Normalization by Evaluation

Midlands Graduate School
in the Foundations of Computer Science
Leicester, UK

Peter Dybjer

Chalmers University of Technology
Gothenburg, Sweden

30 March - 3 April, 2009

What is "evaluation"?

The process which obtains a *value* of an *expression*.
E g evaluation of an arithmetic expression in primary school:
Suppose we are given

$$(11 + 9) \times (2 + 4)$$

We can rewrite this expression in two ways, simplifying either the first bracket or the second. Simplifying the first bracket, we have

$$20 \times (2 + 4) = 20 \times 6 = 120$$

Simplifying the second gives

$$(11 + 9) \times 6 = 20 \times 6 = 120.$$

The value of $(11 + 9) \times (2 + 4)$ is 120.

What is "normalization"?

Simplification in secondary school.

$$\begin{aligned}(a+b)(a-b) &= a(a-b)+b(a-b)\\ &= a^2-ab+ba-b^2\\ &= a^2-b^2\end{aligned}$$

Simplification of *open* expressions, that is, expressions with *free* variables.

Normalization by evaluation?

To do secondary school simplification through primary school simplification ...?? Yes! But there is more to say.
The word "evaluation" has a second meaning ...

Normalization by evaluation in a model?

Normalization by “evaluation” in a model.

$$\textit{syntax} \xrightarrow{[\![-]\!]} \textit{model}$$

(We can ”evaluate” open expressions too, in this sense.)
”There is a striking similarity between computing a program and assigning semantics to it”, P. Landin (1964): The mechanical evaluation of expressions.

Normalization by evaluation in a model

Normalization by "evaluation" in a model.

$$\begin{tikzcd} \mathit{syntax} \arrow[r, shift left, "{[\![-]\!]}"] & \mathit{model} \arrow[l, shift left, "\mathit{reify}"] \end{tikzcd}$$

reify is a left inverse of $[\![-]\!]$ - the "inverse of the evaluation function":

$$\mathit{nbe}\ a = \mathit{reify}\ [\![a]\!]$$

Moreover, we are doing *metaprogramming*: both syntax and model are represented as data structures in a computer! We are doing *constructive metamathematics* - it looks like maths but is actually programming ...

Example of a model

Let *Exp* be a type of arithmetic expressions with free variables

$$Exp \xrightarrow{[\![-]\!]} (Var \to Int) \to Int$$

We would like to perform some magic! Write a function *reify* which extracts a normal form from the meaning:

$$\begin{tikzcd} Exp \arrow[r, shift left, "{[\![-]\!]}"] & (Var \to Int) \to Int \arrow[l, shift left, "reify"] \end{tikzcd}$$

Is this really possible?? Perhaps not ...
Before explaining the magic, let's look at more examples of "secondary school simplification"!

Partial evaluation - program simplification

Let us define $\mathit{power}\ m\ n = m^n$.

$$\mathit{power} : \mathit{Nat} \rightarrow \mathit{Nat} \rightarrow \mathit{Nat}$$

$$\begin{aligned} \mathit{power}\ m\ 0 &= 1 \\ \mathit{power}\ m\ (n+1) &= m * (\mathit{power}\ m\ n) \end{aligned}$$

Let $n = 3$. Simplify by using the reduction rules for power, $*$, and $+$:

$$\mathit{power}\ m\ 3 = m * (m * m)$$

$m * (m * m)$ is the normal form (the "residual program") of $\mathit{power}\ m\ 3$.

Normalization of types in dependent type theory

In Martin-Löf's intuitionisitic type theory we can define the type-valued function $Power\ a\ n = a^n$. Let *Set* be the type of *sets* or *small types*. (The "ordinary types" are small, but *Set* itself is a *large* type.)

$$Power : Set \rightarrow Nat \rightarrow Set$$

$$\begin{array}{rcll} Power\ a\ 0 & = & Unit & -\ a\ one\ element\ type \\ Power\ a\ (n+1) & = & a \times (Power\ a\ n) & -\ a\ product\ type \end{array}$$

Let $n = 3$. Simplify by using the reduction rules for *Power*:

$$Power\ a\ 3 = a \times (a \times (a \times Unit))$$

$a \times (a \times (a \times Unit))$ is the normal form of the type $Power\ a\ 3$; it is a *normal type*. Can we simplify further?

Agda - an implementation of a dependent type theory

- is a functional programming language with *dependent* types
- is also a language for formalizing *constructive* mathematics, a cousin of the *Coq*-system developed at INRIA in France.
- is based on intuitionistic type theory, and extends it with a number of programming language features:
    - definitions of new *data types* a la Haskell and ML, but with dependent types including *inductive families* and *inductive-recursive* definitions
    - a general form of *pattern matching* with dependent types
    - a fairly powerful *termination checker*
    - an emacs-interface which allows the successive refinement of programs and proofs while maintaining type-correctness
- is described in more detail on the *Agda wiki*.

Normalization during type-checking

To check that

$$(2009, (3, (30, ()))) : \mathit{Power}\ \mathit{Nat}\ 3$$

we need to normalize the type:

$$(2009, (3, (30, ()))) : \mathit{Nat} \times (\mathit{Nat} \times (\mathit{Nat} \times 1))$$

Normalization (by evaluation) is used in proof assistants for intuitionistic type theory (Coq, Agda, Epigram, ...). An important application!

Evaluation, partial evaluation and normalization

Evaluation: to simplify a closed term, a complete program where all inputs are given.

Partial evaluation: (from programming languages) to simplify code using the knowledge that some of the inputs are known. The purpose is to get more efficient code.

Normalization: (from proof theory) to simplify a proof or a term, including open terms. Normalization is among other things used during type-checking in proof assistants based on intensional type theory such as *Agda* and *Coq*.

Plan for the lectures

Normalization in monoids. A simple yet "deep" example, connection with algebra and category theory.

Normalization in typed combinatory logic. Historically, the first example of nbe, simpler because no variables. Curry-Howard. Program extraction from constructive proof.

Normalization in untyped combinatory logic. Computing lazy Böhm trees. Neighbourhoods of programs.

Normalization in the simply typed lambda calculus. The Berger-Schwichtenberg algorithm. De Bruijn's nameless dummies. Higher-order abstract syntax.

Normalization in the dependently typed lambda calculus.

Normalization and foundations.

Why should you care?

Why should you care?

- A new approach to normalization: "reduction-free" instead of "reduction-based". Central topic in cs and proof theory.

Why should you care?

- A new approach to normalization: ”reduction-free” instead of ”reduction-based”. Central topic in cs and proof theory.
- A case study in constructive thinking!
  Constructive metamathematics = metaprogramming.

Why should you care?

- A new approach to normalization: "reduction-free" instead of "reduction-based". Central topic in cs and proof theory.
- A case study in constructive thinking!
  Constructive metamathematics = metaprogramming.
- A functional programming exercise ...

Why should you care?

- A new approach to normalization: "reduction-free" instead of "reduction-based". Central topic in cs and proof theory.
- A case study in constructive thinking!
  Constructive metamathematics = metaprogramming.
- A functional programming exercise ...
- ... with dependent types :-)

Why should you care?

- A new approach to normalization: "reduction-free" instead of "reduction-based". Central topic in cs and proof theory.
- A case study in constructive thinking!
  Constructive metamathematics = metaprogramming.
- A functional programming exercise ...
- ... with dependent types :-)
- ... in Agda :-)

Why should you care?

- A new approach to normalization: "reduction-free" instead of "reduction-based". Central topic in cs and proof theory.
- A case study in constructive thinking!
  Constructive metamathematics = metaprogramming.
- A functional programming exercise ...
- ... with dependent types :-)
- ... in Agda :-)
- ... and Haskell

Why should you care?

- A new approach to normalization: "reduction-free" instead of "reduction-based". Central topic in cs and proof theory.
- A case study in constructive thinking!
  Constructive metamathematics = metaprogramming.
- A functional programming exercise ...
- ... with dependent types :-)
- ... in Agda :-)
- ... and Haskell
- Relates to, and applies the knowledge of many of the other courses.

Why should you care?

- A new approach to normalization: "reduction-free" instead of "reduction-based". Central topic in cs and proof theory.
- A case study in constructive thinking!
  Constructive metamathematics = metaprogramming.
- A functional programming exercise ...
- ... with dependent types :-)
- ... in Agda :-)
- ... and Haskell
- Relates to, and applies the knowledge of many of the other courses.
- Has (perhaps) foundational significance: interplay between "meta" and "object" level.

Constructivism in practice

Georges Gonthier: A computer-checked proof of the four colour theorem:

> *The approach that proved successful for this proof was to turn almost every mathematical concept into a data structure or a program, thereby converting the entire enterprise into one of program verification.*

I. Monoids

- A warm-up example: how to normalize monoid expressions!
- A very simple program with some interesting mathematics (algebra, category theory)
- Illustrates some of the underlying principles behind the normalization by evaluation technique.

Monoid expressions

The set *Exp a* of *monoid expressions* with atoms in a set *a* is generated by the following grammar:

$$e ::= (e \circ e) \mid id \mid x$$

where x is an atom. Cf Lisp's S-expressions:

$$e ::= (e.e) \mid NIL \mid x$$

The free monoid

The *free monoid* is obtained by identifying expressions which can be proved to be equal from the associativity and identity laws:

$$\begin{aligned} (e \circ e') \circ e'' &\sim e \circ (e' \circ e'') \\ id \circ e &\sim e \\ e \circ id &\sim e \end{aligned}$$

We call the relation $\sim$ *convertibility* or *provable equality*. Note that it is a congruence relation (equivalence relation and substitutive under the $\circ$ sign).
The distinction between *real* and *provable* equality is crucial to our enterprise!
(Strictly speaking we should say *a* free monoid, since any monoid isomorphic to the above is a free monoid.)

Normalization of monoid expressions

What does it mean to normalize a monoid expression?

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

Nbe/reduction-free view: Find unique representative from each $\sim$-equivalence class! A way to solve the decision problem, write a program which decides whether $e \sim e'$!

How to solve the decision problem for equality?

Given two monoid expressions e and e', is there an algorithm to decide whether $e \sim e'$?

The mathematician's answer: "Just shuffle the parentheses to the right, remove the identities and check whether the resulting expressions are equal".

The programmer's objection: "Yes, but how do you implement this in an elegant way, so that the correctness proof is clear?"

The programmer's answer

$$[\![-]\!] : Exp\ a \to [a]$$

$$\begin{array}{rcl} [\![e \circ e']\!] & = & [\![e]\!] +\!\!+ [\![e']\!] \\ [\![id]\!] & = & [\,] \\ [\![x]\!] & = & [x] \end{array}$$

$$\sim\ : Exp\ a \to Exp\ a \to Bool$$

$$e \sim e' \quad = \quad [\![e]\!] == [\![e']\!]$$

Normal forms as expressions

The lists are here "normal forms", except usually we want our normal forms to be special expressions. Hence we represent lists as right-leaning expression trees (cf Lisp):

$$reify : [a] \longrightarrow Exp\ a$$

$$\begin{array}{rcl} reify\ [\,] & = & id \\ reify\ (x :: xs) & = & x \circ (reify\ xs) \end{array}$$

Here we have syntax = tree, meaning = list ... seems like cheating!

A real interpretation - no cheating!

Alternatively, we can interpret monoid expressions as functions (the "intended" meaning!)

$$[\![-]\!] : \mathit{Exp}\ a \to (\mathit{Exp}\ a \to \mathit{Exp}\ a)$$

$$\begin{aligned} [\![e \circ e']\!]e'' &= [\![e]\!]([\![e']\!]e'') \\ [\![id]\!]e'' &= e'' \\ [\![x]\!]e'' &= x \circ e'' \end{aligned}$$

Can we compare functions for equality?

A real interpretation - no cheating!

Alternatively, we can interpret monoid expressions as functions (the "intended" meaning!)

$$[\![-]\!] : Exp\ a \to (Exp\ a \to Exp\ a)$$

$$\begin{array}{rcl} [\![e \circ e']\!]e'' & = & [\![e]\!]([\![e']\!]e'') \\ [\![id]\!]e'' & = & e'' \\ [\![x]\!]e'' & = & x \circ e'' \end{array}$$

Can we compare functions for equality? No, not in general. However, let's try to turn functions into expressions:

$$reify : (Exp\ a \to Exp\ a) \to Exp\ a$$

$$reify\ f = f\ id$$

Correctness property

The aim of the function

$$nbe : Exp\ a \longrightarrow Exp\ a$$

$$nbe\ e = reify\ [\![e]\!]$$

is to pick out unique representatives from each equivalence class:

$$e \sim e'\ \ iff\ \ nbe\ e = nbe\ e'!$$

Prove this!

Correctness proof

if-direction. Prove that

$$e \sim e' \textit{ implies } nbe\ e = nbe\ e'!$$

Lemma: prove that

$$e \sim e' \textit{ implies } [\![e]\!] = [\![e']\!].$$

Straightforward proof by induction on $\sim$ (convertibility).

only if-direction. It suffices to prove

$$e \sim nbe\ e.$$

Because if we assume $nbe\ e = nbe\ e'$, then

$$e \sim nbe\ e = nbe\ e' \sim e'$$

Correctness proof, continued

To prove

$$e \sim nbe\ e.$$

we prove the following lemma

$$e \circ e' \sim [\![e]\!] e'.$$

(Then put $e' = id$). Proof by induction on e! All cases are easy, the identity follows from the identity law, atoms are definitional identities, composition follows from associativity.

What makes the proof work?

1. A "representation theorem": "Each monoid is isomorphic to *a* monoid of functions" (cf Cayley's theorem in group theory and the Yoneda lemma in category theory).
2. The monoid of functions is "strict" in the sense that equal elements are extensionally equal functions, whereas the syntactic monoid has a conventionally defined equality. The functions are sort of "normal forms".

Cayley's theorem in group theory

Theorem (Cayley). Every group is isomorphic to a group of permutations.
"The theorem enables us to exhibit any *abstract group* in terms of something more *concrete*, namely, as a group of mappings."
(Herstein, Topics in Algebra, p 61).

Cayley's theorem for monoids

Theorem. Every monoid is isomorphic to a monoid of functions.
Proof. Let M be a monoid. Consider the homomorphic embedding

$$\begin{tikzcd} M \arrow[r, shift left, "e \mapsto \lambda e'.e \circ e'"] & M \to M \arrow[l, shift left, "f \mapsto f\ id"] \end{tikzcd}$$

Thus M is isomorphic to the submonoid of functions which are in the image of the embedding.

Nbe and Cayley's theorem for monoids

Consider now the special case that $M = Exp\ a/\sim$, the free monoid of monoid expressions up to associativity and identity laws. In this case we proved that

$$e \circ e' \sim [\![e]\!]e'.$$

Hence, the embedding that we used for nbe

$$\begin{tikzcd} M \arrow[r, shift left, "{[\![-]\!]}"] & M \to M \arrow[l, shift left, "reify"] \end{tikzcd}$$

is the same as the one in Cayley's theorem for monoids!

$$\begin{tikzcd} M \arrow[r, shift left, "e \mapsto \lambda e'.e \circ e'"] & M \to M \arrow[l, shift left, "f \mapsto f\ id"] \end{tikzcd}$$

But can we normalize with the latter? (Try it!)

A role for constructive glasses

Answer: no, because

$$e \circ e' \sim [\![e]\!] e'.$$

does not mean that the results are *identical* expressions, they are only *convertible*, that is, *equal up to associativity and identity laws*. But this fact is invisible if we render the free monoid as a quotient in the classical sense! The equivalence classes hide the representatives.

Classical quotients and constructive setoids

- In constructive mathematics (at least in type theory) one does not form quotients.
- Instead one uses *setoids*, that is, pairs $(M, \sim)$ of *constructive sets* and *equivalence relations* $\sim$. And constructive "sets" are the same as data types in functional languages (more or less).
- Constructively, one defines a *monoid* as a setoid $(M, \sim)$ together with a binary operation $\circ$ on M which preserves $\sim$ and which has an identity and is associative up to $\sim$.
- Note that some setoids (and monoids) are *"strict"* in the sense that $\sim$ is the underlying (extensional) *identity* on the underlying sets. The monoid of functions is strict in this sense, and this is what makes the nbe-technique work!! This is reminiscent of a "coherence theorem" in category theory: each monoidal category is equivalent to a strict monoidal category (Gordon, Power, Street)

Strict and non-strict monoids

$(M \to M, =)$ is a *strict* monoid.
$(M, \sim)$ and $(M \to M, \sim)$ are *non-strict*.
Suggestive terminology?

$\sim$	$=$
non-strict	strict
abstract	concrete
syntactic	semantic
formal	real
static	dynamic

Compare category theory: $\cong$ vs $=$!

The Yoneda lemma - special case for monoids

The *Yoneda lemma* is a theorem which generalizes Cayley's theorem for monoids to categories. It also characterizes the submonoid of functions.
A monoid is a category with one object. The Yoneda embedding is an isomorphism which restricts the Cayley embedding:

$$\begin{tikzcd} M \arrow[r, shift left, "e \mapsto \lambda e'.e \circ e'"] & \{f : M \to M | f \ natural\} \arrow[l, shift left, "f \mapsto f\ id"] \end{tikzcd}$$

Naturality means that f commutes with composition to the right:

$$f(e' \circ e'') \sim (f\ e') \circ e''$$

The general condition in category theory is that f is a *natural transformation*.

What did we learn from this?

- The mathematics of a simple program for "shuffling parentheses".
- The normalization algorithm exploits the fact that monoid expressions really denote functions. The expressions are in one-to-one correspondence with certain well-behaved "endo-functions" (in fact the "natural transformations").
- The situation is more complex but fundamentally analogous for the simply typed lambda calculus, when analyzed categorically as a representation of the free cartesian closed category. Cf Cubric, Dybjer, Scott 1997: "Normalization and the Yoneda embedding" and Altenkirch, Hofmann, Streicher 1995: "Categorical reconstruction of a reduction-free normalization proof".

Exercises

- The nbe-algorithm for monoids (the version that interprets expressions as functions on expressions) returns right-leaning trees as normal forms. Change it so that it returns left-leaning trees instead!
- Rewrite the algorithm so that the model is $[a] \rightarrow [a]$ instead of $Exp\ a \rightarrow Exp\ a$! Why are elements of $[a]$ suitable as representations of the normal forms in $Exp\ a$?
- Why is it possible to write a "generic" nbe-algorithm for normalizing elements in an arbitrary free monoid and also use this to decide equality? This assumes that the free monoid in question is presented "constructively". Discuss exactly what is required! Assume you have such a generic nbe-algorithm. What does it do for the free monoid $[a]$ of lists?
- Work out the details on paper of the proof of correctness for the nbe-algorithm for monoids.

Exercises

- Consider the monoid laws as left-to-right rewrite rules. Prove that each term has a unique normal form with respect to this rewrite rule system! Hint: prove that the system is terminating and confluent!
- Explain why the nbe-program does not return normal forms in the sense of the rewrite system!
- One can use the nbe-technique for getting an alternative proof of uniqueness of normal forms for the rewrite rule system. First, modify the nbe-algorithm so that it returns normal forms in the sense of the rewrite rule system! Then prove that *e* reduces to *nbe e* using a similar technique as in the correctness proof for nbe.

II. Typed combinators

- Typed combinatory logic; historically the first version of nbe (Martin-Löf 1973).
- Simpler than the typed lambda calculus because variable-free
- Add natural numbers and primitive recursion and we get Gödel system T, an expressive language where all programs terminate
- Discuss the traditional approach to normalization via rewriting and the "reduction-free" approach of nbe
- Program extraction from constructive proof

The power example in the typed lambda calculus with natural numbers (Gödel system T)

Recall the program *power*:

$$\begin{aligned} power\ m\ 0 &= 1 \\ power\ m\ (n+1) &= m * (power\ m\ n) \end{aligned}$$

This can be written in *Gödel system T* - the simply typed lambda calculus with natural numbers and a primitive recursion combinator *rec*:

$$power = \lambda m.\lambda n.rec\ 1\ (\lambda xy.m * y)\ n$$

Gödel system T based on the lambda calculus

Grammar for types and terms of Gödel system T:

$$\begin{array}{rcl} a & ::= & a \rightarrow a \mid Nat \\ e & ::= & x \mid e\ e \mid \lambda x.e \mid 0 \mid succ \mid rec \end{array}$$

We have the typing and reduction rules (β and η reduction) for the simply typed lambda calculus. The natural number constructors have the following types:

$$\begin{array}{rcl} 0 & : & Nat \\ succ & : & Nat \rightarrow Nat \end{array}$$

Types and recursion equations for the primitive recursion combinator:

$$rec \quad : \quad a \rightarrow (Nat \rightarrow a \rightarrow a) \rightarrow Nat \rightarrow a$$

$$\begin{array}{rcl} rec\ e\ f\ 0 & \sim & e \\ rec\ e\ f\ (succ\ n) & \sim & f\ n\ (rec\ e\ f\ n) \end{array}$$

History of nbe

We will postpone the treatment of lambda calculus version of Gödel's T and instead begin with a combinatory version. Historically earlier and conceptually simpler:

- Martin-Löf 1973: combinatory version of intuitionistic type theory (variation of Tait's reducibility method)
- Berger and Schwichtenberg 1991: simply typed lambda calculus with eta long normal forms. Used for the Minlog system implemented in Scheme.
- Coquand and Dybjer 1993: implementation of combinatory nbe in Alf system, data types, formal correctness proof.
- Danvy 1994: application of nbe to type-directed partial evaluation; nbe for non-terminating programs
- Coquand: application of nbe to type-checking dependent types
- ... variety of systems, categorical aspects, ...

Gödel system T based on combinators

A grammar for the types and terms of combinatory Gödel system T:

$$\begin{array}{rcl} a & ::= & a \rightarrow a \mid \mathit{Nat} \\ e & ::= & e\ e \mid K \mid S \mid 0 \mid \mathit{succ} \mid \mathit{rec} \end{array}$$

Type schemata:

$$\begin{array}{rcl} K & : & a \rightarrow b \rightarrow a \\ S & : & (a \rightarrow b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c \end{array}$$

Conversion rules:

$$\begin{array}{rcl} K\ x\ y & \sim & x \\ S\ x\ y\ z & \sim & x\ z\ (y\ z) \end{array}$$

Type schemata and reduction rules for $0, \mathit{succ}$, and rec as before.

Schönfinkel and Curry

Schönfinkel 1924 introduced combinators S, K, I, B, C,(and U) to show that it was possible to eliminate variables from logic.

$$\begin{array}{rcl} K & : & a \to b \to a \\ S & : & (a \to b \to c) \to (a \to b) \to a \to c \\ I & : & a \to a \\ B & : & (b \to c) \to (a \to b) \to a \to c \\ C & : & (a \to b \to c) \to b \to a \to c \end{array}$$

He also showed that I, B, C could be defined in terms of S and K. We have

$$g \circ f = B\ g\ f$$

Curry developed combinatory logic during several decades from the 1930s and onwards. He also noticed that the types of the combinators corresponded to axioms of minimal (implicational) logic.

The Curry-Howard correspondence

- type - proposition
- combinator - name of axiom
- term - proof
- expression reduction - proof simplification ("normalization")

Howard 1969 introduced dependent types and extended this correspondence to formulas in predicate logic.
Martin-Löf 1971, 1972 (cf also Scott 1970) extended this correspondence to inductively defined sets and predicates. This is the basis for his *intuitionistic type theory*.

Bracket abstraction

An algorithm for translating lambda calculus to combinatory logic:

$$
\begin{array}{rcl}
T[x] & = & x \\
T[(e_1\ e_2)] & = & (T[e_1]\ T[e_2]) \\
T[\lambda x.E] & = & (K\ T[E])\ \ (\mathit{if}\ x\ \mathit{is\ not\ free\ in}\ E) \\
T[\lambda x.x] & = & I \\
T[\lambda x.\lambda y.E] & = & T[\lambda x.T[\lambda y.E]]\ \ (\mathit{if}\ x\ \mathit{is\ free\ in}\ E) \\
T[\lambda x.(e_1\ e_2)] & = & (S\ T[\lambda x.e_1] T[\lambda x.e_2])\ \ (\mathit{if}\ x\ \mathit{is\ free\ in\ both}\ e_1\ \mathit{and}\ e_2 \\
T[\lambda x.(e_1\ e_2)] & = & (C\ T[\lambda x.e_1]\ T[e_2])\ \ (\mathit{if}\ x\ \mathit{is\ free\ in}\ e_1\ \mathit{but\ not}\ e_2) \\
T[\lambda x.(e_1\ e_2)] & = & (B\ T[e_1]\ T[\lambda x.e_2])\ \ (\mathit{if}\ x\ \mathit{is\ free\ in}\ e_2\ \mathit{but\ not}\ e_1)
\end{array}
$$

The power function in combinatory system T

$$\begin{aligned} add\ m\ n &= rec\ m\ (K\ succ)\ n \\ mult\ m\ n &= rec\ 0\ (K\ (add\ m))\ n \\ power\ m\ n &= rec\ 1\ (K\ (mult\ m))\ n \end{aligned}$$

Hence:

$$\begin{aligned} power &= \lambda m.rec\ 1\ (K\ (mult\ m)) \\ &= (rec\ 1) \circ (\lambda m.K\ (mult\ m)) \quad - compose\ rule \\ &= (rec\ 1) \circ (K \circ mult) \quad - compose\ rule + eta \end{aligned}$$

Exercise: reduce power m 3 using the reduction rules for power!

Normalization and normalization by evaluation

We shall now normalize expressions (programs) in Gödel system T!
As for monoids we have two approaches

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

Nbe/reduction-free view: Find unique representative from each $\sim$-equivalence class! class! A way to solve the decision problem, write a program which decides whether $e \sim e'$!

Normalization as analysis of a binary relation of one step reduction

Note: Turing-machines have a next state *function* but lambda calculus and combinatory logic have next state *relations* because several possible reduction strategies.
History of normalization in logic:

- Proof simplification: (Gentzen) cut-elimination; consistency proofs
- Normalization of lambda terms (Church)
- The simply typed lambda calculus (Church 1940), weak normalization theorem (Turing)

Reduction to normal form - some terminology

- e *is a normal form* iff e is irreducible: there is no e' such that $e \ \mathtt{red}_1 \ e'$.
- e *has normal form* e' iff $e \ \mathtt{red} \ e'$ and e' *is a normal form*, where $\mathtt{red}$ is n-step reduction, the transitive and reflexive closure of $\mathtt{red}_1$.
- $\mathtt{red}_1$ is *weakly normalizing* if all terms have normal form.
- $\mathtt{red}_1$ is *strongly normalizing* if $\mathtt{red}_1$ is a well-founded relation, that is, there is no infinite sequence:

  $$e \ \mathtt{red}_1 \ e_1 \ \mathtt{red}_1 \ e_2 \ \mathtt{red}_1 \ \cdots$$

  ad infinitum.

Confluence

`red` is *Church-Rosser* iff e `red` e_0 and e `red` e_1 implies that there is e_2 such that

$$\begin{tikzcd} & e \arrow[dl, "\mathtt{red}"'] \arrow[dr, "\mathtt{red}"] & \\ e_0 \arrow[dr, "\mathtt{red}"'] & & e_1 \arrow[dl, "\mathtt{red}"] \\ & e_2 & \end{tikzcd}$$

Church-Rosser implies uniqueness of normal forms: If e has normal forms e_0 and e_1, then $e_0 = e_1$.

The decision problem for conversion

- Convertibility $\sim$ is the least congruence relation containing $\mathtt{red}_1$.
- Weak normalization plus Church-Rosser of `red` yields solution of decision problem for convertibility (provided there is an effective reduction strategy which always reaches the normal form).

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

$$WN_a(e) = \exists e' : a . e \; red \; e' \;\&\; Normal(e')$$

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 $e \; red \; e'$ and $Normal(e')$. (This algorithm simultaneously manipulates terms and *proof objects*, but we can perform *program extraction* from this constructive proof and eliminate the proof objects.)

Tait's reducibility method

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.

$$\begin{aligned} Red_{Nat}(e) &= WN_{Nat}(e) \\ Red_{a \to b}(e) &= WN_{a \to b}(e) \;\&\; \forall e' : a . Red_a(e') \supset Red_b(e\ e') \end{aligned}$$

One then proves that

$$\forall e : a . Red_a(e)$$

by induction on e.

Martin-Löf's version of Tait's proof

The constructive proof of

$$\forall e : a.Red_a(e)$$

is an algorithm which for all e computes a proof-object for $Red_a(e)$.

- In the base case $a = Nat$ such a proof object is a triple (e', p, q), where e' is a normal term, p is a proof that e *red* e' and q is a proof that e' is normal.
- In the function case $a = b \rightarrow c$ such a proof object has the form $((e', p, q), r)$, where the triple (e', p, q) is as above, and r is a proof that e maps reducible arguments to reducible results.

Program extraction by removing proof objects

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

$$\begin{array}{rcl} Red_{Nat}(e) & = & WN_{Nat}(e) \\ Red_{a\to b}(e) & = & WN_{a\to b}(e) \,\&\, \forall e' : a. Red_a(e') \supset Red_b(e\ e') \end{array}$$

is thus simplified to

$$\begin{array}{rcl} [\![Nat]\!] & = & Exp_{Nat} \\ [\![a \to b]\!] & = & Exp_{a\to b} \times ([\![a]\!] \to [\![b]\!]) \end{array}$$

where Exp_a is the type of expressions of type a.

Formalizing typed combinatory logic in Martin-Löf type theory

Note that the evaluation function $[\![-]\!]_a : Exp_a \rightarrow [\![a]\!]$ is indexed by the type a of the object language (typed combinatory logic). It is a *dependent type*! Let's program it in Martin-Löf type theory.
We have a small type $Ty : Set$ of object language types. Its constructors are.

$$\begin{array}{lcl} \mathrm{Nat} & : & Ty \\ (\Rightarrow) & : & Ty \rightarrow Ty \rightarrow Ty \end{array}$$

We here use $\Rightarrow$ for *object language (Gödel's T)* function space to distinguish it from *meta language (Martin-Löf type theory)* function space $\rightarrow$.

The inductive family of expressions indexed by types

Constructors for $Exp : Ty \to Set$:

$$\begin{aligned}
\mathsf{K} &: (a, b : Ty) \to Exp\ (a \Rightarrow b \Rightarrow a) \\
\mathsf{S} &: (a, b, c : Ty) \to Exp\ ((a \Rightarrow b \Rightarrow c) \Rightarrow (a \Rightarrow b) \Rightarrow a \Rightarrow c) \\
\mathsf{App} &: (a, b : Ty) \to Exp\ (a \Rightarrow b) \to Exp\ a \to Exp\ b
\end{aligned}$$

In this way we only generate well-typed terms. *Exp* is often called an *inductive family*.

Exercise. Add constructors for 0, succ, rec!

Intended semantics

Just translate object language notions into corresponding meta language notions:

$$
\begin{aligned}
[\![\mathsf{Nat}]\!] &= \mathit{Nat} \\
[\![a \Rightarrow b]\!] &= [\![a]\!] \to [\![b]\!]
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathsf{K}]\!] &= \lambda xy.x \\
[\![\mathsf{S}]\!] &= \lambda xyz.x\ z\ (y\ z) \\
[\![\mathsf{App}\ f\ e]\!] &= [\![f]\!]\ [\![e]\!] \\
[\![\mathsf{Zero}]\!] &= 0 \\
[\![\mathsf{Succ}]\!] &= \mathit{succ} \\
[\![\mathsf{Rec}]\!] &= \mathit{rec}
\end{aligned}
$$

Note that we have omitted the type arguments of K, S, ...

Glueing and reification

$$\begin{aligned} [\![a \Rightarrow b]\!] &= \mathrm{Exp}\ (a \Rightarrow b) \times ([\![a]\!] \rightarrow [\![b]\!]) \\ [\![\mathrm{Nat}]\!] &= \mathrm{Exp\ Nat} \end{aligned}$$

$$\mathit{reify} : (a : \mathit{Ty}) \rightarrow [\![a]\!] \rightarrow \mathrm{Exp}\ a$$

$$\begin{aligned} \mathit{reify}\ (a \Rightarrow b)\ (c, f) &= c \\ \mathit{reify}\ \mathrm{Nat}\ e &= e \end{aligned}$$

Interpretation of terms

$$\begin{aligned}
[\![a \Rightarrow b]\!] &= \mathrm{Exp}\ (a \Rightarrow b) \times ([\![a]\!] \to [\![b]\!]) \\
[\![\mathrm{Nat}]\!] &= \mathrm{Exp\ Nat}
\end{aligned}$$

$$[\![\ ]\!] : (a : Ty) \to \mathrm{Exp}\ a \to [\![a]\!]$$

$$\begin{aligned}
[\![\mathrm{K}]\!] &= (\mathrm{K}, \lambda p.(\mathrm{App\ K}\ (\mathit{reify}\ \ p), \lambda q.p)) \\
[\![\mathrm{S}]\!] &= (\mathrm{S}, \lambda p.(\mathrm{App\ S}\ (\mathit{reify}\ \ p)), (\ldots, \ldots))) \\
[\![\mathrm{App}\ c\ a]\!] &= \mathit{appsem}\ [\![c]\!]\ [\![a]\!] \\
[\![\mathrm{Zero}]\!] &= \mathrm{Zero} \\
[\![\mathrm{Succ}]\!] &= (\mathrm{Succ}, \lambda e.\mathrm{App\ Succ}\ e) \\
[\![\mathrm{Rec}]\!] &= (\mathrm{Rec}, \lambda p.(\mathrm{App\ Rec}\ (\mathit{reify}\ \ p)), (\ldots, \ldots)))
\end{aligned}$$

where

$$\mathit{appsem}\ (c, f)\ q \quad = \quad f\ \ q$$

A decision procedure for convertibility

$$nbe\ a\ e = reify\ [\![e]\!]_a$$

Let $e, e' : Exp\ a$.

- Prove that $e \sim e'$ implies $[\![e]\!]_a = [\![e']\!]_a$!
- It follows that $e \sim e'$ implies $nbe\ a\ e = nbe\ a\ e'$
- Prove that $e \sim nbe\ a\ e$ using the glueing (reducibility) method!
- Hence $e \sim e'$ iff $nbe\ a\ e = nbe\ a\ e'$

Exercises

- Implement the bracket abstraction algorithm in a functional programming language!
- Reduce the combinatory version of *power* m 3 by hand
- Add the combinators I and B to the combinatory language and extend the nbe-algorithm accordingly!
- What happens if you extend the language with a Y-combinator with the conversion rule $Y\ f \sim f\ (Y\ f)$?
- Extend the language of types on with products $a \times b$! Add combinators for pairing and projections, and the equations for projections. Do not add *surjective pairing*, however. Extend the nbe-algorithm accordingly.
- Similarly, extend the language with sums $a + b$, injections and case analysis combinators, and extend the nbe-algorithm.

Exercises

- Modify the algorithm, so that the clause for natural numbers instead is
$$[[\mathit{Nat}]] = (\mathit{Exp}\ \mathit{Nat}) \times N$$
where N is the type of metalanguage natural numbers!
- Modify the nbe-algorithm so that it returns combinatory head normal forms instead of full normal forms.
- Define the dependent type (inductive family) $\mathit{No}\ a$ of terms in normal forms of type a. Then write an application function
$$\mathit{app} : \{a\ b : \mathit{Ty}\} \to \mathit{No}\ (a \Rightarrow b) \to \mathit{No}\ a \to \mathit{No}\ b$$
Note that $a \Rightarrow b$ is the *object language* function space, whereas $\to$ denotes the *meta language* function space. (The above is Agda syntax, but you can do it on paper.)

Exercises

- Work out the details of the normalization and confluence proofs for the reduction system for typed combinatory logic!
- We explained that nbe arises by extracting an algorithm from a constructive proof of *weak normalization*. What would happen if we instead start with a constructive proof of *strong normalization*? What would such an algorithm return?

III. Untyped combinators

- What happens if we apply our normalization algorithm to untyped combinatory terms?

III. Untyped combinators

- What happens if we apply our normalization algorithm to untyped combinatory terms?
- Not all terms will have normal form, so the algorithm may fail to terminate! Is this interesting?

III. Untyped combinators

- What happens if we apply our normalization algorithm to untyped combinatory terms?
- Not all terms will have normal form, so the algorithm may fail to terminate! Is this interesting?
- This is relevant for type-directed partial evaluation, where one wants to treat languages with non-termination.

III. Untyped combinators

- What happens if we apply our normalization algorithm to untyped combinatory terms?
- Not all terms will have normal form, so the algorithm may fail to terminate! Is this interesting?
- This is relevant for type-directed partial evaluation, where one wants to treat languages with non-termination.
- If we use lazy evaluation the nbe-algorithm computes combinatory Böhm trees (a kind of partial and infinitary notion of normal form)! If the program does not have a "head" normal form, then the Böhm tree is undefined, if it has a normal form, then the Böhm tree is that normal form (drawn as a tree), if an infinite regress of head normal forms are computed then we get an infinite Böhm tree. (The usual notion of Böhm tree is for lambda calculus. Here we use the analogue for combinatory logic.)

Correctness of untyped nbe

- What is correctness criterion for the nbe-program on untyped terms?

Correctness of untyped nbe

- What is correctness criterion for the nbe-program on untyped terms?
- Correspondence between an operational and denotational definition of Böhm trees (computational adequacy theorem)! Nbe gives the denotational definition.

Correctness of untyped nbe

- What is correctness criterion for the nbe-program on untyped terms?
- Correspondence between an operational and denotational definition of Böhm trees (computational adequacy theorem)! Nbe gives the denotational definition.
- Proof uses Scott domain theory in a presentation due to Martin-Löf 1983 (in the style of "formal topology")

Haskell as meta-programming language

- We will now consider a program which may not terminate and we will need a data structure which is not well-founded!

Haskell as meta-programming language

- We will now consider a program which may not terminate and we will need a data structure which is not well-founded!
- In Agda (without "codata") all programs *terminate*, and all data structures are *well-founded trees*.

Haskell as meta-programming language

- We will now consider a program which may not terminate and we will need a data structure which is not well-founded!
- In Agda (without "codata") all programs *terminate*, and all data structures are *well-founded trees*.
- So we'd better not use Agda.

Haskell as meta-programming language

- We will now consider a program which may not terminate and we will need a data structure which is not well-founded!
- In Agda (without "codata") all programs *terminate*, and all data structures are *well-founded trees*.
- So we'd better not use Agda.
- Let's use Haskell instead. The standard lazy functional programming language with general recursion and data types definable by general type equations. Non-termination and non-wellfoundedness are allowed!

Formalizing syntax and semantics in Haskell

The Haskell type of untyped combinatory expressions

```
data Exp = K | S | App Exp Exp | Zero | Succ | Rec
```

(We will later use $e \,@\, e'$ for `App e e'`.)
Note that Haskell types contain programs which do not terminate at all or lazily compute infinite values, such as

```
App K (App K (App K ... ))
```

Semantics of untyped combinators in Haskell

Haskell is a typed lambda calculus, not an untyped one. However, untyped lambda expressions can be modelled by a "reflexive" type (Scott's terminology):

```
data D = Lam (D -> D)


app :: D -> D -> D
app (Lam f) d = f d
```

We can interpret untyped combinators as elements of `D`:

```
eval :: Exp -> D
eval K = Lam (\x -> Lam (\y -> x))
eval S = Lam (\x -> Lam (\y -> Lam (\z ->
         app (app x z) (app y z))))
eval (App e e') = app (eval e) (eval e')
```

The nbe program in Haskell

The untyped glueing model as another reflexive type:

```
data D = Gl Exp (D -> D)
```

We can interpret an untyped combinator in this model

```
reify :: D -> Exp
reify (Gl e f) = e

eval :: Exp -> D
eval K = Gl K (\x -> Gl (App K (reify x))
              (\y -> x))
eval S = Gl S (\x -> Gl (App S (reify x))
              (\y -> Gl (App (App S (reify x)) (reify y))
              (\z -> appD (appD x z) (appD y z))))
eval (App e e') = appD (eval e) (eval e')
```

Exercise. Add clauses for Zero, Succ, Rec!

Application in the model

The semantic application function is

```
appD :: D -> D -> D

appD (Gl e f) x = f x
```

Now we can define the untyped version of the nbe program:

```
nbe :: Exp -> Exp

nbe e = reify (eval e)
```

The nbe program computes the Böhm tree of a term

Theorem. `nbe` e computes the combinatory Böhm tree of e. In particular, `nbe` e computes the normal form of e iff it exists.

- What is the combinatory Böhm tree of an expression? An *operational* notion: the Böhm tree is defined by repeatedly applying the *inductively defined* head normal form relation.
- Note that `nbe` gives a *denotational* (*computational*) definition of the Böhm tree of e
- The theorem is to relate an operational (inductive) and a denotational (computational) definition.

Combinatory head normal form

Inductive definition of relation between terms in $\mathtt{Exp}$

$$\mathsf{K} \Rightarrow^{\mathrm{h}} \mathsf{K} \qquad \mathsf{S} \Rightarrow^{\mathrm{h}} \mathsf{S}$$

$$\frac{e \Rightarrow^{\mathrm{h}} \mathsf{K}}{e \mathbin{@} e' \Rightarrow^{\mathrm{h}} \mathsf{K} \mathbin{@} e'} \qquad \frac{e \Rightarrow^{\mathrm{h}} \mathsf{K} \mathbin{@} e' \qquad e' \Rightarrow^{\mathrm{h}} v}{e \mathbin{@} e'' \Rightarrow^{\mathrm{h}} v}$$

$$\frac{e \Rightarrow^{\mathrm{h}} \mathsf{S}}{e \mathbin{@} e' \Rightarrow^{\mathrm{h}} \mathsf{S} \mathbin{@} e'} \qquad \frac{e \Rightarrow^{\mathrm{h}} \mathsf{S} \mathbin{@} e'}{e \mathbin{@} e'' \Rightarrow^{\mathrm{h}} (\mathsf{S} \mathbin{@} e') \mathbin{@} e''}$$

$$\frac{e \Rightarrow^{\mathrm{h}} (\mathsf{S} \mathbin{@} e') \mathbin{@} e'' \qquad (e' \mathbin{@} e''') \mathbin{@} (e'' \mathbin{@} e''') \Rightarrow^{\mathrm{h}} v}{e \mathbin{@} e''' \Rightarrow^{\mathrm{h}} v}$$

Formal neighbourhoods

To formalize the notion of combinatory Böhm tree we make use of Martin-Löf 1983 - the domain interpretation of type theory (cf intersection type systems). Notions of

- formal neighbourhood $=$ finite approximation of the canonical form of a program (lazily evaluated); in particular Δ means no information about the canonical form of a program.
- The denotation of a program is the set of all formal neighbourhoods approximating its canonical form (applied repeatedly to its parts).
- Remark. Two possibilities: *operational neighbourhoods* and *denotational neighbourhoods*. Different because of the *full abstraction problem*, Plotkin 1976.

Expression neighbourhoods

An expression neighbourhood U is a finite approximation of the canonical form of a program of type $\mathtt{Exp}$. Operationally, U is the set of all programs of type $\mathtt{Exp}$ which approximate the canonical form of the program. Notions of *inclusion* $\supseteq$ and *intersection* $\cap$ of neighbourhoods.

A grammar for expression neighbourhoods:

$$U ::= \Delta \mid \mathsf{K} \mid \mathsf{S} \mid U \mathbin{@} U$$

A grammar for the sublanguage of normal form neighbourhoods:

$$U ::= \Delta \mid \mathsf{K} \mid \mathsf{K} \mathbin{@} U \mid \mathsf{S} \mid \mathsf{S} \mathbin{@} U \mid (\mathsf{S} \mathbin{@} U) \mathbin{@} U$$

Approximations of head normal forms

$$e \rhd^{\mathrm{Bt}} \Delta$$

$$\frac{e \Rightarrow^{\mathrm{h}} \mathsf{K}}{e \rhd^{\mathrm{Bt}} \mathsf{K}} \qquad \frac{e \Rightarrow^{\mathrm{h}} \mathsf{K} \mathbin{@} e' \qquad e' \rhd^{\mathrm{Bt}} U'}{e \rhd^{\mathrm{Bt}} \mathsf{K} \mathbin{@} U'}$$

$$\frac{e \Rightarrow^{\mathrm{h}} \mathsf{S}}{e \rhd^{\mathrm{Bt}} \mathsf{S}} \qquad \frac{e \Rightarrow^{\mathrm{h}} \mathsf{S} \mathbin{@} e' \qquad e' \rhd^{\mathrm{Bt}} U'}{e \rhd^{\mathrm{Bt}} \mathsf{S} \mathbin{@} U'}$$

$$\frac{e \Rightarrow^{\mathrm{h}} (\mathsf{S} \mathbin{@} e') \mathbin{@} e'' \qquad e' \rhd^{\mathrm{Bt}} U' \qquad e'' \rhd^{\mathrm{Bt}} U''}{e \rhd^{\mathrm{Bt}} (\mathsf{S} \mathbin{@} U') \mathbin{@} U''}$$

The Böhm tree of a combinatory expression

The Böhm tree of an expression e in $\mathtt{Exp}$ is the set

$$\alpha = \{U \mid e \triangleright^{\mathrm{Bt}} U\}$$

One can define formal inclusion and formal intersection and prove that α is a *filter* of normal form neighbourhoods:

- $U \in \alpha$ and $U' \supseteq U$ implies $U' \in \alpha$;
- $\Delta \in \alpha$;
- $U, U' \in \alpha$ implies $U \cap U' \in \alpha$.

Denotational semantics: the neighbourhoods the nbe program

$\mathtt{nbe}\, e \in U$ iff U is a finite approximation of the canonical form of $\mathtt{nbe}\, e$ when evaluated lazily. For example,

- $\mathtt{nbe}\, e \in \Delta$, for all e
- $\mathtt{nbe}\, \mathsf{K} \in \mathsf{K}$
- $\mathtt{nbe}\, (\mathsf{Y} @ \mathsf{K}) \in \mathsf{K} @ \Delta$
- $\mathtt{nbe}\, (\mathsf{Y} @ \mathsf{K}) \in \mathsf{K} @ (\mathsf{K} @ \Delta)$, etc

Y is a fixed point combinator.
One can define the neighbourhoods of an arbitrary Haskell program, but we will not do that here. (This is a way of defining the *denotational semantics* of Haskell, following the style of Martin-Löf 1983 and Scott 1981, 1982.) In this way we will define what the neighbourhoods of the nbe program are.

Untyped normalization by evaluation computes Böhm trees

One can now prove, using a variation of Tait reducibility (or glueing) that

$$e \rhd^{\mathrm{Bt}} U \ \text{ iff } \ \mathtt{nbe}\, e \in U$$

The main difficulty is to deal with the *reflexive domain*

```
data D = Gl Exp (D -> D)
```

Remark. This theorem relates an "operational" notion (Böhm tree obtained by repeated head reduction) and a "denotational" notion (the approximations of the nbe program). An *operational adequacy theorem*!

Summary

- Nbe-algorithm for typed combinatory logic generalizes immediately to one for untyped combinatory logic.
- In the typed case it computes normal forms. In the untyped case it computes Böhm trees
- In the typed case the proof falls out naturally in the setting of constructive type theory (a framework for total functions). In the untyped case we need domain theory.
- In the typed case we prove correctness by "glueing" - a variant of Tait-reducibility. In the untyped case we need to adapt the glueing method to work on a "reflexive" domain.

IV. Typed lambda terms

- Simply typed lambda calculus with $\beta\eta$-conversion
- The Berger-Schwichtenberg 1991 algorithm, the most famous of nbe-algorithms, performs η-expansion
- Add natural numbers and primitive recursion and we get another version of Gödel system T
- Haskell implementation uses de Bruijn indices and term families
- Correctness proof using types as partial equivalence relations (pers)

Combinators for natural numbers and primitive recursion

Gödel system T has natural numbers as base types, combinators for zero and successor,

$$\begin{aligned} 0 &: Nat \\ succ &: Nat \rightarrow Nat \end{aligned}$$

and a combinator for primitive recursion:

$$rec_a : a \rightarrow (Nat \rightarrow a \rightarrow a) \rightarrow Nat \rightarrow a$$

$$\begin{aligned} rec_a\ e\ f\ 0 &\sim e \\ rec_a\ e\ f\ (n+1) &\sim f\ n\ (rec_a\ e\ f\ n) \end{aligned}$$

Gödel system T based on the lambda calculus

A (new) grammar for the types and terms of Gödel system T:

$$
\begin{array}{lcl}
a & ::= & a \rightarrow a \mid Nat \\
e & ::= & x \mid e\ e \mid \lambda x : a.e \mid 0 \mid succ\ e \mid rec_a\ e\ e\ e
\end{array}
$$

This grammar differs from the ones given before in the following (minor) ways:

- it is a Church-style definition ($\lambda x : a.e$) rather than Curry-style ($\lambda x.e$);
- *succ* is not a constant, it is a unary operation;
- *rec* is not a constant, it takes 4 arguments;
- the first argument of *rec* is the return *type* of the function.

The power example in the lambda calculus version of Gödel system T

Recall the program *power*:

$$\begin{array}{rcl} power\ m\ 0 & = & 1 \\ power\ m\ (n+1) & = & m * (power\ m\ n) \end{array}$$

This can be written in *Gödel system T* - the simply typed lambda calculus with natural numbers and a primitive recursion combinator *rec*:

$$power\ m\ n = rec_{Nat}\ 1\ (\lambda x : Nat.\lambda y : Nat.m * y)\ n$$

$\beta\eta$-conversion and η-long normal forms

We shall consider the simply typed lambda calculus with β and η conversion.

$$\begin{array}{rcll} (\lambda x : a.e)\ e' & \sim & e[x := e'] & \qquad (\beta) \\ e & \sim & \lambda x : a.e\ x & \qquad (\eta) \end{array}$$

We shall use η *expansion* and produces η-long normal forms, where a normal form of type $a \to b$ always has the form

$$\lambda x : a.e$$

where e is a normal form of type b.
Note that $\beta\eta$-conversion is stronger than the weak conversion of combinatory logic (translated into lambda calculus via bracket abstraction).

The history of the Berger-Schwichtenberg algorithm

Schwichtenberg discovered nbe when implementing his proof system MINLOG. "It was just a very easy way to write a normalizer for the simply typed lambda calculus with $\beta\eta$-conversion". He used the untyped programming language SCHEME and the GENSYM function.

- "An inverse of the evaluation functional" by Berger and Schwichtenberg 1991 is about the pure simply typed lambda calculus with no extra constants and reduction rules.
- Berger 1993 showed how to formally extract the algorithm from a Tait-style normalization proof. Berger used *realizability* semantics of intuitionistic logic.
- Berger, Eberl, Schwichtenberg 1997 showed how to extend the Berger-Schwichtenberg algorithm if you extend the lambda calculus with new constants and reduction rules, like in Gödel system T.

The Berger-Schwichtenberg algorithm

Use the following semantics of types:

$$\begin{aligned} [\![a \Rightarrow b]\!] &= [\![a]\!] \rightarrow [\![b]\!] \\ [\![Nat]\!] &= Exp\ Nat \end{aligned}$$

Note that this is the *standard meaning* of a function space, but a *non-standard meaning* of the base type!
Remark. We had the opposite situation for combinatory logic.
We can then write a meaning function for terms

$$[\![\ ]\!]_a : Env \rightarrow Exp\ a \rightarrow [\![a]\!]$$

where *Env* assigns an element $d_i \in [\![a_i]\!]$ to each variable $x_i : a_i$ which may occur free in the expression.
We will define this evaluation functional later!

Reification: the inverse of the "evaluation functional"

Let's perform some magic! Let's build code from input-output behaviour!

$$\begin{aligned} reify_a &: [\![a]\!] \to Exp\ a \\ reify_{Nat}\ e &= e \\ reify_{a \Rightarrow b}\ f &= \lambda x : a. reify_b\ (f\ (reflect_a\ x)) \end{aligned}$$

Since $f \in [\![a \Rightarrow b]\!] = [\![a]\!] \to [\![b]\!]$, we need an element of the set $[\![a]\!]$ to produce an element of $[\![b]\!]$! But we only have a term of type a: the variable x. We thus need an auxiliary "dual" function

$$\begin{aligned} reflect_a &: Exp\ a \to [\![a]\!] \\ reflect_{Nat}\ e &= e \\ reflect_{a \Rightarrow b}\ e &= \lambda d : [\![a]\!]. reflect_b\ (e\ (reify_a\ d)) \end{aligned}$$

Note however ...

Two issues

- Note that the codes of *reify* and *reflect* are the same except that the roles of terms and values have been exchanged! Note also that we have used the same notation for λ and application in the object and in the metalanguage.
- Note also that we need a GENSYM function for generating the variable x!

$$\begin{array}{rcl} reify_a & : & [\![a]\!] \longrightarrow Exp\ a \\ reify_{Nat}\ e & = & e \\ reify_{a \Rightarrow b}\ f & = & \lambda x : a.reify_b\ (f\ (reflect_a\ x)) \end{array}$$

$$\begin{array}{rcl} reflect_a & : & Exp\ a \longrightarrow [\![a]\!] \\ reflect_{Nat}\ e & = & e \\ reflect_{a \Rightarrow b}\ e & = & \lambda d : [\![a]\!].reflect_b\ (e\ (reify_a\ d)) \end{array}$$

Let's resolve these issues by writing the nbe program in Haskell. (Alternatively, we could use a dependently typed language.)

De Bruijn indices

We shall follow de Bruijn and represent lambda terms using "nameless dummies". The idea is to replace a variable x by a number counting the number of λ-signs one needs to cross (in the abstract syntax tree) before getting to the binding occurence.
If we write v_i for the variable with de Bruijn index i, we represent the lambda term

$$power = \lambda m : Nat.\lambda n : Nat.rec_{Nat}\ 1\ (\lambda x : Nat.\lambda y : Nat.m * y)\ n$$

by the de Bruijn term

$$\lambda Nat.\lambda Nat.rec_{Nat}\ 1\ (\lambda Nat.\lambda Nat.v_3 * v_0)\ v_0$$

Nbe for Gödel System T written in Haskell

Syntax of types

```
data Type = NAT | FUN Type Type
```

Syntax of terms

```
data Term = Var Integer | App Term Term | Lam Type Term
          | Zero | Succ Term | Rec Type Term Term Term
```

where `Var i` is the de Bruijn variable v_i.

An element of the type of Terms

For example:

$$\lambda Nat.\lambda Nat.rec_{Nat}\ 1\ (\lambda Nat.\lambda Nat.v_3 * v_0)\ v_0$$

is represented by the Haskell expression

```
Lam NAT
    (Lam NAT
         (Rec NAT
              (Succ Zero)
              (Lam NAT (Lam NAT (times (Var 3) (Var 0))))
              (Var 0)))
:: Term
```

where `times :: Term -> Term -> Term` represents *.

The GENSYM problem and term families

We will deal with the GENSYM problem by working with *term families* rather than terms. A term family $(a_k)_k : Int \longrightarrow Term$, is a family of de Bruijn terms, which differ only with respect to the "start index. The term a_k has start index k.

Syntactic normal forms

We want to obtain normal forms. It will be useful to consider a grammar for normal forms. Let's write it in Haskell

```
data No =  Lam Type No | Zero | Succ No | Ne Ne
```

where

```
data Ne = Var Integer | App Ne No | Rec Type No No Ne
```

are the *neutral* terms, that is, the normal terms which are not on constructor form, but because reduction got stuck by a variable in the "major" position.

A semantic domain

We would really like to interpret terms of type Nat as normal terms (families) and and terms of function type as functions. If we have dependent types, we can build an appropriate semantic domain for each type. However, when working in Haskell, we need to put all semantic values together in one type (a "universal semantic domain") of normal forms in "higher order abstract syntax":

```
data D = LamD Type (D -> D) -- semantic function
       | ZeroD              -- normal 0
       | SuccD D            -- normal successor
       | NeD TERM           -- neutral term family
```

Term families

```
type TERM = Integer -> Term
```

If `t :: TERM`, then `t k` is a de Bruijn term with indices beginning with `k`.

The semantic domain as normal forms in higher order abstract syntax

Grammar for normal (irreducible terms)

$$t ::= \lambda x : a.t \mid 0 \mid succ\ t \mid s$$

where s ranges over the *neutral* terms:

$$s ::= x \mid s\ t \mid rec_a\ t\ t\ s$$

Note that the semantic domain can be viewed as the normal terms in higher order abstract syntax:

```
data D = LamD Type (D -> D) -- semantic function
       | ZeroD              -- normal 0
       | SuccD D            -- normal successor
       | NeD TERM           -- neutral term family
```

Reification and reflection

We can now omit the type *a* in $reify_a\ e$:

```
reify :: D -> TERM

reify (LamD a f) k
  = Lam a (reify (f (reflect a (freevar (-(k+1))))) (k+1))
reify ZeroD       k = Zero
reify (SuccD d)   k = Succ (reify d k)
reify (NeD t)     k = t k

reflect :: Type -> TERM -> D

reflect (FUN a b) t
  = LamD a (\d -> reflect b (app t (reify d)))
reflect NAT         t = NeD t
```

Interpretation of terms

```
eval :: Term -> (Integer -> D) -> D

eval (Var k)         xi = xi k
eval (App r s)       xi = appD (eval r xi)(eval s xi)
eval (Lam a r)       xi = LamD a (\d -> eval r (ext xi d))
eval (Zero)          xi = ZeroD
eval (Succ r)        xi = SuccD (eval r xi)
eval (Rec c r s t) xi = recD c
                                   (eval r xi)
                                   (eval s xi)
                                   (eval t xi)
```

where we need to define `appD` and `recD`, application and primitive recursion in the model.

Application and primitive recursion in the model

```
appD :: D -> D -> D

appD (LamD a f) d = f d
appD (NeD t)    d = NeD (app t (reify d))

app :: TERM -> TERM -> TERM
app r s k = App (r k) (s k)

recD :: Type -> D -> D -> D -> D

recD c z s ZeroD     = z
recD c z s (SuccD d) = s `appD` d `appD` (recD c d z s)
recD c z s d         = reflect c (Rec c
                                        (reify d)
                                        (reify z)
                                        (reify s))
```

Correctness of the nbe-function

We finally define the normalization function

```
nbe t = reify (eval t idenv) 0
```

where `idenv` is an "identity environment".
Correctness means, as usual, that the nbe-function picks unique representatives from each convertibility class:

$$t \sim_a t' \quad \text{iff} \quad \texttt{nbe}\ t = \texttt{nbe}\ t'$$

And as usual we prove this as a consequence of two lemmas:
Convertible terms have equal normal forms

$$t \sim_a t' \quad \text{implies} \quad \texttt{nbe}\ t = \texttt{nbe}\ t'$$

A term is convertible to its normal form

$$t \sim_a \texttt{nbe}\ t$$

Typed equality of semantic values

Both lemmas are proved by reasoning about the values in the semantic domain `D`. We need for example to prove that

$$t \ \sim_a \ t' \quad \text{implies} \quad \texttt{eval}\ a\ t = \texttt{eval}\ a\ t'$$

But what does "=" mean here? It turns out that we need a typed notion of equality $\approx_a$. This equality will be a *partial equivalence relation (per)* on `D`. Hence we prove

$$t \ \sim_a \ t' \quad \text{implies} \quad \texttt{eval}\ a\ t \approx_a \texttt{eval}\ a\ t'$$

Partial equivalence relations (pers) as types

A per is a symmetric and transitive relation.
A per R does not need to be reflexive. If a R a then a is in the *domain* of R.
A partial setoid is a pair (A,R) where A is a set and R is a per.
Pers and partial setoids are useful for representing "sub-quotients" (quotients on a subset).

Convertibility and syntactic identity of terms

We also use two families of partial equivalence relations on syntactic terms:

- $t \equiv_a t'$, t and t' are identical totally defined terms of type a, where a is a totally defined type. (The per is also indexed by a context Γ which assigns types to the free variables; i e de Bruijn indices, but we omit this.)
- $t \sim_a t'$, t and t' are convertible totally defined terms of type a, where a is a totally defined type.

We can lift these pers to term families.

Semantic types as partial equivalence relations

We introduce a family of partial equivalence relations $\approx_a$ on D such that a term of type a will be interpreted as an element of the domain of $\approx_a$ and two convertible terms of type a will be interpreted as related elements of $\approx_a$.

- The partial equivalence relation for natural numbers is $d \approx_{Nat} d'$ iff there are equivalent normal term families $t \equiv_{Nat} t'$ such that $d = \mathtt{NoD}\ t$ and $d' = \mathtt{NoD}\ t'$.
- The partial equivalence relation for functions is defined by

  $$\mathtt{LamD}\ a\ f \approx_{a \to b} \mathtt{LamD}\ a\ f' \text{ iff } \forall d, d' \in D . d \approx_a d' \supset f\ d \approx_b f'\ d'$$

  (Although we can define partial elements of any type in Haskell we here require that a and b are total elements of the type *Type* of types.)

Nbe maps convertible terms to equal normal forms

We first show that nbe maps convertible terms to equal normal forms (cf Church-Rosser):

$$t \sim_a t' \quad \text{implies} \quad \mathtt{nbe}\ t \equiv_a \mathtt{nbe}\ t'$$

which is an immediate consequence of the following lemmas:

$$
\begin{array}{rcl}
t \sim_a t' & \text{implies} & \xi \approx_\Gamma \xi' \text{ implies } \mathtt{eval}\ t\ \xi \approx_a \mathtt{eval}\ t'\ \xi' \quad (1) \\
d \approx_a d' & \text{implies} & \mathtt{reify}\ d \equiv_a \mathtt{reify}\ d' \quad (2) \\
t \equiv_a t' & \text{implies} & \mathtt{reflect}\ a\ t \approx_a \mathtt{reflect}\ a\ t' \quad (3)
\end{array}
$$

where t and t' are neutral term families in (3). Note that $\equiv_a$ is a relation between term families in (2) and (3).
(1) is proved by induction on the convertibility relation, and (2) and (3) are proved simultaneously by induction on (total) types a.

Nbe preserves convertibility

To prove that

$$t \sim_a \texttt{nbe}\ t$$

we use the method of *logical relations*. We define a family of relations

$$\mathrm{R}_a \subseteq \mathit{TERM} \times D$$

by induction on a, such that we can prove

1. $t\ \mathrm{R}_a\ (\mathit{reflect}\ a\ t)$, for neutral t
2. $t\ \mathrm{R}_a\ d$ implies $t\ \sim_a\ (\mathit{reify}\ d)$
3. $\mathit{ts}\ \mathrm{R}_\Gamma \xi$ implies $\mathit{lift}\ t[\mathit{ts}]\ \mathrm{R}_a\ (\mathit{eval}\ t\ \xi)$

Soundness follows by combining 2 and 3.

V. Dependent types

- Martin-Löf type theory - a dependently typed lambda calculus with $\beta\eta$-conversion
- Now we must normalize both types and terms!
- The nbe-algorithm here is novel research (Martin-Löf 2004; Abel, Aehlig, Dybjer 2007; Abel, Coquand, Dybjer 2007)
- Haskell implementation uses de Bruijn indices and term families
- Towards a transparent correctness proof for the type-checking algorithm for dependent types

Normalization of types in dependent type theory

In Martin-Löf type theory we can define the type-valued function `Power` $a\ n = a^n$. Let U be the type of *small types*:

$$Power : U \to Nat \to U$$

$$\begin{array}{rcll} Power\ a\ 0 & = & 1 & -\ a\ one\ element\ type \\ Power\ a\ (n+1) & = & a \times (Power\ a\ n) & -\ a\ product\ type \end{array}$$

In Martin-Löf type theory 1972 the *Power* program will be represented by the term

$$\lambda a : U.\lambda n : Nat.rec\ U\ \hat{1}\ (\lambda x; Nat.\lambda y : U.a \hat{\times} y)\ n$$

Syntax of a our version of Martin-Löf type theory

We have some new types (for simplicity we omit unit types and product types):

dependent function types (also called Π-types) $(x : a) \rightarrow a$;

the type of small types U;

small types $T\ e$

We also have some new terms: the codes for small types

codes for small types $(x : a)\hat{\rightarrow}a, \mid \hat{Nat}$;

code for the natural number type N

The new grammar is

$$\begin{array}{rcl} a & ::= & (x : a) \rightarrow a \mid a \times a \mid Nat \mid 1 \mid U \mid T\ e \\ e & ::= & x \mid (ee) \mid \lambda x : a.e \mid 0 \mid succ\ e \mid rec\ a\ e\ e\ e\ e \\ & \mid & (x : e)\hat{\rightarrow}e \mid \ \mid a\hat{\times}a \mid \hat{Nat} \mid \hat{1} \end{array}$$

Nbe for Martin-Löf type theory written in Haskell

Syntax of types (types may now depend on term variables)

```
data Type = NAT | FUN Type Type   -- Pi-type
          | U | T Term            -- new types
```

Syntax of terms

```
data Term = Var Integer | App Term Term | Lam Type Term
          | Zero n| Succ Term | Rec Type Term Term Term
          | Nat | Fun Term Term -- new terms (small types)
```

Type and term families

```
type TYPE = Integer -> Type -- type families
type TERM = Integer -> Term
```

Definition of the judgements

We need to define typing and equality judgements. Probably not untyped convertibility. Should equality of terms be indexed by two types?

An element of the type of Types

If we enlarge our universe by adding some more small types

```
data Term = ...
          | Unit | Times Term Term -- more small types
```

then we can represent

$$Power\ m\ n = T\ (rec\ U\ \hat{1}\ (\lambda x : Nat.\lambda y : U.m \hat{*} y)n)$$

by

```
Lam NAT
    (Lam NAT
         (T (Rec U
                 Unit
                 (Lam NAT (Lam NAT (Times (Var 3) (Var 0))
                 (Var 0))))
:: Type
```

Semantic domain for types

```
data DT = FUND DT (D -> DT)  -- semantic function types
        | NATD               -- normal Nat type
        | UD                 -- normal U type
        | NED TYPE           -- neutral type family
```

Neutral types have the form `T t`, where `t` is a neutral term.
In mathematical notation:

$$DT = DT \times (D \to DT) + 1 + 1 + TYPE$$

Semantic domain for terms

```
data D = LamD Type (D -> D)
       | ZeroD
       | SuccD D
       | NatD               -- normal code for N
       | FunD D (D -> D) -- normal code for FUN
       | NeD TERM
```

In mathematical notation:

$$D = DT \times (D \to D) + 1 + D + 1 + D \times (D \to D) + TERM$$

Reification

Reifying terms, also two new clauses for reifying small types

```
reify :: D -> TERM

...

reify (FunD a f) k
  = Fun (reify a k)
        (reify (f (reflect (semt a) (freevar (-(k+1)))))
               (k+1))
reify NatD k = Nat
```

Reflection

Same as before but we have dependent function types

```
reflect :: DT -> TERM -> D

reflect (FUND a f) t =
   LamD a (\ d -> reflect (f d) (app t (reify d)))
reflect _          t = NeD t
```

Reification for types

If we want to normalize type expressions we must be able to reify semantic types.

```
reifyT :: DT -> TYPE

reifyT (FUND a f) k
  = FUN (reifyT a k)
        (reifyT (f (reflect a (freevar (-(k+1))))) (k+1))
reifyT NATD       k = NAT
```

Interpretation of types

```
evalT :: Type -> Valuation -> DT

evalT NAT       xi = NATD
evalT U         xi = UD
evalT (FUN a b) xi = FUND (evalT a xi)
                          (\d -> evalT b (ext xi d))
evalT (T t)     xi = semt (eval t xi)
```

where

```
semt :: D -> DT

semt (FunD a f) = FUND (semt a) (\d -> semt (f d))
semt NatD       = NATD
```

Interpretation of terms

As before, but we must also interpret the small types

```
eval :: Term -> Valuation -> D

eval Nat         xi = NatD
eval (Fun r s) xi = FunD (eval r xi)
                         (\d -> eval s (ext xi d))
```

Semantic types as partial equivalence relations

- As for the case of Gödel System T, we represent semantic types as partial equivalence relations on D.
- However, not all elements of the datatype Type of type expressions are well-formed types, and we will only define partial equivalence relations for the well-formed ones. We therefore define by a simultaneous inductive-recursive definition the well-formed types.
- We will not only define the well-formed types, but also the partial equivalence relation of equivalent well-formed types. This is again given by an inductive-recursive definition together with equivalence of terms of two given equivalent types.

VI. Nbe and foundations

Constructive foundations build on the notion of *evaluation* and *not* on *normalization*. BHK-semantics as refined and extended by Martin-Löf. Type soundness as foundation!

Extensional type theory (Martin-Löf 1979) can be justified by Martin-Löfian semantics (meaning explanations). But it does not have the normalization property and its judgements are not decidable. (Cf NuPRL system)

Intensional type theory (Martin-Löf 1972, 1986; Coquand and Huet 1984) has the normalization property and its judgements (in normal form) are decidable. (Cf Agda, Epigram and Coq)

Nbe and foundations

- Normalization by evaluation is related to Martin-Löfian semantics, but it provides meanings as normal forms also for open expressions. This is not part of the usual 1979/1984 Martin-Löfian meaning explanations.
- The big issue is whether intensional or extensional type theory provides the proper foundation. Decidability is considered important by Martin-Löf and Coquand. It is also a cornerstone of the proof assistants Coq, Agda and Epigram. It makes it possible to use the reflexive tactic.
- Prerequisite: what is Martin-Löfian semantics? What is BHK-semantics?

Meaning explanations based on the evaluation of open expressions

On 19 March 2009 Martin-Löf gave a talk on this topic:

> *Abstract: The informal, or intuitive, semantics of type theory makes it evident that closed expressions of ground type evaluate to head normal form, whereas metamathematics, either the method of computability or the method of normalization by evaluation, is currently needed to show that expressions which are open or of higher type can be reduced to normal form. The question to be discussed is: Would it be possible to modify the informal semantics in such a way that it becomes evident that all expressions, also those that are open or of higher type, can be reduced to full normal form?*