Dependent Types in Programming

Peter Dybjer
Chalmers

TYPES summer school
Giens, 5 September 2002
Constructive mathematics
and computer programming
– the original paradigm

Curry-Howard for programming:

\[ a :: A \]

- element belongs to set/type
- proof proves proposition
- program satisfies specification

Martin-Löf type theory:

- a functional language with dependent types where all programs terminate;
• a specification language including predicate logic;

• a full-scale constructive set theory – a “ZF” for constructive mathematics!
Example: sorting

$$\text{SortProp} \; = \; \forall xs :: [\text{Nat}]. \exists ys :: [\text{Nat}]. \; \text{Sorted} \; ys \; \land \; \text{Perm} \; xs \; ys$$

Prove this proposition!

$$\text{sortProof} :: \text{SortProp}$$

Extract a program

$$\text{sortProg} :: [\text{Nat}] \rightarrow [\text{Nat}]$$
with its proof

\[
\text{sortProgProof} :: \forall xs :: [\text{Nat}]. \\
\left(\text{Sorted (sortProg \, xs)} \land \text{Perm \, xs (sortProg \, xs)}\right)
\]
Program extraction

**Set vs Prop.** Distinguish between computationally relevant and irrelevant parts by using Set/Prop-distinction. The sorting proposition becomes eg

$$\Pi x :: [\text{Nat}].\{y :: [\text{Nat}] \mid \text{Sorted } y \land \text{Perm } x \ y \}$$

**One-element types** are not computationally relevant. Eg use that Sorted (and Perm) are decidable:

sorted :: [Nat] -> Bool
Sorted :: [Nat] -> Set
Sorted xs = T (sorted xs)
where

T :: Bool -> Set
T True  = Unit = ()
T False = Empty
Constructive mathematics
and computer programming
- what happened?


1984 - The calculus of constructions. Logical frameworks. INRIA and Edinburgh groups. Implementations of intensional type theory.

1989 - The Logical Framework/TYPES consortium. Lego, Coq, Alf. Inductive definitions, pattern matching, records, ...
2002 Impressive progress. But dependent type theory has not (yet?) revolutionized programming.
The next 700 MLs

Extensions of the Hindley-Milner type system (polymorphic typed lambda calculus with recursive type and function definitions):

- Equality types in ML.
- ML’s module system. Haskell’s class system.
- Arrays. Sized types. Embedded ML.
- Metaprogramming. Meta-ML.
• Specification language for testing. QuickCheck.
Dependent types in practical programming

Dependent types from the point of view of the functional programmer (ML, Haskell):

**Cayenne** Augustsson 1998.


Series of workshops on DTP: Göteborg 1999, Ponte de Lima 2000, Schloss Dagstuhl 2001, ...
What is Cayenne?

Augustsson 1998: “Although dependent types have been used before in proof systems, e.g., [CH88], to our knowledge this is the first time that the full power of dependent types has been integrated into a programming language.”

- A lazy functional language with dependent types, similar to Agda, but intended to be used as a “real” programming language, like Haskell. Unlike Agda it has predefined types Int, String, etc.

- Intended to be used as a partial type theory with unrestricted recursion in type and function definitions.

- Type-checking undecidable, but nevertheless practical.
• Compiled by removing types and translating to LML, which has a compiler producing efficient code.
What is DML?

Xi and Pfenning 1999: “To our knowledge, no previous type system for a general purpose programming language such as ML has combined dependent types with features including datatype declarations, higher-order functions, general recursions, let-polymorphism, mutable references, and exceptions.”

- A strict functional language with dependent types. DML = Dependent ML.
- A conservative extension of ML. Translate to ML by removing dependent types.
- Decidable type checking is obtained by restricting types to depend on index expressions, eg arithmetic expressions, with decidable constraint solving.
Plan

1. Haskell classes and dependent records.
2. Dependently typed datastructures.
3. Testing and dependent types.
6. Coping with general recursion.
Notation - logical framework

Inspired by Haskell, Cayenne, Agda.
here
\[
x :: a
\]
\[
a \rightarrow b
\]
\[
(x :: a) \rightarrow b
\]
\[
f c
\]
\[
\lambda x \rightarrow e
\]
\[
(a,b)
\]
\[
(x::a,b)
\]
\[
(x::a,y::b)
\]
\[
r.x, r.y
\]
\[
(c,d), (x=c, y=d)
\]
\[
()
\]
\[
\text{Bool}
\]
\[
\text{Set}
\]
\[
T :: \text{Bool} \rightarrow \text{Set}
\]

other
\[
x : a
\]
\[
(a)b
\]
\[
(x:a)b, \Pi x:a.b
\]
\[
f(c)
\]
\[
\lambda x.e, (x)e
\]
\[
a \times b
\]
\[
\Sigma x:a.b
\]
\[
\text{sig}\{x:a,y:b\}
\]
\[
\text{fst}, \text{snd}
\]
\[
\text{struct}\{x=c,y=d\}
\]
\[
N_1,1, \text{Unit}
\]
\[
N_2,2
\]
\[
#, *, \text{Prop}
\]
\[
\text{Lift}
\]
Notation - datatypes
<table>
<thead>
<tr>
<th>here</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nat :: Set</td>
<td>data Nat = Zero</td>
</tr>
<tr>
<td>Zero :: Nat</td>
<td>Succ Nat</td>
</tr>
<tr>
<td>Succ :: Nat -&gt; Nat</td>
<td></td>
</tr>
<tr>
<td>[a]</td>
<td>List a</td>
</tr>
<tr>
<td>Vect a n</td>
<td>a^n</td>
</tr>
<tr>
<td>[]</td>
<td>Nil</td>
</tr>
<tr>
<td>x : xs</td>
<td>x.xs, Cons x xs</td>
</tr>
<tr>
<td>xs ++ ys</td>
<td>append xs ys</td>
</tr>
<tr>
<td>BT a</td>
<td>BinaryTree a</td>
</tr>
<tr>
<td>BST a</td>
<td>BinarySearchTree a</td>
</tr>
<tr>
<td>IsBST t</td>
<td>T (isBST t)</td>
</tr>
<tr>
<td>(==)</td>
<td>eq</td>
</tr>
</tbody>
</table>
Also argument hiding, overloading, Haskell (-) notation for infix operations, etc.
1. Haskell classes and dependent records
The $\text{Eq}$-class in Haskell

class Eq a where
  (==) :: a -> a -> Bool

instance Eq Bool where
  (==) = eqBool

where

eqBool :: Bool -> Bool -> Bool

is defined by

eqBool True True = True
eqBool False False = True
eqBool _ _ = False
Eq with records

Haskell’s class declaration corresponds to

Eq :: Set -> Set

Eq a = a -> a -> Bool

If we want to specify the name (==) of the operation we can instead use a record type:

Eq a = ((==) :: a -> a -> Bool)

(This is the Eq “class” in Cayenne.)

The instance declaration corresponds to
eqBool :: Eq Bool

or, as a record with one field,

((==) = eqBool)) :: Eq Bool
Overloading via classes

Wadler’s original purpose of Haskell-classes was to have a systematic approach to overloading. Having defined an instance `Eq a` we can simply use

```haskell
(==) :: a -> a -> Bool
```

for the equality on `a`.

This is not captured by our dependent records. If

```haskell
r :: Eq a
```

is a record, we have to write

```haskell
r.(==) :: a -> a -> Bool
```
I will however in some future examples assume that we can hide the $r$ also when we work in dependent type theory.
Eq with properties

The Eq-record with properties (decidable setoids = “datoids”):

```
Eq a = (==) :: a -> a -> Bool,
    ref :: (x :: a)
        -> T (x == x),
    sym :: (x,y :: a)
        -> T (x == y)
        -> T (y == x),
    tra :: (x,y,z :: a)
        -> T (x == y) -> T (y == z)
        -> T (x == z)
)
```
Deriving equality

What about writing a function

eq :: (a :: Set) -> Eq a

In total type theory this means that we should define a decidable equality for all
a :: Set. But equality is not decidable for all sets! However, we could have

eq :: (a :: EqSet) -> Eq a

where EqSet is a universe of sets for which we can “derive” decidable equality (cf
ML’s equality types). This leads us towards “generic programming” – more later.
Subclasses

Records are first-class citizens in dependent type theory. This helps us model some further class-related language constructs.

In Haskell Ord is a subclass of Eq. A simplified version:

class Eq a => Ord a where
    (<) :: a -> a -> Bool

In dependent type theory this corresponds to

Ord a = (r :: Eq a,
        (<) :: r.a -> r.a -> Bool)


List equality

In Haskell:

\begin{verbatim}
instance Eq a => Eq [a] where
  []     == []      = True
  []     == (y:ys) = False
  (x:xs) == []      = False
  (x:xs) == (y:ys) = x == y && xs == ys
\end{verbatim}

In dependent type theory we write

\[
\text{listEq :: (a :: Set) -> Eq a -> Eq [a]}\]

ie essentially a function
listEq :: (a :: Set) -> (a -> a -> Bool)
    -> [a] -> [a] -> Bool

listEq f []     []     = True
listEq f []     (y:ys) = False
listEq f (x:xs) []     = False
listEq f (x:xs) (y:ys) = f x y && listEq xs ys
2. Dependently typed datastructures
The zip-function

Haskell library function

```haskell
zip :: [a] -> [b] -> [(a,b)]
```

```haskell
zip []     []    = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _       _     = []
```

exceptional cases when lists are of unequal length.
Vectors

\textbf{Vect} :: \textbf{Set} \rightarrow \textbf{Nat} \rightarrow \textbf{Set}

\textbf{Vect} a n is the set of lists of length n, i.e., the set of n-tuples. Then \texttt{zip} gets the type

\texttt{zip} :: (a, b :: \textbf{Set}) \rightarrow (n :: \textbf{Nat})
  \rightarrow \text{Vect} a n \rightarrow \text{Vect} b n
  \rightarrow \text{Vect} (a, b) n

Note that vectors can be defined either inductively (as an “inductive family”)

\texttt{Nil} :: (a :: \textbf{Set}) \rightarrow \text{Vect} a \text{ Zero}
\texttt{Cons} :: (a :: \textbf{Set}) \rightarrow (n :: \textbf{Nat})
\[- \rightarrow a \rightarrow \text{Vect } a \ n \]
\[- \rightarrow \text{Vect } a \ (\text{Succ } n) \]

or recursively (using "large elimination")

\[\text{Vect } a \ \text{Zero} \quad = \ (\) \]
\[\text{Vect } a \ (\text{Succ } n) \quad = \ (a, \ \text{Vect } a \ n) \]
Balanced binary trees

Bal a h is the set of balanced binary trees of height h and with a-elements in the nodes. “Balanced” here means “as in AVL-trees”.

Bal :: Set -> Nat -> Set

Empty :: (a :: Set)
    -> Bal a 0
BranchE :: (a :: Set) -> (h :: Nat)
    -> a -> Bal a h -> Bal a h
    -> Bal a (h+1)
BranchL :: (a :: Set) -> (h :: Nat)
    -> a -> Bal a (h+1) -> Bal a h
-> Bal a (h+2)

BranchR :: (a :: Set) -> (h :: Nat)
-> a -> Bal a h -> Bal a (h+1)
-> Bal a (h+2)
Binary search trees

Empty :: (min,max :: Nat) -> T (min < max)
    -> BST min max
Branch :: (min,max,root :: Nat)
    -> T (min < root) -> T (root < max)
    -> BST min root -> BST root max
    -> BST min max

Branch min max root p q left right

is a binary search tree with bounds min and max, root root, left and right
subtrees left and right, and p and q proofs that root is between min and max.

binSearch :: (min,max,key :: Nat)
\[
\begin{align*}
\text{insert} & \quad :: (\text{min}, \text{max}, \text{key} :: \text{Nat}) \\
\to & \quad T \ (\text{min} < \text{key}) \to T \ (\text{key} < \text{max}) \\
\to & \quad \text{BST} \ \text{min} \ \text{max} \\
\to & \quad \text{BST} \ \text{min} \ \text{max}
\end{align*}
\]
Correctness of binary search

\[ \text{BST} \xrightarrow{\text{min max}} \overline{\text{BT}} \]

\[ \text{binSearch} \quad \text{member} \]

\[ \text{Nat} \rightarrow \text{Bool} \]

where BT is the set of binary trees with natural numbers in the nodes:

\[
\begin{align*}
\text{Empty} & : \text{BT} \\
\text{Branch} & : \text{Nat} \rightarrow \text{BT} \rightarrow \text{BT} \rightarrow \text{BT}
\end{align*}
\]
and

\[
\begin{align*}
\text{member} & : \ BT \to \ Nat \to \ Bool \\
\text{forget} & : (\text{min}, \text{max} : \ Nat) \to \ BST \ \text{min} \ \text{max} \to \ BT
\end{align*}
\]

are the obvious membership and binary search tree structure forgetting functions.
Correctness of insertion in binary search trees

\[
\begin{align*}
\text{BST} \; \text{min} & \quad \text{max} \quad \text{insert} \; \text{key} \\
\text{binSearch} & \\
\text{Nat} \to \text{Bool} & \quad \{\text{key}\} \cup - \\
\text{binSearch} & \\
\text{Nat} \to \text{Bool} & \\
\end{align*}
\]

(\cup) :: (\text{Nat} \to \text{Bool}) \to (\text{Nat} \to \text{Bool})
\( \rightarrow \text{Nat} \rightarrow \text{Bool} \)

\( \{ - \} \) :: \( \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Bool} \)
The binary search tree property

isBST :: Nat -> Nat -> BT -> Bool

isBST min max Empty = min < max
isBST min max (Branch root left right)
  = min < root && root < max
    && isBST min root left
    && isBST root max right

The dependently typed “integrated” representation is isomorphic to the “external” one:

BST min max \cong (t :: BT, T (isBST min max t))
3. Testing and dependent types
Random testing

QuickCheck (Claessen and Hughes 2000) is a tool for testing Haskell programs automatically. The programmer provides a specification of the program, in the form of properties which functions should satisfy, and QuickCheck then tests that the properties hold in a large number of randomly generated cases. Specifications are expressed in Haskell, using combinators defined in the QuickCheck library. QuickCheck provides combinators to define properties, observe the distribution of test data, and define test data generators.
From the ICFP programming contest

Tom Moertel wrote the following ...

Lesson 4. Test early, test often.

This one I got right. Early on I decided to invest a substantial portion of my time on correctness. I benefited from Haskell’s wicked-powerful type system, which catches a lot of problems all by itself, and then I used QuickCheck, an automatic testing tool, to further automate away the pain of testing. Here are a few examples from my log that show how a tool like QuickCheck can be your best friend in a tight coding corner:
Thu 17:13 EDT. QuickCheck is revealing that something is going wrong with either the Parser or my Show instances. . . .

Thu 17:37 EDT. QuickCheck to the rescue! QuickCheck found a test case that falsified my RoundTrip property for Show->Parse->Show, and I was able to hand-feed that case to my parser to determine the error. . . .

Fri 12:19 EDT. My naive optimizer is done. Not so fast, QuickCheck spotted a corner case that causes the optimizer to discard untagged text at the end of a document. Oops.

QuickCheck found these problems and more, many that I wouldn’t have found without a massive investment in test cases, and it did so quickly and easily. From now on, I’m a QuickCheck man!
QuickCheck can test conditional properties written

\[ p \, x \implies q \, x \]

where

\[ p, q :: D \rightarrow \text{Bool} \]

are decidable predicates written in Haskell.

In dependent type theory:

\[ (x :: D) \rightarrow T (p \, x) \rightarrow T (q \, x) \]
The user writes a generator of random D-elements. QuickCheck uses this to check the conditional property for 100 cases, where only cases which pass p are counted. If a counterexample is found, QuickCheck stops and reports it.
Testing binary search

\[
\text{binSearch} :: \text{BT} \to \text{Nat} \to \text{Bool}
\]

is a Haskell version of binary search (now forgetting about bounds).

The correctness criterion written in QuickCheck’s specification language is
\[
\text{isBST } t \implies \text{binSearch } t \text{ key } =\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!\!!
Combining QuickCheck and dependent type theory

Methodology:

- Debug the goal by running QuickCheck. (The specification may as often be wrong as the program!)

- If successful refine the goal, until you get some subgoal that seems hard to prove. Run QuickCheck on this. And so on.

Hayashi used testing when doing proofs in his PX system already in the 1980-ies.

Other benefit of combining QuickCheck and dependent type theory:
• Prove surjectivity ("coverage") properties of generators of random elements.
4. Generic programming and universal algebra
Generic equality

Recall that we said that we would like a function

$$\text{eq} :: (a :: \text{EqSet}) \to a \to a \to \text{Bool}$$

inside our language, which derives an equality for each set in \text{EqSet} - a universe of “equality sets” (like ML’s equality types).

We could even derive the equality with properties:

$$\text{eq} :: (a :: \text{EqSet})$$
$$\quad \rightarrow ((=) :: a \to a \to \text{Bool},$$
$$\quad \quad \text{ref} :: (x :: a)$$
$$\quad \quad \quad \rightarrow T (x == x),$$
sym  ::  (x,y :: a)  
    ->  T (x == y)  
    ->  T (y == x),  
tra  ::  (x,y,z :: a)  
    ->  T (x == y) ->  T (y == z)  
    ->  T (x == z)  
)
Generic map

Another motivating example of generic programming.

\[
\text{map} :: (a \to b) \to [a] \to [b]
\]

is one of the most basic functions for list programming. But we have an analogous function for binary trees:

\[
\text{mapBT} :: (a \to b) \to \text{BT} a \to \text{BT} b
\]

In general we can define a generic map

\[
\text{map} :: (a \to b) \to \text{D} a \to \text{D} b
\]

where
D :: Set -> Set

is a unary datatype constructor. However, we cannot define map for an arbitrary such D, but only ones which are drawn from a suitable universe of “regular” datatypes.
Regular datatypes in PolyP

PolyP (Jansson and Jeuring, 1996 - ) as in “polytypic” (= generic) programming, is an extension of Haskell.

Polytypic functions are defined by induction on a universe of codes for “regular datastructures”. These are unary type constructors of the form

\[ DX = \mu Y. FXY \]

where \( F \) is a “pattern functor” built up from variables and constants by sum, product, and composition. Eg the type of lists of \( X \)’s is a regular datastructure defined by

\[ [X] = \mu Y.() + (X,Y) \]
The functionality of PolyP can be simulated in dependent type theory.
Generic programming
and universal algebra

We consider term algebras $T_\Sigma$ for one-sorted signatures $\Sigma$.

A signature is just a list of arities:

\[ \text{Sig} = [\text{Nat}] \]

and

\[ T :: \text{Sig} \rightarrow \text{Set} \]

takes a signature and returns its term algebra.

Some signatures and their term algebras:
<table>
<thead>
<tr>
<th></th>
<th>Empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,0]</td>
<td>Bool</td>
</tr>
<tr>
<td>[0,1]</td>
<td>Nat</td>
</tr>
<tr>
<td>[0,1,1]</td>
<td>[Bool]</td>
</tr>
<tr>
<td>[0,2]</td>
<td>BT ( )</td>
</tr>
</tbody>
</table>
A generic size function

How to program

size :: (Sigma :: Sig) -> T Sigma -> Nat

by induction on Sig?
Generic formation, introduction, elimination, and equality rules

Use the well-known initial algebra diagram:

\[
\begin{array}{ccc}
F_\Sigma T_\Sigma & \xrightarrow{\text{Intro}_\Sigma} & T_\Sigma \\
\downarrow & & \downarrow \\
F'_\Sigma(\text{iter}_\Sigma d) & & \text{iter}_\Sigma d \\
\downarrow & & \downarrow \\
F_\Sigma C' & \xrightarrow{d} & C
\end{array}
\]

\[T :: \text{Sig} \rightarrow \text{Set}\]
Intro :: (Sigma :: Sig)
    -> F Sigma (T Sigma) -> T Sigma
iter :: (Sigma :: Sig) -> (C :: Set)
    -> (F Sigma C -> C) -> T Sigma -> C

iter Sigma d (Intro Sigma x)
= d (F' Sigma (T Sigma) C (iter Sigma d) x)
The pattern functor

Object and arrow parts

\[ F : \text{Sig} \to \text{Set} \to \text{Set} \]
\[ F' : (\text{Sigma} \to \text{Sig}) \to (X,Y \to \text{Set}) \]
\[ \to (X \to Y) \]
\[ \to F \text{ Sigma } X \to F \text{ Sigma } Y \]

are defined by

\[ F_{[n_1,\ldots,n_m]}X = X^{n_1} + \cdots + X^{n_m} \]
\[ F'_{[n_1,\ldots,n_m]}XYf = f^{n_1} + \cdots + f^{n_m} \]
Remark. It is possible to modify the elimination rule to account for generic primitive recursion rather than just iteration.
Generic dependent type theory

It is possible to encode a large class of inductive types (including the $T_{\Sigma}$) using well-orderings, but to derive the rules we need extensional type theory (see Dybjer 1997).

A generic formulation of dependent type theory with inductive-recursive definitions was given by Dybjer and Setzer 1999 and for indexed inductive-recursive definitions by Dybjer and Setzer 2001.

The rules for initial algebras can be derived in this theory. In fact, the Dybjer-Setzer axiomatization is obtained by considering a more general universe of signatures and a modified initial algebra diagram.
Generic size

A special case of the initial algebra diagram. Let $\Sigma = [n_1, \ldots, n_m]$.

$$
\begin{array}{c}
T^n_1 + \cdots + T^n_m \xrightarrow{\text{Intro}_\Sigma} T^\Sigma \\
\text{size}_\Sigma^n + \cdots + \text{size}_\Sigma^m \downarrow \downarrow \\
\text{Nat}^n_1 + \cdots + \text{Nat}^n_m \xrightarrow{\text{step}_\Sigma} \text{Nat}
\end{array}
$$

where

$$
\text{step}_\Sigma(\text{In}_i(x_1, \ldots, x_{n_i})) = 1 + x_1 + \cdots + x_{n_i}
$$
can be defined by induction on $\Sigma$. 
5. Metaprogramming.
Well-typed interpreters
and partial evaluators
A well-typed interpreter

Consider a small typed programming language based on combinators. With dependent types we can formalize this object language as a type-indexed family of terms.

\[
\begin{align*}
\text{Ty} & :: \text{Set} \\
\text{Te} & :: \text{Ty} \rightarrow \text{Set}
\end{align*}
\]

Some types

\[
\begin{align*}
\text{NAT} & :: \text{Ty} \\
(\rightarrow) & :: \text{Ty} \rightarrow \text{Ty} \rightarrow \text{Ty}
\end{align*}
\]
Some terms

(@) :: (A,B :: Ty)
    -> Te (A => B) -> Te A -> Te B
K    :: (A,B :: Ty)
    -> Te (A => B => A)
S    :: (A,B,C :: Ty)
    -> Te ((A => B => C)
            => (A => B) => A => C)
ZERO :: Te NAT
SUCC :: Te (NAT => NAT)
ITER :: (C :: Ty)
      -> Te ((C => C) => C => NAT => C)

We hide type arguments and write eg f @ c rather than (@) A B f a.
Semantics = interpretation

Interpretation of types:

Eval :: Ty -> Set

Eval NAT = Nat
Eval (A -> B) = Eval A -> Eval B

Interpretation of terms:

eval :: (A :: Ty) -> Te A -> Eval A

eval (f @ c) = eval f (eval c)
eval K = k = \x y -> x
eval S = s = \( x \ y \ z \to (x \ z) \ y \ z \)
eval ZERO = Zero
eval SUCC = Succ
eval ITER = iter

\text{as usual hiding the type argument. iter is the iterator}

iter :: (C :: Set) \to (C \to C) \to C \to \text{Nat} \to C
Partial evaluation

Consider the function

\[
\text{power} :: \text{Nat} \to \text{Nat} \to \text{Nat}
\]

\[
\text{power} \ m \ n = \text{iter} \ (\text{mult} \ m) \ 1 \ n
\]
\[
\text{mult} \ m \ n = \text{iter} \ (\text{add} \ m) \ 0 \ n
\]
\[
\text{add} \ m \ n = \text{iter} \ \text{Succ} \ m \ n
\]

where we have hidden the first argument \text{Nat} of \text{iter}. 
Static and dynamic arguments

In partial evaluation we distinguish between binding-times, that is,

**static** arguments, which are known, and

**dynamic** arguments, which are not known

at specialization time. If \( m \) is dynamic and \( n \) is static in `power m n` then we can specialize and simplify the definition. Eg \( n = 3 \)

\[
\text{power } m \ 3 = \ \text{iter } (\text{mult } m) \ 1 \ 3
\
= \ \text{mult } m \ (\text{mult } m \ (\text{mult } m \ 1))
\
= \ \text{mult } m \ (\text{mult } m \ m)
\]
The simplified program is called the *residual program*.
2-level lambda calculus

In 2-level lambda calculus types and terms are given binding-time annotations. Eg the type \texttt{Nat} exists in both a static version \texttt{Nat} and a dynamic version \texttt{Nat}.

The function \texttt{power} with a first dynamic and a second static argument thus gets the type:

\[
\text{powerDS} :: \texttt{Nat} \rightarrow \texttt{Nat} \rightarrow \texttt{Nat} \\
\text{powerDS} \: m \: n = \texttt{iter} (\texttt{mult} \: m) (\$ \: 1) \: n
\]

where

\[
\$ :: \texttt{Nat} \rightarrow \texttt{Nat}
\]

transforms a static number into the corresponding dynamic one.
Binding-times

There are four different versions of \texttt{power} depending on the binding-times of the arguments:

\texttt{powerDS} :: \texttt{Nat} \rightarrow \texttt{Nat} \rightarrow \texttt{Nat}

\texttt{powerSD} :: \texttt{Nat} \rightarrow \texttt{Nat} \rightarrow \texttt{Nat}

\texttt{powerDD} :: \texttt{Nat} \rightarrow \texttt{Nat} \rightarrow \texttt{Nat}

\texttt{powerSS} :: \texttt{Nat} \rightarrow \texttt{Nat} \rightarrow \texttt{Nat}
Binding-times again

With dependent types and the correspondences

<table>
<thead>
<tr>
<th>dynamic</th>
<th>object language</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>meta language</td>
</tr>
</tbody>
</table>

to get types for “generating extensions” corresponding to the binding time annotations:

\[
\text{powerDS} :: \text{Nat} \rightarrow \text{Te (NAT} \rightarrow \text{NAT)}
\]

\[
\text{powerSD} :: \text{Nat} \rightarrow \text{Te (NAT} \rightarrow \text{NAT)}
\]

\[
\text{powerDD} :: \text{Te (NAT} \rightarrow \text{NAT} \rightarrow \text{NAT)}
\]
powerSS :: Nat -> Nat -> Nat

Note the analogy between the binding-time annotated types and the dependent types, except that powerDS exchanges the order of its arguments, because a static argument is always given before a dynamic one.
Terms-in-context

Combinatory logic is not quite suitable for partial evaluation, and we want to work in lambda calculus instead. Therefore we need to formalize terms-in-context. We use a name-free approach with

\[ \text{Te} :: [\text{Ty}] \to \text{Ty} \to \text{Set} \]

so that \( \text{Te} [A_1, \ldots, A_n] \) \( A \) is the set of terms of type \( A \), where the variables (de Bruijn indices) have the types \( A_1, \ldots, A_n \).

Pure typed lambda terms are then generated by the following rules:

\[ (@) :: (A :: [\text{Ty}]) \to (A, B :: \text{Ty}) \]
\[ \to \text{Te} A (A \to B) \to \text{Te} A A \to \text{Te} A B \]
LAM :: (As :: [Ty]) -> (A,B :: Ty)
    -> Te (A:As) B -> Te As (A => B)
VAR :: (As :: [Ty]) -> (A :: Ty)
    -> Member A As -> Te As A

We hide context and type arguments.
A well-typed interpreter for typed lambda terms

The interpretation of types is as before, but now we need to interpret contexts too:

Eval :: [Ty] -> Set

Eval [] = ()
Eval (A:As) = (Eval A, Eval As)

Interpretation of terms:

eval :: (As :: [Ty]) -> (A :: Ty)
   -> Eval As -> Eval A
\begin{verbatim}
    eval (f @ c) as = eval f as (eval c as)
    eval (LAM e) as = \x -> eval e (x,as)
    eval (VAR n) as = proj n as
\end{verbatim}

again hiding the context and type arguments.
Partially evaluating power again

When we specialize power with a static second argument \( n \) we get the term

\[
\lambda x \to \text{iter} \ (\text{mult} \ x) \ 1 \ n \ :: \ \text{Nat} \to \text{Nat}
\]

which we want to simplify by using that \( \text{iter} \) is a static operation.

The corresponding object language term is:

\[
\text{LAM} \ (\text{ITER} \ @ \ (\text{MULT} \ @ \ (\text{VAR \ X0})) \ @ \ ($ \ 1) \ @ \ ($ \ n)) \\
\ :: \ \text{Te} \ [] \ (\text{NAT} \Rightarrow \text{NAT})
\]

where

\[
$ \ :: \ (\text{As} :: \ [\text{Ty}]) \to \text{Nat} \to \text{Te} \ \text{As} \ \text{NAT}
\]
is the injection of a metalanguage natural number into the object language:

\[
\begin{align*}
\$ \text{Zero} &= \text{ZERO} \\
\$ (\text{Succ } n) &= \text{SUCC } (\$ n)
\end{align*}
\]
Some typings

In the term

\[
\text{LAM} \ (\text{ITER} \ \odot \ (\text{MULT} \ \odot \ (\text{VAR} \ \text{X0})) \ \odot \ (\$ \ 1) \ \odot \ (\$ \ n))
\]

:: Te [] (NAT => NAT)

we use the following instances:

\[
\text{ITER} :: \ Te \ [\text{NAT}]
\]

\[
\quad ((\text{NAT} => \text{NAT}) => \text{NAT} => \text{NAT} => \text{NAT})
\]

\[
\text{MULT} :: \ Te \ [\text{NAT}] \ (\text{NAT} => \text{NAT} => \text{NAT} => \text{NAT})
\]

\[
\text{X0} :: \ \text{Member} \ \text{NAT} \ [\text{NAT}]
\]
Executing a static operation

The fact that \texttt{iter} is a static operation here is expressed by the fact that for all \( n :: \text{Nat} \) the object language term obtained by executing

\[
\text{iter} (\lambda t \rightarrow \text{MULT} @ (\text{VAR X0}) @ t) (@ 1) \ n
\]

has the same semantics as

\[
\text{ITER} @ (\text{MULT} @ (\text{VAR X0})) @ (@ 1) @ (@ n)
\]

For \( n = 2 \):

\[
\text{iter} (\lambda t \rightarrow \text{MULT} @ (\text{VAR X0}) @ t) (@ 1) 2
= \text{MULT} @ (\text{VAR X0}) @ (\text{MULT} @ (\text{VAR X0}) @ (@ 1))
\]
This term can be further simplified to the semantically equal term

\[ \text{MULT} \circ (\text{VAR} \, \text{XO}) \circ (\text{VAR} \, \text{XO}) :: \text{Te} \, [\text{Nat}] \, \text{Nat} \]

This is the residual program (normal form).
6. Coping with general recursion
Recursion in type theory

**Total type theory.** Recursion in Martin-Löf type theory is primitive (structural) recursion. General recursive algorithms are not directly typable in total type theory. This is one of the main obstacles to making total type theory a programming language.

**Partial type theory.** There is a version of Martin-Löf type theory with general recursion (Martin-Löf’s domain interpretation of type theory 1983, Palmgren 1991). Cayenne can be said to be an implementation of partial type theory. Partial type theory allows non-terminating computations and does not support the Curry-Howard correspondence.

We want both logic and general recursion. How? There are many suggestions.
We shall look at a method for generating special purpose accessibility predicates for general recursive definitions (Bove 1999 and Bove and Capretta 2001).
Quicksort in Haskell

qSort :: [Nat] -> [Nat]

qSort [] = []
qSort (x : xs)
    = qSort (filter (< x) xs)
    ++ x : qSort (filter (>= x) xs)

(A more efficient version which partitions xs in one pass can easily be written.)
Quicksort in total type theory

We can define a termination predicate for quicksort:

\[
D :: [\text{Nat}] \to \text{Set}
\]

\[
\text{CO} :: D []
\]

\[
\text{C1} :: (x :: \text{Nat}) \to (xs :: [\text{Nat}])
\]

\[
\to D (\text{filter} (< x) xs)
\]

\[
\to D (\text{filter} (\geq x) xs)
\]

\[
\to D (x : xs)
\]

Quicksort can then be represented as a function of two arguments: a list and a proof that quicksort terminates for this list.
qSort :: (xs :: [Nat]) -> D xs -> [Nat]

qSort [] C0 = []
qSort (x : xs) (C1 x xs p q)
    = qSort (filter (< x) xs) p
    ++ x : qSort (filter (>= x) xs) q
Termination of quicksort

Quicksort terminates for all lists:

\[ q\text{SortTerm} :: (xs :: [\text{Nat}]) \rightarrow D\; xs \]

Hence

\[
\forall xs \rightarrow q\text{Sort} \; xs \; (q\text{SortTerm} \; xs) \\
:: [\text{Nat}] \rightarrow [\text{Nat}]
\]
Treesort

treeSort = buildBST . preorder

buildBST :: [Nat] -> BST
preorder :: BST -> [Nat]

This is essentially the same algorithm as functional quicksort, but it is structurally recursive!
(C. McBride)

The idea of strong functional programming!
(D. Turner)
McCarthy’s 91-function in Haskell

The Bove-Capretta method is applicable to nested recursion as well.

Haskell code for McCarthy’s 91-function:

\[
f_{91} :: \text{Nat} \to \text{Nat}
\]

\[
f_{91} n = \text{if } n > 100 \text{ then } n - 10 \\
\quad \text{else } f_{91} (f_{91} (n + 11))
\]
McCarthy’s 91-function
in total type theory

We get a simultaneous inductive-recursive definition of the termination predicate and the structural recursive version of \(f_{91}\):

\[
D :: \text{Nat} \rightarrow \text{Set} \\
f_{91} :: (n :: \text{Nat}) \rightarrow D\ n \rightarrow \text{Nat}
\]

\[
C_0 :: (n :: \text{Nat}) \rightarrow T\ (n > 100) \rightarrow D\ n \\
C_1 :: (n :: \text{Nat}) \rightarrow T\ (n \leq 100) \\
\quad \rightarrow (p :: D\ (n + 11)) \\
\quad \rightarrow D\ (f_{91}\ (n + 11)\ p) \\
\quad \rightarrow D\ n
\]
\[ f_{91} n \ (C_0 \ n \ r) = n - 10 \]
\[ f_{91} n \ (C_1 \ n \ r \ p \ q) \]
\[ = f_{91} \ (f_{91} \ (n + 11) \ p) \ q \]
Summary

• Dependent types for ML/Haskell style programming:
  – classes, modules
  – arrays, sized types, simple datatype invariants
  – generic programming
  – metaprogramming

• Correctness in the short term:
  – modest use of dependent types (eg a la DML)
  – combining testing and dependent types

• General recursion and dependent types.