1971 tried a dependent type theory with a type of all types. It was inconsistent
- Martin-Löf 1972 published the first definite version of type theory
- Martin-Löf 1979 wrote the seminal paper "Constructive Mathematics and Computer Programming"
- Early 1980s first proof assistants based on Martin-Löf type theory at Cornell (NuPRL) and Chalmers
- Coquand and Huet 1984 implemented first version of the Coq system. Many other system based on intensional dependent type theory followed – Agda 2 is the latest!
What’s the big deal?

- To write programs that are correct by construction!
- To let logic and programs live together in harmony! The logic is also a programming language!
- To extract programs from proofs!
- To make mathematics a subfield of computer science!
Two styles of using dependent type theory systems:

**As proof assistant** Alternative to systems based on classical set theory (Mizar) or classical type theory (HOL, Isabelle, PVS)

**As programming language** Alternative to mainstream functional languages (Haskell, SML, OCaml)

Community split into two: the second is “Dependent Types in Programming” and Agda is mainly aimed at this.

Key question: how to reconcile logical purity via Curry-Howard with the needs of practical programming?
What’s the price?
What’s the price?

You’ll see.
Plan for the lectures

1. Warming up. Simply typed functional programming in Agda;
2. The main point. Dependently typed functional programming in Agda;
3. Some boring stuff. The Curry-Howard correspondence in Agda;
4. More main points. Agda as a programming logic, reasoning about binary search trees;
5. Some interesting stuff. General recursion and partial functions in Agda;
6. Mathematics = programming, in Agda
It’s time to see dependent types at work!
If you want to download your own version, go to the Agda wiki.
Theoreticians’ and Practitioners’ languages

<table>
<thead>
<tr>
<th>&quot;Real” languages</th>
<th>Logicians’ languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Haskell</td>
<td>Plotkin’s PCF</td>
</tr>
<tr>
<td>?</td>
<td>Gödel system $T$</td>
</tr>
<tr>
<td>Agda</td>
<td>Martin-Löf type theory</td>
</tr>
</tbody>
</table>

Note
- ? can be either "strong fp" subset of Haskell, or "simply typed" subset of Agda (the sort of thing Ana just showed you)
- Logicians’ languages are essentially subsets of the ”real” languages
- Is Agda a logic?
Often used for analysing theoretical properties of functional languages.

Types:

\[ A ::= A \rightarrow A \mid \text{Bool} \mid \text{Nat} \]

Terms:

\[ a ::= x \mid a\ a \mid \lambda x \rightarrow a \]
\[ \mid \text{true} \mid \text{false} \mid \text{if\_then\_else\_} \]
\[ \mid \text{zero} \mid \text{succ} \mid \text{pred} \mid \text{isZero} \]
\[ \mid \text{fix} \]

*General* recursive definitions expressed using `fix`. As an Agda program:

\[
\text{fix} : \{A : \text{Set}\} \rightarrow (A \rightarrow A) \rightarrow A
\]

\[
\text{fix \ f} = \ f \ (\text{fix \ f})
\]

`fix (\x \rightarrow x)` does not terminate.
System T - Gödel 1958

Types:

\[ A ::= A \to A \mid \text{Bool} \mid \text{Nat} \]

Terms:

\[ a ::= x \mid a\ a \mid \lambda x \to a \]
\[ \mid \text{true} \mid \text{false} \mid \text{if}_\text{then}_\text{else}_\]
\[ \mid \text{zero} \mid \text{succ} \mid \text{natrec} \]

Primitive recursive definitions expressed using natrec. As an Agda program:

\[
\text{natrec} : \{C : \text{Set}\} \to C \to (\text{Nat} \to C \to C) \to \text{Nat} \to C
\]
\[
\text{natrec} \ p \ h \ \text{zero} = p
\]
\[
\text{natrec} \ p \ h \ (\text{succ} \ n) = h \ n \ (\text{natrec} \ p \ h \ n)
\]

All programs in system T terminate.
Addition in T and in PCF

In system T:

\[
\text{plus : Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \\
\text{plus m n = natrec m (} \lambda x y \rightarrow \text{succ y} ) n
\]

In PCF:

\[
\text{plus : Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \\
\text{plus m = fix (} \lambda \text{plus n} \rightarrow \text{if isZero n} \\
\text{then m} \\
\text{else succ (plus m (pred n)))}
\]

because

\[
\text{plus m n = if isZero n} \\
\text{then m} \\
\text{else succ (plus m (pred n))}
\]
What is in Haskell but not in PCF?

- recursive *data* types (this is important!)
- lots of syntactic sugar (pattern matching, ...)
- structuring facilities (modules, classes)
- input/output, etc (”the awkward squad”)

---

*What are dependent types? Real and idealized languages Propositions as types Maths in Agda Metamathematics = metaprogramming*
What is in $\mathbb{R}$ but not in System $T$?

Remember: $\mathbb{R}$ is the "strong fp" subset of Haskell. This is a subset in Haskell where:

- all programs terminate because of termination discipline/termination checker
- only well-founded or "strictly positive" data types are allowed

Now:

- well-founded recursive data types - in $T$ there are only Nat and Bool (this is logically important!)
- lots of syntactic sugar (pattern matching, ...)
- structuring facilities (modules, classes)
- input/output, etc
What is in Agda but not in Martin-Löf type theory?

- well-founded recursive data types in general - in Martin-Löf type theory there are ”only” Nat, finite types (including ∅, 1, Bool), $A \times B$, $A + B$, $\prod_{x:A} B$, $\sum_{x:A} B$, $\text{Id} A a b$, $\text{W} x:A B$, $\text{U}_n$
- more general pattern matching and termination check
- lots of syntactic sugar
- structuring facilities (modules, classes)
- there is not (yet) input/output in Agda (logical issues?)
The Agda dilemma

- To find a balance between usability and logical clarity.
- A programming language can be *ugly*, but a program always computes *something*!
- A logic can be *wrong*! You may be able to prove something which is simply not true. This is *bad*.
- Agda wants to be both a ”real” programming language and a ”logic”
Curry-Howard in Agda

Some boring stuff.
Conjunction = Cartesian product

1. A & B is a set (of proofs), representing the conjunction of the sets (of proofs) A and B— the formation rule

2. The elements (proofs) of A & B have the form < a , b >, where (a : A) and (b : B) — the introduction rule

```agda
data _&_ (A B : Set) : Set where
  <_,_> : A -> B -> A & B
```

We could equally well have defined

```agda
_&_ : Set -> Set -> Set where
A & B = A X B
```
Canonical and non-canonical proofs

Are there no other proofs of $A \land B$ than $\langle a, b \rangle$?
Building non-canonical proofs

The rules of &-elimination:

\[
\begin{align*}
\text{fst} & : \{A \ B : \text{Set}\} \rightarrow A \ & B \rightarrow A \\
\text{fst} \ < \ a \ , \ b > & = a \\
\text{snd} & : \{A \ B : \text{Set}\} \rightarrow A \ & B \rightarrow B \\
\text{snd} \ < \ a \ , \ b > & = b
\end{align*}
\]

\(\text{fst} < a \ , \ b >\) is a non-canonical proof, but it can be simplified to a canonical one. (BHK-Martin-Löf justification of this rule?)
Disjunction = Disjoint union

Formation and introduction rules:

\[
data _\lor_ (A \ B : \mathsf{Set}) : \mathsf{Set} \ \mathsf{where}
\]
\[
inl : A \to A \lor B
\]
\[
inr : B \to A \lor B
\]

Elimination and equality rules:

\[
\text{case} : \{A \ B \ C : \mathsf{Set}\}
\]
\[
\to A \lor B \to (A \to C) \to (B \to C) \to C
\]
\[
\text{case} (\text{inl } a) d e = d a
\]
\[
\text{case} (\text{inr } b) d e = e b
\]
The true and the false

True has only one proof

```haskell
data True : Set where
    <> : True
```

False has no proof (no constructor, introduction rule)

```haskell
data False : Set where
```

Proof by no cases (the second line means that there is no case):

```haskell
nocase : {A : Set} -> False -> A
nocase ()
```

Note the important difference with `true, false : Bool!`
Implication is ordinary (non-dependent) function space

_==>_ : (A B : Set) -> Set
A ==> B = A -> B

Universal quantification is dependent function space:

Forall : (A : Set) -> (B : A -> Set) -> Set
Forall A B = (x : A) -> B x
Existential quantifier \( = \) dependent product

\[
\text{data Exists} \ (A : \text{Set}) \ (B : A \rightarrow \text{Set}) : \text{Set} \ \text{where}
\]
\[
[_,_] : (a : A) \rightarrow B \ a \rightarrow \text{Exists} \ A \ B
\]

You can extract the witness

\[
dfst : \{A : \text{Set}\} \ \{B : A \rightarrow \text{Set}\} \rightarrow \text{Exists} \ A \ B \rightarrow A
dfst \ [ \ a, b \ ] = a
\]

and the proof that the proposition is indeed true for that witness:

\[
dsnd : \{A : \text{Set}\} \ \{B : A \rightarrow \text{Set}\} \rightarrow (p : \text{Exists} \ A \ B) \rightarrow B (dfst \ p)
dsnd \ [ \ a, b \ ] = b
\]
Equality

Propositional equality. The constructor (introduction rule) is reflexivity:

\[
data \_==\_ \{A : \text{Set}\} : A \to A \to \text{Set} \text{ where}
     \text{refl} : (a : A) \to a == a\]

Equality-elimination is the substitution law.

\[
\text{subst} : \{A : \text{Set}\} \to \{C : A \to \text{Set}\} \to
     (a' a'' : A) \to a' == a'' \to
     C a' \to C a''
\]

\[
\text{subst} .a .a (\text{refl} a) c = c
\]

Remember the \_\_patterns?
Different notions of equality

- Distinguish between *propositional* $==$ and *definitional* equality $=!$. Two expressions are *definitionally equal* (or *convertible*) iff they have the same normal form. For example:

  $\text{zero } +'\text{ m }==\text{ m}$

  but not

  $\text{zero } +'\text{ m }=\text{ m}$

  if $+'$ is defined by recursion on the second argument.

- *Intensional* and *extensional* equality of functions.

- "Book equality". E.g. if positive rational numbers are represented by pairs of natural numbers, then equality of rationals is not the same as propositional equality of pairs numbers!
A dependently typed version of the primitive recursion combinator

\[
\text{natrec : } \{C : \text{Nat} \rightarrow \text{Set}\} \rightarrow \\
(C \text{ zero}) \rightarrow \\
((m : \text{Nat}) \rightarrow C m \rightarrow C (\text{succ m})) \rightarrow \\
(n : \text{Nat}) \rightarrow \\
C n \\
\text{natrec } p \ h \ \text{zero} = p \\
\text{natrec } p \ h \ (\text{succ }n) = h \ n \ (\text{natrec } p \ h \ n)
\]
An example

A little proof written with the recursion (induction) combinator natrec:

\[
eq\text{-plus-rec} : (n \ m : \text{Nat}) \rightarrow n + m =\text{plus} n m
\]
\[
eq\text{-plus-rec} n m = \text{natrec} (\text{refl} m) (\lambda k' \ i h \rightarrow \text{eq\text{-}succ} \ ih)
\]

where

\[
eq\text{-succ} : \{n \ m : \text{Nat}\} \rightarrow n = m \rightarrow \text{succ} n = \text{succ} m
\]
is one of Peano’s axioms.

The same proof written with pattern matching

\[
eq\text{-plus} \ \text{zero} \ m = \text{refl} \ m
\]
\[
eq\text{-plus} (\text{succ} \ n) \ m = \text{eq\text{-}succ} (\text{eq\text{-}plus} n m)
\]

Essentially the same proof – the second style is common in practice!

“the whole conceptual apparatus of programming mirrors that of modern mathematics (set theory, that is, not geometry) and yet is supposed to be different from it. How come? The reason for this curious situation is, I think, that mathematical notions have gradually received an interpretation, the interpretation which we refer to as classical, which makes them unusable for programming.”
Fortunately, I do not need to enter the philosophical debate as to whether the classical interpretation of the primitive logical and mathematical notions (proposition, truth, set, element, function etc.) is sufficiently clear, because this much is at least clear, that if a function is defined as a binary relation satisfying the usual existence and unicity conditions, whereby classical reasoning is allowed in the existence proof, or a set of ordered pairs satisfying the corresponding conditions, then a function cannot be the same kind of thing as a program. Similarly, if a set is understood in Zermelo’s way as a member of the cumulative hierarchy, then a set cannot be the same thing as a data type.”
Now it is the contention of the intuitionists (or the constructivists, I shall use these terms synonymously) that the basic mathematical notions, above all the notion of function, ought to be interpreted in such a way that the cleavage between mathematics, classical mathematics, that is, and programming that we are witnessing at present disappears.

... What I have just said about the close connection between constructive mathematics and programming explains why the intuitionistic type theory ..., which I began to develop solely with the philosophical motive of clarifying the syntax and semantics of intuitionistic mathematics, may equally well be viewed as a programming language.
The school curriculum is entirely constructive, until you start proving theorems in analysis, like the intermediate value theorem,
6-8 year-olds  Downloading Agda from the Agda wiki. Calculate arithmetic expressions.


12-14 year-olds  List programming. Fractions. Propositional logic.


Integers

An every-day representation of integers

data Int : Set where
  +_ : Nat -> Int
  -_ : Nat -> Int

"Book" equality of integers:

_===_ : Int -> Int -> Set
  + m === + n = m == n
  - m === - n = m == n
  + zero === - zero = True
  - zero === + zero = True
  _ === _ = False

(We could instead let the target be Bool.)
Integers form a setoid

We can prove that equality of integers is an equivalence relation

\[
\text{reflInt : } (n : \text{Int}) \rightarrow n \equiv n
\]
\[
\text{symmInt : } (m \ n : \text{Int}) \rightarrow m \equiv n \rightarrow n \equiv m
\]
\[
\text{transInt : } (m \ n \ p : \text{Int}) \rightarrow
\quad m \equiv n \rightarrow n \equiv p \rightarrow m \equiv p
\]

A set together with an equivalence relation is called a setoid. Constructive mathematicians tend to use the word ”set” for our setoids and ”preset” for our sets. (Note that we do not form quotients in constructive type theory!)
The abstract notion of setoid

We would like to be able say something like

```
IntSetoid : Setoid
```

where `IntSetoid` consists of the set `Int` and the equivalence relation `===`, and `Setoid` is the "type" of sets with equivalence relations.
To this end we use Agda’s `records`!
The record of setoids

record Setoid : Set1 where
  field
    preset : Set
    _===_: preset -> preset -> Set
    refl : (x : preset) -> x === x
    symm : (x y : preset) -> x === y -> y === x
    trans : (x y z : preset) ->
      x === y -> y === z -> x === z

Set1 is the type of large sets. We have eg Set : Set1.
Cf the Eq-class in Haskell. Differences?
The integer instance of setoids

IntSetoid : Setoid
IntSetoid = record
    { preset = Int
    ; _===_ = _===_
    ; refl = reflInt
    ; symm = symmInt
    ; trans = transInt
    }
Metamathematics = metaprogramming

Metamathematics: mathematics of mathematical languages
Metaprogramming: programming programming languages
Gödel system $T$ in combinatory form

The type of object-language types

```haskell
data Ty : Set where
  nat : Ty
  _=>_ : Ty -> Ty -> Ty
```

The type of terms.

```haskell
data Tm : Set where
  K : Tm
  S : Tm
  _@_ : Tm -> Tm -> Tm
  Zero : Tm
  Succ : Tm
  Natrec : Tm
```

You could do this in Haskell.
The type-checking algorithm

\[
\text{hasType} : \text{Tm} \to \text{Ty} \to \text{Bool} \\
\text{hasType} \ K \ (a \Rightarrow b \Rightarrow c) = a \equiv c \\
\text{hasType} \ K \ _ = \text{False} \\
\ldots
\]

\[
\_\equiv\_ : \text{Ty} \to \text{Ty} \to \text{Bool}
\]

You can write this algorithm in Haskell too.
The type system: an inductive family

Each constructor is a "witness" of a rule of the type system:

```agda
data _:::_ : Tm -> Ty -> Set where
  Kty : {a b : Ty} ->
    K :: a => b => a
  Sty : {a b c : Ty} ->
    S :: (a => b => c) => (a => b) => a => c
  @ty : {a b : Ty} ->
    f :: a => b -> x :: a -> f @ x :: b
  Zeroty : Zero :: nat
  Succty : Succ :: nat => nat
  Natrecty : {c : Ty} ->
    Natrec :: c => (nat => c => c) => nat => c

A type-checking algorithm which computes its own correctness proof

hasType : (t : Tm) -> (a : Ty) -> t :: a \/ not (t :: a)
The family of well-typed terms

Only well-typed terms are generated:

data Tm : Ty \to Set where
    \begin{align*}
    K &: \{a \ b : Ty\} \to \\
    & \quad \text{Tm (a \to b \to a)} \\
    S &: \{a \ b \ c : Ty\} \to \\
    & \quad \text{Tm ((a \to b \to c) \to (a \to b) \to a \to c)} \\
    _@_ &: \{a \ b : Ty\} \to \\
    & \quad \text{Tm (a \to b) \to \text{Tm a} \to \text{Tm b}} \\
    \text{Zero} &: \text{Tm nat} \\
    \text{Succ} &: \text{Tm (nat \to nat)} \\
    \text{Natrec} &: \{c : Ty\} \to \\
    & \quad \text{Tm (c \to (nat \to c \to c) \to nat \to c)}
    \end{align*}
Standard semantics of types "Tarski"

\[
[[\_]] : \text{Ty} \rightarrow \text{Set} \\
[[\text{nat}]] = \text{Nat} \\
[[\text{a} \rightarrow \text{b}]] = [[\text{a}]] \rightarrow [[\text{b}]]
\]

Standard semantics of terms "Tarski"

\[
[[\_]]' : \{\text{a} : \text{Ty}\} \rightarrow \text{Tm a} \rightarrow [[\text{a}]] \\
[[\text{K}]]' = \lambda x y \rightarrow x \\
[[\text{S}]]' = \lambda x y z \rightarrow x z (y z) \\
[[\text{f} \circ x]]' = [[\text{f}]]' [[\text{x}]]' \\
[[\text{Zero}]]' = \text{zero} \\
[[\text{Succ}]]' = \text{succ} \\
[[\text{Natrec}]]' = \text{natrec}
\]
Normalization and normalization by evaluation

We shall now normalize expressions (programs) in Gödel system T!

Two approaches

**Traditional reduction-based view:** Use the equations as *simplification/rewrite rules* replacing subexpressions matching the LHS by the corresponding RHS.

”Metamathematical” point of view

**Nbe/reduction-free view:** Write an algorithm which returns the normal form! ”Metaprogramming point of view”.
The weak normalization theorem

A normalization by evaluation algorithm can be extracted from a constructive reading of a proof of weak normalization.

\[ \forall e : a. WN_a(e) \]

where

\[ WN_a(e) = \exists e' : a. e \text{ red } e' \& \text{Normal}(e') \]

Constructive reading (via the BHK-interpretation, constructive axiom of choice), states that a constructive proof of this theorem is an algorithm which given an \( e : a \) computes an \( e' : a \) and proofs that \( e \text{ red } e' \) and \( \text{Normal}(e') \). (This algorithm simultaneously manipulates terms and proof objects, but we can perform program extraction from this constructive proof and eliminate the proof objects.)
There is a well-known technique for proving normalization due to Tait 1967: the *reducibility method*. If one tries to prove the theorem directly by induction on the construction of terms one runs into a problem for application. Tait therefore found a way to strengthen the induction hypothesis.

\[
\begin{align*}
\text{Red}_{\text{nat}}(e) &= \text{WN}_{\text{nat}}(e) \\
\text{Red}_{a \rightarrow b}(e) &= \text{WN}_{a \rightarrow b}(e) \& \forall e': a.\text{Red}_a(e') \supset \text{Red}_b(e, e')
\end{align*}
\]

One then proves that

\[\forall e : a.\text{Red}_a(e)\]

by induction on \(e\).
Normalization by evaluation from Tait’s reducibility method

The constructive proof of

\[ \forall e : a. \text{Red}_a(e) \]

is an algorithm which for all \( e \) computes a proof-object for \( \text{Red}_a(e) \).

- In the base case \( a = \text{nat} \) such a proof object consists of a normal term \( e' \) of type \( \text{nat} \) and a proof that \( e \ \text{red} \ e' \) and \( e' \) normal.

- In the function case \( a = b \Rightarrow c \) such a proof object consists of a normal term (as above) and a function mapping proofs for the reducibility of an argument \( e'' \) to the reducibility of the result \( e \ e'' \).
Martin-Löf’s version of Tait’s proof

To any term $e$ associate three things:

1. the normal form $e'$ of the term
2. a proof $p$ that $e \text{ red } e'$
3. a proof $q$ that $e$ is reducible in the sense of Tait

Constructively, we get a program which maps $e$ to a triple $(e', p, q)$. 
Extracting a program from Tait’s proof

One can now *extract* a program \( nbe \) which just returns a normal form (and no proof object) from the Tait/Martin-Löf style constructive proof of weak normalization. One deletes all intermediate proof objects which do not contribute to computing the result (the normal form) but are only there to witness some property.

Tait’s definition

\[
\begin{align*}
Red_{\text{nat}}(e) &= WN_{\text{nat}}(e) \\
Red_{a \Rightarrow b}(e) &= WN_{a \Rightarrow b}(e) \land \forall e' : a. Red_a(e') \supset Red_b(e e')
\end{align*}
\]

is thus simplified to:

\[
\begin{align*}
\llbracket - \rrbracket & : \text{Ty} \rightarrow \text{Set} \\
\llbracket \text{nat} \rrbracket &= \text{Tm nat} \\
\llbracket a \Rightarrow b \rrbracket &= \text{Tm (a } \Rightarrow \text{ b) } \times \llbracket a \rrbracket \rightarrow \llbracket b \rrbracket
\end{align*}
\]
The resulting normalization algorithm

\[
[\_\_] : \text{Ty} \to \text{Set} \\
[\_\_ \text{nat} \_] = \text{Tm nat} \\
[\_\_ \text{a} \to \text{b} \_] = \text{Tm (a \to b)} \times ([\_\_ \text{a} \_] \to [\_\_ \text{b} \_])
\]

Reification: extracting the normal forms from the semantic values:

\[
\text{reify} : (\text{a} : \text{Ty}) \to [\_\_ \text{a} \_] \to \text{Tm a} \\
\text{reify nat n = n} \\
\text{reify (a \to b) < t , f > = t}
\]

Glueing semantics of terms:

\[
[\_\_]' : \{a : \text{Ty}\} \to \text{Tm a} \to [\_\_ \text{a} \_] \\
[\_\_ \text{K} \_]' = < \text{K} , \text{x} \to < \text{K} @ \text{reify x} , \text{y} \to \text{x} > > \\
... \\
\]

Normalization function:

\[
\text{nbe} : \{a : \text{Ty}\} \to \text{Tm a} \to \text{Tm a} \\
\text{nbe t = reify [\_\_ \text{t} \_]}'
\]