Formal Structures for Computation and Deduction

Gérard Huet

First Edition
May 1986

Preface

This document is the first edition of the notes for a graduate-level course given in the Computer Science Department of Carnegie-Mellon University during the Spring 1986. I thank Dana Scott for inviting me for a stimulating sabbatical year at CMU, and encouraging me to write these notes. I thank all the people who attended my class and contributed by their questions and remarks. Rick Statman deserves special credit for his most pertinent observations.

Many people helped me on these notes. I thank especially Guy Cousineau, Jean-Jacques Lévy, and Thierry Coquand for their significant contribution. The work described here is to a great extent directly issued from the research of the Formel project, and especially of Ascander Suarez, Philippe Le Chenadec, Pierre-Louis Curien, and Jean-Marie Hullot. This research was in turn inspired from previous work of Peter Landin, Robin Milner, Gilles Kahn, Gérard Berry, John Reynolds, Gordon Plotkin, Allan Robinson, Alain Colmerauer, Nicolas de Bruijn, Per Martin-Löf, Jean-Yves Girard, Don Knuth, Dana Scott, Gerhart Gentzen, Haskell Curry, Alonzo Church, Jacques Herbrand and Bertrand Russell.

Pittsburgh
May 1986.

These notes are distributed for comments. Send me your remarks and corrections.

Gérard Huet
Inria
Domaine de Voluceau
78150-Rocquencourt
France

uucp address: mcvax!inria!huet

Introduction

The topics covered include basics of proof theory, such as sequent calculus, equational logic, canonical rewriting and natural deduction, and foundations of applicative programming languages, such as lambda calculus, combinators, sequential computations, types and polymorphism. The course ends with a more advanced section on constructive type theory. The material of the course is completely self-contained.

The main paradigm of the course is the Curry-Howard correspondance between propositions and types. This analogy permits to make sense of a well-typed program as a proof of the proposition which corresponds to its type. Conversely, a constructive proof corresponds to an algorithm, whose specification is the theorem proved.

The course uses this paradigm at the level of the meta-theory, in that the main notions are defined algorithmically, using a functional meta-language. This language `ML` is defined in Chapter 1. The main data-structures for representing the logical concepts of propositions and proofs are defined in Chapter 2: trees, terms, and schemas are obtained by successive refinements. Logic Programming is introduced in Chapter 3. The resolution rule of inference restricted to Horn clauses is explained as a composition rule for polymorphic operators, and the Prolog interpreter is justified through the Principal Type Theorem. Chapter 4 is a digression on constructive methods used to prove the termination of rewrite rules. Term rewriting systems and their application to solving word problems in equational theories are systematically studied in Chapter 5. This theory, originated with the pioneer work of Knuth and Bendix, relies on normalisation. This requirement is dropped in Chapter 6, in favor of conditions of linearity and non-ambiguity of rewrite systems. It is argued that such regular term rewriting systems are computationally meaningful, and a theory of sequential computations is developed, aimed at the design of efficient interpreters for applicative programming languages. Chapter 7 develops rudiments of category theory, insisting on the connection with intuitionistic logic. This paves the way for the study of λ-calculus structures in Chapter 8, and natural deduction in Chapter 9. Polymorphism is studied in Chapter 10 at two levels: first, simple polymorphism of operators typed as schemas, like in `ML` . Then, full polymorphism in the second-order calculus of Girard-Reynolds. Constructive type theory is studied in Chapter 11, through the Calculus of Constructions and various extensions. Finally, Chapter 12 discusses axiomatizations of various inductive notions in non-predicative type theories.

This edition is very unsatisfactory. Completely explicit `ML` definitions have been worked out only for the first 5 chapters. Many topics of interest have been omitted for lack of time: full resolution theory, rewriting modulo congruences, implementation techniques of λ-calculus, pure λ-calculus theory including foundations of recursion, Kripke models and Lindenbaum algebras, ordinal notations, modal and temporal axiomatisations are among the ominous omissions. On the other hand, model theory has been purposedly excluded from a pure proof-theoretic study.

This set of notes is aimed at computer scientists rather than mathematicians. It is our thesis that formal elegance is a prerequisite to efficient implementation.

Chapter 1

Functional Programming in CAML

This chapter explains the meta-language `CAML` used in the rest of the course.

1.1 What is `CAML`

1.1.1 ISWIM

The ancestor of ML is ISWIM (If you see what I mean!), a notation devised by P. Landin for describing recursive procedures [18]. ISWIM is a notational variant of λ-calculus with a recursion operator, allowing the description of recursive procedures which accept functional arguments and may return functional results. The "let" construction permits us to state definitions, such as:

```
let I x = x
and K x y = x
and S x y z = (x z (y z)) in (S K K);
```

Basically, the construction `let x = M in N` or its synonym `N where x = M` is equivalent to the λ-calculus redex $(\lambda x \cdot N \quad M)$, except that here the *applicative* order of evaluation is assumed: the expression `M` is evaluated and bound to the identifier `x` *before* the evaluation of the body `N`. This evaluation regime is at variance with the *normal* order of evaluation corresponding to the standardization theorem of λ-calculus, where redexes are reduced in a leftmost-outermost fashion. In the ALGOL terminology, ISWIM procedures evaluate their arguments *by value* and not *by name*. Thus the expression

```
let D x = (x x) and K x y = x in
    let Bottom = (D D) in (K K Bottom);;
```

will loop on the evaluation of `(D D)`, whereas the corresponding λ-expression possesses a normal form.

Remark 1. If you have no acquaintance with the λ-notation, all you need to know at this point is that the expression $\lambda x \cdot M$ denotes the algorithm M with formal parameter x. It is a convenient notation for functions and functionals. In ISWIM we write `fun x -> M`, and thus `let I x = x` is equivalent to `let I = \fun x -> x`. λ-calculus will be studied at length further in the course.

Functional application is noted by juxtaposition. Thus `f x`, `f(x)` and `(f x)` are equivalent expressions. However, application associates to the left. Thus `f x y`, equivalently `(f x y)`, or `((f x) y`, is distinct from `f (x y)`. It is a common mistake to write `f g(x)` and to expect `f(g(x))` instead of `(f(g))(x)`.

Remark 2. Binding. ISWIM is faithful to λ-calculus in its use of variables, with a discipline of *static* binding. That is, the names or *identifiers* used to denote variables do not matter. For instance,

```
let f = let x=1 in fun y -> x
in let x = 2 in f 0;
```

evaluates correctly to 1. This is in sharp contrast with the *dynamic* binding discipline of a language such as LISP [19] which, although syntactically similar to λ-calculus, evaluates

```
(let ((f (let ((x 1)) (function (lambda (y) x)))))
     (let ((x 2)) (apply f (list 0))))
```

incorrectly to 2. This has been however considered an anomaly, and more recent dialects of LISP have converted to static binding [28, 27]. Note also that functions are special types of values in LISP, whence the use of an explicit `apply` operator.

Global variables may however be defined in ISWIM. Evaluating at top-level the declaration

```
let I x = x;;
```

declares the combinator `I` with a global scope extending forever in the future.

Remark 3. It may be worthwhile to point out that the two problems of applicative versus normal order of evaluation and static versus dynamic binding of variables pertain only to the functional (or applicative) features of programming languages. These problems don't have anything to do with imperative features such as assignment, which raise memory allocation and sharing problems which are of no concern to us initially.

Remark 4. Recursion. It is possible to define explicitly a recursion operator in ISWIM, since a fixpoint combinator may be defined as :

```
let Y f = let loop x = f (x x) in loop(loop);;
```

However, this would be of no interest because of the applicative order of evaluation. Instead, a recursion operator is built-in in ISWIM, and we may write `let rec f = M` instead of `let f = Y M`. A similar convention allows `N where rec f = M`.

Thus one may define for instance the iterate functional as (assuming a minimal amount of boolean and integer primitives):

```
let rec power n f x = if n=0 then x else f (power (n-1) f x);;
```

A better (why?) definition would be:

```
let power n f x = pow n
    where rec pow n = if n=0 then x else f (pow (n-1));;
```

1.1.2 LCF's ML

LCF is a proof assistant for a logic for computable functions PPλ due to R. Milner, and modeled after Scott's continuous domain theory. Experience with an early prototype developed in Stanford showed the necessity of a good meta-linguistic capability allowing the user to control his proof strategy by programming. The Edinburgh implementation [14] was thus developed around a meta-language inspired from ISWIM, and called `ML` [13]. `ML` possesses the basic data-types `void`, `bool`, `num` and `string`. A product type-formation operator `#` permits the manipulation of structured data. Thus `(1,true)` is a pair of values, of type `num&bool`. Products are allowed in binders as well. Thus one may define the binary combinator :

```
let app (f,x) = f(x);;
```

Remark that `app` is of any type $((\alpha \to \beta) \times \alpha) \to \beta$, whereas the internal application operator (i.e. functional identity) is of any type $(\alpha \to \beta) \to (\alpha \to \beta)$.

The main innovation consisted in endowing `ML` with a polymorphic type system, allowing variable type identifiers in type expressions (such as α and β above). Actually the `ML` interpreter would answer to the above declaration:

```
Val app = - : (('a -> 'b) & 'a -> 'b)
```

That is, the `*`'s stand for type variables. More importantly, this example shows that, although `ML` has a strong type discipline, the user is not obliged to specify any type for his variables. Instead, the system synthesizes the most general type compatible with proper type consistency of the expression. The justification for such "parametric polymorphism synthesis" will be explained later in the course.

Thus we shall assume the following functions:

```
fst : ('a & 'b -> 'a)   (* first  projection *)
snd : ('a & 'b -> 'b)   (* second projection *)
= : ('a & 'a -> bool)   (* (infix) equality *)
```

The following (non-strict) operators are also provided:

```
if_then_else : (bool & 'a & 'a -> 'a)  (* conditional *)
&  : (bool & bool -> bool) (* infix left-sequential conjunction *)
or : (bool & bool -> bool) (* infix left-sequential disjunction *)
```

1.1.3 The next 700 CAML 's

The initial `ML` language defined in LCF possessed other valuable features, most notably exceptions and abstract data types. For instance the (polymorphic, recursive) abstract data type of lists may be defined, with constructors `nil` and `cons`, discriminator `null` and destructors `hd` and `tl`. This abstract data type facility was crucial for the initial application, since the consistency of the LCF proof system relied on proper encapsulation of the inference rules of PPλ as primitive constructors of the pre-defined `CAML` abstract data type `thm` of the LCF theorems.

Imperative features permitted the use of assignable variables and iteration. However, records and variants were missing from the initial design. Luca Cardelli added them in his `ML` implementation in Pascal [7]. The original LISP implementation was enhanced by Guy Cousineau to allow "concrete" data types of constructor signatures, allowing easy definition of abstract syntax constructions. The resulting `ML` implementation, documented in [1], was used as the basis of the new LCF implementation at Cambridge University [24]. An interface with Yacc, developed by Dave MacQueen and Philippe Le Chenadec, permits the direct manipulation of object language constructions at the level of an (almost) arbitrary concrete syntax.

The `ML` design effort is coordinated by Robin Milner, who initiated a standardization effort [21]. Besides implementations of Standard ML now in progress, several "non standard" dialects are running. The meta-language used in this course is the `CAML` implementation at Inria, based on the Categorical Abstract Machine [10], and documented in [2]. Other interesting related languages are the "lazy" implementations of normal-order evaluating `ML` dialects [17] and of Miranda[29], and the closely related Amber language, which possesses hierarchical types and remanent data values [8]. Research on `ML` -like languages is regularly announced in the Polymorphism Newsletter, edited by Dave MacQueen at Bell Laboratories.

1.2 A few CAML examples

From now on, we assume that the meta-language of the course is CAML Version 2.2 implemented at INRIA, and documented in [2], to which we refer the reader for precise definitions.

1.2.1 Recursion

Here is the standard definition of the factorial function:

```
let rec fact n = if n=0 then 1 else n*fact(n-1);;
```

It is also possible to program factorial iteratively, using the while construct:

```
let fact n = let count=ref n and result=ref 1
in while !count>0 do count,result:=!count-1,!count*!result done;
   !result;;
```

Here is another recursive example, the well-known Hanoi's towers puzzle:

```
let move from to = let p_s=print_string and p_n=print_newline in
p_s "I move a disc from peg ";p_s from;p_s " to peg ";p_s to;p_n ();;

let rec Hanoi from middle to n =
if n>0 then (Hanoi from to middle (n-1);move from to;Hanoi middle from to (n-1));;

Hanoi "A" "B" "C" 3;;
```

Finally, let us show Ackermann's function:

```
let Ack n = A(n,n)
where rec A(n,m) = if n=0 then m+1
                   if m=0 then A(n-1,1)
                   else A(n-1,A(n,m-1));;
```

Note the syntax of the multiple if.

1.2.2 Combinators

Here is a part of the CAML standard prelude file.

```
let equal x y = (x=y)
and pair x y = (x,y);;

let curry f x y = f(x,y)
and uncurry f(x,y) = f x y;;

infix "o";; (* new infixes may be user-defined *)

let op o (f,g)= fun x -> f(g(x));;

(* Here combinators like in Curry & Feys *)
```

```
let I x = x             (* identity *)
and K x y = x           (* the kestrel , or cancellator*)
and C f x y = f y x     (* the cardinal, or permutator *)
and W f x   = f x x     (* the warbler,  or duplicator *)
and B f g x = f (g x)   (* the bluebird, or compositor (curried o) *)
and S f g x = f x (g x) (* the starling *)
and T x f = f x         (* the thrush, or transpositor *)
;;

infix "Co";;
let op Co f g x y = f (g y) x;; (*     permutation-composition   *)
(* named so because  (f Co g) = C (f o g) *)
```

1.2.3 List manipulations

In CAML , the `list` (abstract) type constructor is predefined. A list is noted with square brackets, like:

```
let L1 = [1;2;3];;
```

The empty list is `[]`. The identifier `nil` is initially bound to this value. `null` is the obvious predicate testing equality to `[]`. The list constructor is `cons : 'a & 'a list -> 'a list`. It may also be denoted by a double colon `::` used in infix notation. The destructors are `hd : 'a list -> 'a` and `tl : 'a list -> 'a list`.

Many list operations are pre-defined in CAML . For instance, `append` concatenates two lists, and may be used with an infix syntax, like in:

```
[1;2;3]@[4;5;6] = [1;2;3;4;5;6];
```

The `length` function gives the length of a list. It could have been defined as:

```
let rec length l = if (null l) then 0 else 1+length (tl l);;
```

Here are a few other useful list operators:

```
let rev = revrec []
where rec revrec l = fun
        [] -> l
 |  (x::l') -> revrec (x::l) l';;
let rec flat = fun
        [] -> []
 |  (l::ll) -> l @ (flat ll);;
```

This gives another syntax, for definitions by cases according to the constructors of a type.

The `map` functional maps a function on a list, so that

```
map f [l1; l2 ... ln] = [f l1; f l2; ... f ln].
```

We could have defined `map` as:

```
let rec map f = fun
      [] -> []
 | (h::t) -> (f h) :: map f t;;
```

Note that CAML infers that this functional is doubly polymorphic:

```
Val map = - : (('a -> 'b) -> 'a list -> 'b list).
```

For instance, we get:

```
map fact L1 = [1;2;6];
```

A more general list iterator is `it_list`, which performs iterated compositions of its first argument, as follows:

```
it_list f x [l1; l2 ... ln] = (f ... (f (f x l1) l2) ... ln)
it_list : (('a -> 'b -> 'a) -> 'a -> 'b list -> 'a).
```

It is convenient to be able to iterate the list in the reverse order, using the functional `list_it`:

```
list_it f [l1; l2 ... ln] x = (f l1 (f l2  ... (f ln x)...))
list_it : (('a -> 'b -> 'b) -> 'a list -> 'b -> 'b)
```

`list_it` is an CAML primitive. It could have been defined as:

```
let list_it f l b = itfb l where rec
    itfb = fun [] -> b | (a::l) -> f a (itfb l);;
```

For instance, we could have defined `append` as `list_it cons`;;
We could also have defined:

```
let map f l = list_it (cons o f) l [];;
```

Here is a useful cross between `map` and `it_list`:

```
let num_map f list = let consf (i,l) li = (i+1,(f i li)::l) in
    rev (snd (it_list consf (1,[]) list));;
```

For instance:

```
num_map pair ["a";"b";"c"] = [1,"a"; 2,"b"; 3,"c"]
```

The `it_list` and `list_it` operators are similar to APL's reduce. Here are more examples.

```
let add x y = x+y and mult x y = x*y;;
let sigma = it_list add 0
and pi = it_list mult 1;;
```

Let us define the `range` function:

```
(* range n = [1; 2 ... n] *)
let range = interval 1
where rec interval from to = if from>to then [] else from::(interval (from+1) to);;
```

We may thus define the factorial function in a non-standard way:

```
let fact = pi o range;;
```

Remark that you now have a choice between writing your list processing algorithms in the recursive manner, or using `it_list` in an iterative manner.

Let us now define a function which partitions a list according to a predicate given as its argument. That is, `partition p l` returns a pair `(l1,l2)`, where `l1` (resp. `l2`) consists of the elements of `l` which satisfy `p` (resp. `not p`).

```
let partition p = it_list fork ([],[])
where fork (yes,no) x = if p x then (x::yes),no else yes,(x::no);;
```

For instance, here is the function that discards element `x` from list `l`:

```
let discard x l = snd (partition (equal x) l);;
```

Here is a simple permutation algorithm:

```
let rec permut =
    let perms l x = map (cons x) (permut (discard x l)) in
    fun [] -> [[]]
      |  l -> flat (map (perms l) l);;
```

Now you may scientifically compose your love letters:

```
let love_letter =
permut [" belle marquise";" vos beaux yeux";" me font";" mourir";" d'amour"]
in
(map (fun l -> map print_string l;print_newline()) love_letter);();;
```

This example is due to Pierre Weis, after Molière.

Exercise. Find out what the following function computes.

```
let weird n m =
let Cstar = C I
    in sigma (map (Cstar n) (map (Cstar fact) (map power (range m))));;
```

Same question for:

```
let strange n m =
let f(x,y) = (z,z+y) where z=fact x
    in snd(power m f (n,0));;
```

The reader should carefully study these examples, and try to derive general algebraic laws relating the various combinators introduced. More examples of recursive list operators are given in [2, 5]. A good introduction to recursive programming is Burge [6]. Functional programming application and implementation is explained in Henderson [15]. A good textbook is Abelson-Sussman [3].

1.2.4 Defining an object language in CAML

We now give a simple example where we define an object language in CAML . Our goal is to define a small calculator of arithmetical expressions. This example is due to Philippe Le Chenadec.

First we define an CAML type for the abstract syntax of our language.

```
type exp =    Nat   of num
    | Minus of exp
    | Plus  of exp & exp
    | Diff  of exp & exp
    | Times of exp & exp
    | Quot  of exp & exp
    | Fact  of exp;;
```

Such a declaration defines first a new type, here the type `exp`, and second the various constructors of that type:

```
New constructors declared:
Nat : (num -> exp)
Minus : (exp -> exp)
Plus : ((exp & exp) -> exp)
Diff : ((exp & exp) -> exp)
Times : ((exp & exp) -> exp)
Quot : ((exp & exp) -> exp)
Fact : (exp -> exp)
```

It is convenient to program by cases according to constructors, as in:

```
let rec size = function
  Nat (n)    -> 1
| Minus (x)  -> 1+(size x)
| Plus(x,y)  -> 1+(size x)+(size y)
| Diff(x,y)  -> 1+(size x)+(size y)
| Times(x,y) -> 1+(size x)+(size y)
| Quot(x,y)  -> 1+(size x)+(size y)
| Fact (x)   -> 1+(size x);;
```

We may now define a concrete syntax as a Yacc file [16], containing context-free grammar rules for our object language. The semantic actions associated with each rule are `CAML` expressions specifying the corresponding abstract syntax value. The special identifiers `$i` refer to the semantic value associated with the i-th token in the production.

In this very simple example, we have only one non-terminal symbol `exp`. What follows is the contents of a file named "calc.mly". Note the special syntax allowing binomial coefficients $C(n,p) = \frac{n!}{p!(n-p)!}$:

```
%mlescape
/* C */
%token BINOMIAL
%token NUM
%left '+' '-'
%left '*' ':'
%nonassoc '~'
%%
exp : mlescape {$1}   /* ANTI-QUOTATION */
| NUM {Nat($1)}
```

```
| '~' exp {Minus($2)}
| exp '+' exp {Plus($1,$3)}
| exp '-' exp {Diff($1,$3)}
| exp '*' exp {Times($1,$3)}
| exp ':' exp {Quot($1,$3)}
| exp '!' {Fact($1)}
| BINOMIAL '(' exp ',' exp ')' {Quot(Fact($3),Times(Fact($5),Fact(Diff($3,$5))))}
| '(' exp ')' {$2}
;
%%
```

The concrete syntax is then compiled and loaded:

```
compile_syntax `calc`;;
```

A more detailed description of the CAML to Yacc interface can be found in [2]. We may now use our calculator expressions between quotation marks, as in:

```
let E = "C(4!,3*2-1):2";;
```

The returned CAML value is thus the abstract syntax tree of the above expression (note how the parser has macro-generated the binomial coefficient):

```
E =
Quot (Quot (Fact (Fact (Nat 4)),
            Times (Fact (Diff (Times (Nat 3,Nat 2),Nat 1)),
                   Fact (Diff (Fact (Nat 4),
                               Diff (Times (Nat 3,Nat 2),Nat 1))))),
      Nat 2)
: exp
```

We may now define an interpreter for our language, describing a little arithmetic calculator:

```
let rec calc = function
  Nat (n)    -> n
| Minus (x)  -> - (calc x)
| Plus(x,y)  -> (calc x)+(calc y)
| Diff(x,y)  -> (calc x)-(calc y)
| Times(x,y) -> (calc x)*(calc y)
| Quot(x,y)  -> (calc x)/(calc y)
| Fact (x)   -> fact (calc x);;
Val calc = - : (exp -> num)
```

We may now compute:

```
calc E;;
21252 : num
```

We may also write, using the anti-quote character ^:

```
calc "C(^x,^y-1):2" where x="4!" and y = "3*2";;
```

This facility gives us a powerful macro-expansion mechanism, useful for formal computation.

The next section will develop a more complete example, describing a mini-PASCAL.

1.3 Denotational semantics

This section gives the complete treatment of the syntax and semantics of a small PASCAL-like language. The method of semantic definition is the Scott-Strachey denotational style [25, 20]. A good introductory textbook is Gordon [12]. This example was developed by Guy Cousineau.

1.3.1 Abstract Syntax

```
type       program = Prog of declaration & command

   and     declaration =
                Vardecl of string
              | Procdecl of string & string & string & declaration & command
              | Fundecl of string & string & expression
              | Compdecl of declaration list

   and     command =
                Asscom of string & expression
              | Proccom of string & string & expression
              | Ifcom of  expression & command & command
              | Whilecom of expression  & command
              | Writecom of expression
              | Readcom of string
              | Compcom of command list

   and      expression =
                Varexp of string
              | Funexp of string & expression
              | Unopexp of string & expression
              | Binopexp of expression & string & expression
              | Intconstexp of num
              | Boolconstexp of bool
              | Ifexp of expression & expression & expression;;
```

1.3.2 Concrete Syntax

The syntax of our mini-language follows PASCAL. A certain number of restrictions have been made. For instance, a procedure has always two parameters, the first one called by reference, the second one called by value. There are no arrays or records, and no goto's.

```
/* program procedure function if then else while do write read begin end var := suc */
%token PROGRAM PROCEDURE FUNCTION IF THEN ELSE WHILE DO WRITE READ BEGIN END V ASSYM SUC
%token NUM BOOL IDENT
%right ELSE
%left '+' '-'
%left '*'
%left UMINUS
%%
```

```
program : PROGRAM block '.' {Prog ($2)}
;
block : declaration BEGIN command END {($1,$3)}
;
declaration : declarationl {(fun l -> if null (tl l) then (hd l) else (Compdecl l))$1}
;
declarationl : declelem  {[$1]}
| declelem declarationl  {$1 :: $2}
;
declelem : V IDENT ';' {Vardecl $2}
| PROCEDURE IDENT '(' IDENT ',' IDENT ')' ';' block ';' {Procdecl ($2,$4,$6,$9)}
| FUNCTION IDENT '(' IDENT ')' ';' expression ';' {Fundecl ($2,$4,$7)}
;
command : commandl {(fun l -> if null (tl l) then (hd l) else (Compcom l))$1}
| BEGIN commandl END {(fun l -> if null (tl l) then (hd l) else (Compcom l))$2}
;
commandl : commandelem {[$1]}
| commandelem ';' commandl {$1 :: $3}
;
commandelem : IDENT ASSYM expression {Asscom ($1,$3)}
| IDENT '(' IDENT ',' expression ')' {Proccom($1,$3,$5)}
| IF expression THEN command ELSE command {Ifcom ($2,$4,$6)}
| WHILE expression DO command {Whilecom ($2,$4)}
| WRITE expression {Writecom $2}
| READ IDENT {Readcom $2}
;

expression : NUM {Intconstexp $1}
| BOOL {Boolconstexp $1}
| IDENT {Varexp $1}
| IF expression THEN expression ELSE expression {Ifexp ($2,$4,$6)}
| IDENT '(' expression ')' {Funexp ($1,$3)}
| expression '+' expression {Binopexp ($1,"+",$3)}
| expression '*' expression {Binopexp ($1,"*",$3)}
| expression '-' expression {Binopexp ($1,"-",$3)}
| expression '=' expression {Binopexp ($1,"=",$3)}
| expression '>' expression {Binopexp ($1,">",$3)}
| '-' expression %prec UMINUS {Unopexp ("-",$2)}
| SUC expression {Unopexp ("suc",$2)}
| '(' expression ')' {$2}
;
%%
```

1.3.3 Semantics

Semantic Domains

```
type        env = Env of (string -> dval)
```

```
   and      dval = Locdval of loc
                 | Procdval of proc
                 | Fundval of func
   and      loc = Loc of num
   and      proc = Proc of (string & value -> store -> store)
   and      func =  Fun of (value -> store -> value)
   and      value = Intval of num
                | Boolval of bool
                | Fileval of num list
   and      store = Store of loc & (loc -> value);;


New constructors declared:
Env : ((string -> dval) -> env)
Locdval : (loc -> dval)
Procdval : (proc -> dval)
Fundval : (func -> dval)
Loc : (num -> loc)
Proc : (((string # value) -> store -> store) -> proc)
Fun : ((value -> store -> value) -> func)
Intval : (num -> value)
Boolval : (bool -> value)
Fileval : (num list -> value)
Store : ((loc # (loc -> value)) -> store)
```

Utilities

```
(* complist : (('a -> 'a) list -> 'a -> 'a) *)
let rec complist  =  function
      [] -> I
 | (h::t) -> h o (complist t)
(* appfunlist : (('a -> 'b) list & 'a -> 'b list) *)
and appfunlist = function
      [],_ -> []
 | (h::t),v -> (h v) :: appfunlist (t,v);;

(* emptystore : store  *)
let emptystore = Store (Loc 2,
                    fun (Loc i) -> if i= 0 or i=1 then Fileval []
                                   else failwith "unused");;

(* initenv : env  *)
let initenv = Env (fun t -> if t="input" then Locdval (Loc 0)
                            if t= "output" then Locdval (Loc 1)
                            else failwith (t ^ " unbound"));;

(* valof : (env & store & string -> value)  *)
```

```
let valof (Env e,Store (u,s),t) =  s l where (Locdval l) = e t;;

(* dvalof : (env & string -> dval)  *)
let dvalof (Env e,t) = e t;;

(* pushstore : (value & store -> store)  *)
let pushstore (v,s) = let (Store (Loc i,f)) = s in
     Store (Loc (i+1),fun (Loc j)->if i=j then v else f (Loc j)) ;;

(* updstore : (env & store & string & value -> store)  *)
let updstore (Env e,Store (l,f),t,v) = Store (l,ff)
      where ff = fun l -> if l= l' where (Locdval l') = e t
                          then v else f l ;;

(* extendstore : (store -> store)  *)
let extendstore (Store (l,f)) = Store (Loc (i+1), f)
      where (Loc i) = l ;;

(* updenv : ((env & string & dval) -> env)  *)
let updenv (Env e,t,dv) = Env (fun tt -> if tt=t then dv else e tt);;

(* pushenv : (string -> value -> (env & store) -> (env & store))  *)
let pushenv t v (e,s) = updenv(e,t,Locdval l) , pushstore (v,s)
      where (Store (l,u)) = s ;;
```

The semantic functions

```
(* sembinop : (string -> (value & value) -> value)  *)
let sembinop = fun
      "+" -> (fun (Intval v1,Intval v2) -> Intval (v1 + v2))
    | "-" -> (fun (Intval v1,Intval v2) -> Intval (v1 - v2))
    | "*" -> (fun (Intval v1,Intval v2) -> Intval (v1 * v2))
    | "=" -> (fun (Intval v1,Intval v2) -> Boolval (v1=v2)
                | (Boolval v1,Boolval v2) -> Boolval (v1=v2)
                | _ -> failwith "wrong args to operator = ")
    | ">" -> (fun (Intval v1,Intval v2) -> Boolval (v1>v2)
                | _ -> failwith ("wrong args to operator > "))
    |  t  -> failwith ("wrong operator : " ^ t);;

(* semunop : (string -> value -> value)  *)
let  semunop = fun "suc" -> (fun  (Intval v) -> Intval (1+v))
       | "-"  -> (fun  (Intval v) -> Intval (-v))
                  |  t   ->  failwith ("wrong operator : " ^ t);;

(* semexp : (expression -> env -> store -> value)  *)
let rec semexp = function
        Unopexp (opr,E1) -> ((fun e s -> sop (S e s))
```

```
                 where S=semexp E1 and sop = semunop opr)

      | Binopexp (E1,opr,E2) -> ((fun e s -> sop (S1 e s,S2 e s))
                 where sop = sembinop opr
                   and S1 = semexp E1
                   and S2 = semexp E2)

      | Funexp (fn,arg) -> ((fun e s-> ff (S e s) s
                 where  (Fundval (Fun ff)) = envf fn  where (Env envf) = e )
 where S = semexp arg)

      | Varexp t -> (fun e s -> valof (e,s,t))
      | Intconstexp i -> (fun e s -> Intval i)
      | Boolconstexp b -> (fun e s -> Boolval b)
      | Ifexp (test,exp1,exp2) ->
((fun e s -> let (Boolval tt) = Stest e s in
     if tt then S1 e s else S2 e s)
   where Stest = semexp test
     and S1 = semexp exp1
     and S2 = semexp exp2)
      | _ ->  failwith "wrongexp";;

(* semcom : (command -> env -> store -> store)  *)
let rec semcom = function
        Asscom (id,exp) -> ((fun e s->updstore (e,s,id,S e s))
where S = semexp exp)

      | Proccom (pn,id,exp) ->
       ((fun e s -> pp (id,S e s) s
            where (Procdval (Proc pp)) = envf pn
            where (Env envf) = e)
         where S = semexp exp)

      | Ifcom (test,com1,com2) ->
((fun e s -> let (Boolval tt) = Stest e s in
      if tt then S1 e s else S2 e s)
   where Stest = semexp test
     and S1 = semcom com1
     and S2 = semcom com2)

      | Whilecom (test, comm) -> ((fun e ->
               let rec wh s = let (Boolval tt) = St e s in
   if  tt then wh (S e s) else s in wh)
where  S = semcom comm
    and  St = semexp test)

      | Writecom exp ->
```

```
((fun e s -> updstore (e,s,"output",Fileval (v :: prevoutput))
           where (Intval v) = S e s
          and (Fileval prevoutput) = valof (e,s,"output"))
where S = semexp exp)

      | Readcom id ->
          (fun e s -> updstore (e,updstore (e,s,"input",Fileval vl),id,Intval v)
   where (Fileval (v::vl)) = valof (e,s,"input"))

      | Compcom coml -> ((fun e s -> complist (appfunlist (SS,e)) s)
where SS = map semcom (rev coml))

      | _ -> failwith "wrongcom";;

(* semdecl : (declaration -> (env & store) -> (env & store))  *)
let rec semdecl = function
         Vardecl id -> (fun (e,s) -> updenv (e,id,Locdval i) , extendstore s
where (Store (i,uu)) = s)

      | Fundecl (fn,id,exp) ->
       ((fun (e,s) -> (updenv (e,fn,Fundval (Fun ff)) , s)
    where rec ff = fun  v1 s1 -> S (updenv (e2,fn,Fundval (Fun ff))) s2
where (e2,s2) = pushenv id v1 (e,s1) ) %s1 et pas s%
  where S = semexp exp)

     | Compdecl decl -> ((fun (e,s) -> complist SL (e,s))
where SL = map semdecl (rev decl))

     | Procdecl (pn,id1,id2,decl,comm) ->
((fun (e,s) -> updenv (e,pn,Procdval (Proc pp)) , s
where rec pp = fun  (t1,v1)  s1 ->
  (SC (updenv (e2,pn,Procdval (Proc pp))) s2)
   where (e2,s2) = SD (e3,s3)
    where (e3,s3) = pushenv id2 v1 (e4,s1) %s1, not s%
      where  e4 = updenv (e,id1,dvalof(e,t1)) )
where SD = semdecl decl and SC = semcom comm)

     | _ -> failwith "wrongdecl";;

(* semprog : (program -> int list -> int list)  *)
let semprog (Prog (decl,comm))  = (fun l -> l'
where (Fileval l') = valof (e,ss,"output")
where ss = c e (updstore (e,s,"input",Fileval l)))
        where (e,s) = semdecl decl (initenv,emptystore)
        and c = semcom comm;;
```

Note that `semprog` transforms a program into a function mapping a list of inputs to a list of outputs.

Polymorphism was not used in this example, except for utility functions: all semantic functions are of ground type. Note that the treatment of `Ifexp` and `Ifcom` are similar, suggesting the use of a polymorphic treatment of conditionals, which would share the common code.

1.3.4 Examples

Factorial

```
let fact_prog = "
program
var x;
function fact (x);
if x = 1 then 1 else x * fact(x-1);
begin
read x ; write fact (x)
end.
";;

let FACT = semprog fact_prog;;

FACT[20];;
[2432902008176640000] : num list
```

Fibonacci

```
let fib_prog = "
program
var x ;
function fib (x);
if x = 1 then 1 else
if x = 2 then 1 else
fib (x-1) + fib (x-2);
begin
read x ; write fib (x)
end.
";;

let FIB = semprog fib_prog;;

FIB[10];;
[55] : num list
```

Remarks. This approach is close in spirit to a system such as P. Mosses' SIS [22] or L. Paulson's compiler generator [23]. However, we avoid one level of interpretation here, since our denotational definitions are directly executable as `CAML` programs, whereas Mosses generated λ-calculus expressions which were later reduced by a graph-reduction algorithm (Paulson used a stack abstract machine). In this way, we bridge the gap between denotational definitions and operational definitions: the `CAML` definition fulfills both purposes.

Compared to the closely related approach followed by Wand [9], using a typed language such as `CAML` as opposed to an untyped one such as Scheme has the advantage of giving an additional level of confidence in the semantic definition, since the adequacy of the semantic definitions with respect to the semantic domains is automatically ensured from `CAML` 's type checking.

This application of `CAML` as a programming environment meta-language is characteristic of its usefulness for rapid software prototyping. Once a programming language definition has been agreed upon, after extensive testing on benchmarks of user programs, the systematic development of further tools (compilers, debuggers, proof assistants) can proceed mechanically from the official `CAML` definition.

Designing compiler-generators from denotational definitions is an active research area [30, 26, 4, 11].

References

[1] "The ML Handbook, Version 6.1." Internal document, Projet Formel, Inria (July 1985).

[2] "The CAML Primer, Version 2.2." Internal document, Projet Formel, Inria (March 1987).

[3] H. Abelson and G. J. Sussman. "Structure and Interpretation of Computer Programs." MIT Press (1985).

[4] A. W. Appel. "Semantics-Directed Code Generation." Twelfth ACM POPL Symposium, New-Orleans (Jan. 1985) 315–324.

[5] R. Bird. "An Introduction to the Theory of Lists." Course Notes, International Summer School on Logic of Programming and Calculi of Discrete Design, Marktoberdorf, Aug. 86.

[6] W. H. Burge. "Recursive Programming Techniques." Addison-Wesley (1975).

[7] L. Cardelli. "ML under UNIX." Bell Laboratories, Murray Hill, New Jersey (1982).

[8] L. Cardelli. "Amber." Bell Laboratories Technical Memorandum TM 11271-840924-10 (1984).

[9] W. Clinger, D. P. Friedman and M. Wand. "A Scheme for a Higher-Level Semantic Algebra." US-French Seminar on the Application of Algebra to Language Definition and Compilation, Fontainebleau (1982).

[10] G. Cousineau, P.L. Curien and M. Mauny. "The Categorical Abstract Machine." In Functional Programming Languages and Computer Architecture, Ed. J. P. Jouannaud, Springer-Verlag LNCS 201 (1985) 50–64.

[11] G. Cousineau and P. Weis. "Semantic Combinators for Programming Languages Definitions." In preparation (1986).

[12] M. J. C. Gordon. "The Denotational Description of Programming Languages." Springer-Verlag (1979).

[13] M. Gordon, R. Milner, C. Wadsworth. "A Metalanguage for Interactive Proof in LCF." Internal Report CSR-16-77, Department of Computer Science, University of Edinburgh (Sept. 1977).

[14] M. J. Gordon, A. J. Milner, C. P. Wadsworth. "Edinburgh LCF" Springer-Verlag LNCS **78** (1979).

[15] P. Henderson. "Functional Programming Application and Implementation." Prentice-Hall (1980).

[16] S. C. Johnson "Yacc: Yet Another Compiler-Compiler." Unix Programmer's Manual, Vol. 2B, Bell Laboratories.

[17] T. Johnsson "Efficient compilation of lazy evaluation." ACM Symposium on Compiler Construction, Montreal (June 1984) 58–69.

[18] P. J. Landin. "The next 700 programming languages." Comm. ACM **9,3** (1966) 157–166.

[19] J. Mc Carthy. "Recursive functions of symbolic expressions and their computation by machine." CACM **3,4** (1960) 184–195.

[20] R. Milne and C. Strachey. "A Theory of Programming Language Semantics." Chapman & Hall (1976).

[21] R. Milner. "A proposal for Standard ML." Report CSR-157-83, Computer Science Dept., University of Edinburgh (1983).

[22] P. D. Mosses. "SIS – Reference and user's guide." DAIMI MD-30, Computer Science Dept., University of Aarhus, Denmark (1979).

[23] L. Paulson. "A Semantics-Directed Compiler Generator." Ninth ACM POPL Symposium (1982) 224–233.

[24] L. Paulson. "Tactics and Tacticals in Cambridge LCF." Technical Report No 39, Computer Laboratory, University of Cambridge (July 1983).

[25] D. S Scott and C. Strachey. "Towards a mathematical semantics for computer languages." Proc. Symp. Computers and Automata, 19–46, Polytechnic Press (1971).

[26] R. Sethi. "Control Flow Aspects of Semantics Directed Compiling." Trans. Prog. Lang. and Syst. **5,4** 554–595 (1983).

[27] G. L. Steele Jr. "Common LISP. The Language." Digital Press (1984).

[28] G. L. Steele Jr. and G. J. Sussman. "The revised report on SCHEME: A Dialect of LISP." AI Memo 452, MIT Artificial Intelligence Lab (Jan. 1978).

[29] D.A. Turner. "Miranda: A non-strict functional language with polymorphic types." In Functional Programming Languages and Computer Architecture, Ed. J. P. Jouannaud, Springer-Verlag LNCS 201 (1985) 1–16.

[30] M. Wand. "Deriving target code as a representation of continuation semantics." ACM Trans. Prog. Lang. and Syst. **4,3** 496–517 (1982).

Chapter 2

Hierarchical Structures and Pattern Matching

This chapter develops the theory of first-order terms. The formalism is progressively introduced. First we consider trees. Then labeled trees over a ranked alphabet. Finally terms with free variables.

2.1 Utilities on standard data structures

In this section we review a few basic algorithms on lists, sets and association lists.

2.1.1 Lists

A list L of length n over type τ may be considered as a finite function, of type $n \rightarrow \tau$, with n identified with the cardinal $\{0, 1, ..., n-1\}$. The i-th element L_i of L, with $i \leq n$, is thus obtained as $L_i = L(i-1)$. We can get a similar coercion with `CAML` 's lists, using the operator:

```
% nth : 'a list -> num -> 'a %
let nth l n = if n<0 then fail else (nthrec l n ? failwith "nth")
  where rec nthrec (hd::tl) n = if n=1 then hd else nthrec tl (n-1);;
```

The standard `CAML` list search primitive is `find`, which returns the first element of a list which satisfies a given predicate:

```
(* find : ('a -> bool) -> 'a list -> 'a *)
let rec find p = fun [] -> fail
                   | (x::l) -> if (p x) then x else find p l;;
```

The primitive `filter_pos` and `filter_neg` filter a list according to a predicate:

```
(* filter_pos, filter_neg : ('a -> bool) -> 'a list -> 'a list *)
let filter_pos p = fst o (partition p)
and filter_out p = snd o (partition p);;
```

Lists can also be used to represent updatable sequences. The following operators may be defined:

change $[l_1; l_2; ...l_n]\ i\ f \ = \ [l_1; l_2; ...l_{i-1}; f\ l_i; l_{i+1}...l_n]$
update $[l_1; l_2; ...l_n]\ i\ x \ = \ [l_1; l_2; ...l_{i-1}; x; l_{i+1}...l_n]$.

```
(* change : 'a list -> num -> ('a -> 'a) -> 'a list *)
let change l n f = (changerec l n ? failwith "change")
    where rec changerec (h::t) n = if n=1 then f(h)::t else h::changerec t (n-1);;
(* update : 'a list -> num -> 'a -> 'a list *)
let update l n x = change l n (K x);;
```

We also assume known the combinators split and combine, which transform lists of pairs into pairs of lists and conversely:

```
(* split : ('a & 'b) list -> ('a list & 'b list) *)
let rec split = fun
        []      -> ([],[])
 | ((x,y) :: l) -> let lx,ly = split l in (x::lx , y::ly)
(* combine : ('a list & 'b list) -> ('a & 'b) list *)
and combine = function
 (h1::t1),(h2::t2) -> (h1,h2)::combine(t1,t2)
    |    [],[]     -> []
    |      _       -> failwith "combine";;
```

2.1.2 Sets

We could define `'a set` as an abstract data type, with appropriate primitives. Here we choose rather to represent finite sets as lists.

```
let singleton x = [x]
and make_set l = list_it (fun a s -> if mem a s then s else a::s) l []
and intersect l1 l2 = filter_pos (fun x -> mem x l2) l1
and subtract l1 l2 = filter_neg (fun x -> mem x l2) l1;;
let union l1 l2 = subtract l1 l2 @ l2;;
```

2.1.3 Association lists

Association lists are lists of pairs (name,value) used as dictionaries. The standard association-list search primitive is:

```
let assoc x = find(eq_fst x) where eqfst x (y,z) = (x=y);;
```

For instance:

```
assoc 2 [(1,4);(3,2);(2,5);(2,6)] = (2,5);;
```

The primitive `assoc` is inspired from its LISP synonym. However, it is not quite as useful, since the fact that when it fails the failure token is trapped by a mechanism distinct from the conditional makes sometimes programming with association lists awkward, as will be evident from the examples below.

2.2 Trees

2.2.1 Abstract Syntax

We consider trees as oriented graphs whose nodes are labeled by strings:

```
type tree = Tree of string & tree list;;
```

2.2.2 Concrete Syntax

```
%mlescape
%token IDENT

%%
tree : mlescape {$1}
| IDENT trees {Tree($1,rev $2)}
;

trees : {[]}
| '(' t_list tree ')' {($3::$2)}
;

t_list : {[]}
| t_list tree ',' {($2::$1)}
;

%%
```

So now we may parse trees according to the usual mathematical convention:

```
<<F(A,G(B))>> = Tree ("F",[Tree ("A",[]); Tree ("G",[Tree ("B",[])])])
```

Conversely, we define a simple tree pretty-printer:

```
let unparse M = (unparserec M;print_newline())
   where rec unparserec (Tree (oper,sons)) =
        print_string oper;
        match sons with
                []  -> ()
        | (t :: lt) -> print_string "(";
                       unparserec t;
                       map (fun t -> print_string ",";unparserec t) lt;
                       print_string ")";;
```

2.2.3 Occurrences

Occurrences are lists of positive integers, denoting a tree address. For instance, occurrence `[2;3]` denotes the 3rd subterm of the 2nd subterm.

```
let rec above = function
      _,[]         -> false
   |  [],_         -> true
   | (n::u),(m::v) -> n=m & above(u,v);;

(* The subterm of M at occurrence u is noted M at u *)

directive infix "at";;
```

```
(*Tree(oper,sons) at [n1;...;nk] = nth ( ... (nth sons n1) ... ) nk *)

let op at(M,u) = it_list (fun (Tree(_,sons)) -> nth sons) M u ? failwith "at";;
let head M u = oper where (Tree(oper,_)) = M at u;;

(* let T1 = <<F(G(A,H(B)),H(C))>>;;
  T1 at [1;2] = <<H(B)>>;;
  head T1 [1;2] = "H";; *)

(* Tree replacement M[u<-N] *)
let replace M u N = replacerec (M,u)
  where rec replacerec = function
        _,[] -> N
   |    Tree(oper,sons),(n::u) -> Tree(oper,change sons n (fun P -> replacerec(P,u)));;

(* unparse (replace T1 [1;2] T2 where T2 = T1 at [2;1]);; ==>
F(G(A,C),H(C)) *)
```

2.2.4 Tree traversal

We generalize the functionals `it_list` and `list_it` to trees as general tree traversing functionals.

Think of `f` (resp. `g`) as a vertical (resp. horizontal) mapping.

```
(* treetrav : ((string & 'a) -> 'b) -> ('b -> 'a -> 'a) -> 'a -> tree
-> 'b *)
let treetrav f g start  = travrec
  where rec travrec (Tree(oper,sons)) = f(oper,list_it (g o travrec) sons start);;

let preorder = treetrav (op ::) append []
and postorder = treetrav post append [] where post(x,y) = y@[x];;

(* preorder T1 = ["F"; "G"; "A"; "H"; "B"; "H"; "C"]
 postorder T1 = ["A"; "B"; "H"; "G"; "C"; "H"; "F"] *)

(* The other traversal *)
(* left_treetrav : ((string & 'a) -> 'b) -> ('b -> 'a -> 'a) -> 'a ->
tree -> 'b *)
let left_treetrav f g start = travrec
  where rec travrec (Tree(oper,sons)) = f(oper,it_list (g Co travrec) start sons);;

(*  The set of all occurrences in a tree, sorted lexicographically in a
list *)
let occsin = left_treetrav (snd o snd) putprefix (0,[[]])
where putprefix occsson (rankleft,occsleft) = let rank=1+rankleft in
                        (rank,occsleft @ (map (cons rank) occsson));;

(* occsin T1 = [[]; [1]; [1; 1]; [1; 2]; [1; 2; 1]; [2]; [2; 1]] : num
```

```
list list *)

map (fun occ -> unparse (T1 at occ)) (occsin T1);;  ==>
F(G(A,H(B)),H(C))
G(A,H(B))
A
H(B)
B
H(C)
C
```

2.3 Terms

Terms are trees over a ranked alphabet. We could axiomatize them by changing slightly the type `tree` above, in such a way that the operators would have an arity in addition to a name. This would have the disadvantage of allowing distinct operators with the same name. And in a trivial sense, every `tree` is a term over an alphabet constituted by the tree operator names indexed by natural numbers.

We shall rather assume that an arity function has been defined, which tells what is the arity of the operators of the alphabet, named unambiguously by strings. Then, any `tree` which is well-formed according to the following definition, is a legitimate term.

```
type signature == string -> num;;

(* legim : signature -> tree -> bool *)
let legim arit = treetrav check count 0
where check(oper,n) = if arit oper = n then true else failwith oper ^ " has wrong arity"
and count x y = y+1;;

(* Example *)
let Sigma = fun
  "F" . 2
| "G" . 1
| "A" . 0
|  x  . failwith x ^ " unknown operator";;

let Sigma_term = legim Sigma;;

Sigma_term "F(G(A),A)";; ==>  true
Sigma_term "F(A)";;      ==>  evaluation failed     F wrong arity
Sigma_term T1;;          ==>  evaluation failed     C unknown operator
```

For terms, `preorder` is injective. This is what is generally referred to as “Polish prefix notation”. Here is how to retrieve a tree from its preorder list, given the signature.

```
let parse arity polish =
  let rec parse1 (oper::lops) = let (trees,rest) = parsen([],lops,arity oper)
    where rec parsen(accu,rest,n) =
```

```
      if n=0 then (accu,rest)
      else let N,r = parse1 rest in  parsen(N.::ccu,r,n-1)
  in (Tree(oper,rev trees),rest)
in match parse1 polish ? failwith "parse"
   with (M,[]) -> M
   |  (_,rest) -> failwith "Excess input: " ^ concat_list rest;;
```

Exercise. Prove by induction on the tree `M` that for an arbitrary signature `sigma` we have `parse sigma (preorder M) = M` $\Longleftrightarrow$ `legim sigma M`.

2.4 Terms with variables

Note that our anti-quotation mechanism gives us a way of doing macro-processing. For instance, `let X = <<G(A)>> and Y = <<B>> in <<F(^X,^Y)>>` is equal to `<<F(G(A),B)>>`. Better yet, the `CAML` expression `fun X Y -> <<F(^X,^Y)>>` denotes a function which, applied to any terms `M` and `N`, yields the corresponding instance of the "schema" `<<F(^X,^Y)>>`. We shall now internalize such term schemas as terms with variables.

In many respects, variables will act as constants, i.e. operators of arity 0. However, we shall carefully distinguish them from operators in the abstract syntax.

2.4.1 Abstract Syntax

Variables are represented abstractly as integers, to emphasize the fact that their concrete names are irrelevant.

```
type term = Var of num
          | Term of string & term list;;
```

2.4.2 Concrete Syntax

The names of variables is remembered in a dictionary, here an association list.

```
type dict == (string & num) list;;
```

Parsing returns a tuple (M,D,n) with M a term and D a dict of length n.

```
type concrete_term == term & dict & num;;
```

By convention, variable names start with lower-case letters.

```
let var_name string = let n=ascii_code string in n>96 & n<123;;
```

Here are the parsing routines.

```
let var D name = let n=assoc name D in (Var(n),D)
       ? let n=1+length D in (Var(n),(name,n)::D);;

let collect f g l0 = let t1,l1 = g l0 in let ts,l = f l1 in (t1::ts),l;;

let mk_terml ((sl,fl),(s,f)) = (sl @ s),(f::fl);;
```

```
let mk_term (oper,sl,fl) = sl,
(fun l0 -> let tl,l = it_list collect (fun l -> [],l) fl l0 in
Term(oper,tl),l);;

(* map_to_num ["x";"y";"x"] = ([1; 2; 1],["y",2; "x",1],2) *)
let map_to_num l =
let search s (l,(ass,n as pair)) = ((assoc s ass)::l,pair)
? let n'=n+1 in n'::l,(s,n')::ass,n'
in list_it search l ([],[],0);;

let mk_end_term (sl,f) = let il,dl,n = map_to_num sl
        in let tl,[] = f il in tl,dl,n;;

let const_or_var name = if var_name name then [name],fun (n::l)->Var(n),l
         else mk_term(name,[],[]);;

load_syntax "term";;
```

Here is the mlyacc file `term.mly`:

```
%mlescape
%token IDENT NUM
%%
term : term1 {mk_end_term($1)}
;
term1 : IDENT {const_or_var($1)}
     | NUM {mk_term(string_of_num($1),[],[])}
| IDENT terms {mk_term($1,$2)}
;
terms : '(' t_list term1 ')' {mk_terml($2,$3)}
;
t_list : {[],[]}
| t_list term1 ',' {mk_terml($1,$2)}
;

%%
```

Examples.

```
 <<F(A,G(x))>> = (Term ("F",[Term ("A",[]); Term ("G",[Var 1])]),[("x",1)],1)
 <<F(x,F(x,y))>> = (Term ("F",[Var 2; Term ("F",[Var 2; Var 1])]),
                    [("x",2); ("y",1)],2)
```

Let us now give unparsing routines.

```
(* Inversing an association list *)
let inverse = map (fun (x,y) -> (y,x));;

let unparse D M = let Dop = inverse D in unparserec M;print_newline()
```

```
  where rec unparserec = function
Var(n) -> print_string (assoc n Dop)
     |  Term(oper,sons) -> (
print_string oper;
match sons with
[] -> ()
| (t :: lt) -> print_string "(";
     unparserec t;
     map (fun t -> print_string ",";unparserec t) lt;
     print_string ")");;
```

For example, we get:

```
let M1,D1,n = <<F(x,G(x,y))>>;;
  M1 = Term ("F",[Var 1; Term ("G",[Var 1; Var 2])]) : term
  D1 = ["y",2; "x",1] : dict
  n = 2 : num
unparse D1 M1;;  ==>  F(x,G(x,y))
```

2.4.3 Occurrences

We just adjust our algorithms to the new abstract syntax. This section is a slight adaptation of 2.2.3 and 2.2.4 above.

```
(* Occurrences are lists of positive integers, denoting a term address
*)
(* For instance, occurrence [2;3] denotes the 3rd subterm of the 2nd
subterm *)

let rec above = function  u,[] -> false
        |   [],_ -> true
                | (n::u),(m::v) -> n=m & above(u,v);;

(* The subterm of M at occurrence u is noted M at u *)
directive infix "at";;

(*Term(oper,sons) at [n1;...;nk] = nth ( ... (nth sons n1) ... ) nk *)

let op at (M,u) = it_list (function Term(_,sons) -> nth sons) M u ? failwith "at";;

let head M u = oper where (Term(oper,_)) = M at u;;

(* let N1,D1,n1 = <<F(G(A,H(x)),H(y))>>;;
   let unparse1 = unparse D1;;
   unparse1 (N1 at [1;2]) ==> H(x)
   head N1 [1;2] = "H";; *)

(* Term replacement M[u<-N] *)
let replace M u N = replacerec (M,u)
```

```
  where rec replacerec = function
_,[] -> N
   | Term(oper,sons),(n::u) -> Term(oper,change sons n (fun P -> replacerec(P,u)));;

(* unparse1 (replace N1 [1;2] N2 where N2 = N1 at [2;1]);; ==> F(G(A,y),H(y)) *)

(* We generalize it_list to terms as a general term traversing functional *)
(* The last argument v tells what to do to variables *)
(* termtrav :
    (string & 'a -> 'b) -> ('b -> 'a -> 'a) -> 'a -> (num -> 'b) -> term -> 'b  *)
let termtrav f g start v  = travrec
  where rec travrec = function
     Term(oper,sons) -> f(oper,list_it (g o travrec) sons start)
  |  Var(n) -> v(n);;

let preorder = termtrav (op ::) append [] (singleton o string_of_num)
and postorder = termtrav post append [] (singleton o string_of_num)
    where post(x,y) = y@[x];;

(* preorder N1 = ["F"; "G"; "A"; "H"; "1"; "H"; "2"]
 postorder N1 = ["A"; "1"; "H"; "G"; "2"; "H"; "F"] *)

(* Copying a term *)
let copy = termtrav Term cons [] Var;;
(* For every term M, we have M = copy M *)

(* An unparser as a function from terms to strings *)
let pretty = termtrav wrappar args ("",0) string_of_num
  where wrappar(s1,s2,n) = s1 ^ (if n=0 then "" else "(" ^ s2 ^ ")")
  and args s1 (s2,n) = (s1 ^ (if n=0 then "" else ",") ^ s2),n+1;;

(* (let M,D,n = <<F(G(x,y),H(x),A)>> in pretty M) = "F(G(1,2),H(1),A)" *)

(* The other traversal *)
(* left_termtrav :
    (string & 'a -> 'b) -> ('b -> 'a -> 'a) -> 'a -> (num -> 'b) -> term -> 'b *)
let left_termtrav f g start v = travrec
  where rec travrec = function
     Term(oper,sons) -> f(oper,it_list (g Co travrec) start sons)
   | Var(n) -> v(n);;

(*  The set of all occurrences in a term, sorted lexicographically in a list *)

(* occsin : term -> num list list  *)
let occsin = left_termtrav (snd o snd) putprefix (0,[[]]) occstriv
where putprefix occsson (rankleft,occsleft) = let rank=1+rankleft in
                        (rank,occsleft @ (map (cons rank) occsson))
```

```
and occstriv n = [[]];;

(* occsin N1 = [[]; [1]; [1; 1]; [1; 2]; [1; 2; 1]; [2]; [2; 1]] : num list list *)

(* The set of subterms of a term *)
let subterms M = make_set (map (fun occ -> (M at occ)) (occsin M));;

(* map unparse1 (subterms N1)  ==>
F(G(A,H(x)),H(y))
G(A,H(x))
A
H(x)
x
H(y)
y *)
```

2.4.4 Signatures

Similarly to trees above, we may check that terms with variables are consistent with a signature declaration.

```
type signature == string -> num;;

(* legim : signature -> term -> bool *)
let legim arit = termtrav check count 0 (K true)
where check(oper,n) = if arit oper = n then true else failwith oper ^ " has wrong arity"
and count x y = y+1;;

(* Example *)
let Sigma = fun
  "F" -> 2
| "G" -> 1
| "A" -> 0
| x -> if var_name x then 0 else failwith x ^ " unknown operator";;

let Sigma_term = legim Sigma;;
(*
Sigma_term T where T,_="F(G(A),A)";; ==>  true
Sigma_term T where T,_="F(A)";; ==>  evaluation failed   F has wrong arity
Sigma_term N1;;          ==>  evaluation failed     H unknown operator
*)
let parse arity polish =
  let rec parse1 (oper :: lops) = let (terms,rest) = parsen([],lops,arity oper)
    where rec parsen(accu,rest,n) =
      if n=0 then (accu,rest)
      else let N,r = parse1 rest in  parsen(N::accu,r,n-1)
  in (Term(oper,rev terms),rest)
in match parse1 polish ? failwith "parse"
```

```
   with (M,[]) -> M
   |  (_,rest) -> failwith "Excess input: " ^ concat_list rest;;

(* parse Sigma (preorder M) = M where M,D,n="F(G(A),A)";;*)
```

2.4.5 Various measures on terms

```
(* The length of a term : len M = length (preorder M) *)
let len = termtrav snd add 1 (K 1);;

(* The set of variables of a term *)
let vars = termtrav snd union [] singleton;;

(* nu M is the number of distinct variables occurring in M *)
let nu = length o vars;;

(* The complexity of terms *)
let complex M = (len M) - (nu M);;

(* Linear terms do not contain multiple occurrences of variables *)
let linear M = ((termtrav snd disjoint_union [] singleton M; true) ? false)
   where disjoint_union l1 l2 = list_it cons' l1 l2
where cons' x l = if mem x l then fail else x::l;;
```

If `M` is linear, we have `complex M = termtrav snd add 1 (K 0) M`. We shall call *pattern* a linear term. `CAML` patterns (in the `fun` and `match` constructions) are patterns in this sense. This restriction to linear terms comes from the fact that `CAML` uses pattern-matching as a *binding* mechanism. More generally, we shall define pattern-matching for arbitrary terms. To this end, we study *substitutions*.

2.4.6 Substitutions

Substitutions are functions from variables to terms. Since all terms have a finite number of variables, we only need to consider substitutions with a finite domain.

```
(* We shall represent substitutions by association lists :
  [(var1,term1);...;(varN,termN)] *)

type subst == (num & term) list;;

let pretty_subst =
    map (fun (v,t) ->(string_of_num v) ^ " --> " ^ (pretty t));;

let bindng n subst = (assoc n subst) ? Var n;;

??let bound n subst = (assoc n subst;true)?false;;

let rec substitute subst = function
```

```
  Term(oper,sons) -> Term(oper,(map (substitute subst) sons))
| Var(n)          -> bindng n subst;;

(* occurs n t = mem n (vars t)  *)
let occurs n t = let found m = if m=n then fail else false in
                 (termtrav snd K false found t) ? true;;

let compsubst subst1 subst2 =
 (map (fun (v,t) -> (v,substitute subst1 t)) subst2) @ subst1;;
```

Substitutions as functions can be easily extracted from our concrete substitutions. We recall the `C` combinator, defined as: `C f x y = f y x`.

```
let subst_map = C bindng;;
(* (compsubst subst1 subst2) = (subst_map subst2) o (subst_map subst1) *)
```

Any such mapping in `num -> term` may be extended as as a Σ-morphism over the current signature Σ:

```
(* morphism_ext : (num -> term) -> (term -> term) *)
let morphism_ext = termtrav Term cons [];;
(* For instance, copy = morphism_ext Var *)

let subst_morphism = morphism_ext o subst_map;;
(* subst_morphism is  same as substitute above *)

(* More generally, let a Sigma algebra be defined by a type alpha, and
  a function Operators : string -> (alpha list) -> alpha. Any function
  Interpretation : num -> alpha extends uniquely to a morphism
  (morphism Operators Interpretation) : term -> alpha, with : *)
let morphism Operators = termtrav (uncurry Operators) cons [];;

(* Example: other versions of measure len above *)
let len' = let operators _ = succ o sigma and interpretation _ = 1 in
           morphism operators interpretation;;
```

2.4.7 Matching

We shall need frequently to iterate on two lists in parallel. A convenient operator to do this is:

```
(* it_list2 : (('a -> 'b & 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a
*)
let it_list2 f init list1 list2 =
        it_list f init (combine (list1,list2) ? failwith "it_list2");;

(* matching = - : (term -> term -> subst) *)
let matching term1 term2 =
  let rec matchrec subst =
   fun (Var v,M) -> if bound v subst then let N = assoc v subst in
```

```
                    if M=N then subst else failwith "matching"
     else (v,M)::subst
   | (Term(op1,sons1),Term(op2,sons2)) ->
  if op1 = op2 then it_list2 matchrec subst sons1 sons2
  else failwith "matching"
  in matchrec [] (term1,term2);;

(*  matching returns the matching substitution, or fails *)

let M1,D1,n1 = <<F(G(A,x),H(x,y))>>
and M2,D2,n2 = <<F(G(x,y),z)>>
and M3,D3,n3 = <<F(x,y)>>;;

let Sub1 = matching M2 M1;;
(* Sub1 = [(1,Term("H",[Var 2; Var 1])); (2,Var 2); (3,Term("A",[]))]  *)
let Sub2 = matching M3 M1;;
(*  Sub2 = [(1,Term("H",[Var 2; Var 1])); (2,Term("G",[Term("A",[]); Var 2]))]  *)
```

The control structure in the variable case is not very nice, since we must evaluate twice the expression `(assoc v subst)`.

Note that for linear terms the variable case can be greatly simplified:

```
let matching_linear term1 term2 =
  let rec matchrec subst =
   fun  (Var v,M) -> (v,M)::subst
   |    (Term(op1,sons1),Term(op2,sons2)) ->
         if op1=op2 then it_list2 matchrec subst sons1 sons2 else failwith "matching"
  in matchrec [] (term1,term2);;
```

Thus pattern-matching of patterns is linear in the size of the pattern, independently from the size of the matched term.

Matching defines a quasi-ordering on terms: we write $M \leq N$ iff `match M N` does not fail. In that case, there exists a unique substitution σ of domain `vars M` such that $N = \sigma(M)$. σ is precisely the result of `match M N`. As we saw, σ may be uniquely extended as a Σ-morphism, where Σ is the current signature. The isomorphisms are exactly the variable renamings, which apply an arbitrary permutation to the variable indexes of a term. Let us write $\equiv$ for the corresponding equivalence: $M \equiv N \Leftrightarrow M \leq N \wedge N \leq M$. The classes of Σ-terms under $\equiv$ are partially ordered by matching. We shall see that this partial ordering admits glb's, and conditional lub's. That is, if two terms admit a common substitution instance (we then say they are *unifiable*), they admit a most general such instance.

2.4.8 Unification

The partial ordering $\leq$ extends to substitutions: $\sigma \leq \tau$ iff for some ρ we have $\tau = \rho \ o \ \sigma$. Let us call *unifier* of M and N any substitution σ such that $\sigma(M) = \sigma(N)$. If M and N are unifiable, they admit a *principal* unifier, which is the lub of all their unifiers.

```
(* A naive unification algorithm *)
```

```
let rec unify =
 fun (Var n1,term2)-> if Var n1 = term2 then []
       if occurs n1 term2 then fail
else [n1,term2]
| (term1,Var n2)-> if occurs n2 term1 then fail
else [n2,term1]
| (Term(op1,sons1),Term(op2,sons2)) ->
  if op1 = op2 then
   (it_list2 unifylist [] sons1 sons2
    where unifylist s (t1,t2) =
      compsubst (unify(substitute s t1,substitute s t2)) s)
  else fail ? failwith "unify";;


let T1 = Term("F",[Var 1;Term("G",[Var 1])])
and T2 = Term("F",[Var 2;Var 2])
and T3 = Term("F",[Term("H",[Var 2]);Var 3]);;

(* unify(T1,T1) = [] : subst;;
unify(T1,T2);;  ==> evaluation failed     unify
pretty_subst (unify(T1,T3));; ==> ["1 --> H(2)"; "3 --> G(H(2))"] *)
```

2.4.9 Anti-unification

Two arbitrary terms possess a least general generalization, or maximal factorization, which is a glb with respect to the ordering $\leq$.

```
(* L is list of tuples ((Mi,Ni),ni)*)
let enter x L = (let k=assoc x L in Var k,L)
                ? let n=length(L)+1 in Var n,(x,n)::L;;

type subst2 == ((term & term) & num) list;;

(* fctrec : (term & term) -> subst2 -> (term & subst2) *)
let rec fctrec pair L = match pair with
  Term(op1,terms1),Term(op2,terms2) ->
    if op1=op2 then let l',L'= it_list2 collect ([],L) terms1 terms2
                         where collect (l,S) p = let T,S' = fctrec p S in T::l,S'
                     in Term(op1,rev l'),L'
    else enter pair L
  | _  -> enter pair L;;

(* factor : (term & term) -> term *)
let factor pair = fst(fctrec pair []);;

(* let M,_="F(G(A),A,A,A,B)" and N,_="F(G(B),B,B,A,C)" in pretty(factor(M,N));;
==> "F(G(1),1,1,A,2)" *)
```

Note that the second component of the pair computed by fctrec is a proof that `factor` computes a common factor, in the form of a pair of substitutions. More precisely, with `(P,L) = fctrec (M,N) []`, the list `L:subst2` may be interpreted as a pair of substitutions:

```
(* factorization : (term & term) -> (term & subst & subst) *)
let factorization (pair) = let (P,L) = fctrec pair [] in
    let L1,L2 = split L in
          let L3,L4=split L1 in P,combine(L2,L3),combine(L2,L4);;
```

And it is easy to prove by induction that `factorization (T1,T2) = (P,S1,S2)` with `substitute S1 P = T1` and `substitute S2 P = T2`, and that `P` is the greatest such term in ordering $\leq$.

Exercise. Complete the proof above, showing the existence of greatest lower bounds with respect to ordering $\leq$. Prove that the ordering $\leq$ is well-founded (i.e. that there are no infinite strictly decreasing sequences), by showing that $M < N$ implies $complex(M) < complex(N)$. (We write $M < N$ for $M \leq N \wedge \not N \leq M$). Conclude from these two facts the following theorem.

Theorem. Let $\mathcal{T}_\Sigma$ be the set of equivalence classes of Σ-terms, ordered by the match partial ordering $\leq$, to which we add a maximum element $\top$. $\mathcal{T}_\Sigma$ is a complete lattice.

Corollary. Any unifiable set of terms possesses a minimum common instance.

Unification appeared in Herbrand's thesis [3]. It has become well-known since Robinson's seminal paper on resolution [8]. The algebraic theory of term structures was developed in [6, 7, 4, 2].

2.5 Infinite rational terms and recursion equations

It is possible to generalize the formalism of standard finite first-order terms to infinite rational terms described by recursion equations. Ordinary unification generalizes to unification of infinite rational terms. Substitutions are replaced by systems of recursive equations. For unification, this amounts to suppress the "occur test":

A set of recursive equations is a list of pairs (V,T) where V is a set of variables and T is a term. This set is an equivalence class of variables bound together. When T=Var(0), this means the class of variables V is *free*. Otherwise, T is a non-variable term, and the class is *bound* to term T. The first variable in a class is its *canonical* representative.

```
type rec_eqs == (num list & term) list;;

let class n = find (function V,_ -> mem n V);;

(* n is assumed canonical *)
let rem_class n = remrec
   where rec remrec (p::E) = let (i::V,_) = p in if i=n then E else p::remrec E;;

let rec reduce_eqs E = fun
    [] -> E
  | (p::L) -> let E' = reduce_eqs E L in (match p with
       Var(n1),Var(n2) -> merge_classes n1 n2 E'
     | Var(n1),term2 -> put_class n1 term2 E'
```

```
     | term1,Var(n2) -> put_class n2 term1 E'
     | Term(op1,sons1),Term(op2,sons2) ->
          if op1=op2 then reduce_eqs E' (combine(sons1,sons2)) else fail)
where merge_classes i j E = let (ci::I),T = class i E and (cj::J),U = class j E
           in if ci=cj then E
              else let E' = rem_class ci (rem_class cj E)
                   and IJ = ci::cj::(I @ J) in match T with
                         Var(_) -> (IJ,U)::E'
                       | _ -> let E''=(IJ,T)::E' in match U with
                             Var(_) -> E''
                           | _ -> reduce_eqs E'' [T,U]
and put_class i M E = let (ci::I),N = class i E in match N with
                Var(_) -> (ci::I,M)::rem_class ci E
              | _ -> reduce_eqs E [M,N];;

let rat_unify (M,N) =
  let inieqs = map initclass (union (vars M) (vars N))
      where initclass v = (singleton v,Var(0)) in reduce_eqs inieqs [M,N];;

(* rat_unify(T1,T2) = [[1; 2],Term ("G",[Var 1])] : rec_eqs *)
(* rat_unify(T1,T3) = [[1],Term ("H",[Var 2]); [3],Term ("G",[Var 1]); [2],Var 0] *)
```

Remark that we could simplify `rat_unify` if we assumed all rational terms are defined by a pair `(n,E)` with `E` a set of recursive equations. The right-hand side of an equation would no longer be a general term, but just a pair `(oper,vars):string & num list`.

It is possible to interpret the result of `rat_unify` as a principal unifier for rational terms. The algebraic theory of rational terms is developed in [4].

Exercise. Write a function `acyclic` which tests whether a set of recursive equations contains cycles or not. Use it as a post-processor to `rat_unify` in order to get a new unification algorithm for finite terms. Compare with `unify` above.

2.6 Practical Considerations

The `CAML` algorithms given above should be regarded as functional specifications. Actual algorithms used in practice use special data-structures, with destructive operators, in order to optimize these very basic routines. We shall not discuss efficient implementations nor complexity considerations here.

References

[1] C. Dwork, P. C. Kanellakis and J. C. Mitchell. “On the Sequential Nature of Unification.” Technical Report No CS-83-26, Brown University (1983).

[2] E. Eder. “Properties of Substitutions and Unifications.” J. Symbolic Computation **1,1** 31–46 (1985).

[3] J. Herbrand. "Recherches sur la théorie de la démonstration." Thèse, Université de Paris (1930). Reprinted in "Ecrits Logiques", Presses Universitaires de France (1968), translated as "Logical writings", Harvard University Press (1971).

[4] G. Huet. "Résolution d'équations dans des langages d'ordre 1,2, ... ω." Thèse d'Etat, Université Paris VII (1976).

[5] M. S. Paterson, M.N. Wegman. "Linear Unification." J. of Computer and Systems Sciences **16** (1978) 158–167.

[6] G. D. Plotkin. "Lattice-Theoretic Properties of Subsumption ". Memo MIP-R-77, U. of Edinburgh (1970).

[7] J. Reynolds. "Transformational Systems and the Algebraic Structure of Atomic Formulas". Machine Intelligence **5** 135–152 (1970), American Elsevier.

[8] J. A. Robinson. "A Machine-Oriented Logic Based on the Resolution Principle ." JACM **12** (1965) 32–41.

[9] J. A. Robinson. "Computational Logic: the Unification Computation ." Machine Intelligence **6** Eds B. Meltzer and D.Michie, American Elsevier (1971).

Chapter 3

Logic Programming

As an application of the term structures algorithms, we shall present in this chapter various variations on the theme of logic programming. The theoretical problem to be solved is the construction of a term of a given type in a theory of polymorphic operators. Each operator is presented with its signature, which may be considered as a definite clause, that is a conditional statement universally quantified:

$$Op \ := \ Concl \leftarrow Hyp_1; \ldots ; Hyp_n.$$

Given arguments $M_1, \ldots M_n$ of respective types $Typ_1, \ldots, Typ_n$, we may form the term $M = Op(M_1, \ldots, M_n)$ iff for some substitution σ we have $Typ_i = \sigma(Hyp_i)$ for $i = 1, \ldots, n$. In that case, we say that M is well-formed, and of type $\sigma(Concl)$. So we address here the problem of finding one (or all) terms whose type is an instance of some given goal type.

Analogously, we may view this problem as searching for an existential proof in a logical inference system (viewing each operator as a schematic rule of inference). The extracted answer (the substitution applied to the original goal) is the "result" computed by the proof-synthesis procedure, seen as the interpreter of a "logic program", i.e. the set of definite clauses.

The main synthesis paradigm is the Prolog procedure, which proceeds by systematic backtracking in a depth-first top-down way.

3.1 A mini-Prolog interpreter

This section explains the Prolog [4, 27] proof-synthesis procedure, as a mini-Prolog interpreter in `CAML` .

3.1.1 Abstract syntax

A (definite) clause is composed of a term (its conclusion), a list of terms (its hypotheses), and an integer (the number of variables appearing in all the terms). These variables are assumed to be bound together at the level of the clause.

```
type clause == term & (term list) & num;;
```

3.1.2 Concrete syntax

Parsing returns a clause and its variables dictionary.

```
type concrete_clause == clause & dict;;
```

The typical syntax of a definite clause is: `Concl <- Hyp1; ... Hypn.`

Parsing utilities.

```
let mk_clause ((s,f),(sl,fl)) = let (il,dl,n) = map_to_num (s @ sl)
   in let (h,il') = f il
    in let (t,[]) = it_list collect (fun l -> [],l) fl il'
       in (h,t,n),dl;;
```

A *goal* clause is a list of terms preceded by a question mark. It is used to make queries in specifications described as definite clauses. Thus the variables appearing in a clause should be considered existentially bound. A goal clause containing variables $x_1, ..., x_n$ is represented as a clause: `Answer(x1,...,xn) <- goal`.

```
let mk_goal (sl,fl) = let il,D,n = map_to_num sl
    in let lt,[] = it_list collect (fun l -> [],l) fl il
       in (Term("Answer",map (Var o snd) D),lt,n),D;;

load_syntax "prolog";;
```

Here is the contents of the `prolog.mly` file:

```
%mlescape
%token IDENT NUM
%left '='
%left '*'
%%
clause  : cl {mk_clause($1)}
| '?' tail {mk_goal($2)}
;
cl : term '<' '-' tail {$1,$4}
| term {$1,([],[])}
;
tail : term {mk_terml(([],[]),$1)}
| tail ';' term  {mk_terml($1,$3)}
;
term : IDENT {const_or_var($1)}
| NUM {mk_term(string_of_num($1),[],[])}
| IDENT terms {mk_term($1,$2)}
| term '=' term {mk_term("=",mk_terml(mk_terml(([],[]),$1),$3))}
| term '*' term {mk_term("*",mk_terml(mk_terml(([],[]),$1),$3))} ;
| '(' term ')' {$2}
;
terms : '(' t_list term ')' {mk_terml($2,$3)}
;
t_list : {[],[]}
| t_list term ',' {mk_terml($1,$2)}
;
```

```
%%

Example of a definite clause:
<<P(x,y)<-Q(x);R(y)>> =
(Term ("P",[Var 2; Var 1]),[Term ("Q",[Var 2]); Term ("R",[Var 1])],
2),["x",2; "y",1] : concrete_clause
```

You may think of the above clause as a logical statement: "For all x and y, we have $P(x,y)$ if $Q(x)$ and $R(y)$". Its "procedural" interpretation is: "In order to show $P(x,y)$, you must show first $Q(x)$ and then $R(y)$".

```
Example of a goal clause:
<<?P(A,x)>> =
(Term ("Answer",[Var 1]),[Term ("P",[Term ("A",[]); Var 1])],
1),["x",1] : concrete_clause
```

You may think of the above clause as a logical interrogation: "Does there exist an x such that $P(A,x)$?" or as a procedural *query*: "Find an x such that $P(A,x)$".

```
let unparse_clause ((conc,hyps,n),D) =
 let U = unparse D in
  let U' x = (print_string ";";U x) in
(U conc;match hyps with
   [] -> ()
 | (h::t) -> (print_string "<-";U h;map U' t;print_newline()));;

(* New variables are printed as v1, v2, ... *)
let gensym n = concat "v" (string_of_num n);;

let unparse_answer D (Term("Answer",lt)) =
 let newvars = (subtract (list_it (union o vars) lt []) (map snd D)) in
 let newD = union D (num_map (pair o gensym) newvars)
 in (let unparseD = unparse newD in
     map (fun ((name,_),term) -> print_string name;print_string " = ";
                             unparseD term;print_newline())
         (combine(D,lt)));();;
```

3.1.3 Resolution

Integers for variables allow easy renaming. This renaming is necessary to ensure that variables pertaining to different clauses are not confused.

```
let instance n = morphism_ext (fun k -> Var(k+n));;
```

The resolution rule is here the following. Let C be a definite clause:

$$C \ = \ P \leftarrow Q_1 \ ; \ Q_2 \ ... \ Q_n$$

and let G be a goal clause:

$$G \ = ?\, G_1 \ ; \ G_2 \ ... \ G_p.$$

We assume that C has been renamed in such a way as to have no variable in common with G. Let terms P and G_1 be unifiable, with principal unifier σ. We infer by *resolution* of C and G the new goal clause:

$$?\ Q'_1\ ;\ Q'_2\ ...\ Q'_n\ ;\ G'_2\ ...\ G'_p$$

where everywhere M' denotes $\sigma(M)$.

```
(* Resolution *)
(* 2nd arg and result are goals *)
(* resolve : clause -> clause -> clause *)
let resolve (c,h,max) (ans,(goal1::goals),n) =
  let conc = instance n c and hyps = map (instance n) h in
  let subst = substitute (unify(goal1,conc)) in
  (subst ans, map subst (hyps @ goals),max+n)
  ? failwith "resolve";;

(* Definite clauses are organized in a theory *)
type theory == clause list;;

(* Example *)
let Append = [Append1;Append2] (* : theory *)
where Append1,_ = <<Append(Nil,x,x)>>
and   Append2,_ = <<Append(Cons(u,x),y,Cons(u,z)) <- Append(x,y,z)>>;;
```

3.1.4 Finding all solutions in a depth-first manner

```
(* The Prolog_all loop *)
(* Takes a theory and a list of goals, and returns a list of answers *)
(* realize_all : theory -> clause -> term list *)
let realize_all theory = rev o real []
  where rec real answers g = match g with
   (answer,[],_) -> (answer :: answers)
 | _ -> (it_list (fun ans cl -> real ans (resolve cl g) ? ans) answers theory
       ? answers);;

(* This loop returns the list of all solutions *)
(* Prolog_all : theory -> concrete_clause -> void *)
let Prolog_all theory (goals,D) =
    match realize_all theory goals with
[] -> message "No solution"
     | answers -> it_list (fun n ans -> print_string "Solution ";print_num n;
      print_newline();unparse_answer D ans;n+1) 1 answers;();;

(*
Prolog_all Append <<?Append(Nil,x,y)>>;;
Solution 1
x = v1
y = v1
```

```
Prolog_all Append <<?Append(x,y,Cons(A,Cons(B,Nil)))>>;;
Solution 1
x = Nil
y = Cons(A,Cons(B,Nil))
Solution 2
x = Cons(A,Nil)
y = Cons(B,Nil)
Solution 3
x = Cons(A,Cons(B,Nil))
y = Nil
```

But note how `Prolog_all Append <<?Append(x,y,z)>>` would loop trying to generate an infinite list of solutions! This is particularly irritating, since Append is an obviously deterministic theory whose solution space can be spanned without any backtracking. Let us now program a deterministic Prolog loop.

3.1.5 A deterministic loop

We assume known the `try_find` CAML primitive; `try_find f l` returns `f(li)` for the first i such that `f(li)` succeeds. It is equivalent to:

```
let rec try_find f = fun [] -> fail
                       | (h::t) -> (try f h with _ -> try_find f t);;

(* realize_det : theory -> goals -> term list *)
let realize_det theory = real
  where rec real g = match g with
    (answer,[],_) -> answer
  | _ -> real (try_find (fun cl -> resolve cl g) theory);;

(* Prolog_det : theory -> concrete_clause -> void *)
let Prolog_det theory (goals,D) =
    (let answer1 = realize_det theory goals in
(message "Solution found";unparse_answer D answer1))
    ? message "No solution found";;

Prolog_det Append <<?Append(x,y,Cons(A,Cons(B,Nil)))>>;;
Solution found
x = Nil
y = Cons(A,Cons(B,Nil))
```

Note that `Prolog_det` is a natural generalization of the CAML `fun` and `match` constructs. Instead of doing a simple matching of linear patterns against constructor trees without variables, we may here have to do full unification of possibly non-linear terms with actual arguments which are themselves terms with variables. It is this feature which makes Prolog a revolutionary programming language, since procedure call is not conceived of as simply handing down arguments for a procedure to compute and return a result. Rather, this is “hand-shaking” between two procedures which share information in a symmetric way. The fundamental Prolog data-type is the first-order term, considered as an "incomplete" record-like object. Use of this data-type allows a free mixture of symbolic and concrete computations.

The other salient feature of Prolog considered as a programming language is of course its implicit treatment of non-determinism. Let us now consider a typical non-deterministic theory.

```
let Zero_or_One = [Ax1;Ax2;Ax3;Ax4] (* : theory *)
where Ax1,_ = <<Zero_or_One(x) <- Zero(x)>>
and   Ax2,_ = <<Zero_or_One(x) <- One(x)>>
and   Ax3,_ = <<Zero(0)>>
and   Ax4,_ = <<One(1)>>;;
```

Now we get a failure using the deterministic loop:

```
Prolog_det Zero_or_One <<?Zero_or_One(1)>>;;
No solution found
```

So we now build a (non-deterministic) Prolog loop that returns the first solution it finds.

3.1.6 Looking for the first solution

```
(* The Prolog_1st loop *)
(* Takes a theory and a list of goals, and returns its 1st answer *)
(* realize_1st : theory -> clause -> term *)
let realize_1st theory = real
  where rec real g = match g with
    (answer,[],_) -> answer
  | _ -> try_find (fun cl -> real (resolve cl g)) theory;;
(* Note that we have simply commuted try_find and real *)

(* Prolog_1st : theory -> concrete_clause -> void *)
let Prolog_1st theory (goals,D) =
    let answer1 = realize_1st theory goals in
(message "Solution found";unparse_answer D answer1)
    ? message "No solution found";;
```

And now :

```
Prolog_1st Zero_or_One <<?Zero_or_One(1)>>;;
Solution found
```

So now we may get a solution when there are infinitely many, without any assumption on the theory to being deterministic. Still, we would like to be able to enumerate progressively all solutions, in a "stream" rather than in a list. `Prolog_1st` is of no help, since after it returns, all information is lost about the state of its backtrack.

It is not possible to compute streams directly in our non-lazy `CAML` . However, it is quite possible to describe coroutine-like computations with functions returning closures, which are left un-evaluated. All we need is to refine our Prolog interpreter in such a way that the backtrack structure is explicit. This way we shall be able to extract a Prolog coroutine that will compute the "next" solution.

3.1.7 A Prolog coroutine

A *state* consists of an answer term, a goal history represented as a list of pairs (goal,th), where th is the list of all definite clauses from the theory not yet matched to goal, and a variables counter. A *resumption* is a stack of states.

```
type state == term & (term & theory) list & num;;

(* A resumption is a stack of states. *)
type resumption == state list;;

(* depth-first resolution *)
(* next_state : theory -> state -> state *)
let next_state theory (ans,(goal1,(c,h,max)::_)::rest,n) =
    let conc = instance n c and hyps = map (instance n) h in
    let subst = substitute (unify(goal1,conc)) in
        let first = map (fun M -> (subst M,theory)) hyps
        and last = map (fun (g,th) -> (subst g,th)) rest in
    (subst ans,first @ last,max+n) ? failwith "next_state";;

(* start : theory -> clause -> resumption *)
let start theory (term,terms,n) =
   singleton(term,map (fun M -> (M,theory)) terms,n);;

(* A Prolog coroutine *)
(* resume : theory -> resumption -> (term & resumption) *)
let resume theory = let next = next_state theory in resumerec
where rec resumerec = fun
   [] -> failwith "No more solution"
 | (state::states) -> let answer,stack,n = state in match stack with
       [] -> answer,states         (* A solution has been found *)
     | ((_,[])::_) -> resumerec states (* Backtrack *)
     | ((goal,th)::rest) ->
let next_states = (answer,(goal,tl th)::rest,n)::states
in resumerec((next state)::next_states ? next_states);;

(* A Prolog loop calling repeatedly the coroutine *)
let Prolog_loop theory (goal,dict) =
    let coroutine = resume theory
    and print_answer = unparse_answer dict in
    let register = ref ((start theory goal):resumption)
    and count = ref 1 in
while true do let answer,next = coroutine !register in
(print_string "Solution ";print_num !count;print_newline();
 print_answer answer;register:=next;count:=!count+1;()) done
 ? print_string "Finished";;
```

Example.

```
let Permutations = map fst [
<<Perm(Nil,Nil)>>;
<<Perm(l1,Cons(x2,l2))<-Pick(l1,x2,l3);Perm(l3,l2)>>;
<<Pick(Cons(x,l),x,l)>>;
<<Pick(Cons(x,l),y,Cons(x,l'))<-Pick(l,y,l')>>];;
Prolog_loop Permutations <<?Perm(Cons(A,Cons(B,Cons(C,Nil))),p)>>;;
Solution 1
p = Cons(A,Cons(B,Cons(C,Nil)))
Solution 2
p = Cons(A,Cons(C,Cons(B,Nil)))
Solution 3
p = Cons(B,Cons(A,Cons(C,Nil)))
Solution 4
p = Cons(B,Cons(C,Cons(A,Nil)))
Solution 5
p = Cons(C,Cons(A,Cons(B,Nil)))
Solution 6
p = Cons(C,Cons(B,Cons(A,Nil)))
```

An example of progressive use of the coroutine:

```
next_perm : (resumption -> (term & resumption))
let next_perm = resume Permutations;;
let goal,dict = <<?Perm(Cons(A,Cons(B,Cons(C,Nil))),p)>>;;
let present = unparse_answer dict;;
let R0 = start Permutations goal;;
let P1,R1 = next_perm R0;;
present P1;;  ==> p = Cons(A,Cons(B,Cons(C,Nil)))
let P2,R2 = next_perm R1;;
present P2;;  ==> p = Cons(A,Cons(C,Cons(B,Nil)))
```

But we have still other reasons for non-termination. For instance, consider the simple example of transitive-reflexive closure:

```
let Ancestor1,_ = <<Ancestor(x,x)>>
and Ancestor2,_ = <<Ancestor(g,s)<-Ancestor(g,f);Father(f,s)>>;;
```

Now

```
Prolog_1st [Ancestor1;Ancestor2] <<?Ancestor(John,John)>>;;
```

succeeds, whereas

```
Prolog_1st [Ancestor2;Ancestor1] <<?Ancestor(John,John)>>;;
```

loops in the "recursive call" to `Ancestor`.

The problem lies here in the Prolog choice of iterating "vertically" (recursive calls by resolution) before iterating "horizontally" (non-deterministic choice of a clause in the theory). In other words, Prolog has a depth-first search, as opposed to a breadth-first search. This is its main cause of incompleteness.

Remark. The Prolog interpreter described above is more general than usual Prolog implementations. A clause is here any signature ¡¡Q ¡- P1;P2 ... Pn¿¿ where Q and the Pi's are general first-order terms. Thus there is no distinction between predicate and function symbols. Also these terms may be reduced to a single variable. For instance :

```
let Kax,_ = <<F(a,F(b,a))>>
and Sax,_ = <<F(F(a,F(b,c)),F(F(a,b),F(a,c)))>>
and MP,_ = <<b<-F(a,b);a>>;;
```

Note that `Kax` is the `CAML` type of K, `Sax` is the type of S, and `MP` is the type of application. You may think of `F` as the function type-formation operator "-¿", or as (intuitionistic) logical implication. Thus, the above clauses specify minimal logic:

```
(* Min_Logic : theory *)
let Min_Logic = [Kax;Sax;MP];;
```

Let us use our Prolog interpreter to find a (proof) term of type $(a \to a)$ for some (type) term a :

```
Prolog_1st Min_Logic <<?F(a,a)>>;;
Solution found
a = F(v2,F(v1,v2))
```

However, the simple-minded depth-first strategy of Prolog loops when asking for a term of polymorphic type $(a \to a)$, that is, "Skolemizing" the variable a into a constant `A` :

```
Prolog_1st Min_Logic <<?F(A,A)>>;;
... loops ...
```

Acknowledgement. The Prolog interpreter routines are based on earlier programs of Ascander Suarez.

3.2 Definite clauses theorem proving

This section explains a complete theorem prover for definite clauses, and gives a general theorem explaining the underlying theory. We assume known the Prolog abstract and concrete syntax.

3.2.1 Looking for satisfiability of a goal

The problem we have been encountering with the dependency on the order of the clauses in the theory is a simple problem of search strategy: we do not have a fair scheduler for the various non-deterministic choices.

It is not immediate how to derive a fair scheduler from the above algorithms, which are really geared to depth-first search. For instance, if we exchange the role of first and last in the `next_state` function above, we would be able to get a proof of:

```
Prolog_1st [Ancestor2;Ancestor1] <<?Ancestor(John,John)>>
```

since the `Father` subgoal fails before the recursion loops, allowing the clause `Ancestor1` to conclude. But this does not give us a fair scheduler strategy, and we would still loop on goals such as:

```
Prolog_1st Min_Logic <<?F(A,A)>>;;
```

So we reconsider the problem, and we add a stack of goals, in order to implement a fair scheduler using a level-saturation stategy: The scheduler will consider all non-deterministic choices for the goals of level n, before considering the goals of level $n+1$, which are kept in a separate stack.

```
(* Takes a theory and a list of goals, and returns its 1st answer *)
(* schedule : theory -> clause -> term *)
let schedule theory goal = schrec [] [goal]
  where rec schrec stack = function
[] -> if stack = [] then failwith "No solution" else schrec [] stack
   | (g::goals) -> match g with
  (answer,[],_) -> answer
        | _ -> schrec (it_list (fun q cl -> ((resolve cl g)::q) ? q) stack theory) goals
;;

(* Remark that now it would be trivial to make a coroutine giving the successive
solutions, by returning as result a triple (answer,goals,stack) *)

let search_solution theory (conjecture,dict) =
let answer = schedule theory conjecture
        in (message "Solution found";unparse_answer dict answer)
 ? (message "No solution");;

(* Example
search_solution [Ancestor2;Ancestor1] <<?Ancestor(John,John)>>;;
Solution found

And now we may check proofs in Min_Logic:
search_solution Min_Logic <<?F(A,A)>>;;
Solution found
```

So we have found a proof of `F(A,A)`. Unfortunately, the "proof" has vanished: it existed only temporarily in the control structure of the scheduler. This is rather frustrating. Let us now build a real proof-generator, as a trace of the scheduler. This trace is represented as a list of integers (the indexes of the clauses in the theory).

3.2.2 Proof construction

```
type trace == num list;;

(* build_proof : theory -> clause -> (trace & term) *)
let build_proof theory goal = buildrec [] [[],goal]
  where rec buildrec stack = fun
[] -> if stack = [] then failwith "No solution" else buildrec [] stack
   | (proof::goals) -> match proof with
  (trace,answer,[],_) -> trace,answer
        | (trace,g) ->
```

```
let stack_goals (n,q) cl = n+1,(((n::trace,resolve cl g)::q) ? q) in
    buildrec (snd (it_list stack_goals (1,stack) theory)) goals;;

Example :
let goal,_ = <<?F(A,A)>> in build_proof Min_Logic goal;;
[1; 1; 2; 3; 3],Term ("Answer",[]) : (trace & term)
```

The trace obtained in case of success is the (reverse) polish notation of a proof tree. In order to construct this proof tree, we have to look up the arities of the various clauses in the current theory.

```
(* arity : theory -> num list *)
let arity = map (fun (_,hyps,_) -> length hyps);;

Example
arity Min_Logic = [0; 0; 2]
```

In order to give a tree structure to our proofs, we must associate names with the clauses of a theory.

```
type alphabet == string list;;
```

Now we build a proof by parsing the trace, given the logic. In general we shall return a list of proofs, one for each sub-goal.

```
(* parse_proof : alphabet -> theory -> trace -> tree list *)
let parse_proof operators theory trace =
  let arit = nth (arity theory)
  and oper = nth operators in parserec [] trace
    where rec parserec stack = fun
      [] -> stack
      | (n::rest) -> let ar = arit n and ope = oper n
 in parserec (Tree(ope,rev sons)::trees) rest
 where sons,trees = let transfer (l,h::t) = h::l,t in
                  iterate transfer ar ([],stack);;

Example
parse_proof ["K";"S";"A"] Min_Logic [1; 1; 2; 3; 3];;
==> [Tree ("A",[Tree ("A",[Tree ("S",[]); Tree ("K",[])]); Tree ("K",[])])]
```

We now combine an alphabet and a theory into a logic.

```
type logic == (string & clause) list;;

let mk_logic = map (fun (name,cl,_) -> name,cl);;

(* Example *)
let Combinatory_Logic = mk_logic [
    "K",<<F(a,F(b,a))>>;
    "S",<<F(F(a,F(b,c)),F(F(a,b),F(a,c)))>>;
```

```
    "A",<<b<-F(a,b);a>>];;

(* pretty : tree -> void  is the pretty-print of the proofs in logic *)

let search_proof logic pretty (conjecture,dict) =
    let operators,theory = split logic in
        let trace,answer = build_proof theory conjecture
            in (unparse_answer dict answer;
                let proofs = parse_proof operators theory trace
                and _,thms,_ = conjecture
                and print_result (proof,thm) =
                            (pretty proof;print_string " : ";unparse dict thm)
                in map print_result (combine(proofs,thms));
                print_newline()) ? (message "No proof");;

(* Example: the standard pretty-print of Combinatory Logic *)
let print_combinator tree = print_string (mkopen [] tree)
    where rec mkopen stack = function
        Tree("A",[f;x]) -> mkopen (x::stack) f
      | Tree(c,[]) -> c ^ list_it (fun t s -> s ^ " " ^ (mkclose t)) stack ""
    and mkclose = function
        Tree("A",[f;x]) -> "(" ^ (mkopen [x] f) ^ ")"
      | Tree(c,[]) -> c;;

Example:
search_proof Combinatory_Logic print_combinator <<?F(A,A)>>;;
S K K : F(A,A)
```

3.2.3 Resolution theory and Horn clauses

Let us pause a moment to explain the generality of the method. The initial motivation came from considering a complete proof procedure for Horn sentences, i.e. for first-order sentences which can be transformed, through Skolemization (removal of existential quantifiers by introduction of new function symbols) and normalization to conjunctive normal form, to a conjunction of conditional propositions:

$$Q_1 \wedge Q_2 \wedge ... \wedge Q_n \Rightarrow P$$

where the P's and Q_i's are atomic formulas of the form: $R(t_1, ..., t_p)$ where R is a predicate (i.e. a relation symbol) and the t_j's are terms. We also allow the case where P is $False$. The two cases obviously correspond to respectively our definite clauses and our goal clauses.

The inference rule that we have programmed so far is what is known in resolution theory as "selected negative input resolution", a complete proof method for Horn clauses. Thus, if some proof of a given goal exists, we are sure to be able to find such a proof using a finite number of applications of the inference rule. Our fair scheduler strategy ensures that we properly enumerate deterministically the search space of such a non-deterministic inference rule. Of course, this does not preclude the fact that we may loop indefinitely in the search of a proof for an unsatisfiable goal, since Horn clauses satisfiability is an undecidable problem.

The theory of resolution applies more generally to first-order logic. In the general case, a clause is a (finite) set of literals, i.e. of atomic formulas given with a sign. We write $C = C_+ \leftarrow C_-$. The resolution rule is an algorithm deriving a clause from two clauses, as follows. Let C and C' be two clauses, assumed to be renamed so that they have no variable in commun. Let $\mathcal{P} \subseteq C_+$ and $\mathcal{N}' \subseteq C'_-$ be two non-empty sets of atomic formulas such that $\mathcal{P} \cup \mathcal{N}'$ is unifiable, with principal unifier σ. Let $D_+ = \sigma(C'_+ \cup (C_+ - \mathcal{P}))$, and $D_- = \sigma(C_- \cup (C'_- - \mathcal{N}'))$. We say that $D = D_+ \leftarrow D_-$ follows from C and C' by *resolution* [24].

Using Skolemization and conjunctive-normalization, any first-order formula can be transformed into a (finite) set of clauses, and this transformation respects satisfiability [2, 26]. Satisfiability (i.e. truth in some first-order structure) is equivalent to satisfiability in a Herbrand structure (whose domain is the set of ground terms and where the function symbols are kept un-interpreted). Herbrand structures are characterized by the set of ground atomic formulas which are interpreted as true. By Herbrand's theorem [14], a set of clauses is unsatisfiable iff some finite set of its ground instances is unsatisfiable. This result may be interpreted as a completeness theorem for the resolution rule.

Applications of the resolution rule may thus be interpreted semantically as exploring the search-space of Herbrand interpretations [11]. An unsatisfiable set of clauses will lead to discovery of a refutation after a finite number of applications of the resolution rule. A satisfiable set of clauses may lead to non-termination of the procedure. Such a refutation procedure may of course be turned into a semi-decision procedure for validity in an arbitrary first-order theory T: in order to show that $T \vdash P$, show equivalently that $T \wedge \neg P$ is unsatisfiable.

Many refinements of the resolution rule have been studied [2, 23, 26]. Usually full resolution is replaced by two simpler rules: binary resolution and binary factoring. Several restrictions of binary resolution are known to be complete. For instance, one may impose one of the clauses to consist only of positive (resp. negative) literals [25]. For Horn clauses, this corresponds to *positive unit* resolution (resp. *negative input* resolution). It can be shown that factoring (i.e. merging by unification of literals with the same sign) is not necessary for Horn clauses [13].

Finally, it is usually sufficient to replace clauses by ordered sequences of literals, leading to complete strategies of *selected* resolution [20, 21]. This theory validates the procedure called above `search_solution` as a correct implementation of "selected negative input" resolution. (The Prolog depth-first strategy being an incorrect implementation of the same).

Over general clauses, neither input nor unit resolution is complete. Actually, a set of clauses is refutable by input resolution iff it is refutable by unit resolution iff it is renamable into a set of Horn clauses [22]. Semantically, this can be explained by a very strong closure property of Horn clauses: the intersection of two Herbrand models is itself a model. Thus a set of Horn clauses possesses a minimum Herbrand model. Horn clauses axiomatizations may therefore be seen as systems of inductive definitions, defining a standard interpretation of the predicate symbols over free term structures [8, 1]. Algebraists say that Horn axiomatizations define *quasi-varieties*, whose class of first-order models forms a category admitting an initial object [15, 9].

Note that, at the propositional level, the distinction between general and Horn satisfiability is quite spectacular from the complexity classes of the two problems: NP-complete for the general case [5], linear for the Horn case [7].

All these phenomena reflect the restriction of expressiveness of Horn sentences with respect to full predicate logic: the non-Horn clause $P(A) \vee P(B)$ has incompatible Herbrand models $\{P(A)\}$ and $\{P(B)\}$. And the query $?P(x)$ over such an axiomatization can only lead to a non-deterministic answer $x = A \vee x = B$.

3.2.4 Polymorphic type-checking

However, all these considerations are to a great extent irrelevant to what we are actually doing. In our term-manipulation algorithms above, no distinction was made between predicate symbols and function symbols, and between term and atomic formula. We even accepted atomic formulas reduced to a variable, as in clause `A` of `Combinatory_Logic`. The essence of the computation was to search for a proof tree, with nodes labeled by polymorphic operators. That is, we may regard each definite clause

$$C \;:\; Q_1 \wedge Q_2 \wedge ... \wedge Q_n \Rightarrow P$$

as declaring an operator C of arity n with its signature: Q_i is the type of its i-th argument, and P is the type of its result. The operator is *polymorphic* in that free variables may appear in the types Q_i and P. This polymorphism is *parametric* in Reynolds' terminology, as opposed to ad hoc polymorphism (overloading). This means that clauses are really *schemas*, denoting a family of operators, whose types are all instances of the clause. Type-checking is explained simply in terms of instantiation.

Let Σ be the set of clauses in the current theory. We define what it means for a tree T to be *consistently typed* of type τ in theory Σ, which we write $\Sigma \models T : \tau$. The definition is by induction on the size of T. Assume that $C \;=\; P \leftarrow Q_1; Q_2; ...Q_n$ is in Σ, and that for some substitution σ we have $\Sigma \models T_i : \sigma(Q_i)$ for all $1 \leq i \leq n$. Then we get $\Sigma \models C(T_1, ..., T_n) : \sigma(P)$.

Remark that we have now two levels of term structure: terms such as P formed with the original *functor* alphabet Φ and variables, and ground terms such as T formed from the clause alphabet Σ. We look at the Φ-terms as *types* of the Σ-trees. A Σ *theorem* is a Φ term P which possesses a Σ-proof T, i.e. $\Sigma \models T : P$.

When the clauses of Σ are (schematic) rules defining some logical inference system, these notions correspond to the traditional ones. The point of view of considering propositions as types is not new [16], and it was well-known that combinatory logic is the algebra of proofs of minimal logic [6]. But our paradigm is completely general over a very simple formalism which encompasses all Horn-definable theories. And the Prolog point of view corresponds to generalizing proof-checking to proof synthesis. More generally, we have:

The Principal Type Theorem. Let T be a Σ-proof: $\Sigma \models T : M$. Then T possesses a *principal* type τ, such that $\Sigma \models T : \tau$, and for every M such that $\Sigma \models T : M$, we have $\tau \leq M$.

Proof. Simple induction, using the properties of the principal unifier. Let $T \;=\; C(T_1, ..., T_n)$, with $\Sigma \models T : M$. This means that $C \;:\; P \leftarrow Q_1; Q_2; ...Q_n$ is in Σ, and that $M \;=\; \sigma(P)$, with $\Sigma \models T_i : \sigma(Q_i)$. By the induction hypothesis, $\Sigma \models T_i : \tau_i$, with τ_i principal. Thus for some ρ_i we have $\sigma(Q_i) \;=\; \rho_i(\tau_i)$. We may assume without loss of generality that the τ_i are renamed so that they have no variable in common, and no variable in common with clause C. Thus the tuples $< ..., Q_i, ... >$ and $< ..., \tau_i, ... >$ are simultaneously unifiable, and their principal unifier θ gives a tuple $< ..., N_i, ... >$ such that $N_i \;=\; \theta(Q_i) \;=\; \theta(\tau_i)$. The construction defines $\tau \;=\theta(P)$ having the required properties.

Now we can explain our proof generator as follows. Let $?Q_1; ...; Q_n$ be a query (goal clause). We say that this query is *satisfiable* iff there exist proofs T_i such that $\Sigma \models T_i : P_i$ with $P_i \;=\; \sigma(Q_i)$ for some substitution σ. In that case, consider the principal types τ_i of each T_i, expressed with distinct variables not occurring in the Q_i's. Let θ be the principal unifier of tuples $< ..., \tau_i, ... >$ and $< ..., Q_i, ... >$. Then $\theta(Answer(..., x_j, ...))$ is a *principal answer* to the query, where $< ..., x_j, ... >$ is the tuple of all variables occurring in the Q_i's. The procedure `search_solution` returns a principal answer if the query is satisfiable. The procedure `search_proof` returns the corresponding proof tuple $< ..., T_i, ... >$ as well. The coroutine extension of these procedures would progressively generate all principal answers.

Remark. This point of view of proof trees as operator terms over a polymorphic definite clause alphabet may be extended to general clauses. However, a non-Horn clause must then be interpreted as the type specification of a non-deterministic operator.

References

[1] K. R. Apt and M. H. van Emden. "Contributions to the Theory of Logic Programming . JACM **29,3** (1982) 841–862.

[2] C. Chang and R. Lee. "Symbolic Logic and Mechanical Theorem-Proving." Academic Press (1973).

[3] K. Clark and S. Tärnlund. A first-order theory of data and programs. IFIP 77 proceedings, North-Holland (1977) 939–944

[4] A. Colmerauer, H. Kanoui, R. Pasero, Ph. Roussel. "Un système de communication homme-machine en francais." Rapport de recherche, Groupe Intelligence Artificielle, Faculté des Sciences de Luminy, Marseille (1973).

[5] S. A. Cook. "The Complexity of Theorem-proving Procedures." Proc. Third ACM Symp. on Theory of Computing (1971) 151–158.

[6] H. B. Curry, R. Feys. "Combinatory Logic Vol. I." North-Holland, Amsterdam (1958).

[7] W. F. Dowling and J. H. Gallier. "Linear-time algorithms for testing the satisfiability of propositional Horn formulæ." J. Logic Programming **1,3** (1984) 267–284.

[8] M. van Emden and R. Kowalski. "The semantics of predicate logic as a programming language." JACM **23,4** (1976) 733-742.

[9] F. Galvin. "Horn sentences." Annals of Math. Logic **1,4** (1970) 389–422.

[10] J. Guard. "Automated Logic for Semi-Automated Mathematics." Scientific Report 1, AFCRL (1964).

[11] P. Hayes. "Semantic trees." Ph. D. Thesis, University of Edinburgh (1973).

[12] L. Henschen. "Semantic resolution for Horn sets." 4th IJCAI, Tbilisi (1975).

[13] L. Henschen and L. Wos. Unit refutations and Horn sets. JACM **21,4** (1974) 590–605.

[14] J. Herbrand. "Recherches sur la théorie de la démonstration." Thèse, U. de Paris (1930). In: Ecrits logiques de Jacques Herbrand, PUF Paris (1968).

[15] A. Horn. "On sentences which are true of direct unions of algebras." Journal of Symbolic Logic **16,1** (1951).

[16] W. A. Howard. "The formulæ-as-types notion of construction." Unpublished manuscript (1969). Reprinted in to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Eds J. P. Seldin and J. R. Hindley, Academic Press (1980).

[17] N. D. Jones and W. T. Laaser. "Complete Problems for Deterministic Polynomial Time." Theor. Comp. Sci. **3** (1977) 107-117.

[18] R. Kowalski. "Logic for Problem Solving." North-Holland (1979).

[19] R. Kowalski and M. H. van Emden. "The Semantics of Predicate Logic as a Programming Language." JACM **23,4** (1976) 733-742.

[20] R. Kowalski and D. Kuehner. "Linear Resolution with Selection Function." Artificial Intelligence **2** (1970) 227-260.

[21] D. Kuehner. "Some special-purpose resolution systems". Machine Intelligence **7** (1972) Edinburgh U. Press, 117–128.

[22] H. Lewis. "Renaming a set of clauses as a Horn set". JACM **25,1** (1978) 134–135.

[23] D. Loveland. "Automated theorem proving: a logical basis." North-Holland (1978).

[24] J. A. Robinson. "A Machine-Oriented Logic Based on the Resolution Principle ." JACM **12** (1965) 32–41.

[25] J. A. Robinson. "Automatic deduction with hyper-resolution." Intern. Jour. of Computer Math **1** (1965) 227–234.

[26] J. A. Robinson. "Logic: Form and Function." North-Holland (1979).

[27] Ph. Roussel. "PROLOG: manuel de référence et d'utilisation." Groupe d'Intelligence Artificielle, Faculté des Sciences de Luminy, Marseille (1973).

Chapter 4

Termination

This chapter deals with methods for proving the finiteness of sequences of structures such as trees and terms, and thus for proving the termination of computations seen as term rewriting. This version is very preliminary, and consists of notes written by J. J. Lévy.

4.1 Kruskal's theorem

4.1.1 The tree theorem

The following conjecture of Varojny has been proved by Kruskal [5]. Let Σ be a finite set of labels and T be trees labeled by elements of Σ. The embedding relation on $T \times T$ is the minimum quasi-ordering satisfying:

- $t \leq f(..., t, ...)$ where f is in Σ and t in T,
- $f(..., ...) \leq f(..., t, ...)$, where t is in T,
- $f(..., t, ...) \leq f(..., u, ...)$ if t $\leq u$ where t and u are in T.

Then in any infinite sequence of terms $\{t_1, t_2, ...\}$, there are at least two integers i and j such that $t_i \leq t_j$ and $i < j$.

Proof: not very constructive. Consider a minimal counterexample $\{u_1, u_2, ...\}$ such that the size of u_i is minimal for every i. We start with two remarks.

First, in any infinite sequence $\{w_1, w_2, ...\}$ of direct subtrees of the u_i's, there are i and j such that $i < j$ and $w_i \leq w_j$. This is proved as follows. Consider the initial sequence $\{u_1, u_2, ..., w_1, w_2, ..\}$ where we just changed u_k by its corresponding subtree w_1 and continued with w_2, w_3, ... It cannot be a counterexample since the size of w_i is strictly less than the size of u_k. Which elements are related? It can't be $u_i \leq u_j$ for $i < j$. It cannot either be $u_i \leq w_j$, since $w_j \leq u_l$ for the u_l of which w_j is a subtree and then $u_i \leq u_l$ for $i < l$. The only possible solution is $w_i \leq w_j$ for some i and j such that $i < j$.

Second, note that in any of these sequences $\{w_1, w_2, ...\}$ of subtrees, there is an infinite ascending subsequence for $\leq$. We saw that $w_i \leq w_j$ for some i and j such that $i < j$. We go on with w_j trying to make an ascending chain. Either this process is infinite and we are done. Either, it stops at w_k and we iterate the argument on the sequence following w_k. Again either there is an infinite ascending chain in it and we are done. Or again we are stuck at some maximal point. And the iteration goes on ... If the iteration does not terminates, it means that we found an infinite number of maximal trees w.r.t to their followers. But these form an infinite subsequence of $\{w_1, w_2, ...\}$ and there must be two related trees. So the iteration stops and there is an infinite ascending chain subsequence of $\{w_1, w_2, ...\}$.

Now, since the set Σ of labels is finite, there is an infinite subsequence of $\{u_1, u_2, ...\}$ labelled at the top level with the same f. Let $\{v_1, v_2, ...\}$ be this subsequence. We claim that $v_i \leq v_j$ for some i and j such that $i < j$. Which will lead to a contradiction.

Consider the number n_i of direct subtrees of the v_i. Either there is a an infinite subsequence of n_i with the same value (say n), or there is an an infinite subsequence strictly increasing (starting with say n). In both cases we focus on this value n. Let $\{w_1, w_2, ...\}$ be the corresponding infinite subsequence of $\{v_1, v_2, ...\}$. Consider the sequence of the leftmost direct substrees. It contains an infinite ascending subsequence w.r.t. $\leq$. We can iterate the argument on the second subtree from the left until n. Finally we get $f(t_1, t_2, ...t_n) \leq f(t'_1, t'_2, ...t'_n)$. When the chain of the n_i's is strictly increasing, we have also $f(t_1, t_2, ...t_n) \leq f(t'_1, t'_2, ...t'_n, ...t_p)$. Thus the minimum counterexample $\{u_1, u_2, ...\}$ cannot exist. And the proof is finished.

4.1.2 Formal statement

There is a theory of well quasi orderings defined by Kruskal in which the tree theorem can be rephrased very simply.

Definition: A well quasi ordering (w.q.o.) is any relation $\leq$ such that in any infinite sequence there is an an infinite ascending subsequence.

An example is the set of positive integers with the usual $\leq$. Also, an easy corollary is that w.q.o. are well founded (i.e. without any infinite strictly descending chain). This definition is similar with the one imposing only two elements to be related in any infinite sequence, that is i and j such that $t_i \leq t_j$ and $i < j$. We did the argument inside the proof of the tree theorem. In short, if we consider the set of maximal points with respect to their followers, this set must be finite. Otherwise the corresponding subsequence will violate the definition. Finally, in the case of the free monoid, Kruskal's theorem is known as Higman's.

Theorem: Well quasi orderings are closed under embedding.

This is the formal way of describing the proof previously done on trees, and obviously it is the same proof. The main remark is that the equality on finite sets forms a w.q.o.

4.1.3 Functionals on orderings

Given an ordering $>$ on a set T, one can look at orderings on T^n (cartesian product) or on TM (multisets of T). There are straightforward ones which preserves well-foundedness.

- Lexicographic for the product: $(t_1, t_2) > (u_1, u_2)$ if either $t_1 < u_1$ or $t_1 = u_1$, and $t_2 > u_2$.

- Multiset ordering: $\{t_1, t_2, ...t_n\} > \{u_1, u_2, ...u_p\}$ if either the t_i's are equal to the u_i's except at least one of the u_i's is missing, or one the u_i has been replaced by an arbitrary multiset of strictly less terms of T.

It is easy to prove that both these functionals on orderings keep well foundness. For multisets, it is an easy consequence of Kœnig's lemma.

4.2 W.q.o. and rewriting systems

In "The Art of Programming" [4] (p.385), the tree theorem is quoted as exercise 8 [M39] and it is said "This fact may be used to prove that certains algorithms must terminate". Dershowitz [2] made the connexion with termination of term rewriting systems. The rest of the notes reports mainly his paper.

Remark 1: Any partial ordering $<$ on first order terms is well-founded as soon as it verifies the following two rules:

- $t < f(..., t, ...)$
- $t < u$ implies $f(..., t, ...) < f(..., u, ...)$

Obvious, since it contains the embedding relation. In fact a stronger form of it can be proved using quasi-orderings with the same two rules.

Remark 2: Any rewriting system such that $t > u$ for every instantiation of a rewrite rule $t \rightarrow u$ terminates when $>$ is an ordering verifying the rules of the previous remark.

Finding a more contructive form of these so-called "simplification orderings" in [2] will be the rest of the notes. There could seem to be a trouble with the universal quantifier on instantiations of each of the (supposed finite) number of rewrite rules. But clearly one can take advantage of the subexpression rule.

4.3 Recursive path orderings

The following definition of an ordering on first order terms is due to Dershowitz [2] and is a significant simplification of an ordering named "recusive path ordering" by Plaisted [9, 10]. The idea is to give weight to functions symbols in Σ. Intuitively, $f \succ g$ if the function represented by f is defined in terms of g like in primitive recursive functions. We can assume Σ finite and it is clear then that $\preceq$ is a w.q.o. on Σ.

Definition 1: Recursive path ordering (r.p.o.). Let $t = f(t_1, t_2, ..., t_n)$ and $u = g(u_1, u_2, ..., u_p)$. Then the r.p.o. $t > u$ induced by $\succ$ is recursively defined by:

- (1) $f \succ g$ and $t > u_i$ for all i, or
- (2) $t_i > u$ or $t_i = u$ for some i, or
- (3) $f = g$ and $\{t_1, t_2, ...t_n\} \gg_M \{u_1, u_2, ...u_p\}$ in the multiset ordering induced by $>$ on multiset of terms.

Now, it is easy to check that r.p.o is a partial ordering. And it is well founded (using remark 1).

Example 1: (Disjunctive normal forms)

- $\neg(\alpha \vee \beta) \rightarrow \neg\alpha \wedge \neg\beta$,
- $\neg(\alpha \wedge \beta) \rightarrow \neg\alpha \vee \neg\beta$,
- $\alpha \wedge (\beta \vee \gamma) \rightarrow (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$,

- $(\alpha \vee \beta) \wedge \gamma \to (\alpha \wedge \gamma) \vee (\beta \wedge \gamma)$,
- $\neg\neg\alpha \to \alpha$,
- $\alpha \vee \alpha \to \alpha$,
- $\alpha \wedge \alpha \to \alpha$,

This system can easily be proved to terminate by ordering $\{\neg, \wedge, \vee\}$ from left to right in a decreasing chain and considering the r.p.o. induced by this relation on terms.

Example 2: (Free Groups, completed by the Knuth-Bendix technique)

- $(x * y) * z \to x * (y * z)$,
- $U * x \to x$,
- $I(x) * x \to U$,
- $I(x) * (x * y) \to y$,
- $I(U) \to U$,
- $x * U \to x$,
- $I(I(x)) \to x$,
- $x * I(x) \to U$,
- $x * (I(x) * y) \to y$,
- $I(x * y) \to I(y) * I(x)$.

Taking $\{I, *, U\}$ ordered by $\succ$ from left to right seems to induce a valid r.p.o. except for the associativity rule which needs some lexicographic property instead of the multiset ordering. And the multiset ordering has never been used here. More generally, some function symbols are recursively defined by lexicographic ordering on their arguments and other behave better with the multiset way. So given an ordering $>$ on the Cartesian product T_n or on multisets TM, one wants to associate an ordering $O(>)$ on the corresponding space. As O may depend on the top function symbol and on special values of arguments, it is easier to take O as being a functional of ordering on terms to orderings to terms, with the idea that it could be multiset of lexicographic orderings on the arguments. Also, if we have in mind that O could be the lexicographic ordering, then we have to ensure that the arguments skept by the lexicographic order should not dominate the left hand side. So O should verifies:

- **Axiom 1**: O preserves transitivity,
- **Axiom 2**: O preserves irreflexivity,
- **Axiom 3**: If $t > u$, then $f(..., t, ...)O(>)f(..., u, ...)$,
- **Axiom 4**: O is a continuous function on relations.

All these axioms are used in the following definition. Axiom 3 makes the definition to be monotonic on terms. Axiom 4, which may look curious, but which is true of any relation you may think of, means that we can make an easy inductive definition.

Definition 2: Recursive path ordering (revision 1) [3]. Let $t = f(t_1, t_2, ..., t_n)$ and $u = g(u_1, u_2, ..., u_p)$. Then the r.p.o. $t > u$ induced by $\succ$ is recursively defined by:

(1) $f \succ g$ and $t > u_i$ for all i, or

(2) $t_i > u$ or $t_i = u$ for some i, or

(3) $f = g$, and $tO(>)u$ and $t > u_i$ for all i.

We prove that this r.p.o is an ordering containing the embedding relation. First we remark that the continuity of O allows to define the r.p.o as the union of $>_n$ for all n, where $t >_n u$ is recursively defined, when $t = f(t_1, t_2, ..., t_k)$ and $u = g(u_1, u_2, ..., u_k)$, by:

(1_n) $f \succ g$ and $t >_n u_i$ for all i, or

(2_n) $t_i >_n u$ or $t_i = u$ for some i, or

(3_n) $f = g$, $tO(>_{n-1})u$, and $t >_n u_i$ for all i.

Proof for **transitivity**. By induction on $< n, |t|, |u|, |v| >$.

If $n = 0$, the empty relation is transitive. Suppose $>_n$ transitive. Let $t >_n u >_n v$. We have 9 cases to consider to prove $t >_n v$.

Case $(1_n - 1_n)$: $f \succ g \succ h$. Thus $f \succ g$. Also $t >_n u >_n v_i$ for all i. So by induction, $t >_n v_i$, and rule (1_n) gives $t >_n v$.

Case $(1_n - 2_n)$: $t >_n u_i$ for some u_i, and $u_i >_n v$ or $u_i = v$. So by induction $t >_n v$.

Case $(1_n - 3_n)$: $f \succ g = h$. Thus $f \succ h$. Also, as in case $(1_n - 1_n)$, induction gives $t >_n v_i$ for all i and again $t >_n v$ by rule (1_n).

Case $(2_n - *)$: $t_i >_n u$ or $t_i = u$ for some i. Thus by induction $t_i >_n v$ and by rule (2_n) $t >_n v$.

Case $(3_n - 1_n)$: $f = g \succ h$. Thus $f \succ h$. Then $t >_n u >_n v_i$ for all i. Thus $t >_n v_i$ by rule (1_n).

Case $(3_n - 2_n)$: $t >_n t_i$ for all i. But $t_i >_n u$ or $t_i = u$ for some i. This implies $t >_n u$ by induction.

Case $(3_n - 3_n)$: $f = g = h$. And $tO(>_{n-1})uO(>_{n-1})v$. By induction on n, we know that $>_{n-1}$ is transitive. Moreover O preserves transitivity. Thus $tO(>_{n-1})v$. Now, $t >_n u >_n v_i$ for all i. Thus again by induction $t >_n v_i$. So by rule (3_n), $t >_n v$.

Proof for **irreflexivity**. By induction on $< n, |t| >$.

If $n = 0$, the empty relation is clearly irreflexive. Suppose by induction that $>_n$ is irreflexive, then we can't have $>_n$ by rule (1_n), since $\succ$ is irreflexive. If by rule (2_n), we have $t_i >_n t$. Then, as $t >_n t_i$ also by rule (2_n), and as transitivity (just proved) is true for $>_n$. Then $t_i >_n t_i$. Impossible by induction. If $t >_n t$ by rule (3_n), then $tO(>_{n-1})t$. But $>_{n-1}$ is irreflexive and O is supposed to preserve irreflexivity. Contradiction.

Proof for **well-foundedness**. We show that $>$ is a simplification ordering.

First, $t > t_i$ for any subexpression t_i of t. Second, let $t > u$. In order to prove $f(..., t, ...) > f(..., u, ...)$, we have to use rule (3). And the monotonicity axiom on O gives the solution.

Example 2 (groups revisited). We just take the lexicographic ordering from left to right on $*$.

Example 3 (Ackerman's function).

- $A(0,x) \rightarrow ...$,
- $A(s(x),0) \rightarrow ...$,
- $A(s(x),s(y)) \rightarrow A(x,A(s(x),y))$

Example 4:

- $f(g(x),y,y) \rightarrow g(f(x,x,y))$,

4.4 Extended recursive path orderings

The principle of the extension is twofold: first we want to treat accurately our first extension. (In fact, the careful reader can see that Definition 2 is not an extension of the one in [2].) Second, we would like to take advantage of some semantic information like some well understood well-founded ordering such as the traditional order on integers as in the following example:

- $fact(sx) \rightarrow x * fact(p(sx))$
- $p(sx) \rightarrow x$ ...

So the idea will be to capture with $\simeq$ and $\succeq$ both the previous order on definitions of symbols and the natural interpretation (at the semantic level) of the rewrite rules. First, we remark that the embedding relation induced by a w.q.o. $\succeq$ is the least q.o. such that:

- $f(...,t,...) \geq t$,
- $t = f(t_1,t_2,...,t_n) \geq g(u_1,u_2,...,u_p) = u$ if $t \succeq u$ and $t_{k_i} \geq u_i$ for $1 < i < p$ and $1 \leq k_1 < k_2 < ... < k_p \leq n$.

So the embedding with respect to a w.q.o. $\succeq$ will also be a w.q.o. by the same proof as the one of the tree theorem. Thus we may restart the r.p.o. definition but w.r.t. to a given $\succeq$. We assume that $\succeq$ is a w.q.o., that $\simeq$ is its associated equivalence, and $\succ$ the associated strict partial ordering. In the following, we will need this tedious list of axioms:

- **Axiom 1**: O preserves strict orderings,
- **Axiom 2**: O is a continuous function on relations.
- **Axiom 3**: E preserves equivalences,
- **Axiom 4**: E is a continuous functions on relations,
- **Axiom 5**: $>\equiv$ contained in $>$ implies $O(>)E(\equiv)$ contained in $O(>)$,
- **Axiom 6**: $\equiv>$ contained in $>$ implies $E(\equiv)O(>)$ contained in $O(>)$,
- **Axiom 7**: $f(t_1,t_2,...t_n)E(\equiv)g(u_1,u_2,...u_p)$ implies that for all u_i, there is t_j such that $t_j \equiv u_i$.
- **Axiom 8**: $f(t_1,t_2,...t_n)E(\equiv)g(u_1,u_2,...u_p)$ implies that for all t_i, there is u_j such that $t_i \equiv uj$.
- **Axiom 9**: If $t = f(t_1,t_2,...t_n) \simeq u = g(u_1,u_2,...u_p)$ and $t_{k_i} \geq u_i$ for $1 < i < p$ and $1 \leq k_1 < ... < k_p \leq n$, then $tO(>)u$ or $tE(\equiv)u$.

- **Axiom 10**: If $t > u$, then $f(..., t, ...)O(>)f(..., u, ...)$,
- **Axiom 11**: If $t \to u$, then $f(..., t, ...) \succeq f(..., u, ...)$,

Definition 3: Recursive path ordering (revision 2) [3].
Let $t = f(t_1, t_2, ..., t_n)$ and $u = g(u_1, u_2, ..., u_p)$. Then the r.p.o. $t > u$ induced by $\succ$ and $\simeq$ is recursively defined by:

(1) $t \succ u$ and $t > u_i$ for all i, or

(2) $t_i > u$ or $t_i \equiv u$ for some i, or

(3) $t \simeq u$, and $tO(>)u$ and $t > u_i$ for all i.

and

(4) $t \equiv u$ iff $t \simeq u$ and $tE(\equiv)u$, or $t = u$.

The following proofs are done by parallel induction on $< n, |t|+|u|+|v| >$. Also t_i, u_i and v_i stand for direct subterms of t, u and v.

Proof of **transitivity**. If $n = 0$, the empty relation is transitive. Suppose $>_{n-1}$ transitive. Let $t >_n u >_n v$. We have 16 cases to consider to prove $t >_n v$.

Case $(1_n - 1_n)$: $t \succ u \succ v$. Thus $t \succ v$. Also $t >_n u >_n v_i$ for all i. So by induction, $t >_n v_i$, and rule (1_n) gives $t >_n v$.

Case $(1_n - 2_n)$: $t >_n u_i$ for some u_i, and $u_i >_n v$ or $u_i \equiv_n v$. So by induction one has $t >_n v$.

Case $(1_n - 3_n)$: $t \succ u \simeq v$. Thus $t \succ v$. Also, as in case $(1_n - 1_n)$, induction gives $t >_n v_i$ for all i and again $t >_n v$ by rule (1_n).

Case $(1_n - 4_n)$: $t \succ u \simeq v$. Thus $t \succ v$. Also $t >_n u_i$ for all i. Thus by axiom 7, there is v_j such that $u_i \equiv_n v_j$. By induction $t >_n v_i$ for all i. And by rule (1_n), one gets $u >_n v$.

Case $(2_n - *_n)$: $t_i >_n u$ or $t_i \equiv_n u$ for some i. Thus by induction on left compatibility of $\equiv_n$, one has $t_i >_n v$ and by $t >_n v$ by rule (2_n).

Case $(3_n - 1_n)$: $t \simeq u \succ v$. Thus $t \succ v$ by left compatibility of $\simeq$. Then $t >_n u >_n v_i$ for all i. Thus $t >_n v_i$ by rule (1_n).

Case $(3_n - 2_n)$: $t >_n t_i$ for all i. But $t_i >_n u$ or $t_i \equiv_n u$ for some i. This implies $t >_n u$ by induction.

Case $(3_n - 3_n)$: $t \simeq u \simeq v$. And $tO(>_{n-1})uO(>_{n-1})v$. By induction on n, we know that $>_n$ is transitive. Moreover O preserves transitivity. Thus $tO(>_n)v$. Now, $t >_n u >_n v_i$ for all i. Thus again by induction $t >_n v_i$.

Case $(3_n - 4_n)$: $t \simeq u \simeq v$. Thus $u \simeq v$. Also $tO(>_{n-1})uE(\equiv_{n-1})v$. Thus by axiom 5, $tO(>_{n-1})v$. Also $u >_n v_i$ for all i. So by induction $t >_n v_i$ and $t >_n v$ by rule (3_n).

Case $(4_n - 1_n)$: $t \simeq u \succ v$. Thus $t \succ v$. Also we have $u >_n v_i$ for all i. Then $t >_n v_i$ by induction. And $t >_n v$ by rule (1_n).

Case $(4_n - 2_n)$: $t \equiv_n u$ and $u_i \equiv_n v$ or $u_i >_n v$. By axiom 8, one has $t_j \equiv u_i$ for some j. By induction $u_j \equiv_n v$, and $u >_n v$ by rule (2_n). Otherwise, still by induction $u_j >_n v$, and $u > v$ by rule (2_n).

Case $(4_n - 3_n)$: $t \simeq u \simeq v$. Thus $t \simeq v$. Also, $tE(\equiv_{n-1})u$ and $uO(>_{n-1})v$. Thus by axiom 6, $tO(>_{n-1})v$. As $u >_n v_i$ for all i. Then by induction $t > v_i$ for all i, and $t > v$ by rule (1_n).

Case $(4_n - 4_n)$: $t \simeq u \simeq v$. Thus $t \simeq v$. Also $tE(\equiv_{n-1})uE(\equiv_{n-1})v$. Thus as E preserves transitivity by Axiom 3, $tE(\equiv_{n-1})v$. Thus $t \equiv_n v$ by rule (4_n).

Proof of **reflexivity** of $\equiv_n$. Obvious since the equality is contained in $\equiv$.

Proof of **irreflexivity** of $>_n$.

If $n = 0$, the empty relation is clearly irreflexive. Otherwise:

Case (1_n): Impossible since $\succ$ is irreflexive.

Case (2_n): We have $t_i \equiv_n t$ or $t_i >_n t$. Then also, $t >_n t_i$ by rule (2_n) since $t_i \equiv_n t_i$ by reflexivity of $\equiv_n$. Thus $t_i >_n t_i$ by transitivity. Impossible by induction.

Case (3_n): We have $tO(>_{n-1})t$. But $>_{n-1}$ is irreflexive and O is supposed to preserve irreflexivity by axiom 1. Contradiction.

Proof of **symmetry** of $\equiv_n$. If n = 0, then $\equiv_0$ is the standard equality, which is symmetric. Otherwise we have $t \simeq u$ and $tE(\equiv_{n-1})u$. By induction and by use of axiom 3, one gets $uE(\equiv_{n-1})v$. Also $u \simeq v$. And $t \equiv_n u$ by rule (2_n).

So we have proved that $>$ is a strict ordering and $\equiv$ a compatible equivalence relation. For this, we needed Axioms (1-8). All of these are rather natural, because they mean preservation of orderings, equivalences, and left-right compatibility. Trouble is with axioms 7-8. They mean more or less that E is not far from the standard equality, but it could be permutations of subterms in the case of the multiset orderings. Axiom 9 reflects the embedding property that we will discuss on. One has just to remember that in general it is applied with $f = g$.

Proof of **well-foundedness** of $>$.
We show that $\geq$ is contained in the embedding defined by $\succeq$. First the subexpression property is trivially true. Second, suppose that $t_{k_i} \geq u_i$ for all $1 < i < p$ where $1 \leq k_1 < k_2 < ...k_p \leq n$ and let $t = f(t_1, t_2, ..., t_n)$, $u = g(u_1, u_2, ..., u_p)$, $t \succeq u$. Then we remark first that $t \geq t_{k_i}$ and $t_{k_i} \geq u_i$ for $1 < i < p$. Thus $t \geq u_i$ for all i. Now, we have two cases:

- $t \succ u$ and $t \geq u$ follows by rule (1), or
- $t \simeq u$. But axiom 9 gives $t \geq u$ by rule (3) or (4).

So, if $\succeq$ is w.q.o., then $\geq$ is a w.q.o. Now if $\succ$ is only a well-founded partial ordering, one can complete it into a total well-founded ordering which is then a w.q.o. (cf. Birkhoff). Moreover, the r.p.o defined by such a well founded partial $\succ$ is contained in the r.p.o induced by the larger $\succ$. And we can still make this new order compatible with $\simeq$ by only ordering the quotient set induced by $\simeq$. Thus the initial r.p.o. is contained in a w.q.o. which is well-founded. So the initial r.p.o is well founded.

Proof of **monotonicity** of $> \wedge \rightarrow$.
It is sufficient to prove $t > u$ for all instantiations of the rewrite rules. For this, we need to show that the intersection of $>$ and $\rightarrow$ is monotonous. So suppose that $t > u$ and $t \rightarrow u$. First $f(..., t, ...) \succeq f(..., u, ...)$ since $t \rightarrow u$ by axiom 11. Now, we can use rules (1) or (3) by use of axioms 9 and 10 (as done in the well-foundedness proof).

Completeness of the method.

Just take $t \succ u$ iff $t \rightarrow v$ and u is a subexpresion of v. Write in short $t \rightarrow C[u]$. Then clearly if $\rightarrow$ is terminating, then $\succ$ is well founded. Now, if $t \rightarrow C[u]$, we have $t > u$ by induction on $|u|$ and using the rules (1) and (2). Thus in the case of the empty contex $C[]$, we have $t > u$ if $t \rightarrow u$.

Example 5.

- $fact(sx) \rightarrow x * fact(p(sx))$

- $p(sx) \to x$
- $\cdots$

Then we can define $t \succ u$ and $t \simeq u$ by:

- $fact(t) \succ u * v$ for all t, u, v,
- $fact(t) \succ p(u)$ for all t, u,
- $fact(t) \succ fact(u)$ if $N \models t > u$ where N is the set of integers.
- $fact(t) \simeq fact(u)$ if $N \models t = u$
- $t \simeq u$ for all t, and u if the top function symbol is in $\{*, p, s\}$.

Clearly, $\succ$ is well-founded and the system checks the r.p.o. when E and O are the identity functional on direct subterms orderings.

4.5 Conclusion

The r.p.o. method as described in [2] is very powerful. However it is not clear whether it could be applied to proving the termination of typed lambda calculus. Furthermore, the method seems to fit the case of conditional rewriting systems (as for defining the operational semantics of programming languages having for instance conditional expressions or based on an inductive definition on the structure of terms). Recently, L. Puel [11, 12] proposed a generalisation by using the notion of the so-called "unavoidable sets". Also we should mention that inside the REVE system, there is another notion of r.p.o. in the manner of Plaisted [6, 7].

References

[1] G. Cousineau. "Preuves de Terminaison des systèmes de réécriture". Notes de Cours de DEA, Université de Paris 7, (1983).

[2] N. Dershowitz. "Orderings for term rewriting systems". FOCS 1979 and TCS, March 1982.

[3] S. Kamin, J. J. Lévy. "Two generalisations of recursive path orderings". Unpublished note, 1980.

[4] D. Knuth. "The Art of Programming", Vol 1, Addison-Wesley (1968).

[5] J. Kruskal. "Well quasi orderings, the tree theorem, and Varsony'is conjecture". Trans. Amer. Math. Soc., 95 (1960), 210–225.

[6] P. Lescanne. "Decomposition ordering as a tool to prove the termination of rewriting systems'. Centre de Recherche en Informatique de Nancy, Rapport RG 2-82 (1982).

[7] P. Lescanne. "Two Implementations of Recursive Path Ordering on Monadic Terms". Centre de Recherche en Informatique de Nancy, Rapport RG 11-82 (Sept. 1982).

[8] C. St. J. A. Nash-Williams. "On Well-quasi-ordering Finite Trees". Proc. Cambridge Phil. Soc. 59 (1963), 833–835.

[9] D. Plaisted. "Well-founded orderings for proving termination of systems of rewrite rules". Report R-78-932, Dept of Computer Science, University of Illinois (1978).

[10] D. Plaisted. "A recursively defined ordering for proving termination of system of rewrite rules". Report R-78-943, Dept of Computer Science, University of Illinois (1978).

[11] L. Puel. "A generalisation of Kruskal's theorem". Submitted to publication (1985).

[12] L. Puel. "A generalisation of the recursive path ordering method". Submitted to publication, (1985).

[13] L. E. Sanchis. "Termination of typed lambda calculus" Notre Dame Journal of Symbolic Logic, 1968.

Chapter 5

Equational Reasoning by Canonical Simplification

We consider in this chapter the theory of equational axiomatizations. The main paradigm is to try and replace equational non-deterministic deductions by deterministic computations by rewrite rules. Rewrite rules are oriented equations, and when a set of such rules is confluent (deterministic) and nœtherian (terminating) it may be used as a decision procedure for the corresponding equational presentation. We present the Knuth-Bendix decision procedure for verifying confluence of a finite set of first-order term rewrite rules. Finally we explain the Knuth-Bendix procedure for attempting the completion of a set of rules to a confluent one. Additional material on equational theories and rewrite rules may be found in the author's survey [11].

5.1 Equational Logic

We first remark that equational logic inference rules can be considered as definite clauses, and thus that it is possible in principle to search for equational proofs using the general mechanism.

```
(* The parser knows that = and * are infix functors *)
let Equational_Logic = mk_logic [
    "Refl",<<x=x>>;
(*  "Sym",<<x=y <- y=x>>; *)
    "Trans",<<x=z <- x=y; y=z>> ];;
let Group_theory = Equational_Logic @ mk_logic [
    "Ass",<<(x*y)*z=x*(y*z)>>;
    "Idl",<<1*x=x>>;
    "Invl",<<I(x)*x=1>>;
    "Congr*",<<x*y=u*v <- x=u; y=v>>;
    "CongrI",<<I(x)=I(y) <- x=y>>];;
*)

let unparse tr = unparserec tr;print_newline()
  where rec unparserec (Tree (ope,sons)) =
print_string ope;
match sons with
```

```
[] -> ()
| (t :: lt) -> print_string "(";
       unparserec t;
       map (fun t -> print_string ",";unparserec t) lt;
       print_string ")";;
search_proof Group_theory unparse <<?(A*1)*B=A*B>>;;
==> Trans(Ass,Congr*(Refl,Idl))
```

This is however an absurdly costly way of doing equational proofs. The next idea is to consider equations as oriented rewrite rules. We assume familiarity with the basic notions of equational logic and term rewriting systems [8, 11].

5.2 Term rewriting

```
(* cross between map and it_list *)
(* num_map : (num -> 'a -> 'b) -> 'a list -> 'b list *)
let num_map f list = let consf (i,l) li = (i+1,(f i li)::l) in
    rev (snd (it_list consf (1,[]) list));;
(* Example
num_map pair ["a";"b";"c"];;
==> [1,"a"; 2,"b"; 3,"c"] : (num # string) list
*)
(* We assume "=" is a binary functor, written concretely in infix notation *)
(* Terms whose main functor is "=" are called equations *)
load_syntax "eq";;

(* Example
<<F(x)=x>> = Term ("=",[Term ("F",[Var 1]); Var 1]),["x",1],1 : concrete_term
*)

type term_pair == term & term;;

(* standardizes an equation so its variables are 1,2,... *)
let mk_rule M N = let all_vars = union (vars M) (vars N) in
  let k,subst = it_list (fun (i,sigma) v -> (i+1,(v,Var(i))::sigma)) (1,[]) all_vars
  in (k-1, substitute subst M, substitute subst N);;

(* Rewrite rules are structures (n,k,L,R) where L and R are terms, k is the
  number of distinct variables in L and R, and n is an identification number *)

type rule ==  (num & num & term_pair);;

(* We need to print terms with variables independently from input terms
  obtained by parsing. We give arbitrary names v1,v2,... to their variables. *)
let INFIXES = ref ["=";"*"];;
let rec pretty_term = fun
(Var n) -> print_string ("v" ^ string_of_num n)
```

```
     |  (Term (oper,sons)) -> if mem oper !INFIXES then
          (pretty_close (hd sons);print_string oper;pretty_close (hd (tl sons)))
                           else (print_string oper;
                                  match sons with
              [] -> ()
                     | (t :: lt) -> print_string "(";
                                    pretty_term t;
                                    map (fun t -> print_string ",";pretty_term t) lt;
                                    print_string ")")
and pretty_close M = match M with
Var(n) -> print_string ("v" ^ string_of_num n)
     |  Term(oper,sons) -> if mem oper !INFIXES then
                (print_string "(";pretty_term M; print_string ")")
                   else pretty_term M;;

let pretty_rule (k,n,M,N) = print_num k;
    print_string " : "; pretty_term M; print_string " = "; pretty_term N;
    print_newline();;

(* Top-level rewriting. Let eq:L=R be an equation, M be a term such that L<=M.
  With sigma = matching L M, we define the image of M by eq as sigma(R) *)
let reduce L M = substitute (matching L M);;

let top_rewrite (L,R) M = reduce L M R;;

(* One step of rewriting in leftmost-outermost strategy *)
(* fails if no redex is found *)
let rewrite1 (L,R) = rewrec
where rec rewrec M = reduce L M R
    ? let (Term(f,sons)) = M in Term(f,tryrec sons)
    where rec tryrec (son::rest) = (let son'=rewrec son in son'::rest)
                                 ? son::tryrec rest;;

(* A more efficient version of can (rewrite1 (L,R)) for R arbitrary *)
let reducible L = redrec
where rec redrec M = (matching L M;true)
                    ? match M with Term(_,sons) -> exists redrec sons
                              | _ -> false;;

(* Iterating rewrite1. Returns a normal form. This may loop forever *)
let rewrite_all eq M = rew_loop M
where rec rew_loop M = rew_loop(rewrite1 eq M) ? M;;

(* Remark that we are obliged to go back to the top of the term after
  each step, since rewriting may create a new redex arbitrary high
  above the current one. Consider for instance eq = <<F(x,x)=x>>, and
  M = <<F(G(G(...(G(A))...)),G(G(...(G(F(A,A)))...)))>>. If the left-hand
```

```
  side L of eq is linear, this phenomenon cannot occur, and a new redex may
  only occur above a reduct at a distance bounded by its height. *)

(* Let us now consider a set of rewrite rules *)
type rules == rule list;;

let pretty_rules = map pretty_rule;;

(* mk_rules : concrete_term list -> rules *)
let mk_rules = num_map (fun n (Term("=",[l;r]),_,k) -> (n,k,l,r));;

(* Example
let Group_rules = mk_rules [
    <<U*x=x>>;
    <<I(x)*x=U>>;
    <<(x*y)*z=x*(y*z)>>];;
*)

(* mreduce : rules -> term -> term *)
let mreduce rules M = let redex (_,_,L,R) = reduce L M R in
    try_find redex rules;;

(* One step of rewriting in leftmost-outermost strategy, with multiple rules *)
(* fails if no redex is found *)
(* mrewrite1 : rules -> term -> term *)
let mrewrite1 rules = rewrec
where rec rewrec M = mreduce rules M
    ? let (Term(f,sons)) = M in Term(f,tryrec sons)
    where rec tryrec (son::rest) = (let son'=rewrec son in son'::rest)
                                 ? son::tryrec rest;;

(* Iterating rewrite1. Returns a normal form. May loop forever *)
(* mrewrite_all : rules -> term -> term *)
let mrewrite_all rules M = rew_loop M
where rec rew_loop M = rew_loop(mrewrite1 rules M) ? M;;

(*
pretty_term (mrewrite_all Group_rules M where M,_=<<A*(I(B)*B)>>);;
==> A*U
*)

(* Inside-out rewriting *)
(* mrewriteio : rules -> term -> term *)
let mrewriteio rules = rewriorec
where rec rewriorec = function
    Term(f,sons) -> let M = Term(f,map rewriorec sons) in
                   (let N=mreduce rules M in rewriorec N)? M
```

```
  | x -> x;;

(* Remark that mrewriteio may loop while mrewrite_all terminates.
For instance, consider F(A) with [F(x)=B;A=A].
Also, mrewrite_all may loop while other sequences of rewriting terminates.
For instance, consider F(A,B) with [A=A;B=C;F(x,C)=D] *)
```

5.3 Local confluence

We recall Newman's lemma:

Newman's lemma. A Nœterian relation is confluent iff it is locally confluent [16].

We check local confluence with critical pairs analysis, along the lines of the Knuth-Bendix method [14, 8].

5.3.1 Superposition

```
(* All (u,sig) such that N/u (&var) unifies with M, with principal unifier sig *)
(* super : term -> term -> (num list & subst) list *)
let super M = suprec
where rec suprec N =
  let insides = match N with
    Term(_,sons) -> (fst(it_list collate ([],1) sons)
            where collate (pairs,n) son =
                     (pairs @ map (fun (u,sig) -> (n::u,sig)) (suprec son)),n+1)
   | _ -> []
  in ((let (Term(_))=N and sig = unify(M,N) in ([],sig)::insides) ? insides);;

(* Ex
let M,_ = <<F(A,B)>>
and N,_ = <<H(F(A,x),F(x,y))>> in super M N;;
==> [[1],[2,Term ("B",[])];                         x <- B
     [2],[2,Term ("A",[]); 1,Term ("B",[])]]        x <- A  y <- B
*)

(* All (u,sig), u&[], such that N/u unifies with M *)
(* super_strict : term -> term -> (num list & subst) list *)
let super_strict M = function
    Term(_,sons) -> (fst(it_list collate ([],1) sons)
              where collate (pairs,n) son =
                    (pairs @ map (fun (u,sig) -> (n::u,sig)) (super M son)),n+1)
   | _ -> [];;
```

5.3.2 Critical pairs

```
(* Critical pairs of L1=R1 with L2=R2 *)
(* critical_pairs : term_pair -> term_pair -> term_pair list *)
let critical_pairs (L1,R1) (L2,R2) =
```

```
let mk_pair (u,sig) = substitute sig (replace L2 u R1),substitute sig R2
in map mk_pair (super L1 L2);;

(* Strict critical pairs of L1=R1 with L2=R2 *)
(* strict_critical_pairs : term_pair -> term_pair -> term_pair list *)
let strict_critical_pairs (L1,R1) (L2,R2) =
let mk_pair (u,sig) = substitute sig (replace L2 u R1),substitute sig R2
in map mk_pair (super_strict L1 L2);;

(* All critical pairs of eq1 with eq2 *)
let mutual_critical_pairs eq1 eq2 =
    (strict_critical_pairs eq1 eq2) @ (critical_pairs eq2 eq1);;

(* rename : num -> term_pair -> term_pair *)
let rename n = distr_pair (instance n);;

(* Returns all critical pairs between a rule and a list of rules *)
(* all_critical_pairs : rule -> rules -> (num & term_pair) list *)
let all_critical_pairs (name,n,eq) rules =
   let self = strict_critical_pairs eq (rename n eq) in
      let init = if null self then [] else [name,self]
         in list_it crit rules init
         where crit (name',_,eq1) list =
              let eq' = rename n eq1 in
              let pairs = (mutual_critical_pairs eq eq') in
                          if null pairs then list else (name',pairs)::list;;

(* all_critical_pairs (4,n,l,r) Group_rules
       where (Term("=",[l;r]),_,n)=<<x*U=x>>  =
[(1,[Term("U",[]),Term("U",[])]);
 (2,[Term("U",[]),Term("I",[Term("U",[])])]);
 (3,
  [Term("*",[Var(4); Var(2)]),
    Term("*",[Var(4); Term("*",[Term("U",[]); Var(2)])]);
   Term("*",[Var(4); Term("*",[Var(3); Term("U",[])])]),
    Term("*",[Var(4); Var(3)])])]
*)
(* Generate failure message *)
let unresolved (M,N,n1,n2) =
         pretty_term M; print_string " = ";pretty_term N;print_newline();
         let R1 = string_of_num n1
         and R2 = string_of_num n2 in
             failwith "Irreducible consequence of " ^ R1 ^ " & " ^ R2;;
```

5.3.3 Local confluence

```
(* locally_confluent : rules -> bool *)    (* Answers true or fails *)
```

```
let locally_confluent rules =
  let normal_form = mrewrite_all rules in
    let enter_rule previous rule =
      (for_all check_criticals (all_critical_pairs rule previous);rule::previous)
      where check_criticals (n,pairs) =
            for_all check_pair pairs
            where check_pair (M,N) =
                  let M' = normal_form M and N' = normal_form N in
                  (M'=N') or unresolved(M',N',n,fst rule)
    in (it_list enter_rule [] rules;true);;

(* Example
locally_confluent Group_rules;;
v1 = I(v2)*(v2*v1)
Evaluation Failed: Irreducible consequence of 2 & 3

Another example:
let Twice1 = mk_rules [<<G(x)=F(F(x))>>]
and Twice2 = mk_rules [<<F(F(x))=G(x)>>];;
locally_confluent Twice1;;
==> true
locally_confluent Twice2;;
==> F(G(v1)) = G(F(v1))
Evaluation Failed: Irreducible consequence of 1 & 1

let Twice2_completed = mk_rules [<<F(F(x))=G(x)>>;<<F(G(x))=G(F(x))>>];;
locally_confluent Twice2_completed;;
==> true  (0.11s)

It is also possible to complete Group_rules to a locally confluent set of rules:

let Group_completed = mk_rules [
    <<U*x=x>>;
    <<I(x)*x=U>>;
    <<(x*y)*z=x*(y*z)>>;
    <<x*U=x>>;
    <<x*I(x)=U>>;
    <<I(I(x))=x>>;
    <<I(U)=U>>;
    <<x*(I(x)*y)=y>>;
    <<I(x)*(x*y)=y>>;
    <<I(x*y)=I(y)*I(x)>>];;

mrewrite_all Group_completed M where M,_=<<(A*B)*I(B)>>;;
==> Term ("A",[]) : term  (0.06s)

locally_confluent Group_completed;;
```

```
==> true
```

We shall see below how to generate such complete sets mechanically.

5.4 Checking termination

The test of local confluence above was guaranteed to terminate only when the given set of rules is Nœtherian. In that case, it provides a decision procedure for confluence, using Newman's lemma. Let us now show how one can check termination mechanically, using the recursive path ordering algorithm. Of course, this gives a sufficient but non necessary criterion for termination, the problem being undecidable in general. The theoretical justification of the algorithms below was given in the last chapter.

```
(* Recursive path ordering, after N. Dershowitz, S. Kamin and J.J. Levy *)

type extension = Multiset | Lexico;;

type ordering = Greater | Equal | NotGE;;

let ge_ord order pair = not (order pair = NotGE)
and gt_ord order pair = (order pair = Greater)
and eq_ord order pair = (order pair = Equal);;

let rem_eq equiv = remrec where rec remrec x = fun
      [] -> fail
    | (y::l) -> if equiv (x,y) then l else y::remrec x l;;

let diff_eq equiv (x,y) =
    let rec diffrec (x,y) = match x with
           [] -> (x,y)
         | (h::t) -> diffrec (t,rem_eq equiv h y)
                   ? (h::x',y') where (x',y') = diffrec (t,y)
    in let lx=length x and ly=length y in
           diffrec(if lx>ly then (y,x) else (x,y));;

(* multiset extension of order *)
let mult_ext order (Term(_,sons1),Term(_,sons2)) =
    match diff_eq (eq_ord order) (sons1,sons2) with
         ([],[]) -> Equal
       | (l1,l2) -> if for_all (fun N -> exists (fun M -> order (M,N) = Greater) l1) l2
                   then Greater else NotGE;;

(* lexicographic extension of order *)
let lex_ext order (M,N) = let (Term(_,sons1))=M and (Term(_,sons2))=N
in lexrec(sons1,sons2) where rec lexrec = fun
    ([] , []) -> Equal
  | ([] , _ ) -> NotGE
```

```
  | ( _ , []) -> Greater
  | ((x1::l1),(x2::l2)) -> match order (x1,x2) with
        Greater -> if for_all (fun N' -> gt_ord order (M,N')) l2
             then Greater else NotGE
      | Equal -> lexrec (l1,l2)
      | NotGE -> if exists (fun M' -> ge_ord order (M',N)) l1
             then Greater else NotGE;;

(* recursive path ordering *)
(* rpo : (string -> string -> ordering) -> (string -> extension) -> term_pair -> ordering *)
let rpo op_order ext = rporec where rec rporec (M,N) = match M with
      Var(m) -> if N=Var(m) then Equal else NotGE
    | Term(op1,sons1) -> match N with
         Var(n) -> if occurs n M then Greater else NotGE
       | Term(op2,sons2) -> match (op_order op1 op2) with
            Greater -> if for_all (fun N' -> gt_ord rporec (M,N')) sons2
                      then Greater else NotGE
          | Equal -> (match (ext op1) with
                                Multiset -> mult_ext
                             |  Lexico -> lex_ext) rporec (M,N)
          | NotGE -> if exists (fun M' -> ge_ord rporec (M',N)) sons1
                      then Greater else NotGE;;

(* checking a set of rules is Noetherian with recursive path ordering method *)
let rpo_noetherian op_order ext =
    for_all (fun (_,_,pair) -> ((rpo op_order ext pair) = Greater) or
            (let M,N=pair in pretty_term M; print_string " not > ";
                             pretty_term N; print_newline(); fail));;

(* Exemple
(* I > * > U *)
let Group_order op1 op2 =
    if op1=op2 then Equal
    if (op1="I") or (op2="U") then Greater
    else NotGE;;

rpo_noetherian Group_order (K Lexico) Group_completed;;
==> true  (0.05s)

let Disj_nf_rules = mk_rules [
    <<Not(Not(p)) = p>>;
    <<Not(Or(p,q)) = And(Not(p),Not(q))>>;
    <<Not(And(p,q)) = Or(Not(p),Not(q))>>;
    <<And(p,Or(q,r)) = Or(And(p,q),And(p,r))>>;
    <<And(Or(q,r),p) = Or(And(q,p),And(r,p))>>]

(* Not > And > Or *)
```

```
and Disj_nf_order op1 op2 =
    if op1=op2 then Equal
    if (op1="Not") or (op2="Or") then Greater
    else NotGE;;

rpo_noetherian Disj_nf_order (K Multiset) Disj_nf_rules;;
==> true (0.10s)
```

Note that the `Lexico` extension would work as well for this example.

5.5 Knuth-Bendix Completion

We present in this section the Knuth-Bendix completion procedure. The idea is to use the failure cases of local confluence as interesting derived lemmas. These equations are added as supplementary rewrite rules in the hope to complete the non-confluent system to a confluent one.

5.5.1 Completion without deletions

```
(* checks that rules are numbered in sequence and returns their number *)
let check_rules = it_list (fun n (k,_) -> if k=n+1 then k
                            else failwith "Rule numbers not in sequence") 0;;

(* Generate failure message *)
let non_orientable (M,N) =
    pretty_term M; print_string " = "; pretty_term N; print_newline();
    "Non orientable equation";;

(* rules is the current partial complete set, n is its length, the pair
  (k,l) with k<=l<=n gives the last critical pairs computed.
  Next argument is list of queued equations *)
(* completion : (term_pair -> bool) -> num -> rules -> (num & num) -> term_pair list -> rules *)
let completion greater = completerec
where rec completerec n rules =
  let normal_form = mrewrite_all rules
  and get_rule k = assoc k rules in process
  where rec process (k,l) = processkl
  where rec processkl = fun
          [] -> if k<l then next_criticals (k+1,l)
               if l<n then next_criticals (1,l+1)
               else rules (* successful completion *)
        | ((M,N)::crits) -> let M' = normal_form M and N' = normal_form N in
       if M'=N' then processkl crits
                else let left,right =
                     if greater(M',N') then (M',N')
                     if greater(N',M') then (N',M')
                     else failwith (non_orientable(M',N')) (* completion fails *)
                in let new_rule = n+1,mk_rule left right
```

```
                   in pretty_rule new_rule;
                      completerec (n+1) (new_rule::rules) (k,l) crits
  and next_criticals (k,l) =
      let pairs = if k=l then let (v,eq) = get_rule k in
                         strict_critical_pairs eq (rename v eq)
                  else let (v,ek) = get_rule k and (_,el) = get_rule l
                       in mutual_critical_pairs ek (rename v el)
      in process (k,l) pairs;;

let pretty_complete completed_rules =
    print_newline();
    message "Canonical set found :";
    pretty_rules (rev completed_rules);();;

(* complete_rules is assumed locally confluent, and checked Noetherian with
  ordering greater, rules is any list of rules *)
let complete greater complete_rules rules =
    let n = check_rules complete_rules
    and eqs = map (fun (_,_,pair) -> pair) rules in
    let completed_rules = completion greater n complete_rules (n,n) eqs in
    pretty_complete completed_rules;;


(* Example
let greater pair = (rpo Group_order (K Lexico) pair = Greater) in
complete greater [] Group_rules;; ==>
1 : U*v1 = v1
2 : I(v1)*v1 = U
3 : (v1*v2)*v3 = v1*(v2*v3)
4 : I(v1)*(v1*v2) = v2
5 : I(U)*v1 = v1
6 : I(I(v1))*U = v1
7 : I(v1*v2)*(v1*(v2*v3)) = v3
8 : I(I(v1))*v2 = v1*v2
...
15 : v1*I(v1) = U
16 : v1*(I(v1)*v2) = v2
17 : v1*U = v1
18 : I(I(v1)) = v1
...
```

And we see that there is a lot of redundancy in the generated rules. The next idea is to keep the partial complete set in reduced form. That is, every rule $M = N$ is such that N is irreducible, and M is irreducible by any other rule in the set. Reduced rules are re-introduced as equations, since their orientation may change after reduction (at least if the left-hand side is reduced). When a critical pair is selected, we check that its two parents are still active.

5.5.2 Completion with deletions

```
let deletion_message (k,_) = print_string "Rule ";print_num k;
                             message " deleted";;

(* Knuth-Bendix completion procedure *)
(* kb_completion1 : (term_pair -> bool) -> num -> rules -> (num & num) -> term_pair list -> rules *)
let kb_completion1 greater = kbrec
where rec kbrec n rules =
  let normal_form = mrewrite_all rules
  and get_rule k = assoc k rules in process
  where rec process (k,l) = (processkl
  where rec processkl = fun
          [] -> if k<l then next_criticals (k+1,l)
               if l<n then next_criticals (1,l+1)
               else rules (* successful completion *)
        | ((M,N)::eqs) -> let M' = normal_form M and N' = normal_form N in
       if M'=N' then processkl eqs
                else let left,right =
                     if greater(M',N') then (M',N')
                     if greater(N',M') then (N',M')
                     else failwith (non_orientable(M',N')) (* completion fails *)
                in let new_rule = n+1,mk_rule left right
                   in pretty_rule new_rule;
                      let left_reducible (_,_,L,_) = reducible left L in
                      let redl,irredl = partition left_reducible rules in
                      map deletion_message redl;
                      let irreds = map right_reduce irredl
                             where right_reduce (m,_,L,R) = m,
                                   mk_rule L (mrewrite_all (new_rule::rules) R)
                      in let new_eqs = map (fun (_,_,pair) -> pair) redl in
                         kbrec (n+1) (new_rule::irreds) (k,l) (eqs @ new_eqs))
  and next_criticals (k,l) =
      try (let v,el = get_rule l in
          if k=l then process (k,l) (strict_critical_pairs el (rename v el))
          else try (let _,ek = get_rule k in
                process (k,l) (mutual_critical_pairs el (rename v ek)))
                 with failure "find" (*rule k deleted*) -> next_criticals (k+1,l))
      with failure "find" (*rule l deleted*) -> next_criticals (1,l+1);;

(* complete_rules is assumed locally confluent, and checked Noetherian with
  ordering greater, rules is any list of rules *)
let kb_complete1 greater complete_rules rules =
    let n = check_rules complete_rules
    and eqs = map (fun (_,_,pair) -> pair) rules in
    let completed_rules = kb_completion1 greater n complete_rules (n,n) eqs in
    pretty_complete completed_rules;;
```

```
(* And now:
let greater pair = (rpo Group_order (K Lexico) pair = Greater) in
kb_complete1 greater [] Group_rules;; ==>
1 : U*v1 = v1
2 : I(v1)*v1 = U
3 : (v1*v2)*v3 = v1*(v2*v3)
4 : I(v1)*(v1*v2) = v2
5 : I(U)*v1 = v1
6 : I(I(v1))*U = v1
7 : I(v1*v2)*(v1*(v2*v3)) = v3
8 : I(I(v1))*v2 = v1*v2
Rule 6 deleted
9 : v1*U = v1
10 : I(I(v1*v2)*v1) = v2
11 : I(v1*I(v2))*v1 = v2
12 : I(v1*(v2*v3))*(v1*(v2*(v3*v4))) = v4
13 : I(v1*I(v2))*(v1*v3) = v2*v3
14 : v1*I(v1) = U
15 : v1*(I(v1)*v2) = v2
16 : I(U) = U
Rule 5 deleted
17 : I(I(v1)) = v1
Rule 8 deleted
18 : v1*(I(v2*v1)*v2) = U
...
24 : v1*I(I(v2)*v1) = v2
I(v1*v4)*v1 = I(v3*v4)*v3
Evaluation Failed :   Non orientable equation
```

The completion has generated the equation $I(x*y)*x = I(z*y)*z$ before the rule $I(x*y) = I(y)*I(x)$, and thus failed.

So the next idea is to defer such non-orientable equations until all critical pairs have been generated, in the hope that they will ultimately get simplified by new rules. Thus we add a new argument "failures" to stack these troublesome equations.

5.5.3 Completion with delayed failures

```
(* Improved Knuth-Bendix completion procedure *)
(* kb_completion : (term_pair -> bool) -> num -> rules -> term_pair list -> (num & num) -> term_pair list -> rules *)
let kb_completion greater = kbrec
where rec kbrec n rules =
  let normal_form = mrewrite_all rules
  and get_rule k = assoc k rules in process
  where rec process failures = processf
  where rec processf (k,l) = (processkl
  where rec processkl = fun
     [] -> if k<l then next_criticals (k+1,l)
          if l<n then next_criticals (1,l+1)
```

```
          if null failures then rules (* successful completion *)
          else (message "Non-orientable equations :";
                map non_orientable failures;fail)
   | ((M,N)::eqs) -> let M' = normal_form M
                   and N' = normal_form N
            and enter_rule(left,right) =
          let new_rule = n+1,mk_rule left right
          in pretty_rule new_rule;
             let left_reducible (_,_,L,_) = reducible left L in
             let redl,irredl = partition left_reducible rules in
             map deletion_message redl;
             let irreds = (map right_reduce irredl
                 where right_reduce (m,_,L,R) =
                       m,mk_rule L (mrewrite_all (new_rule::rules) R))
             and eqs' = map (fun (_,_,pair) -> pair) redl in
                 kbrec (n+1) (new_rule::irreds) [] (k,l) (eqs @ eqs' @ failures)
                   in if M'=N' then processkl eqs
                      if greater(M',N') then enter_rule(M',N')
                      if greater(N',M') then enter_rule(N',M')
                      else process ((M',N')::failures) (k,l) eqs)
  and next_criticals (k,l) =
      try (let v,el = get_rule l in
          if k=l then processf (k,l) (strict_critical_pairs el (rename v el))
          else try (let _,ek = get_rule k in
                processf (k,l) (mutual_critical_pairs el (rename v ek)))
        with failure "find" (*rule k deleted*) -> next_criticals (k+1,l))
      with failure "find" (*rule l deleted*) -> next_criticals (1,l+1);;

(* complete_rules is assumed locally confluent, and checked Noetherian with
  ordering greater, rules is any list of rules *)
let kb_complete greater complete_rules rules =
    let n = check_rules complete_rules
    and eqs = map (fun (_,_,pair) -> pair) rules in
    let completed_rules = kb_completion greater n complete_rules [] (n,n) eqs in
    message "Canonical set found :";
    pretty_rules (rev completed_rules);();;

(* Now:
let greater pair = (rpo Group_order (K Lexico) pair = Greater) in
kb_complete greater [] Group_rules;; ==>
1 : U*v1 = v1
2 : I(v1)*v1 = U
3 : (v1*v2)*v3 = v1*(v2*v3)
4 : I(v1)*(v1*v2) = v2
5 : I(U)*v1 = v1
6 : I(I(v1))*U = v1
7 : I(v1*v2)*(v1*(v2*v3)) = v3
```

```
8 : I(I(v1))*v2 = v1*v2
Rule 6 deleted
9 : v1*U = v1
10 : I(I(v1*v2)*v1) = v2
11 : I(v1*I(v2))*v1 = v2
12 : I(v1*(v2*v3))*(v1*(v2*(v3*v4))) = v4
13 : I(v1*I(v2))*(v1*v3) = v2*v3
14 : v1*I(v1) = U
15 : v1*(I(v1)*v2) = v2
16 : I(U) = U
Rule 5 deleted
17 : I(I(v1)) = v1
Rule 8 deleted
18 : v1*(I(v2*v1)*v2) = U
19 : I(I(v1*(v2*v3))*(v1*v2)) = v3
20 : v1*(I(v2*v1)*(v2*v3)) = v3
21 : I(I(v1)*I(v2)) = v2*v1
22 : I(v1*I(v2*v1)) = v2
23 : I(v1*(v2*I(v3)))*(v1*v2) = v3
24 : v1*I(I(v2)*v1) = v2
25 : I(v1*(I(v2*v3)*v2))*v1 = v3
26 : I(v1*(v2*(v3*v4)))*(v1*(v2*(v3*(v4*v5)))) = v5
27 : I(v1*(v2*I(v3)))*(v1*(v2*v4)) = v3*v4
28 : I(v1*(I(v2*v3)*v2))*(v1*v4) = v3*v4
29 : v1*(I(v2*(v3*v1))*(v2*(v3*v4))) = v4
30 : v1*(I(I(v2)*v1)*v3) = v2*v3
31 : v1*(v2*I(v1*v2)) = U
32 : I(v1*v2)*v1 = I(v2)
Rule 28 deleted
Rule 25 deleted
Rule 18 deleted
Rule 11 deleted
Rule 10 deleted
33 : I(v1*(v2*v3))*(v1*v2) = I(v3)
Rule 23 deleted
Rule 19 deleted
34 : I(v1*I(v2)) = v2*I(v1)
Rule 22 deleted
Rule 21 deleted
Rule 13 deleted
35 : v1*(v2*(I(v1*v2)*v3)) = v3
36 : I(v1*v2)*(v1*v3) = I(v2)*v3
Rule 33 deleted
Rule 29 deleted
Rule 27 deleted
Rule 26 deleted
Rule 20 deleted
```

```
Rule 12 deleted
Rule 7 deleted
37 : v1*(v2*I(I(v3)*(v1*v2))) = v3
38 : I(I(v1)*v2) = I(v2)*v1
Rule 37 deleted
Rule 30 deleted
Rule 24 deleted
39 : v1*(v2*(v3*I(v1*(v2*v3)))) = U
40 : v1*I(v2*v1) = I(v2)
Rule 31 deleted
41 : I(v1*v2) = I(v2)*I(v1)
Rule 40 deleted
Rule 39 deleted
Rule 38 deleted
Rule 36 deleted
Rule 35 deleted
Rule 34 deleted
Rule 32 deleted
Canonical set found :
1 : U*v1 = v1
2 : I(v1)*v1 = U
3 : (v1*v2)*v3 = v1*(v2*v3)
4 : I(v1)*(v1*v2) = v2
9 : v1*U = v1
14 : v1*I(v1) = U
15 : v1*(I(v1)*v2) = v2
16 : I(U) = U
17 : I(I(v1)) = v1
41 : I(v1*v2) = I(v2)*I(v1)
```

5.5.4 Discussion

Note that our choice of examining critical pairs in a very specific order is arbitrary. We could use other selection criteria, as long as they are fair, i.e. every critical pair should be eventually examined. This could significantly improve the performance of the procedure, by selecting early interesting consequences which may quickly simplify previous rules and critical pairs. Realistic implementations of the Knuth-Bendix completion procedure use sophisticated heuristics for this selection. Also, critical pairs could be generated one at a time, but the data structure permitting such incremental computation would be rather complicated.

Under a fairness assumption, it can be shown that the Knuth-Bendix completion procedure is a semi-decision procedure, for all systems for which we can decide orientation of the generated equations. That is, maybe at some point we shall fail to give an orientation to an equation, in which case the procedure fails without conclusion (all one may conclude is that all equations are equational consequences of the given equations). Otherwise, there are two cases. Either the procedure will loop, generating an infinite set of equations (but then every equational consequence of the given equations will be rewritten to a tautology $M = M$ at some point). Or else the procedure will stop, and the resulting set of equations is canonical, i.e. it is a Nœtherian and confluent set of rewrite rules which may be used to decide any equation of the original equational theory.

If one wants to keep track of proofs we must keep a trace of superpositions and rewritings. Actually all operations are reductions and narrowings (substitution followed by reduction), and thus a history of a rule consists in a number, a substitution, and two sequences of rewritings, one on the left and the other on the right, represented as lists of pairs (rule,occurrence).

Remark: The Knuth-Bendix completion is a costly process, but it should be considered a compilation process done once and for all. The whole procedure may be seen as a very general way of compiling a (decidable) equational theory into an algorithm of reduction to canonical form.

Numerous generalizations of the Knuth-Bendix completion procedure have been proposed. First, it is possible to generalize the method to deal with rewriting on congruence classes of terms modulo some congruence. This way, permutative axioms such as commutativity can be accomodated [18, 13]. Numerous canonical systems for decidable varieties have been found mechanically with these methods [12]. It is also possible to extend the completion to derive proofs in the initial algebra of equational presentations, as opposed to ordinary equational proofs, valid in the variety of all algebras which are models of the axioms. Such "induction-less" induction proofs are described in [10]. Finally, the method applies to decide word problems in finitely presented algebras, as shown for instance in [2, 1, 15].

References

[1] A.M. Ballantyne, D.S. Lankford. "New Decision Algorithms for Finitely Presented Commutative Semigroups." Report MTP-4, Department of Mathematics, Louisiana Tech. U (May 1979).

[2] G. M. Bergman. "The Diamond Lemma for Ring Theory ." Advances in Mathematics, **29,2** (1978) 178–218.

[3] N. Dershowitz, L. Marcus. "Existence and Construction of Rewrite Systems." Aerospace report No. ATR-82(8478)-3 (Dec. 1982).

[4] N. Dershowitz "Computing with Rewrite Systems." Aerospace Report No ATR-83(8478) (Jan. 1983).

[5] N. Dershowitz. "Applications of the Knuth-Bendix Completion Procedure." Aerospace Report No ATR-83(8478)-2 (May 1983).

[6] J. Hsiang. "Topics in Automated Theorem Proving and Program Generation ." Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign (Nov. 1982).

[7] J. Hsiang, N. Dershowitz. "Rewrite Methods for Clausal and Non-Clausal Theorem Proving." ICALP 1983, Spain.

[8] G. Huet. "Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems." J. Assoc. Comp. Mach. **27,4** (1980) 797–821.

[9] G. Huet. "A Complete Proof of Correctness of the Knuth-Bendix Completion Algorithm ." JCSS **23,1** (1981) 11-21

[10] G. Huet, J.M. Hullot. "Proofs by Induction in Equational Theories With Constructors." JCSS **25,2** (1982) 239–266.

[11] G. Huet, D. Oppen. "Equations and Rewrite Rules: a Survey." In Formal Languages: Perspectives and Open Problems, Ed. Book R., Academic Press (1980).

[12] J.M. Hullot "Compilation de Formes Canoniques dans les Théories Equationnelles." Thèse de 3ème cycle, U. de Paris Sud (Nov. 80).

[13] Jean Pierre Jouannaud, Helene Kirchner. "Completion of a set of rules modulo a set of equations." CSL Technical Note, SRI International (April 1984). To appear, SIAM Journal on Computing.

[14] D. Knuth, P. Bendix. "Simple word problems in universal algebras". In: Computational Problems in Abstract Algebra, J. Leech Ed., Pergamon (1970) 263–297.

[15] Ph. Le Chenadec. "Canonical forms in finitely presented algebras". Lecture Notes in Theoretical Computer Science, Pitman-Wiley (1986).

[16] M.H.A. Newman. "On Theories with a Combinatorial Definition of "Equivalence"." Annals of Math. **43,2** (1942) 223–243.

[17] E. Paul. "Equational Methods in First Order Predicate Calculus." J. of Symbolic Computation, **1,1** (1985) 7–29.

[18] G.E. Peterson, M.E. Stickel. "Complete Sets of Reduction for Equational Theories with Complete Unification Algorithms ." JACM **28,2** (1981) 233–264.

Chapter 6

Sequential computations

These notes sketch the theory of computation by rewrite rules.

6.1 Trading termination against linearity

The confluence of term-rewriting systems obeying the Knuth-Bendix test was based on Newman's lemma:

Newman's lemma. A Nœtherian relation is confluent iff it is locally confluent.

and on a closure property of critical pairs:

The critical pairs lemma. A rewriting system is locally confluent iff its critical pairs are locally confluent (with the obvious definition of a pair (M, N) being locally confluent iff there is a common reduct P of M and N).

If a term rewriting system is not Nœtherian, that is if some term admits an infinite rewriting sequence, then local confluence is not enough to show its confluence. For instance, consider:

Example 1 (Hindley)

$$\begin{cases} A \to B \\ A \to C \\ B \to A \\ B \to D. \end{cases}$$

We get counter-examples even if we forbid cycles in the rewriting relation:

Example 2 (Newman-Huet)

$$\begin{cases} F(x,x) & \to & A \\ C & \to & G(C) \\ F(x,G(x)) & \to & B \end{cases}$$

since the term $F(C, C)$ possesses two distinct normal forms A and B. Note that the system above has no critical pair, since $F(x, x)$ and $F(x, G(x))$ are not unifiable.

It seems that the trouble with the term-rewriting system above arises from the existence of non-linear left-hand sides. We may wonder whether one non-linear left-hand is not sufficient to get a counter-example. Indeed this has been shown by Klop [15]. Here is his counter-example, simplified by Barendregt:

Example 3 (Klop-Barendregt)

$$\begin{cases} F(x,x) & \rightarrow & A \\ C & \rightarrow & G(C) \\ G(x) & \rightarrow & F(x,G(x)) \end{cases}$$

since the term $G(C)$ rewrites to A and $G(A)$, but these two terms have no common reduct (Exercise).

Let us call *left-linear* (resp. *right-linear*) a system such that every left-hand side (resp. every right-hand side) is linear. For linear systems, we have better hopes to show confluence without insisting on termination. For instance, we may hope to use a strong covering of derivation diagrams:

Definition. A relation $\rightarrow$ is *strongly confluent* iff for all x,y,z, $x \rightarrow y$ and $x \rightarrow z$ implies there is u such that $y \rightarrow u$ and $z \rightarrow u$.

Strong confluence may seem to be a very strong requirement. Actually, it leads to a completely general method, where one shows the confluence of $\rightarrow$ by showing the strong confluence of some other relation having the same reflexive-transitive closure as $\rightarrow$ (the method being general since confluence of $\rightarrow$ is strong confluence of $\rightarrow^*$). This method is the basis of classical proofs of confluence in the λ-calculus. See the Tait-Martin Löf method in Barendregt's book [1].

Let us now try and apply directly this method to the reflexive closure of rewriting. We say that a pair (M,N) of terms is strongly confluent in the term rewriting system $\mathcal{R}$ iff there is a P such that $M \rightarrow^\epsilon P$ and $N \rightarrow^\epsilon P$, with $\rightarrow^\epsilon$ the reflexive closure of $\rightarrow_\mathcal{R}$.

Proposition 1. If $\mathcal{R}$ is left-linear, right-linear, and if all its critical pairs are strongly confluent, then $\mathcal{R}$ is confluent.

The proof of this proposition may be found in [12]. Note that right-linearity does not suffice, as shown by Example 2 above. Similarly, left-linearity is not enough:

Example 4 (Lévy)

$$\begin{cases} F(A,A) & \rightarrow & G(B,B) \\ A & \rightarrow & A' \\ F(A',x) & \rightarrow & F(x,x) \\ F(x,A') & \rightarrow & F(x,x) \\ G(B,B) & \rightarrow & F(A,A) \\ B & \rightarrow & B' \\ G(B',x) & \rightarrow & G(x,x) \\ G(x,B') & \rightarrow & G(x,x) \end{cases}$$

since $F(A',A')$ and $G(B',B')$ are interconvertible, but not confluent.

Still, it would be very desirable to find sufficient conditions for a term rewriting system to be confluent that do not depend on right linearity, a rather unnatural condition. We shall do this by strengthening the closure condition. Let us first define the *parallel-disjoint* rewriting relation $\Rightarrow_\mathcal{R}$

associated with a term-rewriting system $\mathcal{R}$. We recall that a relation $\Rightarrow$ is *term compatible* if and only if it is reflexive and verifies:

$$\frac{M_1 \Rightarrow N_1 \quad M_2 \Rightarrow N_2 \ldots M_n \Rightarrow N_n}{F(M_1, M_2 \ldots M_n) \Rightarrow F(N_1, N_2 \ldots N_n)}$$

for every operator F of arity n.

Definition. $\Rightarrow_{\mathcal{R}}$ is the smallest term compatible relation containing $\rightarrow_{\mathcal{R}}$.

It is easy to show that $M \Rightarrow_{\mathcal{R}} N$ iff N is obtained from M by the simultaneous replacement of a set of mutually disjoint redexes, and thus that $\Rightarrow_{\mathcal{R}}$ has the same reflexive-transitive closure as $\rightarrow_{\mathcal{R}}$.

Now we may state:

Proposition 2. If $\mathcal{R}$ is a left-linear term-rewriting system such that $M \Rightarrow_{\mathcal{R}} N$ for every critical pair (M, N), then $\mathcal{R}$ is confluent.

The proof, which shows that $\Rightarrow_{\mathcal{R}}$ is strongly confluent under the given conditions, appears in [12]. Remark that it is crucial here that critical pairs are defined precisely as ordered pairs, since the condition is not symmetric.

6.2 Regular rewriting systems

Proposition 2 above can be used to show the confluence of various rewriting systems used as computational paradigms. For instance, combinatory reduction is a left-linear term-rewriting system without critical pairs:

Combinatory reduction

$$\begin{cases} A(A(K,x),y) & \rightarrow & x \\ A(A(A(S,x),y),z) & \rightarrow & A(A(x,z),A(y,z)). \end{cases}$$

Note that right-linearity cannot be imposed upon a combinatorially-complete set of combinators. By contrast, it is necessary to impose left-linearity if one wants nice computational properties. We saw that matching was linear in the size of the pattern for left-linear rules, and that reduction to normal form was complicated in the presence of non-linear left-hand sides, since a redex could appear at an arbitrary distance above the last reduction. We may also wonder whether term-rewriting systems with critical pairs are meaningful from a computational point of view, since combinatory reduction (for any set of combinators) is non-ambiguous. It is intuitively obvious that the absence of critical pairs should allow better deterministic computations. Let us try to make precise this intuition.

Definition. Let $\mathcal{R}$ be a term-rewriting system, M be any term. We say that a $\mathcal{R}$-*redex* in M is any occurrence u in $\mathcal{D}(M)$ such that $\lambda_k \leq M/u$ for some rule $R_k : \lambda_k \rightarrow \rho_k$ in $\mathcal{R}$. We denote by $\mathcal{R}(M)$ the set of $\mathcal{R}$-redexes of M. If $\mathcal{R}(M) = \emptyset$ we say that M is an $\mathcal{R}$-*normal* term.

A redex determines a possible $\mathcal{R}$-reduction: if $u \in \mathcal{R}(M)$, we get $M \rightarrow_{\mathcal{R}} N$, with $N = M[u \leftarrow \sigma(\rho_k)]$, where the substitution σ is uniquely determined by the condition $M/u = \sigma(\lambda_k)$. When $\mathcal{R}$ is understood from the context, we write $M \rightarrow_{u,k} N$ to indicate that M rewrites at redex u to

N, using rule k. If $\mathcal{R}$ has no critical pairs, k is uniquely determined, and thus a reduction step, or elementary derivation, is uniquely determined by its redex u, and we write simply $M \rightarrow_u N$.

Definition. Let $a : M \rightarrow_{u,k} N$ be an elementary $\mathcal{R}$-derivation, and $v \in \mathcal{D}(M)$. We define the set $v \backslash a$ of *residuals* of v by a as the set of occurrences of N defined as follows:

$$v \backslash a = \begin{cases} \{v\} & \text{if } v|u \text{ or } v \prec u; \\ \{u@w'@v' \mid \rho_k/w' = x\} & \text{if } v = u@w@v' \text{ with } \lambda_k/w = x \in \mathcal{V}; \\ \emptyset & \text{otherwise.} \end{cases}$$

Every occurrence w in N has been either created by the reduction step, if it is inside the right hand side ρ_k, or else it belongs to the residual set of exactly one occurrence of M. Intuitively, the residual map indicates how the new term N *shares* subterms with the old term M.

When u is a redex, its residuals indicate where we should expect to find a corresponding redex in the reduced term N: it may have stayed undisturbed, it may have been duplicated, or it may have vanished. Note that in this case, when there are no critical pairs, the case "otherwise" above can only occur when $u = v$.

Example. Let $\mathcal{R} = \{F(x,y) \rightarrow G(x,x)\}$, $M = G(A,F(B,C))$, $u =$ `[2]` and thus $N = G(A,G(B,B))$. The residual map is given by:

$$\begin{cases} \texttt{[]} \mapsto \{\texttt{[]}\} \\ \texttt{[1]} \mapsto \{\texttt{[1]}\} \\ \texttt{[2]} \mapsto \{\} \\ \texttt{[2;1]} \mapsto \{\texttt{[2;1]}, \texttt{[2;2]}\} \\ \texttt{[2;2]} \mapsto \{\} \end{cases}$$

Definition. A term rewriting system $\mathcal{R}$ is *regular* iff it is left-linear and does not possess critical pairs.

Fact. If $\mathcal{R}$ is regular, then for any $a : M \rightarrow N$, the residuals of any redex of M by a are redexes of N.

Note that both conditions of left-linearity and absence of critical pairs are needed in order to have this very important preservation property. This fact shows that regular systems are computationally sensible.

Remark. If we consider λ-calculus with the β and the η rules (we shall see later in the course how to consider these rules as term rewritings), then we do not get a regular system according to our definition, since we have a critical pair between these two rules. However, this critical pair is trivial, in the sense of being a pair of identical terms, and may thus be safely ignored. Still, this complicates the theory of λ-reduction, and generally the η rule is dealt with separately, since it may be shown that the η-conversions may be delayed after all β-reductions.

Redexes of N which are not residuals of a redex of M are said to be *created* by reduction a. There are three cases of created redexes in regular systems, illustrated by the following examples:
Downward creation. $\mathcal{R} = \{F(x) \rightarrow H(G(x)), G(A) \rightarrow B\}$, reduction $F(A) \rightarrow H(G(A))$. Here redex `[]` creates redex `[1]` downwards.
Upward creation. $\mathcal{R} = \{F(x) \rightarrow G(x), H(G(x)) \rightarrow K(x)\}$, reduction $H(F(A)) \rightarrow H(G(A))$. Here redex `[1]` creates redex `[]` upwards.
Collapse creation. $\mathcal{R} = \{F(G(x)) \rightarrow H(x), K(x) \rightarrow x\}$, reduction $F(K(G(A))) \rightarrow F(G(A))$. Here redex `[1]` creates redex `[]` both upwards and downwards, by "collapse and glueing".

In the systems that we consider now, we do not assume finite termination of rewritings anymore. However, any infinite sequence of rewritings must involve an infinite number of redex creations, since sequences of reductions of residuals only always terminate:

The finite developments theorem. Every sequence of reductions of residuals of a fixed set of redexes must terminate.
Proof hint. Order the original redexes by the prefix ordering, and consider the multiset extension of this ordering.

We saw (Proposition 2) that a regular system is always confluent. Actually, a much stronger structural property of its derivation spaces is true, which is the object of the next section.

6.3 The parallel moves theorem

From now on, we assume that we are studying a given regular term rewriting system $\mathcal{R}$. In order to study the algebra of derivations with $\mathcal{R}$, we consider the category defined by parallel rewritings.

First, we extend the notation $a : M \to_u N$ to parallel (disjoint) reduction, defining one step of parallel reduction of a set $U = \{u_1, \ldots, u_n\}$ of mutually disjoint redexes $A : M \Rightarrow_U N$ as the result of effecting the successive n reductions of the individual redexes in any order. We remark that the set of residuals of a set of mutually disjoint redexes is itself a set of mutually disjoint redexes.

6.3.1 The derivations category

We consider the category whose objects are the terms, and the arrows $A : M \Rightarrow^* N$ the sequences of elementary parallel reductions. We call *derivation* such a parallel reduction sequence. The identity $Id : M \Rightarrow^* M$ is the empty sequence, and composition is simply the concatenation of sequences, written as $A; B : M \Rightarrow^* P$ when $A : M \Rightarrow^* N$ and $B : N \Rightarrow^* P$. A derivation is uniquely determined by its source term (i.e. its domain), and by the sequence $[U_1; \ldots ; U_k]$ of the contracted redexes. In other words, the derivation category is the free category generated by the elementary parallel derivations.

We extend the notion of residual to derivations as follows. First we extend residuals to several steps of reduction, by defining $v\backslash(a; b)$ as $\bigcup\{w\backslash b \mid w \in v\backslash a\}$. This defines the residuals of an occurrence by elementary derivations, and we extend similarly to any derivation, as follows. Let $v \in \mathcal{D}(M)$, and $A : M \Rightarrow^* N$. We define $v\backslash A$ by induction on $|A|$:

$$\begin{cases} v\backslash Id = \{v\} \\ v\backslash(A; B) = \bigcup\{w\backslash B \mid w \in v\backslash A\}. \end{cases}$$

Now we extend the residual map to projections of co-initial derivations. We write $Der(M)$ for the set of derivations starting from term M. Let A and B be two derivations in $Der(M)$, with $A : M \Rightarrow^* N$, and $B : M \Rightarrow_U P$ elementary. We define the *residual* $B\backslash A$ of B by A as the elementary derivation starting from N and contracting all redexes in $\bigcup\{u\backslash A \mid u \in U\}$. Finally, we define the *sum* $A + B$ of derivations A and B as the derivation $A + B = A; (B\backslash A)$. We are now able to refine Proposition 2 above.

6.3.2 Parallel moves

The parallel moves theorem. Let A and B be two elementary derivations in $Der(M)$. Then $A + B$ and $B + A$ are co-terminal, and for every $u \in \mathcal{R}(M)$ we have $u\backslash(A + B) = u\backslash(B + A)$.

Proof: Exercise. Actually the theorem is true for an arbitrary occurrence $u \in \mathcal{D}(M)$, but we shall use it below only when u is a redex.

Corollary. The relation $\Rightarrow$ is strongly confluent, and thus $\mathcal{R}$ is confluent. Thus, every term M admits at most one normal reduct, i.e. an $\mathcal{R}$-normal term N such that $M \rightarrow^* N$.

Example. Let $\mathcal{R} = \{F(x) \rightarrow G(x,x)\ ,\ H \rightarrow K\}$,
$A : G(F(F(H)), H) \Rightarrow_{\{\texttt{[}1\texttt{]},\texttt{[}2\texttt{]}\}} G(G(F(H), F(H)), K)$, and
$B : G(F(F(H)), H) \Rightarrow_{\{\texttt{[}1;1;1\texttt{]},\texttt{[}2\texttt{]}\}} G(F(F(K))), K)$. We get:
$B\backslash A : G(G(F(H), F(H)), K) \Rightarrow_{\{\texttt{[}1;1;1\texttt{]},\texttt{[}1;2;1\texttt{]}\}} G(G(F(K), F(K)), K)$,
$A\backslash B : G(F(F(K)), K) \Rightarrow_{\{\texttt{[}1\texttt{]}\}} G(G(F(K), F(K)), K)$, and the common residual set of redex $\texttt{[}1;1\texttt{]}$ in term $G(F(F(H)), H)$ by $A + B$ and $B + A$ is $\{\texttt{[}1;1\texttt{]}, \texttt{[}1;2\texttt{]}\}$.

The parallel moves theorem allows us to extend the residual relation to arbitrary derivations, as follows.

Definition. Let $A, B \in Der(M)$, with B elementary: $B : M \Rightarrow N$. We define $A\backslash B$ by induction on $|A|$:

$$\begin{cases} Id_M\backslash B & = Id_N \\ (A_1; A_2)\backslash B & = (A_1\backslash B); (A_2\backslash(B\backslash A_1)) \end{cases}$$

Remark that $A_2\backslash(B\backslash A_1)$ is well defined by induction, since $B\backslash A_1$ is elementary, and that its composition with $A_1\backslash B$ makes sense, by the theorem above. Now, for A, B two arbitrary co-initial derivations, we define a derivation $A\backslash B$ composable with B by the following induction on $|B|$:

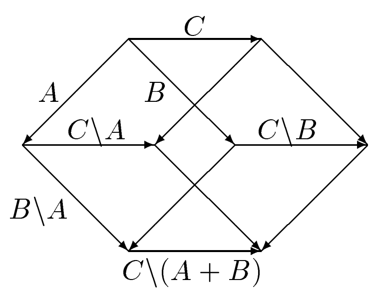
$$\begin{cases} A\backslash Id & = A \\ A\backslash(B_1; B_2) & = (A\backslash B_1)\backslash B_2 \end{cases}$$

We extend the notation $A + B\ =\ A; (B\backslash A)$ to co-initial derivations of any length, and we extend without difficulty the parallel moves theorem:

The generalized parallel moves theorem. Let A and B be two arbitrary derivations in $Der(M)$. Then $A + B$ and $B + A$ are co-terminal, and for every $u \in \mathcal{R}(M)$ we have $u\backslash(A + B)\ =\ u\backslash(B + A)$.

Actually, we may even generalize the preservation of residuals to an arbitrary derivation C starting from M:

The cube theorem.
Let A,B and C be three co-initial derivations. Then $C\backslash(A + B)\ =\ C\backslash(B + A)$.
Proof hint: induction on $|A| + |B| + |C|$.



If we now define, for two co-initial derivations A and B, the *permutation equivalence* as $A \equiv B$ iff for any C co-initial with A and B we have $C \backslash A = C \backslash B$, we get:

Corollary 1. $A + B \equiv B + A$.

Corollary 2. $(A + B) + C \equiv A + (B + C)$.

It is now fairly straightforward to show that $\equiv$ is a congruence respectively to operations $\backslash$,+, and ;. Also for every A we get $A \backslash A \equiv Id$, $Id + A \equiv A$, and $A + A \equiv A$. The only derivations equivalent to Id are the ones that contract at every step an empty set of redexes.

Now, we may define a partial ordering $\sqsubseteq$ of *dominance* between co-initial derivations, by: $A \sqsubseteq B$ iff $A + B \equiv B$. Intuitively, B dominates A if it does at least as much computation, in an implementation where residuals are shared (for instance, over a dag structure). The preceding results show that the set of derivations starting from a common term, and ordered by the dominance ordering $\sqsubseteq$, is an upper semi-lattice with + as its l.u.b. This can be cast elegantly in a categorical framework:

Definition. The *computation category* associated with the regular term rewriting system $\mathcal{R}$ is its derivation category, quotiented by the permutation equivalence. That is, it admits as objects the terms, and as arrows from M to N the permutation class of parallel derivations from M to N.

The computation theorem. The computation category admits pushouts.

Caution! The lattice structure given by the parallel moves theorem is on *derivations*, and *not* on terms. For instance, if we consider the system R consisting solely of the rules $I(x) \rightarrow x$ and $J(x) \rightarrow x$, the following derivations diagram shows that the terms $I(J(K))$ and $J(I(K))$ do not possess a l.u.b.

$$\begin{tikzcd}
 & & I(J(I(K))) \arrow[dll] \arrow[d] \arrow[drr] & & \\
I(J(K)) \arrow[dr] \arrow[drrr] & & J(I(K)) \arrow[dl] \arrow[dr] & & I(I(K)) \arrow[dl, shift left] \arrow[dl, shift right] \\
 & J(K) \arrow[dr] & & I(K) \arrow[dl] & \\
 & & K & &
\end{tikzcd}$$

Note that this phenomenon may be traced to the existence of two non-equivalent derivations between $I(I(K))$ and $I(K)$. This shows that the categorical viewpoint is the right one here: we need to talk in terms of arrows, not just relations between terms. And now the confluence diagrams can be replaced by more informative commuting diagrams expressing permutation equivalences of derivations. For instance, in the diagram above, certain sub-diagrams are confluent only for unimportant syntactic coincidences (due to the double occurrence of I in $I(J(I(K)))$), but the others are commuting diagrams expressing a strong equivalence of computations. Exercise: partition these sub-diagrams into the commuting and non-commuting ones.

This theory of derivations was first developed by J. J. Lévy in the framework of λ-calculus [17]. It was then adapted to recursive program schemas computations in [3], and to the present framework of regular term rewriting systems in [13]. Klop investigated a more general framework of combinatory reduction systems, which allows binding operators and provides a general theory encompassing λ-calculus, in his dissertation [15].

6.4 Standardization

First we remark that all the results of the preceding section can be applied to sequences of rewritings, confusing a reduction $a : M \to_u N$ with the elementary derivation $A : M \Rightarrow_{\{u\}} N$.

We shall now show that every derivation is equivalent to a certain derivation which computes redexes in an outside-in manner. We shall call such derivations *standard*. A similar situation happens in λ-calculus. However, contrarily to the λ-calculus situation, the leftmost outermost reductions do not usually lead to standard derivations in regular term rewriting systems. For instance, with $\mathcal{R} = \{F(x,K) \to K, H \to K, Z \to Z\}$, the standard derivation starting from the term $F(Z,H)$ is:
$F(Z,H) \to_{[2]} F(Z,K) \to_{[]} K$, whereas the left-most outermost rule leads to an infinite derivation $F(Z,H) \to_{[1]} F(Z,H) \to \cdots$

6.4.1 Preservation of occurrences by derivations

Definition. Let $u \in \mathcal{R}(M)$, $\lambda \to \rho \in \mathcal{R}$ the rule applicable at u: $\lambda \leq M/u$ (it is unique, since we assume there are no critical pairs). The occurrences of M below u are partitioned between the occurrences internal to λ, and the ones which pertain to the substitution part; let us denote these two sets respectively $\mathcal{L}(M,u)$ and $\mathcal{S}(M,u)$.

We may assume that $\mathcal{R}$ is not reduced to the trivial set $\{x \to \cdots\}$, and thus we get $u \in \mathcal{L}(M,u)$, and $\forall v \in \mathcal{S}(M,u)\ u \prec v$. Now if $u \prec v \in \mathcal{R}(M)$, the absence of critical pairs imposes $v \in \mathcal{S}(M,u)$. Furthermore, for every derivation $A : M \Rightarrow_V N$ with $v \in V$, we get $u \backslash A = \{u\}$. Also whenever $u, v \in \mathcal{R}(M)$, with $\mathcal{L}(M,u) \cap \mathcal{S}(M,v) \neq \emptyset$, we have $u \in \mathcal{S}(M,v)$.

Now let A be a derivation starting from M, and contracting successively the redex sets $U_1, \ldots, U_n$. For any $u \in \mathcal{D}(M)$, we say that A *preserves* u iff A never contracts a redex above u, i.e. $\forall i \leq n$ $\nexists v \in U_i\ v \prec u$. The following technical lemmas are easy to establish:

Lemma 1. Let $A \in Der(M)$, preserving v. For each $u \in \mathcal{R}(M)$ such that $u \prec v$ we have $u \backslash A = \{u\}$.

Lemma 2. Let A and B be co-initial, both preserving u. Then $A \backslash B$ preserves u.

Lemma 3. Let A and B be co-initial, with $A \sqsubseteq B$. Every occurrence preserved by B is preserved by A too.

Remark. Occurrences play two rôles in the theory. They may be used to talk about subterms, like in the notation M/u. They may also be used to talk about *contexts*, like in the notation $M[u \leftarrow N]$. Here the notion of preservation concerns the context, and thus if A preserves u it does not mean that A preserves the symbol $M(u)$, but rather the context $M[u]$.

We are now ready to present the important definition of an occurrence being external for a derivation.

6.4.2 External occurrences

Definition. Let $A \in Der(M)$, and $u \in \mathcal{D}(M)$. We say that u is *external* for A, and write $u \in \mathcal{X}(A)$, iff:

- either A preserves u
- or else we can decompose A as $A_1; A_2; A_3$ such that $\exists v \prec u$:
  - A_1 preserves u,
  - $A_2 : P \Rightarrow_V Q$ with $v \in V$ and $u \in \mathcal{L}(P, v)$,
  - and $v \in \mathcal{X}(A_3)$.

Remark that in the second case, the decomposition of A and the occurrence v are unique. Intuitively, $u \in \mathcal{X}(A)$ iff A does not contract at any step a redex above u, and to which u does not contribute (where u contributes to $v \in \mathcal{R}(M)$ iff $u \in \mathcal{L}(M, v)$).

Example. With $\mathcal{R} = \{F(x, K) \to x, H \to K\}$, and $A : F(K, H) \Rightarrow F(K, K) \Rightarrow K$, we get $\mathcal{X}(A) = \{\texttt{[]}, \texttt{[2]}\}$.

Now let A be a derivation starting from M, and contracting successively the redex sets $U_1, \ldots, U_n$. For any $u \in \mathcal{R}(M)$, we say that u is a redex *pertaining* to A, and write $u \in \mathcal{R}(A)$, iff for some $i \leq n$ some residual of u by the first $n-1$ steps of A is in U_n. Finally, we define the *external* redexes *pertaining* to A, as: $\mathcal{E}(A) = \mathcal{R}(A) \cap \mathcal{X}(A)$.

We now extend the technical lemmas above to external redexes.

Lemma 4. Let $A \in Der(M)$, and $u \in \mathcal{R}(M)$. If $\mathcal{X}(A) \cap \mathcal{S}(M, u) \neq \emptyset$, we have $u \backslash A = \{u\}$.

Lemma 5. Let A and B be two co-initial derivations. If B preserves $u \in \mathcal{X}(A)$, then $u \in \mathcal{X}(A \backslash B)$.

Lemma 6. Let A and B be two co-initial derivations, with $A \sqsubseteq B$. We have $\mathcal{X}(B) \subseteq \mathcal{X}(A)$.

Corollary. $A \equiv B$ implies $\mathcal{X}(A) = \mathcal{X}(B)$.

External redexes are their own residuals, until they are contracted:

Lemma 7. Let $A \in Der(M)$, and u be a redex of M external for A: $u \in \mathcal{X}(A) \cap \mathcal{R}(M)$. Then either $u \in \mathcal{R}(A)$ and then $u \backslash A = \emptyset$, or else $u \backslash A = \{u\}$.

Corollary. $A \equiv B$ implies $\mathcal{E}(A) = \mathcal{E}(B)$.

Lemma 8. $\mathcal{X}(A)$ is closed by the prefix ordering $\prec$: $u \in \mathcal{X}(A)$ implies $v \in \mathcal{X}(A)$ for all $v \prec u$.

Lemma 9. $\mathcal{E}(A) = \emptyset$ implies $A \equiv Id$.
Proof. Induction on $|A|$. Trivial for $A = Id$. Otherwise, let $A = A_1; A_2$ with $A_1 : M \Rightarrow_U N$. If $A_2 \equiv Id$, then $\mathcal{E}(A) = U \neq \emptyset$. Otherwise, consider $v \in \mathcal{E}(A_2)$, which exists by induction hypothesis. There are two cases:
1. There exists $u \in U$ such that $u \preceq v$. Then $u \in \mathcal{X}(A_2)$ by lemma 8, and thus $u \in \mathcal{X}(A)$ since A_1 preserves u. Therefore $u \in \mathcal{E}(A)$.
2. Otherwise, v is preserved by A_1, and thus $v \in \mathcal{X}(A)$. Again two cases:
   (i) There exists $u \in U \cap \mathcal{L}(M, v)$. Then $u \in \mathcal{X}(A)$, and therefore $u \in \mathcal{E}(A)$.
   (ii) Otherwise $v \in \mathcal{R}(A)$, and therefore $v \in \mathcal{E}(A)$.

6.4.3 Outside-in and standard derivations

We are now ready to use $\mathcal{E}(A)$ to construct an outside-in equivalent to A. The termination of the construction is based on the following notion.

Definition. We say that derivation B is a *strict outside-in suffix* of A, written $A \rhd B$, iff $A \not\equiv Id$, and $B = A\backslash b$ where b is the elementary derivation contracting some $u \in \mathcal{E}(A)$.

Lemma 10. $\rhd$ is a Nœtherian relation.
Proof. Consider $A : M \Rightarrow_{U_1} \Rightarrow_{U_2} \cdots \Rightarrow_{U_n}$, with $A \not\equiv Id$. Let $u \in \mathcal{E}(A)$, $b : M \rightarrow_u$. According to lemma 7, u is preserved by A until it is contracted, say at step k. Therefore $A\backslash b$ will contract $U_k - \{u\}$ at its $k-th$ step, and from there on will be identical to A. Thus $A\backslash b$ is less than A in the lexicographic ordering $(|U_n|, |U_{n-1}|, \ldots, |U_1|)$.

We are now ready for the main definition of this section.

Definition. The derivation A is said to be *outside-in* iff

- either $A = Id$
- or else $A = A_1; A_2$, where A_1 is the elementary derivation contracting an external redex pertaining to A, and A_2 is outside-in.

We can now give an algorithm normalizing a derivation A to an outside-in equivalent B as follows. If $A \equiv Id$, let $B = Id$. Otherwise, pick some $u \in \mathcal{E}(A)$. We define B as composed of the elementary derivation b contracting u, followed by an outside-in equivalent of $A\backslash b$. Lemma 10 tells us that the construction will terminate, by Nœtherian induction. By construction, we have $B \sqsubseteq A$. But then, according to Lemma 9, the construction can stop only when $A \sqsubseteq B$, and thus we get $B \equiv A$.

In order to get a canonical element, we may choose the leftmost u at every stage:

Definition. A derivation A is said to be *standard* iff A is outside-in, and at every stage the redex u is the lefmost in $\mathcal{E}(A)$.

We may now conclude:

Standardization theorem. Every derivation class possesses a unique standard element.

In other words, the computation category is isomorphic to the category of standard derivations.

Of course, the leftmost condition is not important here, any choice function on $\mathcal{E}(A)$ would do.

Exercise. Let $\mathcal{R} = \{F(x, K) \rightarrow G(x, x), H \rightarrow K\}$. Draw all derivations starting from $F(H, H)$. Which of all sub-derivations are standard?

6.4.4 Normal derivations

The results above do not tell us how to compute in a standard way, since $\mathcal{E}(A)$ may depend on the whole of A, not just on its starting term. We shall now relativize the notions above to an initial term, in order to define a notion of outside-in computation rule, generalizing the normal order evaluation rule of recursive program schemes.

Definition. The set of *external occurrences* of a term M is defined as: $\mathcal{X}(M) = \bigcap_{A \in Der(M)} \mathcal{X}(A)$. Also, we define the *external redexes* of M as: $\mathcal{E}(M) = \mathcal{R}(M) \cap \mathcal{X}(M)$.

We need one more technical definition. A derivation is said to be *internal* iff it never reduces a redex at the top occurrence. An elementary derivation is *top* iff it reduces a top redex. Every derivation is internal, or else of the form $A = A_1; A_2; A_3$ with A_1 internal and A_2 top. It follows directly from the definitions that in that case $\mathcal{X}(A) = \mathcal{X}(A_1; A_2)$.

Lemma 11. $\mathcal{E}(M) = \emptyset$ iff M is a normal term.
Proof. If M is normal, $\mathcal{R}(M) = \emptyset$, and thus $\mathcal{E}(M) = \emptyset$. If M is not normal, we prove by induction on $|M|$ that $\mathcal{E}(M) \neq \emptyset$. There are two cases.

1. Every derivation in $Der(M)$ is internal. Thus $M = F(M_1, \ldots, M_n)$ and for some i the term M_i is not normal. By the induction hypothesis there exists u_i in $\mathcal{E}(M_i)$, and therefore $i \cdot u_i \in \mathcal{E}(M)$.

2. There is some $A = A_1; A_2$ starting from M, with A_1 internal and A_2 top. Now take any B starting from M. We get $\mathcal{X}(A + B) = \mathcal{X}(A)$ by the remark above, and thus $\mathcal{X}(A) \subseteq \mathcal{X}(B)$ by lemma 6, which shows that $\mathcal{E}(A) \subseteq \mathcal{E}(M)$. Since $A \not\equiv Id$, we get $\mathcal{E}(M) \neq \emptyset$ by lemma 9.

We may now define the notion of a *normal derivation* $\mathcal{N}(M)$ starting from a term M:

- If M is a normal term, then $\mathcal{N}(M) = Id_M$.
- Otherwise, let u be leftmost in $\mathcal{E}(M)$, $A : M \to_u N$. Then $\mathcal{N}(M) = A; \mathcal{N}(N)$.

We may now state a relativized version of the standardization theorem:

Normal derivation theorem. If M reduces to a normal term N, the normal derivation starting from M will end in N; it is the standard in the class of all derivations from M to N.

It is possible to explain the above results as justifying the existence of a *call by need* computation rule. Let us call *needed* redex of a term M reducing to a normal term N any redex relevant to every derivation from M to N, or equivalently to some outside-in derivation from M to N. The results above show that the external redexes are needed. Of course, there may be other needed redexes; for instance, it is easy to show that if for every rule $\lambda \to \rho$ of $\mathcal{R}$ we have $\mathcal{V}(\rho) = \mathcal{V}(\lambda)$, then every redex is needed. Any interpreter which is fair for needed redexes, in the sense that it will not postpone forever the contraction of its residuals, is correct in the sense that it will lead to the normal form, if it exists. However, we do not have any effective general way of computing a needed redex. In particular, we do not have an effective way of computing an element in $\mathcal{E}(M)$. This problem will be attacked in the next section.

6.5 Sequentiality

6.5.1 Inherently parallel systems

Certain rewriting systems need parallel evaluation. For instance, consider the "parallel or" definition:

Parallel or

$$\begin{cases} True \vee x \to True \\ x \vee True \to True \end{cases}$$

In order to compute $M \vee N$, we must evaluate in parallel M and N, since it will be undecidable in general (there may be many other rules) whether M or N may reduce to the constant $True$. Furthermore, in the case where M and N both reduce to $True$, there is no needed redex in $M \vee N$ in the sense of the last section. This is because the system above is not regular, since there is a critical pair. We know from the previous theory that in a regular system there always exists a needed redex. The problem is how to compute a needed redex without doing a lot of look-ahead

(like computing a normal term using all possible derivations, and then normalizing to a standard one if one is found!)

The next example, due to G.G. Berry [2], will illustrate the problem.

Gustave's algorithm

$$\begin{cases} F(True, False, x) \to True \\ F(x, True, False) \to True \\ F(False, x, True) \to True \end{cases}$$

Note that this example is regular. Thus in any term $M = F(M_1, M_2, M_3)$ there must exist a needed redex. But it is undecidable in general (i.e. in the presence of other rules) if M_i reduces to $True$ or $False$, and thus it seems impossible to compute Gustave's algorithm without some kind of parallel evaluation, which will in general do some un-needed computation. For instance, if you start computing the first argument, you may run into a non-terminating computation, whereas it may be the case that $M_2 \to^* True$, $M_3 \to^* False$, and thus $M \to^* True$. We must find a characterization of such pathological examples, in order to be able to construct sequential evaluators for a restricted class of regular systems.

6.5.2 Ω-terms

We need to develop a minimum of theory to explain how to compute trees and terms by approximation. Intuitively, our interpreter will work in a top-down fashion, and at any point we shall have read only a prefix approximation of the input.

We thus augment our term alphabet by adding to the alphabet Φ a new constant Ω, which will stand for "unknown". Terms (without variables) over this extended alphabet will be called Ω-terms. They are used to talk about partially computed terms. The approximation or prefix ordering over Ω-terms will be the same as the familiar match ordering, but where Ω acts as a "dont-care" variable. Occurrences of Ω in M are called *omegas*, and we write $\mathcal{F}(M)$ for the set of such occurrences (the *frontier* of M). We partition the domain of an Ω-term in its frontier and its set $\mathcal{O}(M)$ of strict occurrences:

$$\mathcal{D}(M) = \mathcal{F}(M) \cup \mathcal{O}(M).$$

Finally, if M is any term, we write M_Ω for the Ω-term obtained by substituting every variable by Ω in M.

Definition. The *prefix* ordering $\preceq$ over Ω-terms is the smallest term compatible relation verifying $\Omega \preceq M$ for all M.

It is easy to verify that $\preceq$ is a partial ordering, which could be alternatively defined by:

$$M \preceq N \Leftrightarrow \mathcal{D}(M) \subseteq \mathcal{D}(N) \wedge \forall u \in \mathcal{O}(M)\ M(u) = N(u).$$

This is of course the specialization to linear terms of the familiar partial ordering of matching. Actually, the set of Ω-terms is order-isomorphic to the set of linear terms with variables.

If $M \preceq P$ and $N \preceq P$, we say that the Ω-terms M and N are *compatible*, which we write $M \uparrow N$.

We may then define $M \sqcap N$ as the P with $\mathcal{D}(P) = \mathcal{D}(M) \cap \mathcal{D}(N)$ such that $P(u) = F$ if $M(u) = N(u) = F$, and $P(u) = \Omega$ otherwise. Similarly, when $M \uparrow N$, we define $M \sqcup N$ as the P

with $\mathcal{D}(P) = \mathcal{D}(M) \cup \mathcal{D}(N)$ such that $P(u) = F$ if $u \in \mathcal{D}(M) \cap \mathcal{D}(N)$ and $M(u) = N(u) = F$, or $u \in \mathcal{D}(M) - \mathcal{D}(N)$ and $M(u) = F$, or $u \in \mathcal{D}(N) - \mathcal{D}(M)$ and $N(u) = F$.

We leave it to the reader to check that $M \sqcap N$ is the greatest lower bound, and that $M \sqcup N$ is the conditional lowest upper bound of M and N. Actually, we do not need to restrict ourselves to finite terms, with finite domains. The whole theory applies to arbitrary terms, with finite or infinite terms possessing finite and infinite occurrences. Then we may generalize $\sqcup$ to $\bigsqcup_E$, where E is a directed set of Ω-terms. Thus Ω-terms form a complete partial ordering. It is an algebraic cpo, with the finite Ω-trees as finite elements: every Ω-tree is the limit of its finite approximants. This is of course a direct consequence of the compactness of its set of occurrences.

Actually, Ω-trees form a domain of a very special kind, a *concrete domain* in the sense of Kahn-Plotkin [14]. For instance, we get:

Proposition. If $M \not\preceq N$, there exists $P \not\preceq M$ such that $M \sqcap N \prec P \preceq N$.
Proof. Let $U = \mathcal{D}(M \sqcap N)$. By hypothesis we get $U \subset \mathcal{D}(N)$. Let u minimum in $\mathcal{D}(N) - U$. We take P as the restriction of N to $U \cup \{u\}$.

6.5.3 Derivation sequences

From now on, we assume that $\mathcal{R}$ is a given regular system. We now consider derivations from M to N, where M and N are Ω-terms.

We shall here abstract a derivation $A : M \Rightarrow_{U_1} \cdots \Rightarrow_{U_n} N$ into its derivation sequence $U^* = [U_1, \ldots, U_n]$. This derivation sequence may be applied to any initial term which contains at least all the occurrences that are used by some reduction step. Let us make that more precise. Let $Used(A) = \{u \in \mathcal{D}(M) \mid u \backslash A = \emptyset\}$ be the set of occurrences in M which are used by derivation A. We now close this set by prefix, and get $Useful(A) = \{u \in \mathcal{D}(M) \mid \exists v \in Used(A)\ u \preceq v\}$. By restricting M to $Useful(A)$ we get an Ω-term $Dom(A)$, and it makes sense to consider the derivation starting from $Dom(A)$ and having as derivation sequence U^*, i.e. $\bar{A} : Dom(A) \Rightarrow^*_{U^*} Codom(A)$. Let $Vars(A) = \mathcal{F}(Dom(A))$. Now N can be constructed uniquely as $N = Codom(A)[u \leftarrow M/v \mid v \in Vars(A) \wedge u \in v\backslash A]$.

Remark. $Dom(A)$ is not invariant by permutation, because some redex $u \in \mathcal{R}(A)$ may be not needed. However, $Dom(st(A))$ is the minimum common prefix to every $Dom(B)$, for $B \equiv A$, with $st(A)$ the standard equivalent to A.

We can number the omegas of $Dom(A)$ (for instance, from left to right), and thus get a term with variables $Left(A) = Dom(A)[u_i \leftarrow i \mid u_i \in Vars(A)]$. Similarly, we get a term with variables: $Right(A) = Codom(A)[u \leftarrow i \mid v_i \in Vars(A) \wedge u \in v_i\backslash A]$. The term $Left(A)$ is linear, but $Right(A)$ is not in general, since one of the occurrences in $Vars(A)$ may have several residuals by A. Now we see that $\bar{A}$ is a substitutive tree-transformation: $\bar{A} : Left(A) \to Right(A)$, i.e. for every substitution σ we get: $\sigma(\bar{A}) : \sigma(Left(A)) \Rightarrow^*_{U^*} \sigma(Right(A))$.

Derivations preserve the term structure, in the sense of the following proposition, whose proof is left as exercise:

Proposition 1. Let M and N be two Ω-terms, and u a common redex in M and N. That is, with λ the corresponding pattern: $\{u@v \mid v \in \mathcal{O}(\lambda)\} \subseteq \mathcal{O}(M \sqcap N)$. Let $M \to_u M'$ and $N \to_u N'$. We have $M \sqcap N \to_u M' \sqcap N'$. Furthermore, when $M \uparrow N$, we get $M' \uparrow N'$ and $M \sqcup N \to_u M' \sqcup N'$.

Furthermore, these preservation properties commute with the permutation equivalence. For instance, let us consider the interference of two redexes u_1 and u_2.

Proposition 2. Let $\{u_1@v \mid v \in \mathcal{O}(\lambda_1)\} \subseteq \mathcal{O}(M \sqcap N)$, and $\{u_2@v \mid v \in \mathcal{O}(\lambda_2)\} \subseteq \mathcal{O}(M \sqcap N)$. Let $M \rightarrow_{u_1} \Rightarrow_{V_1} M'$ and $N \rightarrow_{u_2} \Rightarrow_{V_2} N'$, with $V_1 = u_2 \backslash u_1$, and $V_2 = u_1 \backslash u_2$. We have $M \sqcap N \rightarrow^* M' \sqcap N'$. Furthermore, when $M \uparrow N$, we get $M' \uparrow N'$ and $M \sqcup N \rightarrow^* M' \sqcup N'$.
Proof. Exercise.

Corollary. Let A and B be two equivalent derivation sequences starting from Ω-term P: $A : P \Rightarrow^* P'$, $B : P \Rightarrow^* P'$, $A \equiv B$. Let $M \succeq P$ and $N \succeq P$. Considering A and B as derivation sequences, it makes sense to write: $A : M \Rightarrow^* M'$, $B : M \Rightarrow^* M'$, and similarly: $A : N \Rightarrow^* N'$, $B : N \Rightarrow^* N'$. We have $A : M \sqcap N \Rightarrow^* M' \sqcap N'$. Furthermore, when $M \uparrow N$, we get $M' \uparrow N'$ and $M \sqcup N \Rightarrow^* M' \sqcup N'$.

Thus the structure of terms is preserved by equivalent derivations. In categorical terms, this could be expressed by considering derivations sequences as functors which preserve pullbacks.

6.5.4 Computability

Definition. We say that the Ω-term M is *computable* iff $M \rightarrow^* N$, where N is a normal term (without Ω's).

Computability is increasing with respect to the prefix ordering:
Fact. If M is computable, then every $N \succeq M$ is computable.

Furthermore, computability is a *stable* property in the terminology of Berry [2]:

The Stable Computability Theorem. Let $M \uparrow N$. Then $M \sqcap N$ is computable iff both M and N are.
Proof. Assume M and N are computable. Consider the standard derivation from $M \sqcup N$ to its normal form, which is thus the common normal form P of M and N. We have $Dom(A) \subseteq \mathcal{O}(M)$ and $Dom(A) \subseteq \mathcal{O}(N)$, and thus $Dom(A) \subseteq \mathcal{O}(M \sqcap N)$. Consider the derivation $A : M \sqcap N \Rightarrow^* Q$. We have from the preceding corollary $Q = P \sqcap P = P$ and thus $M \sqcap N$ is computable.

Corollary. Every computable Ω-term M possesses a minimum computable prefix N, and M and N admit the same normal form.

Note that this key property is not true of non-regular systems. For instance, with the parallel-or system above, both Ω-terms $True \vee \Omega$ and $\Omega \vee True$ are computable, but $\Omega \vee \Omega$ is not.

If we reconsider Gustave's algorithm, we get in particular that any computable Ω-term $M = F(M_1, M_2, M_3)$ has a minimum computable prefix. But this does *not* mean that there is a uniform way of computing in all such M's. Actually, $F(True, False, \Omega)$, $F(\Omega, True, False)$ and $F(False, \Omega, True)$ are all three computable, even though $F(\Omega, \Omega, \Omega)$ is not.

Remark. Our notion of computability is absolute. We could define a more general notion of M *computable towards* P, defined as the existence of some $N \succeq P$ such that $M \rightarrow^* N$. This notion could be used to study computation rules for derivations starting from terms without normal forms. But since this would involve considering infinite terms and infinite derivations, we shall not develop this further here.

6.5.5 Sequentiality

Intuitively, Gustave's algorithm is not sequential. Our goal is now to characterize a restriction of regular systems for which we can design a sequential evaluator, which will compute uniformly on prefixes of terms.

Technically, we are going to require the computability predicate to be sequential in the sense of Kahn and Plotkin [14].

Definition. Let $\mathcal{P}$ be a prefix-increasing property on Ω-terms. That is, $M \preceq N$ and $\mathcal{P}(M)$ implies $\mathcal{P}(N)$. We say that $u \in \mathcal{F}(M)$ is an *index* of $\mathcal{P}$ in M iff $\forall N \succeq M\ \mathcal{P}(N) \Rightarrow u \notin \mathcal{F}(N)$.

Thus an index of $\mathcal{P}$ is a place in its argument which we must fill in order to achieve $\mathcal{P}$.

Definition. A predicate $\mathcal{P}$ is *sequential at M* iff

- either $\mathcal{P}(M)$
- or $\neg\mathcal{P}(N)$ for every $N \succeq M$
- or else there is an index of $\mathcal{P}$ in M.

Finally, $\mathcal{P}$ is *sequential* iff it is sequential at every Ω-term.

Intuitively, a sequential predicate is input-driven: it can be fulfilled only by computing at critical, or *needed* places, its indexes. Note that every predicate is sequential at every term (i.e. without Ω's). The notion is interesting only for the Ω terms M such that $\mathcal{P}(M)$ is false but $\mathcal{P}(N)$ is true for some $N \succ M$.

Definition. A term rewriting system is *sequential* iff its computability predicate is sequential.

Lemma 12. If $\mathcal{R}$ is regular, its sequentiality need only be tested on normal Ω-terms.
We do not give here the proof of this lemma, which uses auxiliary technical notions. The interested reader may consult [13] for a proof.

Sequential evaluation theorem. Let $\mathcal{R}$ be sequential. Let $\Omega(M) = M[u \leftarrow \Omega \mid u \in \mathcal{R}(M)]$. If M is not normal, then any index of the computability predicate in $\Omega(M)$ is a needed redex of M.

This theorem gives the existence of a terminating computation strategy for any term reducing to a normal term. Unfortunately, this strategy is not effective in general, since recognizing an index may be an undecidable property. For instance, consider the regular system:

The undecidably sequential system

$$\left\{\begin{array}{ll} F(G(A,x),B) & \rightarrow\ K \\ F(G(x,A),C) & \rightarrow\ K \\ F(D,x) & \rightarrow\ K \\ G(E,E) & \rightarrow \cdots \\ \cdots & \rightarrow \end{array}\right.$$

Now, consider $M = F(G(\Omega,\Omega),\Omega)$. Its occurrence [2] is a computability index iff $G(E,E)$ cannot compute to D, an undecidable property, since there may be many more rules. Note that if indeed [2] is a computability index, then the result at that place (e.g. B or C) will then be used to determine which argument of $G(\Omega,\Omega)$ to evaluate next. Thus a sequential evaluator need not base its strategy solely on the arguments of a given functor, but on the *context* of the computation as well. This shows that our theory of sequentiality is stronger than previous semantical proposals such as Sazonov's [19].

In order to be able to compute effectively computability indexes, and to decide sequentiality of regular systems, we must restrict further the sequentiality condition.

6.5.6 Necessary sequentiality

The main idea is to ignore the right-hand side of rules, and to make sequentiality determined only from the left-hand sides of the rules. This leads us to a modal theory of sequentiality.

Definition. We say that M *possibly reduces* to N, and write $M \rightarrow N$, iff $N = M[u \leftarrow P]$ for some redex $u \in \mathcal{R}(M)$ and some arbitrary Ω-term P. We say that M is *possibly computable* iff $M \rightarrow^* N$ for some normal term N.

We now restrict indexes by enlarging computability:

Definition. A term rewriting system is *necessarily sequential* iff the possible computability predicate is sequential at every normal Ω-term.

Lemma 13. A necessarily sequential system is sequential.
Proof. Let M be any normal Ω-term which is not a normal term. It does not possibly reduce to a normal term, and if there is some $N \succeq M$ such that N reduces to a normal term, it also possibly reduces to it, and thus an index of possible computability at M is also an index of computability. The result follows from Lemma 12.

However, notice that the equivalent of Lemma 12 does not hold, i.e. there may exist non-normal Ω-terms which have no index of possible computability. For instance, with $\mathcal{R} = \{F(True, False, x) \rightarrow K, F(False, x, True) \rightarrow K, H \rightarrow K\}$, consider $M = F(H, \Omega, \Omega)$. Now $\Omega(M) = F(\underline{\Omega}, \Omega, \Omega)$ has an index of possible computability (underlined), but M has not.

Let us note $\mathcal{I}(M)$ for the set of indexes of possible computability. From now on, we shall simply call *indexes* of M the members of $\mathcal{I}(M)$.

Definition. Let M be a non-normal term, and $u \in \mathcal{R}(M)$. The redex u is said to be *necessarily needed* iff $u \in \mathcal{I}(\Omega(M))$.

Remark. A necessarily needed redex is needed.

We now present a correct sequential computation rule.

The sequential evaluator. At every step, select a necessarily needed redex.

We must now explain algorithms to find effectively a necessarily needed redex in every non-normal term, and to decide the necessary sequentiality of regular term-rewriting systems.

6.5.7 Approximation

Definition. Let $\mathcal{R}$ be a regular system. We define $\mathcal{R}_\Omega = \{\lambda_\Omega \mid \lambda \rightarrow \rho \in \mathcal{R}\}$ for the set of Ω-*patterns* of $\mathcal{R}$. We write $M \uparrow \mathcal{R}$ iff M and N are compatible Ω-terms, for some $N \in \mathcal{R}_\Omega$. We use a similar convention for other notations such as $M \prec \mathcal{R}$. Note that, since $\mathcal{R}$ is assumed to be left-linear, M is top-reducible iff $M \succeq \mathcal{R}$.

We are now ready to define the *approximant* $\omega(M)$ of an Ω-term M as follows:

- $\omega^+(\Omega) = \Omega$
- $\omega^+(F(M_1, \ldots, M_n)) = F(\omega(M_1), \ldots, \omega(M_n))$
- $\omega(M) = \text{let } M^+ = \omega^+(M) \text{ in if } M^+ \uparrow \mathcal{R} \text{ then } \Omega \text{ else } M^+$.

Note that for every Ω-term M we have $\omega(M) \preceq \Omega(M) \preceq M$. Intuitively, the approximant $\omega(M)$ is the *minimum information* available without look-ahead concerning the computations starting from M: if it is Ω, then M is possibly ultimately top-reducible. More generally, $\omega(M)$ is a common prefix to all terms to which any extension of M possibly reduces.

Example. Let $\mathcal{R}_\Omega = \{F(G(\Omega, A), B), H(B), C\}$, and $M = G(F(G(\Omega, H(C)), \Omega), A)$. We get $\omega(M) = G(\Omega, A)$.

It is easy to verify that ω is an idempotent increasing function on Ω-terms.

6.5.8 A decision procedure for indexes

Definitions. An Ω-term M is *fixed* iff $M = \omega(M)$. For instance, a term is fixed iff it is normal. A fixed Ω-term M is said to be *solid* iff there is no non-Ω subterm N of an element of $\mathcal{R}_\Omega$ such that $M \uparrow N$. For instance, any constant which does not appear in any pattern is solid. Also, every proper non-Ω subterm of an Ω-pattern is solid, by regularity. Intuitively, a solid Ω-term solidifies locally the approximation function. Finally, we say that M is *ω-maximum* iff $\omega(N) = \omega(M)$ for every $N \succeq M$.

We now have all the tools to connect the notions of approximation and of possible reduction, by showing that an Ω-term is possibly computable iff it is ω-maximum.

Let K be any given solid Ω-term. If u is an occurrence of Ω in M, $\omega(M[u \leftarrow K])$ is the maximum of all $\omega(M[u \leftarrow N])$. We define the *solidified* term $\widehat{M}$ associated with Ω-term M as M where every Ω is replaced by K: $\widehat{M} = M[u \leftarrow K \mid M/u = \Omega]$. An Ω-term is ω-maximum iff $\omega(\widehat{M}) = \omega(M)$, a condition easily testable.

Lemma 14. $M \rightarrow^* N$ iff $\omega(\widehat{M}) \preceq \widehat{N}$.
Corollary. An Ω-term is possibly computable iff it is ω-maximum.

The preceding lemma gives a decision procedure for possible computability. The following lemma gives a decision procedure for an omega to be an index.

Lemma 15. Let M be an Ω-term, and $u \in \mathcal{F}(M)$. Then $u \in \mathcal{I}(M)$ iff $\omega(M) \neq \omega(M[u \leftarrow K])$ iff $u \in \mathcal{D}(\omega(M[u \leftarrow K]))$.

That is, an index is a place where it is possible to increase the approximant.

Example. Let $\mathcal{R}_\Omega = \{F(G(A, \Omega), F(B, \Omega)), F(G(\Omega, A), F(C, \Omega)), G(D, D)\}$. The following Ω-terms have their indexes underlined:
$F(\underline{\Omega}, \underline{\Omega})$
$F(G(\Omega, \Omega), \underline{\Omega})$
$F(\underline{\Omega}, F(\underline{\Omega}, \Omega))$
$F(G(\Omega, \Omega), F(\underline{\Omega}, \Omega))$

Let us give a few immediate consequences of the previous lemmas.

Lemma 16. If $u@v$ is an index of M, then v is an index of M/u. If furthermore $\omega(M/u) = \Omega$, then u is an index of $M[u \leftarrow \Omega]$. Similarly, when v is disjoint from u such that $\omega(M/u) = \Omega$, then v is an index of M iff v is an index of $M[u \leftarrow \Omega]$.

The remaining problem is deciding whether a regular term rewriting system is necessarily sequential or not. The problem is not entirely trivial. For instance, the system given in the last example is *not* necessarily sequential, but the smallest Ω-term with no index is $F(G(\Omega,\Omega),F(G(\Omega,\Omega),\Omega))$.

We shall postpone temporarily this problem, and study in the next section pattern-matching in necessarily sequential systems.

6.5.9 Directions

Definition. Let S be any set of Ω-terms. We consider the predicate $Match_S$ defined by $Match_S(M)$ iff $M \succeq N$ for some $N \in S$. We say that set S is *sequential* iff $Match_S$ is sequential. The set $Dir_S(M)$ of indexes of this predicate at M are called the *directions towards* S in M. In particular, we shall consider the predicate $\mathcal{R}$*-match* defined as $Match_{\mathcal{R}_\Omega}$, and we note $Dir_{\mathcal{R}}(M)$ for the set of its indexes in M, called simply *directions* in M. We say that $\mathcal{R}$ is *match-sequential* iff $\mathcal{R}$-match is sequential.

Exercise. Show that every index of M must be a direction in M.

Lemma 17. An omega u of M is a direction towards S in M iff for every $N \in S$ such that $M \uparrow N$, u is a non-Ω occurrence of N.

Examples. Let $\mathcal{R}_\Omega = \{F(A,\Omega), F(G(K,\Omega),B), F(G(\Omega,K),C)\}$. The directions in the following Ω-terms are the underlined omegas:
$\underline{\Omega}$
$F(\underline{\Omega},\Omega)$
$F(G(\Omega,\Omega),\underline{\Omega})$
$F(G(\underline{\Omega},\Omega),B)$
$F(G(\Omega,\underline{\Omega}),C)$.

In this example, every direction in also an index. If now we consider $\mathcal{R}_\Omega = \{F(G(A,B)), G(\Omega,C)\}$, both omegas in $F(G(\Omega,\Omega))$ are directions, but only the rightmost one is an index.

We shall now see that the (finite) set of proper prefixes of Ω-patterns is sufficient to search for directions (whereas the set of Ω-terms simply compatible with some Ω-pattern is in general infinite).

Lemma 18. Let $S \parallel M = \{N \in S \mid N \uparrow M\}$. If $S \parallel M$ is empty, then $Match_S$ is trivially sequential at M. Otherwise, the directions towards S in M are the directions towards $S \parallel M$ in $M \sqcap N$, for any $N \in S \parallel M$.
Corollary. For any finite S, we may decide whether S is sequential.
Proof: Check that $Match_S$ is sequential at every M proper prefix of some $N \in S$.

Lemma 19. If $\mathcal{R}$ is necessarily sequential, then $\mathcal{R}$ is match-sequential.
Proof. If M is normal, then an index of M is also a direction. Thus every $M \prec \mathcal{R}_\Omega$ has directions. If $M \uparrow \mathcal{R}_\Omega$, use lemma 18.

Lemma 20. Let $M \prec \mathcal{R}_\Omega$, $v \in Dir_{\mathcal{R}}(M)$, and $N = M[v \leftarrow F(\Omega,\ldots,\Omega)]$. If $N \uparrow \mathcal{R}_\Omega$, then $N \preceq \mathcal{R}_\Omega$.

Intuitively, the notion of direction allows us to organize the set of patterns of $\mathcal{R}$ as a trie structure in order to factorize pattern matching over all patterns. When $\mathcal{R}$ is necessarily sequential, it is even possible to construct data-structures, called below matching tries, which factorize tree traversals for both global and local searches.

Note that the converse of the last lemma is true for systems that correspond to recursive definitions by cases on constructors, that is when the functors are divided into *defined* functors and *constructors*, and every pattern consists of a defined functor applied to constructor terms with variables. This is the case for `ML` definitions by cases, whenever the patterns are mutually incompatible: such definitions are sequentially evaluable iff there are directions for pattern matching. The rest of the general theory concerns systems where there is no clear notion of (free) constructor.

6.5.10 Progressive systems

Definition. Let $\mathcal{R}$ be a regular term rewriting system. We say that $\mathcal{R}$ is *progressive* iff there is a function Δ mapping every $M \prec \mathcal{R}_\Omega$ to a non-empty set of its directions $\Delta(M) \subseteq Dir_{\mathcal{R}}(M)$ such that, for every $u \in \Delta(M)$ and for every $N \prec \mathcal{R}_\Omega$, with $P = M[u \leftarrow N]$, whenever $P \prec \mathcal{R}_\Omega$ the set $\{v \mid u@v \in \Delta(P)\}$ is a non-empty subset of $\Delta(N)$.

In particular, a progressive system is match-sequential. But it is so in a very strong way: global and local matches can be factored together, leading to a linear pattern-matching algorithm. In other words, no backtracking is necessary when looking for a redex.

Examples. The example S considered above defines the patterns of a progressive system. Gustave's algorithm is not, since it is not even match-sequential. The undecidably sequential system is match-sequential, but definitely not progressive, since the only direction in $F(\Omega,\Omega)$ is `[1]`, and the only direction in $F(G(\Omega,\Omega),\Omega)$ is `[2]`, even though $G(\Omega,\Omega)$ is the prefix of a pattern. This reflects the fact that this system is possibly sequential, but not necessarily sequential, since we shall see that a regular system is progressive iff it is necessarily sequential.

We are now able to state a non-deterministic sequential evaluator:

Algorithm A. Input: a term M without variables. The algorithm uses two occurrence variables u and v, and one Ω-term variable P. P is a prefix of M, denoting the part of M that has been read so far. The occurrence u is an outer-most potential redex in P, and v is a direction in P/u.
Initialisation: $P \leftarrow \Omega$;
Start global search: Choose u in $\mathcal{F}(P)$;
Get next symbol: Choose v in $\Delta(P/u)$;
let $F = M(u@v)$ in $P \leftarrow P[u@v \leftarrow F(\Omega,\ldots,\Omega)]$. There are two cases:

1. $\exists v' \preceq v \quad (P/u@v')\uparrow\mathcal{R}_\Omega$; Then for the minimum such v', $u \leftarrow u@v'$ and two cases again:
   (i) $P/u \in \mathcal{R}_\Omega$: exit with answer "Redex u".
   (ii) Otherwise: Get next symbol.
2. Otherwise, two cases:
   (i) $\mathcal{F}(P) = \emptyset$: exit with answer "Normal form".
   (iii) Otherwise: Start global search.

Theorem. Let M be any term without variables. If M is normal, algorithm **A** terminates with answer "Normal form". Otherwise, it terminates with answer "Redex u", with $u \in \mathcal{I}(\Omega(M))$ and thus $M/u \succeq \mathcal{R}_\Omega$.

The proof of the theorem above is done using Floyd's method. We shall show that after choosing v the following assertions are true:
(a) $P \prec M$
(b) $P[u \leftarrow \Omega] \preceq \Omega(M)$
(c) $P/u \prec \mathcal{R}_\Omega$
(d) $\forall w \in \mathcal{D}(P) - \mathcal{F}(P) \quad (P/w\uparrow\mathcal{R}_\Omega) \Rightarrow w \succeq u \wedge (P/w \prec \mathcal{R}_\Omega)$
(e) $v \in \Delta(P/u)$

We leave it to the reader to show the invariance of the above assertions, using previous results such as lemma 20. Note that it is essential that Δ be closed by suffix, in the sense that $u@v \in \Delta(M)$ and $M/u \prec \mathcal{R}_\Omega$ imply $v \in \Delta(M/u)$. This property is verified by indexes. Let us now show the converse. More generally, we show that the above assertions entail that $u@v \in \mathcal{I}(P)$:

Lemma 21. Let $P/u \prec \mathcal{R}_\Omega$ with P such that $\forall w \in \mathcal{D}(P) - \mathcal{F}(P) \ \ (P/w{\uparrow}\mathcal{R}_\Omega) \Rightarrow w \succeq u$. Then $u@v \in \mathcal{I}(P)$ for all $v \in \Delta(P/u)$.
Corollary. $\Delta(M) \subseteq \mathcal{I}(M)$.

Let us however remark that $\mathcal{I}(M)$ does not in general satisfy the requirements for $\Delta(M)$. For instance, consider $\mathcal{R}_0 = \{F(G(A,\Omega),B), F(G(\Omega,A),C)\}$, and $\mathcal{R}_1 = \mathcal{R}_0 \cup \{G(E,E)\}$. The system $\mathcal{R}_1$ is necessarily sequential, and both omegas in $F(\Omega,\Omega)$ are indexes. However, choosing the first one as a candidate for v above is wrong, since then at the next step we may get $P = F(G(\Omega,\Omega),\underline{\Omega})$ which has only one index (underlined) and thus assertion (d) cannot be maintained. This in turn may falsify assertion (b) later. For instance, we would miss the redex in term $F(G(E,E),D)$. Thus the members of Δ must be not only indexes, but indexes that may be increased as further indexes, so that we get a linear piling-up of compatible subterms of P.

This does not entail however that we have a linear chain of non-omega symbols below u. For instance, consider $\mathcal{R}_2 = \mathcal{R}_0 \cup \{F(D,x)\}$. Now in $F(\Omega,\Omega)$ only the first omega is an index. If we choose it as v, we get $P = F(G(\Omega,\Omega),\underline{\Omega})$. Now this is consistent with all assertions, since here $G(\Omega,\Omega)$ is not potentially reducible.

Let us now finish the proof of the theorem above. When algorithm **A** terminates at "Normal form", we get $P = M$, and M normal. If it terminates at "Redex u", we get $u \in \mathcal{I}(\Omega(M))$ by lemma 16.

We shall see later that every necessarily sequential system is progressive. Conversely, every progressive system has computable indexes. This provides a decision procedure for necessary sequentiality, since the condition on Δ is finitary.

There are basically two ways of constructing sets candidate for the Δ operation: proceed bottom-up from the empty sets, adding progressively elements with the existence condition. Or else proceed top-down from the sets of directions, subtracting directions which violate the suffix condition. The first method seems preferable, since it will generate a minimum solution when possible. There is no known efficient method of conducting this search, so we may have to backtrack through the whole search space. However, this is done once and for all for a given set of rules, and thus this should be counted as the cost of building a compiler for the language. Once the Δ sets have been computed, we shall see that it is possible to implement efficiently the sequential interpreter.

6.5.11 Fast pattern-matching

The first problem we have to tackle is how to make the evaluator deterministic. There are two non-deterministic choices. The first one is the choice of u among the omegas of P when starting a global search. This is not a problem. We may for instance decide to pick the left-most such omega. This can be implemented efficiently as follows. Assume we mark every node of the original term M as "not traversed". Every time we read a new symbol, we mark it as "traversed". This way we shall get all the choices in a linear time, since term M is traversed just once. This uses just one bit of space for every node of M, plus a traversing stack of length bounded by the depth of M (i.e. $log|M|$).

The second non-deterministic choice is v in $\Delta(P/u)$. Since the domain of Δ is finite, we may have pre-computed it in a table where we have made a deterministic choice once and for all. However, this leads to a cost proportional, for each node of M, to the maximum size of a pattern of $\mathcal{R}$. This seems a waste, since the search in the table storing the subterms of patterns could be done progressively along the traversal of term M.

However there is a difficulty in implementing this approach: we may have to store in this dictionary structure several distinct linearizations of some of these patterns prefixes. This is true even if we make a deterministic choice of the directions in the global search, since local searches may be forced along several distinct partial linearizations. Let us give an example.

The problematic progressive system.
Let $\mathcal{R}_\Omega = \{G(F(A,\Omega)), F(F(\Omega,H(A)),B), G(F(F(\Omega,H(\Omega)),C))\}$. Assume we are looking for a redex in $M \succeq P = G(F(F(\Omega,\underline{\Omega}),\Omega))$. The underlined Ω is the only v where to look next, since the three non-omegas are potential redexes, and the leftmost omega is not a direction toward the second pattern. Now there are two scenarios. Either we read an H at this place, and wind up with $P = G(F(F(\Omega,H(\Omega)),\underline{\Omega}))$ and then we may read a B leading to $P = G(F(F(\Omega,H(\underline{\Omega})),B))$. The second scenario is to read a B, and wind up successively with $P = G(F(F(\underline{\Omega},B),\Omega))$, next $P = G(F(F(F(\Omega,\underline{\Omega}),B),\Omega))$, finally $P = G(F(F(F(\Omega,H(\underline{\Omega})),B),\Omega))$. In both scenarios we end up looking for a global redex in subterm $F(F(\Omega,H(\underline{\Omega})),B)$, but unfortunately with two distinct linearizations: $F;F;H;B$ and $F;B;F;H$ respectively.

This example shows that it is not possible to simplify the definition of progressive system in such a way as to make each Δ set a singleton. What we may hope at best in to replace $\Delta(M)$ by a single direction $\delta(w_M)$, where w_M is a linearization of M. We do not have to examine all such linearizations, since they must be obtainable as possible suffixes of sequences of δ's. This computation can be effected at the same time as the progressivity condition is checked, by growing a *matching trie* as explained below.

A δ-matching trie has nodes of the form (w_M, v) with w_M a linearization of a subterm of a pattern, and $v = \delta(w_M)$. It has also success nodes, labeled with linearizations of patterns, and a failure node labeled Fail. Its arcs are labeled with the functors of the term alphabet. The initial node is labeled $(\texttt{[]},\texttt{[]})$. Consider a node (w_M, v), and a functor F. Let $M' = M\texttt{[}v \leftarrow F(\Omega,\ldots,\Omega)\texttt{]}$. If $M'/u\uparrow\mathcal{R}_\Omega$ for some non-omega $u \in \mathcal{D}(M)$, let u be the minimum such occurrence, and $N = M'/u$. There are two cases, according to lemma 20. Either $N \in \mathcal{R}_\Omega$, in which case we grow an F-arc toward the corresponding success node. Or else, $N \prec \mathcal{R}_\Omega$, and with $w_N = w_M;F$ and $v = \delta(w_N)$ we grow an F-arc toward the node (w_N, v). Finally, if there is no such u, we grow an F-arc toward the failure node.

The only choice when growing the matching trie is the function δ. The essential requirement when choosing $v = \delta(w_N)$ above is that v should be a common direction for every compatible subterm of N. This requirement makes sense, since by construction all such subterms are aligned. If this requirement cannot be met, we have to backtrack on the choice of some earlier choice of δ.

If such a matching trie can be constructed, the system is progressive. Furthermore, an efficient matching algorithm exists. It is enough to remark that the linearization process permits us to represent efficiently $v = \delta(w_M)$, when $M \neq \Omega$, as follows. Let us decompose v as $v1\texttt{@[}k\texttt{]}$. The immediate father $v1$ of v corresponds to some position j in the string w_M, and thus we may represent v as the pair of integers (j,k). j is bounded by the size of a pattern, and k is bounded by the maximum arity of a functor.

We may now replace the matching trie by a finite automaton table. The states of the automaton are the various w_M constructed as nodes of the trie. The initial state 0 corresponds to the initial node and to the failure node. The final states correspond to the success nodes of the trie. For every state S, and every functor F we store the transition state in a table $Next(S,F)$. For every state $S \neq 0$ we also store the information $Index(S) = (j,k)$. The final algorithm uses a stack *Display* for traversing the term M. The pointer $Free$ holds the last non-visited node in M. *Display* holds the linearization of what we previously called P/u, as a list of addresses. We assume two auxiliary functions: $Mark$ which marks the currently visited node of M, and $Searchfree$ which finds the next non-visited node. We now get a fast deterministic evaluator:

Algorithm B.
Initialisation: $Free \leftarrow M$; $Display \leftarrow$ []; $S \leftarrow 0$;
Start global search: $P \leftarrow Free$;
Get next symbol: $F \leftarrow M(P)$; $Mark\ P$; $Display \leftarrow P \cdot Display$;
$S \leftarrow Next(S,F)$. By cases on S:

- $Success_i$: exit with answer "Redex at Q" with $Q = Display(|\mathcal{R}_i|)$.
- 0 : $Free \leftarrow Searchfree(Free)$. If this fails, exit with answer "Normal form", else Start global search.
- Otherwise : Let $(j,k) = Index(S)$ in $P \leftarrow Display(j)_k$; Get next symbol.

Our algorithm is the natural generalization to trees of the Knuth-Morris-Pratt string pattern-matching algorithm [16]. However, note that this algorithm has nothing to do with algorithm D of Hoffmann and O'Donnell [10], which also uses string pattern-matching for tree pattern-matching, but in a completely different way. Instead of matching linearizations of patterns, they rather match in parallel all maximal paths in patterns. Thus for pattern $F(A,B)$, they will not look for string $F;A;B$ but rather for the two strings $F;1;A$ and $F;2;B$. The Hoffmann-O'Donnell algorithm works for arbitrary sets of patterns. On the other hand, it cannot be used for finding an external redex.

Fast tree pattern-matching theorem. Let $\mathcal{R}$ be progressive. For every non-normal term M, the algorithm **B** will find a necessarily needed redex of M, in a time of order $|M|$.

The main remaining problem of the theory above is to find upper bounds for the number of states of the matching automaton, as a function of $|\mathcal{R}_\Omega|$.

Example. Let us consider the problematic system above. We leave it to the reader to construct a matching trie, and to extract from it the following matching automaton tables.

		Next					
S	*Index*	*F*	*G*	*H*	*A*	*B*	*C*
0		*2*	*1*	*0*	*0*	*0*	*0*
1	*(1,1)*	*3*	*1*	*0*	*0*	*0*	*0*
2	*(1,2)*	*2*	*1*	*0*	*0*	*4*	*0*
3	*(1,1)*	*5*	*1*	*0*	S_1	*0*	*0*
4	*(2,1)*	*6*	*1*	*0*	*0*	*0*	*0*
5	*(1,2)*	*2*	*1*	*7*	*0*	*4*	*0*
6	*(1,2)*	*2*	*1*	*9*	*0*	*4*	*0*
7	*(3,2)*	*2*	*1*	*0*	*0*	*8*	S_3
8	*(2,1)*	*2*	*1*	*0*	S_2	*0*	*0*
9	*(1,1)*	*2*	*1*	*0*	S_2'	*0*	*0*

In the transition table, S_1, S_2, S_2' and S_3 denote success states corresponding to the three patterns. S_2 and S_2' are distinguished, since in these two states the linearization of the patterns are distinct, and this information is needed for the reduction algorithm to perform efficiently, since the variables positions may be accessed directly in the *Display* stack.

Using the automaton above, the term $G_1(F_2(F_3(F_7(F_9(A, H_{10}(A_{11})), B_8), H_4(B_6)), B_5))$ is accessed according to the subscripts, and algorithm **B** stops with a redex found at the occurrence labeled 7.

Another interesting feature of algorithm B is that it can be easily adapted to the problem of building an interpreter for a given rewrite rules set. In order to be able to restart dynamically after one reduction, and look incrementally for the next redex, all we have to do is to store pairs $< P, S >$ in the display. After reduction of a pattern of size n, all we have to do is to pop n times the display, restore P and S, and continue at "Get next symbol". Note that this restarting facility obliges us to keep the whole display (of maximum length $|M|$), whereas if one is interested only in finding one redex, the display may be implemented in a circular ring of length the maximum size of a pattern. We remark that the total cost of the interpretation of a term is the total size of M and all the right-hand sides substituted when going to the normal form.

6.5.12 Increasing indexes

We now return to indexes, and are going to show that a regular system is necessary sequential iff it is progressive.

Definition. Let M be an Ω-term. We define its set of *increasing indexes* as:

$$\mathcal{J}(M) = \{u \in \mathcal{I}(M) \mid \forall N \;\; \omega(N) = \Omega \wedge \Omega(N) = N \Rightarrow \exists v \succeq u \;\, v \in \mathcal{I}(M[u \leftarrow N])\}.$$

Intuitively, an increasing index can be increased into an index whenever we substitute it with a potential redex. For instance, with $\mathcal{R}_1$ above, both omegas in $F(\Omega, \Omega)$ are indexes, but only the second one is increasing.

Lemma 22. Let $\mathcal{R}$ be necessarily sequential. Then for every normal Ω-term M we have $\mathcal{J}(M) \neq \emptyset$.

Lemma 23. $u@v \in \mathcal{J}(M) \Rightarrow v \in \mathcal{J}(M/u)$.

An increasing index may, by definition, be increased into an index. Actually, it may even be increased into an increasing index:

Lemma 24. Let N be such that $\omega(N) = \Omega$ and $\Omega(N) = N$. For every $u \in \mathcal{J}(M)$, there exists $v \succeq u$ such that $v \in \mathcal{J}(M[u \leftarrow N])$.

Theorem. Every necessarily sequential system is progressive.
Proof: lemmas 23 and 24 show that $\mathcal{J}$ fulfills the requirements for Δ.

Conversely, it is easy to show that that if a system is progressive it is necessarily sequential. Let M be any normal Ω-term which is not ω-maximum. We show that M possesses an index by processing it with algorithm **A** above, slightly modified so as to stop at "Get next symbol" on omegas. All assertions are valid as before, except that (a) reads now $P \preceq M$. The algorithm may not stop at normal form, since normal forms are ω-maximum. It may not stop at redex either, since M is normal. It must then stop with $w \in \mathcal{F}(M)$ satisfying $w \in \mathcal{I}(P)$, according to lemma 21. Thus $w \in \mathcal{I}(M)$.

6.6 Practical applications

The theory presented above is rather complicated. Here we present two simple subcases for which it is easy to construct the matching automaton tables.

6.6.1 Systems with constructors

We assume the functors are partitioned between parameters and constructors. Every pattern is of the form $F(C_1, \ldots, C_n)$, with F a parameter and the C_i's built only from constructors and variables. For a left linear system with constructors to be regular, we just have to check that no two distinct patterns are unifiable.

Lemma 25. A system with constructors is sequential iff the set $\mathcal{R}_\Omega$ is sequential.
Proof. We check that for every $M \preceq \mathcal{R}_\Omega$, $Dir_{\mathcal{R}}(M)$ satisfies the requirements for Δ.

6.6.2 Simple systems

Definition. Let $\mathcal{R}^*_\Omega = \{\lambda/u \mid \lambda \in \mathcal{R}_\Omega \wedge u \notin \mathcal{F}(\lambda)\}$. We say that $\mathcal{R}$ is a *sequential simple* system iff all subsets of $\mathcal{R}^*_\Omega$ are sequential sets.

Then the set of patterns of $\mathcal{R}$ is a simple forest in the sense of Hoffmann-O'Donnell [10]. That is, any two compatible Ω-terms M and N of $\mathcal{R}^*_\Omega$ are comparable, since the set $\{M, N\}$ is sequential.

Definition. We define $Dir^*(M) = Dir_S(M)$, where $S = \mathcal{R}^*_\Omega$. We then define by induction $\mathcal{I}^*(M)$ for every Ω-term M, with $\mathcal{I}^*(\Omega) = \{[\,]\}$, and $\mathcal{I}^*(M) = \{u@v \mid u \in Dir^*(\Omega(M)) \wedge v \in \mathcal{I}^*(M/u)\}$.

Lemma 26. Let $\omega(M) = \Omega$. Then $\mathcal{I}^*(M) \subseteq \mathcal{I}(M)$.

Lemma 27. Any sequential simple system is necessarily sequential.
Proof. We show that if $M \prec \mathcal{R}_\Omega$, we have $\mathcal{I}^*(M) \neq \emptyset$. Thus, $\mathcal{I}^*$ satisfies the requirements for Δ.

The matching trie, and in turn the automata tables, can be easily built up from $\mathcal{I}^*(M)$ by induction on the size of M. We refer the interested reader to [13] for the details, and more examples.

References

[1] H. Barendregt. "The Lambda-Calculus: Its Syntax and Semantics." North-Holland (1980).

[2] G. Berry. "Stable Models of Typed Lambda-calculi." Proc. 5th ICALP Conf., Udine (1978).

[3] G. Berry, J.J. Lévy. "Minimal and Optimal Computations of Recursive Programs." J. Assoc. Comp. Mach. **26,1** (1979) 148–175.

[4] H. B. Curry, R. Feys. "Combinatory Logic Vol. I." North-Holland, Amsterdam (1958).

[5] J. R. Hindley. "An abstract form of the Church-Rosser theorem, I." J. Symbolic Logic **34** (1969) 545–560.

[6] J. R. Hindley. "An abstract form of the Church-Rosser theorem, II: applications." J. Symbolic Logic **39** (1974) 1–21.

[7] J. R. Hindley. "Standard and Normal Reductions." Trans. Amer. Math. Soc. **240** (1978).

[8] J. R. Hindley. “Reductions of residuals are finite.” Trans. Amer. Math. Soc. **240** (1978) 345–361.

[9] J. R. Hindley, B. Lercher and J. P. Seldin. Introduction to Combinatory Logic.” Cambridge University Press, 1972.

[10] C. M. Hoffmann, M. J. O’Donnell. “Pattern matching in trees.” J. Assoc. Comp. Mach. **29,1** (1982) 68–95.

[11] C. M. Hoffmann, M. J. O’Donnell. “Programming with Equations.” ACM Transactions on Programming Languages and Systems, **4,1** (1982) 83–112.

[12] G. Huet. “Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems.” J. Assoc. Comp. Mach. **27,4** (1980) 797–821.

[13] G. Huet, J.J. Lévy. “Call by Need Computations in Non-Ambiguous Linear Term Rewriting Systems.” Rapport Laboria 359, IRIA (Aug. 1979).

[14] G. Kahn, G. Plotkin. “Domaines concrets .” Rapport Laboria 336, IRIA (Déc. 1978).

[15] J.W. Klop. “Combinatory Reduction Systems.” Ph. D. Thesis, Mathematisch Centrum Amsterdam (1980).

[16] D.E. Knuth, J. Morris, V. Pratt. “Fast Pattern Matching in Strings.” SIAM Journal on Computing **6,2** (1977) 323–350.

[17] J.J. Lévy. “Réductions correctes et optimales dans le λ-calcul.” Thèse d’Etat, Université de Paris VII (1978).

[18] B. K. Rosen. “Tree-Manipulating Systems and Church-Rosser theorems.” J. Assoc. Comp. Mach. **20,1** (1973) 160–187.

[19] V. Yu. Sazonov. “Functionals Computable in Series and in Parallel.” Siberian Math. Journal **17,3** (1976) 498–516.

[20] J. Staples. “Church-Rosser theorems for replacement systems.” in: Algebra and Logic (ed. J.N. Crossley) Lecture Notes in Mathematics **450** (1975) Springer-Verlag, 291–307.

Chapter 7

Categorical Logic

This chapter introduces the main notions of category theory in a formal and constructive manner, using a generalization of equational logic. The development of the theories of finite products and exponentiation gives a categorical presentation of intuitionistic propositional logic.

7.1 General motivation

7.1.1 An axiomatic view of rewriting

Let us review the inference rules for equality rewriting, as an axiomatization of derivations.

We assume given a functor alphabet Φ given with arity function α, in which we distinguish an operator $\rightarrow$ given with arity 2. We have no substitution inference rule, since it is implicit from the polymorphism of other rules. The replacement of equals for equals is decomposed into elementary steps of term replacement rules:

$$Id_A : \quad A \rightarrow A \qquad\qquad \mathit{Reflexivity}$$

$$; : \quad \frac{A \rightarrow B \quad B \rightarrow C}{A \rightarrow C} \qquad\qquad \mathit{Transitivity}$$

which specify that the rewriting arrow $\rightarrow$ is a quasi-ordering. Now we must state that $\rightarrow$ is compatible with the rest of the Φ-structure. That is, for every functor F in $\Phi - \{\rightarrow\}$ of arity n and for every $i \leq n$ we take a congruence rule:

$$Funct_F \; : \; \frac{A_1 \rightarrow B_1 \quad \cdots \quad A_n \rightarrow B_n}{F(A_1, \ldots, A_n) \rightarrow F(B_1, \ldots, B_n)} \qquad\qquad \mathit{Congruence}$$

The rewrite rules are added as supplementary axioms (i.e. nullary rules).

The terms which form the types of derivations possess a rich structure, and there is an interesting interplay between the structure of the derivation space and the term structure. Category theory is the right formalism in which to express this interplay. We saw earlier that certain properties of derivations (for instance, the parallel-moves lemma in the case of regular systems), had a nice categorical characterization. So we shall now recall a minimum of category theory in order to make precise these points.

7.1.2 The categorical viewpoint

This viewpoint gives a prominent role to the monoïd structure of the quasi-ordering $\rightarrow$. Simplifying the presentation, we may present a *category* as presented by a set of *objects* Obj, which we shall here confuse with the set of (closed) terms over some functor alphabet Φ, and by a set of *arrows* (or morphisms) which we shall here confuse with the set of (closed) proofs generated from some inference system Σ containing initially the two rules:

$$Id_A \ : \ A \to A \qquad \textit{Identity}$$

$$; \ : \ \frac{A \to B \quad B \to C}{A \to C} \qquad \textit{Composition}$$

Whenever $f : A \to B$, we say that arrow f has *domain* A and *codomain* B. Furthermore, it is specified that the proofs are quotiented by a congruence $=$ verifying:

$$Idl: \ Id; f \ = \ f$$

$$Idr: \ f; Id \ = \ f$$

$$Ass: \ (f;g);h \ = \ f;(g;h).$$

So we see that a category is a structure obtained as a hybrid of quasi-ordering and of monoïd, to which it reduces in the two degenerate cases (i.e. $|f : A \to B| \leq 1$ and $|Obj| = 1$). In the equations above, the types of f and g are kept implicit. A principal type compatible with the declaration of the inference rules can be inferred as usual, by unification.

If **A** and **B** are two categories, a functor F from **A** to **B** associates to every object A of **A** an object $F(A)$ of **B**, and to every arrow $f : A \to B$ an arrow $F(f) : F(A) \to F(B)$ such that the following functorial conditions hold:

$$F(Id) \ = \ Id$$

$$F(f;g) \ = \ F(f);F(g).$$

We see a great analogy between the notion of rewriting inference system and the main categorical notions. Actually, the categorical viewpoint is richer in that the functors have sorts themselves (i.e., the categories), and poorer in that they do not yet have arities (i.e. we just have monadic functors so far). In order to build-in arities we shall need products, and a full categorical account of minimal logic is obtained by a further adjunction, namely exponentiation. But we shall defer this explanation until later. We have given this elementary development of category theory essentially to justify our terminology. The congruence rule of term formation explains a functoriality condition on the object part, and the functoriality condition on the arrow part of the functor expresses the congruence property for rewriting.

Substitutivity in rewrite rules is expressed by defining them as natural transformations between the functors denoted by the two sides of the rule. That is, a *natural transformation* τ between functors F and G (both from category **A** to category **B**) is a mapping associating to every object A of **A** an arrow $\tau_A : F(A) \to G(A)$ such that

$$\tau_A ; G(f) \ = \ F(f) ; \tau_B.$$

And if we consider equations rather than simply rewrite rules, the symmetry inference rule is interpreted as the existence of inverses to arrows. Equations are thus defined as natural isomorphisms.

Category theory is explained in Mac Lane [12]. The categorical viewpoint for algebra has been developed by Lawvere and others [13]. Its application to proof theory is explained (in a somewhat complicated form) in Szabo [18].

Let us now give a closer look at a possible formalization of category theory along proof-theoretic lines.

7.2 The equational nature of category theory

Category theory reasoning proves equality of arrow compositions, as determined by diagrams. The corresponding equality is given in the model, i.e. in the category under consideration. But the proofs do not appeal to any particular property of the equality relation, such as extensionality. All we assume is that equality is a congruence with respect to the arrow operators.

However we are not dealing with simple homogeneous equational theories, but with typed theories. For instance, every arrow is equipped with its type $f : A \rightarrow B$. Here A and B are expressions denoting objects. These expressions are formed in turn by functorial operations and constants representing distinguished objects. The object terms can be considered untyped only within the context of one category. As soon as several categories are concerned, we must type the objects as well, with sorts representing categories. We thus have implicitly two levels of type structure.

The main difference between typed theories and untyped ones is that in untyped (homogeneous) theories one usually assume the domain of discourse to be non-empty. For instance, a first-order model has a non-empty carrier. Thus a variable always denotes something. In untyped theories one does not usually make this restriction. Thus we do not want to impose the Hom-set $A \rightarrow B$ to be always empty for every A and B in the category, in the same way that we want to consider partial orderings.

This has an unfortunate consequence: the law of substitution of equals for equals does not hold whenever one substitutes an expression containing a variable universally quantified over an empty domain by an expression not containing this variable, since we replace something that does not denote by something which may denote. For instance, with $F, G : 0 \rightarrow 2 = \{T, F\}$, we have for every $x \in \emptyset$ $F(x) = G(x)$, and yet we cannot conclude $T = F$. We shall have to keep this problem in mind in the following.

7.2.1 The general formalism

We have thus a formalism with four levels. At the first level, we have the alphabet of categories $\mathbf{Cat} = \{\mathbf{A}, \ldots, \mathbf{Z}\}$. At the second level, we have the alphabet of objects. Every category is defined over an object alphabet Φ of operators given with an arity. Φ is where the (internal) functors live. We then form *sequents* by pairs of terms $M \rightarrow N$, with $M, N \in \mathbf{T}(\Phi, V)$. V is a set of variables denoting arbitrary objects of the category. At the third level we have the alphabet Σ of arrows. An operator from Σ is given as an *inference rule* of the form:

$$S_1, \ldots, S_n \vdash S$$

where the S_i's and S are sequents. Such an operator is *polymorphic* over the free variables of the S_i's and S, which are supposed to be universally quantified over the inference rule. Such operators are familiar from logic, either as schematic inference rules, or as (definite) Horn clauses. Of course the arrows with domain M and codomain N are represented as terms over $\mathbf{T}(\Sigma, F)$ of type $M \rightarrow N$. Here F is a set of variable arrows, indexed by sequents $A \rightarrow B$. Finally, at the fourth level we have the *proofs* of arrow equalities. The alphabet consists of a set $\mathcal{R}$ of conditional rules of the form:

$$f_1 =_{S_1} g_1, \ldots, f_n =_{S_n} g_n \models f =_S g.$$

Here the f's and g are arrow expressions of type S, and similarly for the f_i's and g_i's. All object and arrow variables appearing in the rule are supposed to be universally quantified in front of the rule.

7.2.2 A simplified formalism

From now on, we shall assume that we are in one category of discourse which is left implicit. We shall therefore deal only with the last three levels. Furthermore, we shall assume that the only proof rules are:

$$Refl: \ f =_{A\to B} f$$

$$Trans: \ f =_{A\to B} g \ , \ g =_{A\to B} h \models f =_{A\to B} h$$

$$Sym: \quad f =_{A\to B} g \models f =_{A\to B} g$$

together with the rules stating that $=$ is a congruence with respect to the operators in Σ, all other rules being given by simple identities, i.e. by rules with an empty set of premisses ($n = 0$).

The further simplification comes from the realization that we are not really obliged to completely specify the types of all variable arrows and equalities, since there is a lot of redundancy. This fact exploits unification, and the following:

Meta-theorem. Let Σ be an arbitrary arrow signature, and E be an arbitrary term formed by operators from Σ and untyped variables. If there is an assignment of types to the variables of E that makes E well-typed with respect to Σ, there is a most general such assignment, independent in each variable, and furthermore the resulting type of E is most general. Here "more general" means "has as substitution instance". We call this assignment, together with the resulting type of E, the *principal* typing of E. More generally, for every type sequent S, if there is an assignment of types which makes E of a type some instance of S, there is a principal such assignment.

The meta-theorem above is most useful. It allows us to omit most of the types. When we write an equation $E = E'$, we shall implicitly refer to the principal typing giving E and E' the same type $A \to B$. So from now on, all equations in $\mathcal{R}$ are written without type, the types being implicit from the principality assumption.

7.2.3 The initial theory Categ

We are now ready to start Category Theory. The initial theory **Categ** has $\Phi = \emptyset$, and $\Sigma = \{Id, \_;\_\}$, given with respective signatures:

$$Id \ : \ A \to A$$

$$\_;\_ \ : A \to B \ , \ B \to C \ \vdash A \to C.$$

The notation $\_;\_$ means that we use the infix notation $f;g$ for the composition of arrows f and g. We can read f *then* g, and follow arrow composition along diagrams with semi-colon as concatenation of the labels. But since more people are accustomed to the standard set-theoretic composition notation, we shall below use $f \circ g$ as an abbreviation for $g; f$.

The equations $\mathcal{R}$ of **Categ** are simply the laws of a monoïd:

$$Ass \ : \ (f \circ g) \circ h \ = \ f \circ (g \circ h)$$

$$Idl \ : \ Id \circ f \ = \ f$$

$$Idr \ : \ f \circ Id \ = \ f$$

It is to be noted that the identification of the Hom-set symbol $\to$ with the sequent entailment arrow is not fortuitious. Actually Id and ; are well-known inference rules of intuitionistic propositional calculus. However, the logic is quite poor at this stage: we have no propositional connective whatsoever, just the basic mechanism for sequent composition, stating that entailment is reflexive and transitive. The rules of $\mathcal{R}$, considered as a left-to-right rewriting system, define a normal form on the sequent calculus proofs, i.e. on the arrow expressions.

Before we embark on more complicated theories, let us give a recipe on how to cook an equational presentation from a categorical statement.

7.2.4 What the category theorists don't say

Open a standard book on category theory, and consider a typical categorical definition. It usually reads: "Mumble, such that the following diagram commutes." Similarly, a typical categorical result states: "If $diagram_1$ and ... and $diagram_n$ commute, then $diagram$ commutes." The first step in understanding such statements is to determine exactly their universality: what is exactly quantified, universally or existentially, what depends on what, what are exactly the parameters of the (frequent) unicity condition. The next step is to realize that the diagram states conditional equalities on arrows, and that it is enough to state the equalities of the inside diagrams in order to get all equalities.

A uniform compilation of such statements as an equational theory proceeds as follows. First write completely explicitly the quantification prefix of the statement, in two lines, one for the objects and one for the arrows. Then Skolemize the statement independently in the first and the second line. That is, for every existentially quantified variable x following the universally quantified $y_1, \ldots y_n$, introduce a new n-ary operator X and substitute throughout x by $X(y_1, \ldots y_n)$. The Skolemization of the object variables determines Φ. The Skolemization of the second line, together with the types implicit from the diagram determine Σ. Finally, following arrows around the inner diagrams determines $\mathcal{R}$. This concerns the *existential* part. For the *unicity* part, proceed as follows. Let f be the arrow whose unicity is asserted. The existence part provided by Skolemization an $F(g_1, \ldots, g_k)$ in place of f. Write a supplementary arrow h on the diagram parallel to f, and use the commutation conditions to eliminate all the g_i's as $G_i(h)$. Add an extra equation $F(G_1(h), \ldots, G_k(h)) = h$.

Once we have convinced ourselves that the category theoretic statements and proofs are of an equational nature, we may ask: why do the category theorists use diagrams at all? The reason is that diagram chasing is a sophisticated way of doing complex equality reasoning, using several equations simultaneously, on a shared data structure (the graph underlying the diagram). So diagrammatic reasoning may be considered a good tool for high level equational reasoning. On the other hand, equality reasoning techniques such as rewrite rules analysis is good for mechanical implementation, and this is why we stress here the equational theories hidden behind the diagrams.

Remark. Let us finally remark that more general categorical concepts than the simple universal statements that we shall now consider may force us to generalize the basic formalism. For instance, more complicated limit constructions such as pullbacks force the dependence of objects on arrows. The Skolemization cannot be effected separately on the object and the arrow variables, and we shall have to place ourselves in a more complicated type theory with dependent types.

7.3 Products

7.3.1 The theory Prod

We shall apply the recipe above to the definition of *product* in a category. We recall that a category possesses a product if for all objects A, B, C there exists arrows fst, snd such that for every object D and every arrows f, g there exists a unique arrow h such that the following diagram commutes:

$$\begin{tikzcd}
A & C \arrow[l, "fst"'] \arrow[r, "snd"] & B \\
& D \arrow[ul, "f"] \arrow[u, "h"] \arrow[ur, "g"'] &
\end{tikzcd}$$

We now get the theory **Prod** by enriching **Categ** as follows. The Skolemization of C gives the binary functor $\times$, and we write with the infix notation $A \times B$ in place of C. So now $\Phi = \{\times\}$. Similarly, we add to Σ the following operators, issued respectively from fst, snd and h:

$$Fst \ : \ A \times B \rightarrow A$$

$$Snd \ : \ A \times B \rightarrow B$$

$$< \_, \_ > : \ D \rightarrow A \ , \ D \rightarrow B \vdash D \rightarrow A \times B$$

and we now have the usual diagram:

$$\begin{tikzcd}
A & A \times B \arrow[l, "Fst"'] \arrow[r, "Snd"] & B \\
& D \arrow[ul, "f"] \arrow[u, "< f, g >"] \arrow[ur, "g"'] &
\end{tikzcd}$$

The existence of h, that is the commutation of the two triangles, gives two new equations in $\mathcal{R}$:

$$\pi_1 \ : \ Fst \circ < f, g > = \ f$$

$$\pi_2 \ : \ Snd \circ < f, g > = \ g.$$

Unicity of h gives one last equation:

$$UniPair \ : < Fst \circ h \, , \, Snd \circ h > = \ h.$$

The arrow part of the functor $\times$ may be defined as a derived operator as follows:

$$\_ \times \_ \ : \ A \rightarrow B \ , \ C \rightarrow D \ \vdash \ A \times C \rightarrow B \times D$$

$$Def\times \ : \ f \times g \ = < f \circ Fst \, , \, g \circ Snd >$$

Remark. There is a possible source of confusion in our terminology. We talked about the elements of Φ as functors. Actually these are just function symbols denoting object constructors. Skolemization of a diagram will determine certain such function symbols, but there is no guarantee that there will be a corresponding functor. For instance, for product, we had to define the arrow part of $\times$ above, and to verify that indeed it obeys the functoriality laws.

7.3.2 The logical point of view

From the logical point of view, specifying a product amounts to defining conjunction. Read $A \times B$ as $A \wedge B$, and recognize Fst, Snd and $< \_, \_ >$ as respectively $\wedge$-elim-left, $\wedge$-elim-right, and $\wedge$-intro respectively [3, 6, 16].

The rules of $\mathcal{R}$ have a computational meaning: they specify how to reduce a proof to its normal form. Here we may apply known results from the theory of term rewriting systems, in order to complete $\mathcal{R}$ to a *canonical* system [7, 4].

The Knuth-Bendix completion procedure, when applied to theory **Prod**, generates two additional rewrite rules:

$$IdPair \;:< Fst, Snd > = \; Id$$

$$DistrPair \;:< f, g > \circ h \; = < f \circ h \, , \, g \circ h > .$$

The resulting system $\mathcal{R}$ is canonical, and can be used to decide the equality of arrows in the theory **Prod**. Of course, the equations above do not modify the theory, since they have been obtained by equational reasoning.

Finally, let us note that other presentations of the same theory are possible. For instance, we could have obtained product as the right adjoint of the diagonal functor. The unit of this adjunction is the *duplicator*, which may be defined here as:

$$D \; = < Id, Id > .$$

Note that type-checking imposes that the two identities are the same, so that $D_A : A \rightarrow A \times A$. As its name suggests, the duplicator duplicates, in the sense that we can prove $D \circ f \; = < f, f >$. The co-unit of the adjunction is the pair of projections (Fst, Snd).

7.3.3 Finite products

We say that a category admits all finite products if it admits products and a terminal object. Equationally, this amounts to enriching the theory **Prod** to a theory **Prods** by adding a constant 1 to Φ, a polymorphic constant $Nil \; : \; A \rightarrow 1$ to Σ, and a unicity equation $Uni1$ to $\mathcal{R}$:

$$Uni1 \; : \; h \; = \; Nil.$$

Note that this does not make the equational theory inconsistent: variable h above is principally typed to $A \rightarrow 1$. However this equation brings up two problems. The first one is the one mentioned in the beginning of these notes, since variable h appears on the left but not on the right of $Uni1$. The second problem is that $Uni1$ cannot be considered as a term rewriting rule in the usual sense, since it would rewrite Nil to itself and therefore does not satisfy the finite termination criterion. Note that $Uni1$ entails with the other equations two consequences:

$$Zero \; : \; Nil \circ f \; = \; Nil$$

$$Id1 \; : \; Id \; = \; Nil.$$

Again, $Id1$ does not identify every Id with every Nil, but only (restoring explicit types) Id_1 with Nil_1. Now it may be checked that $Uni1$ is actually a consequence of $Zero$ and $Id1$. The rule $Zero$ is a bona fide rewrite rule, which leaves the special equality $Id1$ to be dealt with in an ad hoc fashion.

Using operators $\times$ and 1 we may now construct n-tuples of objects, which we shall call *contexts*. 1 is the empty context, and if E is a context of length n and A an object term, $E \times A$ is a context of length $n+1$. We write $|E|$ for the length of an object list. If $\mathcal{C}$ is the current set of (representable) objects, i.e. $\mathbf{T}(\Phi, V)$, we denote by $\mathcal{C}^*$ the set of contexts.

If $1 \leq i \leq |E|$, we define the i-th component E_i of E recursively, as:

$$\begin{array}{lll} (E \times A)_i & = A & \text{if } i = 1 \\ & E_{i-1} & \text{if } i > 1. \end{array}$$

If E and E' are contexts, we define their concatenation $E@E'$ as a context recursively:

$$\begin{array}{lll} E@1 & = & E \\ E@(E' \times A) & = & (E@E') \times A. \end{array}$$

Similarly, using operators $< \_, \_ >$ and Nil we may construct lists, or n-tuples of arrows of same domain D. The empty arrow list is Nil, of length 0 and if $L : D \to E$ is an arrow list of length n then $< L, f >: D \to E \times A$ is an arrow list of length $n+1$, for every $f : D \to A$. Finally, for every object list and every n, with $1 \leq n \leq |E|$ we define recursively the projection arrow $\pi_E(n) \ : \ E \to E_n$, as:

$$\begin{array}{lll} \pi_E(n) & = Snd & \text{if } n = 1 \\ & \pi_{E'}(n-1) \circ Fst & \text{if } n > 1 \text{ and } E \ = \ E' \times A. \end{array}$$

7.4 CCC

7.4.1 The theory Exp

We obtain the theory **Exp** by enriching the theory **Prods** as follows. First, we add a binary operator $\Rightarrow$ to Φ. Next we add two operators to Σ, the constant App (application) and the unary operator $[]$ (abstraction):

$$App \ : \ (B \Rightarrow C) \times B \ \to \ C$$

$$[] \ : \ A \times B \to C \ \vdash \ A \to (B \Rightarrow C)$$

Finally, we add the following equations to $\mathcal{R}$:

$$ExAbs \ : \ App \circ ([]f \times Id) \ = \ f$$

$$UniAbs \ : \ [](App \circ (f \times Id)) \ = \ f$$

As before, this equational theory may be mechanically generated from the following diagram:

$$\begin{tikzcd} B \Rightarrow C & \times & B \arrow[r, "App"] & C \\ A \arrow[u, "{[]f}"] & \times & B \arrow[u, "Id"] \arrow[ur, "f"] & \end{tikzcd}$$

The logical point of view is here that $\Rightarrow$ is the (intuitionistic) implication. The operator App is $\Rightarrow$-introduction. It plays the rôle of the Modus Ponens inference rule (although here it is a constant, and not a binary operator). Abstraction is $\Rightarrow$-elimination, and plays somewhat the rôle of the deduction theorem.

Let us give a few equational consequences of the theory **Exp**:

$$IdExp \ : \ []App \ = \ Id$$

$$Red_1 \ : \ App \circ < []f \circ y \,,\, x > \ = \ f \circ < y \,,\, x >$$

$$Red \ : \ App \circ < []f \,,\, x > \ = \ f \circ < Id \,,\, x >$$

$$DistrAbs \ : \ []f \circ g \ = \ [](f \circ (g \times Id))$$

We can also show that abstraction is a bijection between the arrows of $A \times B \to C$ and those of $A \to B \Rightarrow C$, with inverse:

$$[]^{-1}f \ = \ App \circ < f \times Id > .$$

Thus we could have presented **Exp** in terms of $[]$ and $[]^{-1}$, and defined App as $[]^{-1}Id$. This corresponds to the fact that we could have rather axiomatized exponentiation by an adjunction to the product, whose co-unit is App (the unit being $[]Id$).

Finally, we define the arrow part of the functor $\Rightarrow$ (which is contravariant in its first argument) as:

$$f \Rightarrow g \ = \ [](g \circ App \circ (Id \times f)).$$

Sometimes $[]$ is called Curryfication, in honor of Curry. In fact there is an important relation between combinatory logic and CCC's, which we shall exhibit on λ-calculus in the next chapter.

References

[1] P. L. Curien. "Categorical Combinatory Logic." ICALP 85, Nafplion, Springer-Verlag LNCS 194 (1985).

[2] P. L. Curien. "Categorical Combinators, Sequential Algorithms and Functional Programming." Pitman (1986).

[3] G. Gentzen. "The Collected Papers of Gerhard Gentzen." Ed. E. Szabo, North-Holland, Amsterdam (1969).

[4] G. Huet. "Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems." J. Assoc. Comp. Mach. **27,4** (1980) 797–821.

[5] G. Huet. "Initiation à la Théorie des Catégories." Polycopié de cours de DEA, Université Paris VII (Nov. 1985).

[6] S.C. Kleene. "Introduction to Meta-mathematics." North Holland (1952).

[7] D. Knuth, P. Bendix. "Simple word problems in universal algebras". In: Computational Problems in Abstract Algebra, J. Leech Ed., Pergamon (1970) 263–297.

[8] C. Koymans. "Models of the lambda calculus." Information and Control **52,3** (1982) 306–332.

[9] J. Lambek. "From Lambda-calculus to Cartesian Closed Categories." in To H. B. Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism, Eds. J. P. Seldin and J. R. Hindley, Academic Press (1980).

[10] J. Lambek and P. J. Scott. "Introduction to Higher-Order Categorical Logic." To appear (1986).

[11] J. Lambek and P. J. Scott. "Aspects of Higher Order Categorical Logic." Contemporary Mathematics **30** (1984) 145–174.

[12] S. Mac Lane. "Categories for the Working Mathematician." Springer-Verlag (1971).

[13] E.G. Manes. "Algebraic Theories." Springer-Verlag (1976).

[14] C. Mann. "The Connection between Equivalence of Proofs and Cartesian Closed Categories." Proc. London Math. Soc. **31** (1975) 289–310.

[15] A. Poigné. "On Semantic Algebras ." Universitat Dortmund (March 1983).

[16] D. Prawitz. "Natural Deduction." Almqist and Wiskell, Stockolm (1965).

[17] D. Scott. "Relating Theories of the Lambda-Calculus." in To H. B. Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism, Eds. J. P. Seldin and J. R. Hindley, Academic Press (1980).

[18] M. E. Szabo. "Algebra of Proofs." North-Holland (1978).

Chapter 8

λ-calculus

This chapter gives an overview of λ-calculus, centered principally on the connection between typed λ-calculi, cartesian closed categories, and proof structures of natural deduction.

8.1 Combinatory Algebra

8.1.1 Proofs with variables: sequents

We first come back to the general theory of proof structures. We saw earlier that the Hilbert presentation of minimal logic was not very natural, in that the trivial theorem $A \Rightarrow A$ necessitated a complex proof $S\ K\ K$. The problem is that in practice one does not use just proof terms, but *deductions* of the form

$$\Gamma \ \vdash \ A$$

where Γ is a set of (hypothetic) propositions.

Deductions are exactly proof terms *with variables*. Naming these hypothesis variables and the proof term, we write:

$$\{...[x_i : A_i]... \mid i \leq n\} \ \vdash \ M : A$$

with $V(M) \subseteq \{x_1, ..., x_n\}$. Such formulas are called *sequents*. Since this point of view is not very well-known, let us emphasize this observation:

Sequents represent proof terms with variables.

Note that so far our notion of proof construction has not changed:

$\Gamma \vdash_\Sigma M : A$ iff $\vdash_{\Sigma \cup \Gamma} M : A$, i.e. the hypotheses from Γ are used as supplementary axioms, in the same way that in the very beginning we have defined $T(\Sigma, V)$ as $T(\Sigma \cup V)$.

8.1.2 The deduction theorem

This theorem, fundamental for doing proofs in practice, gives an equivalence between proof terms with variables and functional proof terms:

$$\Gamma \cup \{A\} \ \vdash \ B \quad \Leftrightarrow \quad \Gamma \ \vdash \ A \Rightarrow B$$

That is, in our notations:

a) $\Gamma \ \vdash \ M : A \Rightarrow B \quad \Rightarrow \quad \Gamma \cup \{x : A\} \ \vdash \ (M\ x) : B$

This direction is immediate, using App, i.e. Modus Ponens.
b) $\Gamma \cup \{x : A\} \vdash M : B \;\; \Rightarrow \;\; \Gamma \vdash [x]M : A \Rightarrow B$
where the term $[x]M$ is given by the following algorithm.

Schönfinkel's abstraction algorithm:

$$
\begin{array}{ll}
[x]x = I & (= S\ K\ K) \\
[x]M = K\ M & \text{if } M \text{ atom (variable or constant)} \neq x \\
[x](M\ N) = S\ [x]M\ [x]N &
\end{array}
$$

Note that this algorithm motivates the choice of combinators S and K (and optionally I). Again we stress a basic observation:

Schönfinkel's algorithm is the essence of the proof of the deduction theorem.

Now let us consider the rewriting system R defined by the rules:

$$Def_K : \ K\ x\ y \ = \ x,$$

$$Def_S : \ S\ x\ y\ z \ = \ ((x\ z)\ (y\ z)),$$

optionally supplemented by:

$$Def_I : \ I\ x \ = \ x$$

and let us write $\triangleright$ for the corresponding reduction relation.

Fact. $([x]M \quad N) \quad \triangleright^* \quad M[x \leftarrow N]$.

We leave the proof of this very important property to the reader. The important point is that the abstraction operation, together with the application operator and the reduction $\triangleright$, define a *substitution* machinery. We shall now use this idea more generally, in order to internalize the deduction theorem in a basic calculus of functionality. That is, we forget the specific combinators S and K, in favor of abstraction seen now as a new term constructor.

Remark 1. Other abstraction operations may be defined. For instance, the *strong* abstraction algorithm is more economical:

$$
\begin{array}{ll}
[x]x = I & \\
[x]M = K\ M & \text{if } x \text{ does not occur in } M \\
[x](M\ x) = M & \text{if } x \text{ does not occur in } M \\
[x](M\ N) = S\ [x]M\ [x]N & \text{otherwise.}
\end{array}
$$

Remark 2. The computation relation $\triangleright$ of combinatory algebra is confluent. Actually, it is defined by a particularly simple case of necessarily sequential rewrite rules. It is compatible with the term structure of combinatory algebra, and in particular with application. But it is not compatible with the derived operation of abstraction, and thus the ξ rule of λ-conversion is not valid. That is, combinatory computation simulates only weak β-reduction.

Similarly to λ-calculus, there are typed and untyped versions of combinatory algebra.

Other combinators than K, S and I have been considered. A general combinator is defined by a rewrite rule:

$$C\ x_1\ x_2\ \ldots\ x_n \rightarrow M,$$

where the left-hand side stands for the pattern $App(\cdots App(C, x_1)\cdots, x_n)$ and the right-hand side is an arbitrary term constructed from the x_i's and App.

A set of combinators is said to form a *basis* if it is sufficient to derive an abstraction algorithm (equivalently, if S and K are definable from the set). The state of the art about combinatory completeness is described in Statman [20].

8.2 Lambda-calculus

We now abandon the first-order term structures of combinatory algebra and turn to λ-calculi. We first consider *typed* λ-calculus, where the set of types $\mathcal{T}$ is defined as the set of terms constructed over some functor alphabet Φ containing the binary functor $\Rightarrow$. We write $\mathcal{T}^*$ for the set of finite sequences of types, with 1 the empty sequence and $E \times A$ the sequence obtained from sequence E by adding one more type A.

8.2.1 The (typed) λ-terms

We define recursively a relation $E \vdash M : A$, read "M is a *term* of *type* A in *context* E", where $A \in \mathcal{T}$ and $E \in \mathcal{T}^*$, as follows:

Variable : If $1 \leq n \leq |E|$ then $E \vdash n : E_n$
Abstraction : If $E \times A \vdash M : B$ then $E \vdash \texttt{[}A\texttt{]}M : A \Rightarrow B$
Application : If $E \vdash M : A \Rightarrow B$ and $E \vdash N : A$ then $E \vdash (M\ N) : B$

Thus a term may be a natural number, or may be of the form $\texttt{[}A\texttt{]}M$ with A a type and M a term, or may be of the form $(M\ N)$ with M, N two terms.

We thus obtain typed λ-terms with variables coded as de Bruijn's indexes [2], i.e. as integers denoting their reference depth (distance in the tree to their binder). This representation avoids all the renaming problems associated with actual names (α conversion), but we shall use such names whenever we give examples of terms. For instance, the term $\texttt{[}A\texttt{]}(1\ \texttt{[}B\texttt{]}(1\ 2))$ shall be presented under a concrete representation such as $\texttt{[}x : A\texttt{]}(x\ \texttt{[}y : B\texttt{]}(y\ x))$. In Church's original notation, the left bracket was a λ and the right bracket a dot, typing being indicated by superscripting, like: $\lambda x^A \cdot (x\ \lambda y^B \cdot (y\ x))$.

Note that the relation $E \vdash M : A$ is functional, in that A is uniquely determined from E and M. Thus the definition above may be interpreted as the recursive definition of a function $A = \tau_E(M)$.

The set $\mathcal{T}$ of types used in the λ-terms has been defined as all terms constructed from Φ containing $\Rightarrow$. The ordinary Curry-Church λ-calculus is obtained when $\Phi = \{\Rightarrow\} \cup \mathcal{T}_0$, where $\mathcal{T}_0$ is a finite set of atomic types, for instance $\{bool, int\}$. But we may include other functors in Φ. For instance, $\mathcal{T}$ may be taken as the set of objects in a cartesian closed category.

8.2.2 A translation from λ-terms to CCC arrows

We shall now show how to translate λ-terms to CCC arrows. More precisely, to every term M such that $E \vdash M : A$ we associate an arrow $F_E(M) : E \to A$ as follows:

$$\begin{aligned} F_E(n) &= \pi_E(n) \\ F_E(\texttt{[}A\texttt{]}M) &= \texttt{[]}F_{E\times A}(M) \\ F_E((M\ N)) &= App \circ < F_E(M)\,,\, F_E(N) > \end{aligned}$$

It can be easily proved by induction that $F_E(M)$ is a well-typed arrow expression.

Example. The closed term $M = [f : nat \Rightarrow nat][x : nat](f\ (f\ x))$ of type $A = (nat \Rightarrow nat) \Rightarrow (nat \Rightarrow nat)$ in the empty context $E = 1$, gets translated to:

$$F_E(M) = [][](App \circ < Snd \circ Fst\, ,\ App \circ < Snd \circ Fst\, ,\ Snd >>) : 1 \rightarrow A.$$

8.3 The syntactic theory of λ-terms

The advantage of the name-free terms is that we have no name conflict. The disadvantage is that we have to make explicit the relocation operations for terms containing free variables. For instance, let us define for every term M the term M^{+n} obtained in incrementing its free variables by n. Let $M^{+n} = R_n^0(M)$, with:

$$\begin{array}{lll} R_n^i(k) & = k & \text{if } k \leq i \\ & \ \ k+n & \text{if } k > i \\ R_n^i([A]M) & = & [A]R_n^{i+1}(M) \\ R_n^i((M\ N)) & = & (R_n^i(M)\ R_n^i(N)). \end{array}$$

The reader will check that $E \vdash M : A$ iff $E@E' \vdash M^{+n} : A$, where E' is an arbitrary context of length n.

We now define *substitution* to free variables. Let $E \times A \vdash M : B$, and $E \vdash N : A$. We shall define a term $M\{N\}$, and show that $E \vdash M\{N\} : B$. First we define recursively:

$$\begin{array}{lll} \Sigma_N^n(k) & = k & \text{if } k < n \\ & \ \ N^{+n} & \text{if } k = n \\ & \ \ k-1 & \text{if } k > n \\ \Sigma_N^n([A]M) & = & [A]\Sigma_N^{n+1}(M) \\ \Sigma_N^n((M\ N)) & = & (\Sigma_N^n(M)\ \Sigma_N^n(N)). \end{array}$$

It is easy to show that substitution preserves the types, in the sense that $(E \times A)@E' \vdash M : B$ and $E \vdash N : A$ implies $E@E' \vdash \Sigma_N^n(M) : B$, with $n = |E'|$.

Now we define $M\{N\} = \Sigma_N^0(M)$, and we get that $\tau_E(M\{N\}) = \tau_{E \times A}(M)$, with $A = \tau_E(N)$.

We are now ready to define the *computation* relation $\triangleright$ as follows:

$$([A]M\ N) \triangleright M\{N\} \qquad (\beta)$$

$$M \triangleright M' \Longrightarrow [A]M \triangleright [A]M' \qquad (\xi)$$

$$M \triangleright M' \Longrightarrow (M\ N) \triangleright (M'\ N)$$

$$M \triangleright M' \Longrightarrow (N\ M) \triangleright (N\ M').$$

It is clear that computation preserves the types of terms. But it also preserves their values, in the sense of the translation to CCC arrows: If $E \vdash M : A$ and $M \triangleright N$, then $F_E(M) = F_E(N)$ in the theory **Exp**, as we shall show.

The computation relation presented above is traditionally called (strong) β-reduction. It is confluent and nœtherian (because of the types!), and thus every term possesses a canonical form, obtainable by iterating computation non-deterministically. Another valid conversion rule is η-conversion:

$$[x : A](M\ x) = M \qquad (\eta)$$

whenever x does not appear in M. Let us show that it corresponds to $UniAbs$, using our translation above.

First we define the *relocation* combinators $\rho(i)$ as follows.

$$\begin{aligned}\rho(0) &= Fst\\ \rho(i+1) &= \rho(i)\times Id\end{aligned}$$

It is easy to show that (with appropriate types):

$$\pi(k)\circ\rho(i) = \begin{array}{ll}\pi(k) & \text{if } k\leq i\\ \pi(k+1) & \text{if } k>i\end{array}$$

and thus that $R_1^i(M) = M\circ\rho(i)$. As a particular case we get $M^{+1} = M\circ Fst$ and thus we can read the law $UniAbs$ as $[x](M^{+1}\ x) = M$. Whenever x does not occur in M the expressions M and M^{+1} are concretely identical, and we obtain the η conversion rule. Note however that $UniAbs$ is an algebraic law, whereas η makes sense only relatively to concrete representations.

We are now going to show that Red validates the β-reduction rule. First we define the *substitution* combinators as follows.

$$\begin{aligned}\sigma_N(0) &= <Id, N>\\ \sigma_N(n+1) &= \sigma_N(n)\times Id\end{aligned}$$

Next we check that for every λ-terms M,N and every integer n the following equation is provable in **Exp** (confusing M with $F(M)$, and assuming types are correct):

$$\Sigma_N^n(M) = M\circ\sigma_N(n).$$

This suggests defining in **Exp** the derived operator:

$$\_\{\_\} : A\times B\rightarrow C\ ,\ A\rightarrow B\ \vdash A\rightarrow C$$

with defining equation:

$$Subst\ :\ f\{x\} = f\circ <Id, x>$$

and now the rule Red reads:

$$App\circ <[]f, x> = f\{x\}$$

which clearly validates the computation relation $\triangleright$.

8.4 λ-calculus variations

8.4.1 Weak reduction

There are many variations on λ-calculus. What we have just presented is typed λ-calculus, with Curry-Church types. The notion of computation $\triangleright$ is strong β reduction. It is also interesting to consider a weak reduction, obtained by not allowing rule ξ above. Thus, weak reduction is not compatible with the abstraction operator []. As we have already seen, λ-calculus may be translated into combinatory algebra, but the natural computation rule associated with the set of combinator definitions seen as term rewriting system corresponds then to weak reduction, *not* strong reduction.

8.4.2 Pure λ-calculus

If we remove the types, we get the theory of pure λ-calculus. The set of pure lambda terms is defined as:

$$\lambda = \bigcup_{n \geq 0} \lambda_n$$

where the set λ_n of λ-terms with n potential free variables is defined inductively by:

- $i \in \lambda_n$ if $1 \leq i \leq n$
- $\texttt{[]}M \in \lambda_n$ if $M \in \lambda_{n+1}$
- $(M\ N) \in \lambda_n$ if $M, N \in \lambda_n$

As we did previously, we get readable concrete syntax by sticking variable names in the brackets, as in $\texttt{[}x\texttt{]}x$. The terms in λ_0 are the *closed* pure λ-terms. Analogous untyped versions of the rules above define analogous computation rules. Sometimes syntactic properties are easier to prove in pure λ-calculus. For instance, the confluence property in typed calculi is an easy consequence of the corresponding property in the pure calculus, if we remark that computation preserves typing. The classical method, due to Tait and Martin-Löf [1], consists in proving that the relation $\trianglerighteq$ is strongly confluent, with $\trianglerighteq$ defined as the reflexive and compatible closure of:

$$\frac{M \trianglerighteq M' \quad N \trianglerighteq N'}{(\texttt{[]}M\ N) \trianglerighteq M'\{N'\}}.$$

It is easy to check that indeed $\trianglerighteq$ and $\triangleright$ have the same reflexive-transitive closure, whence the result. As we saw for regular term rewriting system, such a "parallel moves" theorem is actually much stronger than strong confluence, since it corresponds to the existence of pushouts in an appropriate category of computations. The theory of λ-calculus derivations is worked out in detail in J.J. Lévy's thesis [15, 16]. Note that contrarily to the theory of regular term rewriting systems, the parallel reduction $\trianglerighteq$ is not limited to parallel disjoint redexes, since in λ-calculus residuals of a redex may not be disjoint. For instance, consider $(\texttt{[}u\texttt{]}(u\ u)\ \ \texttt{[}v\texttt{]}(\texttt{[}x\texttt{]}v\ y))$.

The theory of β-η-reduction is rather complicated. Actually, note that there is a critical pair between the two rules, since $(\texttt{[}x\texttt{]}(M\ x)\ N)$ contains conflicting redexes for the two rules. Fortunately, the two rules reduce to the same term $(M\ N)$. However, the two rules are usually dealt with separately, since it can be showed that η conversions can be postponed after β reductions. In the following, we write $\triangleright$ for the β-reduction rule, and $\cong$ for its associated congruence. The theory of β-reduction is similar to the theory of regular term rewriting systems. Certain results are simpler. For instance, the standardization theorem has a simpler form, since the standard derivation always reduces the leftmost needed redex. Others are more complicated, due to the residual embedding noted earlier.

Certain theorems are identical for the pure calculus as for the typed case. Other aspects of pure λ-calculus differ from the typed version. In the pure calculus, some terms do not always admit normal forms. For instance, with $\Delta\ =\ \texttt{[}u\texttt{]}(u\ u)$ and $\perp\ =\ (\Delta\ \Delta)$, we get $\perp\ \triangleright \perp\ \triangleright \ldots$ A more interesting example is given by

$$Y\ =\ \texttt{[}f\texttt{]}(\texttt{[}u\texttt{]}(f\ (u\ u))\ \ \texttt{[}u\texttt{]}(f\ (u\ u)))$$

since $(Y\ M)\ \cong\ (M\ (Y\ M))$ shows that Y defines a general fixpoint operator. Y is called the *Curry* fixpoint operator. Other fixpoint operators are known. For instance, the *Turing* fixpoint operator is defined as:

$$\Theta\ =\ (\texttt{[}x\texttt{]}\texttt{[}y\texttt{]}(y\ (x\ x\ y))\ \ \texttt{[}x\texttt{]}\texttt{[}y\texttt{]}(y\ (x\ x\ y))),$$

and it verifies the stronger property that for every M we have $(\Theta\ M) \ \trianglerighteq^* \ (M\ (\Theta\ M))$.

Exercise. Show that $\Phi = [\varphi][f](f\ (\varphi\ f))$ is a generator of fixpoints, in that M is a fixpoint combinator iff $\Phi(M) \cong M$.

The existence of fixpoint operators, and the easy encoding of arithmetic notions in pure λ-calculus, make it a computationally complete formalism: all partial recursive functions are definable. We shall not develop further this aspect of λ-calculus, but we just remark that it entails the undecidability of most syntactic properties. Thus $\cong$ is an undecidable relation, and it is generally undecidable whether a given term is normalisable or not.

What we are mostly concerned here is the application of λ-calculus to logic. And one may worry about the interpretation of fixpoints of propositional connectives such as negation. The next section shows that indeed pure λ-calculus is logically problematic.

8.4.3 Curry's version of Russell's paradox

Our framework is minimal logic, with propositions represented as pure λ-expressions. That is, we assume that $\Rightarrow$ is a constant of the calculus. We assume that we have as rules of inference:

$$A \Rightarrow B\ ,\ A\ \vdash\ B \tag{App}$$

$$\vdash A \Rightarrow A \tag{Id}$$

$$\vdash (A \Rightarrow (A \Rightarrow B)) \Rightarrow (A \Rightarrow B) \tag{W}$$

It is easy to see that (W) is valid in minimal logic (consider $[u : A \Rightarrow (A \Rightarrow B)][v : A](u\ v\ v)$). Now consider an arbitrary proposition X. Let us define $N = [A]A \Rightarrow X$, and let $M = (Y\ N)$. N is in a way the minimal meaning for negation, and M is a fixpoint of it. That is:

$$M \cong (M \Rightarrow X). \tag{$*$}$$

Now we get $M \Rightarrow M$ from Id_M, and thus $M \Rightarrow (M \Rightarrow X)$ by $(*)$. Using App and D we infer $M \Rightarrow X$, and thus M using $(*)$ in the reverse direction. A final use of App yields X, which is an arbitrary proposition, and thus the logic is inconsistent [6].

Thus combinatory completeness of the pure λ-calculus at the level of propositions is not compatible with the logical completeness issued from the typed λ-calculus at the level of proofs.

Half way between the typed and the pure calculus we find typed calculi where additional constants and reduction rules have been added. For instance, it is possible to add typed recursion operators in order to develop recursive arithmetic in a sound way [21]. Let us now consider a particularly important typed calculus.

8.4.4 λ-calculus with products

CCC arrows are richer than λ-terms. This suggests enriching λ-calculus with further operators $fst, snd, pair, nil$ with appropriate supplementary reduction rules, and to allow "varstruct" bindings in the concrete syntax in order to have variables correspond to arbitrary sequences of Fst and Snd, as opposed to just integers coded up in unary notation. For instance, ML (without recursion) can be translated into CCC arrows by a simple extension of the translation F above.

Let us formalize this idea. We now consider a λ-calculus with types formed with $\times$,$\Rightarrow$, and 1. The rules of formation of the calculus are as follows. We define recursively a relation $E \vdash M : A$, read "M is a *term* of *type* A in *context* E", where $A \in \mathcal{T}$ and $E \in \mathcal{T}^*$, as follows:

Variable :	If $1 \leq n \leq \lvert E\rvert$ then $E \vdash n : E_n$
Abstraction :	If $E \times A \vdash M : B$ then $E \vdash [A]M : A \Rightarrow B$
Application :	If $E \vdash M : A \Rightarrow B$ and $E \vdash N : A$ then $E \vdash (M\ N) : B$
Pairing :	If $E \vdash M : A$ and $E \vdash N : B$ then $E \vdash M, N : A \times B$
Projl :	If $E \vdash M : A \times B$ then $E \vdash fst(M) : A$
Projr :	If $E \vdash M : A \times B$ then $E \vdash snd(M) : B$
Nil :	$E \vdash () : 1$

The computation relation extends the one given above, with supplementary rules:

$$fst(M, N) \ \triangleright M$$

$$snd(M, N) \ \triangleright N$$

$$fst(M), snd(M) \ \triangleright M \tag{SP}$$

$$M \ \triangleright () \ \text{ if } \ M : 1 \ \text{ and } \ M \neq ()$$

$$M \ \triangleright M' \Longrightarrow M, N \ \triangleright M', N$$

$$N \ \triangleright N' \Longrightarrow M, N \ \triangleright M, N'$$

$$M \ \triangleright M' \Longrightarrow fst(M) \ \triangleright fst(M')$$

$$M \ \triangleright M' \Longrightarrow snd(M) \ \triangleright snd(M')$$

$$[x : A](M\ x) \ \triangleright M \tag{\eta}$$

This computation relation is confluent [17] and Nœtherian [10].

Let us now consider the set $\lambda_1^\times$ of all such λ-terms defined in a context of length 1, i.e. with at most one free variable. We abbreviate $1 \times A \vdash M : B$ as $M : A \to B$. We associate with every $M : A \to B \in \lambda_1^\times$ an **Exp** arrow $\overline{M} : A \to B$ as follows. We use u for the name of the free variable of M, in order to avoid complex de Bruijn indexes.

$M = u$	$\overline{M} = Id$
$M = [B]N$	$\overline{M} = []P$ where $P = \overline{N\{1 \leftarrow snd(u)\}\{2 \leftarrow fst(u)\}}$
$M = (M_1\ M_2)$	$\overline{M} = App \circ < \overline{M_1}, \overline{M_2} >$
$M = M_1, M_2$	$\overline{M} =< \overline{M_1}, \overline{M_2} >$
$M = fst(N)$	$\overline{M} = Fst \circ \overline{N}$
$M = snd(N)$	$\overline{M} = Snd \circ \overline{N}$
$M = ()$	$\overline{M} = Nil.$

This definition is well-founded, using a weight function ξ such that $\xi(k) = 0$ for k a variable, $\xi(fst(M)) = 0$ if $\xi(M) = 0$ and similarly for snd.

Exercise. Show that the congruence classes of $\lambda_1^\times$ with respect to $\triangleright$ form a CCC.

Conversely, we define an inverse translation from a ground arrow term $M : A \to B$ in **Exp** to a term $\underline{M} : A \to B$ in $\lambda_1^\times$, as follows.

$M = Id$	$\underline{M} = u$
$M = M_1 \circ M_2$	$\underline{M} = \underline{M_1}\{\underline{M_2}\}$
$M = Fst$	$\underline{M} = fst(u)$
$M = Snd$	$\underline{M} = snd(u)$
$M = < M_1, M_2 >$	$\underline{M} = \underline{M_1}, \underline{M_2}$
$M = Nil$	$\underline{M} = nil$
$M = [\,]N$	$\underline{M} = [\,](P\{u \leftarrow (2,1)\})$ where $P = \underline{N}$
$M = App$	$\underline{M} = (fst(u)\ snd(u))$.

Lemma. For every term $M \in \lambda_1^\times$, we have $\underline{(\overline{M})} \cong M$, where $\cong$ is the equivalence generated by $\triangleright$.

Corollary 1. $\lambda_1^\times / \cong$ is isomorphic to the free CCC.

Corollary 2. The theory **Exp** is decidable.
Proof. **Exp** $\vdash M = N$ iff $\underline{M}$ and $\underline{N}$ have the same $\triangleright$-normal form.

This method is due to Lambek and Scott [14]. The decidability of the word problem for free CCC's was first shown by more complicated methods in Szabo [22].

8.4.5 Other sub-theories of CCC's

The **Exp** theory is decidable, as we saw. Unfortunately, no canonical system is known for the full theory. An interesting sub-theory is obtained by restricting $\mathcal{R}$ to the set

$$\mathcal{R}_1 \ = \ \{Ass, Idl, Idr, \pi_1, \pi_2, DistrPair, IdPair, UniPair, Red, DistrAbs\}.$$

Considered as a (typed) term-rewriting system, $\mathcal{R}_1$ is locally confluent. This can be mechanically checked by the Knuth-Bendix decision procedure. Note that this is possible because a system such as $\mathcal{R}_1$ may be considered as an (untyped) equational theory in the ordinary sense, by mixing the arrow structure and the object structure as follows: every arrow sub-term M of type $A \to B$ is represented as $T(M, A, B)$, where T is a special ternary function symbol. Note that variables in the type subparts get instantiated by matching and unification, in the same way as the variables in the arrow subparts. This supports our view of the polymorphic nature of the categorical combinators.

The system $\mathcal{R}_1$ is nœtherian, and thus confluent. It would be interesting to have the termination argument formulated so as to show that the rewriting relation is a simplification ordering. However, note that this argument cannot work on the Σ-trees alone, since the corresponding untyped system is strong enough to code the β-reductions of untyped λ-calculus (and furthermore Klop has shown that λ-calculus with surjective pairing is not Church-Rosser [12]).

Another interesting locally confluent subtheory is:

$$\mathcal{R}_2 \ = \ \{Ass, Idl, Idr, \pi_1, \pi_2, DistrPair, IdPair, UniPair, Red, Red_1\}.$$

The state of the art in the theory of categorical combinatory algebra and its relations with various λ-calculi can be found in Curien's extensive monography [5].

8.4.6 Extensions of the CCC theory

It is also possible to enrich the type structure with sums, corresponding to categorical co-product. The **Exp** theory is enriched with injections and a conditional operator. The corresponding models are the bi-cartesian closed categories.

It is also possible to axiomatize the existence of a natural number object, in order to get primitive recursive functions. We may even axiomatize a general fixpoint operator.

Finally, we may postulate the existence of a universal object U and build the pure λ-calculus in the manner of Scott, as described in [19]. That is, we postulate a retract pair between U and $U \rightarrow U$:

$$Quote : (U \Rightarrow U) \rightarrow U$$

$$Eval : U \rightarrow (U \Rightarrow U)$$

verifying:

$$Retract : Eval \circ Quote = Id.$$

Let us call **Univ** the theory obtained by the corresponding enrichment of **Exp**.

We may now translate any $M \in \lambda_n$ as an arrow $A_n(M) : U^n \rightarrow U$ as follows:

$$A_n(k) \;=\; \pi_{U^n}(k)$$

$$A_n([\,]M) \;=\; Quote \circ [\,]A_{n+1}(M)$$

$$A_n((M\ N)) \;=\; App \circ < Eval \circ A_n(M)\,,\, A_n(N) > .$$

We leave it to the reader to check that the β rule is still an equational consequence of **Univ**. However, note that the η rule is not valid anymore, since it would entail that *Eval* and *Quote* define an isomorphism between U and $U \Rightarrow U$.

References

[1] H. Barendregt. “The Lambda-Calculus: Its Syntax and Semantics.” North-Holland (1980).

[2] N.G. de Bruijn. “Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem.” Indag. Math. **34,5** (1972), 381–392.

[3] G. Cousineau, P.L. Curien and M. Mauny. “The Categorical Abstract Machine.” In Functional Programming Languages and Computer Architecture, Ed. J. P. Jouannaud, Springer-Verlag LNCS 201 (1985) 50–64.

[4] Th. Coquand, G. Huet. “Constructions: A Higher Order Proof System for Mechanizing Mathematics.” EUROCAL85, Linz, Springer-Verlag LNCS 203 (1985).

[5] P. L. Curien. “Categorical Combinators, Sequential Algorithms and Functional Programming.” Pitman (1986).

[6] H. B. Curry, R. Feys. “Combinatory Logic Vol. I.” North-Holland, Amsterdam (1958).

[7] G. Gentzen. “The Collected Papers of Gerhard Gentzen.” Ed. E. Szabo, North-Holland, Amsterdam (1969).

[8] M. J. Gordon, A. J. Milner, C. P. Wadsworth. “Edinburgh LCF” Springer-Verlag LNCS **78** (1979).

[9] L.S. van Benthem Jutting. “The language theory of Λ_∞, a typed λ-calculus where terms are types.” Private communication (1984).

[10] M. Karr. "Delayability in Proofs of Strong Normalizability in the Typed Lambda Calculus." Private communication (1984).

[11] S.C. Kleene. "Introduction to Meta-mathematics." North Holland (1952).

[12] J.W. Klop. "Combinatory Reduction Systems." Ph. D. Thesis, Mathematisch Centrum Amsterdam (1980).

[13] D. Knuth, P. Bendix. "Simple word problems in universal algebras". In: Computational Problems in Abstract Algebra, J. Leech Ed., Pergamon (1970) 263–297.

[14] J. Lambek and P. J. Scott. "Aspects of Higher Order Categorical Logic." Contemporary Mathematics **30** (1984) 145–174.

[15] J.J. Lévy. "Réductions correctes et optimales dans le λ-calcul." Thèse d'Etat, U. Paris VII (1978).

[16] J.J. Lévy. "Optimal Reductions in the λ-calculus." in To H. B. Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism, Eds. J. P. Seldin and J. R. Hindley, Academic Press (1980).

[17] G. Pottinger. "The Church-Rosser Theorem for the Typed λ-calculus with Extensional Pairing." Carnegie Mellon University technical report (March 1979).

[18] D. Prawitz. "Natural Deduction." Almqist and Wiskell, Stockolm (1965).

[19] D. Scott. "Relating Theories of the Lambda-Calculus." in To H. B. Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism, Eds. J. P. Seldin and J. R. Hindley, Academic Press (1980).

[20] R. Statman. "On translating λ-terms into combinators; the basis problem." IEEE Conference on Logic in Computer Science, Boston, June 1986.

[21] S. Stenlund. "Combinators λ-terms, and proof theory." Reidel (1972).

[22] M. E. Szabo. "Algebra of Proofs." North-Holland (1978).

Chapter 9

Propositional Natural Deduction

9.1 Proof theoretic methods

Let us give a short interlude on the general methodology we have been following. Our approach has been to classify logical systems according to the formal structures underlying propositions and proofs. We used analogies drawn from computer science (types and programs) and category theory (objects and arrows). We identified three levels of generality of hierarchical structures:

- terms
- terms with variables
- λ-terms.

For instance, concerning proofs, we get at the three levels:

- Hilbert systems
- Sequent formulations
- Natural deduction, as seen below.

The point of view of computation, by rewriting proofs, is the key to showing many metamathematical properties. In proof theory, a redex schema is called a *cut*. In natural deduction, a cut takes a form of the composition of an introduction rule for some connective with the elimination rule for the same connective. For instance, in minimal logic, this corresponds to the β redex $(\texttt{[}A\texttt{]}M\ N)$. *Cut elimination* consists in showing that this proof rewriting relation is nœtherian. The corresponding λ-calculus property is called strong normalization.

Strong normalization shows that every proof is equivalent to a normal proof, having the structure of a normal λ-term. This proves the equivalence of the original logical system with a simpler system, whose inference rules reflect the construction rules of normal terms. This simpler system is usually useless for actually carrying out deductions. In particular, proofs in the normal system may be much larger than the proofs in the original systems. That is, every lemma is expanded out like a macro, leading to a completely explicit reasoning sequence. However, the simplicity of the normal system makes more apparent certain of its properties.

For instance, in many cases the normal system has the *subformula property*: for every inference rule, every antecedent formula is a subformula of some consequent formula. Such systems are immediately seen to be logically consistent. This was the original application of Gentzen's techniques. Of course, according to Gödel's theorem, this does not establish *absolute* consistency of the logic, but relativizes it to a carefully identified troublesome point, the proof of termination of some reduction relation. This has the additional advantage in providing a hierarchy of strength of inference systems, classified according to the ordinal necessary to consider for the termination proof.

Finally, in the cases where the notion of subformula is such that any formula has only a finite number of subformulas, the subformula property yields directly the decidability of the corresponding system. This permits to give an immediate proof of decidability of intuitionistic propositional calculus.

9.2 Gentzen's systems

Gentzen considered two kinds of systems: the N systems of *natural deduction* and the L systems of *sequent calculus*. Both systems come in two flavors: the intuitionistic systems, and the classical systems (the latter beeing denoted by N_K and L_K respectively). These calculi are defined by inference rules operating not simply on propositions, but rather on *judgements* of the form $\Gamma \vdash A$, for the intuitionistic systems, and of the form $\Gamma \vdash \Delta$, for the classical systems. Here Γ and Δ denote finite *sets* of propositions, and A denotes a proposition. These judgements may be considered as types of deductions, i.e. proofs from hypotheses, i.e. terms with a set of free variables.

9.2.1 The N system

The role of variables is taken by the base sequents:

$$Axiom \ : \ \frac{}{\{A\} \ \vdash \ A}$$

together with the structural *thinning* rule:

$$Thinning \ : \ \frac{\Gamma \ \vdash B}{\Gamma \cup \{A\} \ \vdash B}$$

which expresses that a proof may not use all of the hypotheses. Gentzen's remaining rules give types to proofs according to propositions built as functor terms, each functor corresponding to a propositional connective. The main idea of his system is that inference rules should not be arbitrary, but should follow the functor structure, in explaining in a uniform fashion how to *introduce* a functor, and how to *eliminate* it.

For instance, minimal logic is obtained with $\Phi = \{\Rightarrow\}$, and the rules of $\Rightarrow$ *intro* and $\Rightarrow$ *elim*, that is:

$$\Rightarrow intro \ : \ \frac{\Gamma \cup \{A\} \ \vdash \ B}{\Gamma \ \vdash \ A \Rightarrow B}$$

$$\Rightarrow elim \ : \ \frac{\Gamma \ \vdash \ A \Rightarrow B \qquad \Delta \ \vdash A}{\Gamma \cup \Delta \ \vdash \ B}.$$

Conjunction is obtained by adding:

$$\wedge \ intro \ : \ \frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma \cup \Delta \ \vdash \ A \wedge B}$$

$$\wedge \ elim \ left \ : \ \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}$$

$$\wedge\ \textit{elim right} \quad : \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}.$$

We remark that the rule of thinning may then be dispensed with, since it may be derived from the $\wedge$ and *Axiom* rules. One gets *positive logic* by adding the rules for disjunction:

$$\vee\ \textit{intro left} \quad : \quad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}$$

$$\vee\ \textit{intro right} \quad : \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

$$\vee\ \textit{elim} \quad : \quad \frac{\Gamma \vdash A \vee B \qquad \Delta \cup \{A\} \vdash C \qquad \Theta \cup \{B\} \vdash C}{\Gamma \cup \Delta \cup \Theta \vdash C}.$$

Finally, full (intuitionistic) propositional logic is obtained by adding negation:

$$\neg\ \textit{intro} \quad : \quad \frac{\Gamma \cup \{A\} \vdash B \qquad \Delta \cup \{A\} \vdash \neg B}{\Gamma \cup \Delta \vdash \neg A}$$

$$\neg\ \textit{elim} \quad : \quad \frac{\Gamma \vdash A \qquad \Delta \vdash \neg A}{\Gamma \cup \Delta \vdash B}.$$

Alternatively, we could introduce negation by definition, adding a constant $\perp$ for falsity, defining $\neg A$ as $A \Rightarrow \perp$, and adding a rule:

$$\perp\ \textit{elim} \quad : \quad \frac{\Gamma \vdash \perp}{\Gamma \vdash A}.$$

The classical system N_K is obtained from N by replacing the $\neg$ *elim* rule by:

$$\textit{Classical} \quad : \quad \frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A},$$

which expresses that classical proofs are proofs by contradiction (this is the well known reduction of classical logic to intuitionistic logic by double negation).

Such propositional systems can be extended to first-order logic by giving appropriate rules for quantifiers. Further extensions yield arithmetic, set theory, etc... However we do not have yet the formal apparatus to discuss dependent types, and thus we abstain from presenting these rules; this difficulty leads in standard textbooks to side-conditions restricting the application of the rules such as "where x does not appear free in ..."

9.2.2 The L system

The judgements of the system are sequents $\Gamma \vdash \Delta$ where Γ and Δ are finite sets of propositions. In the intuitionistic version, we have $|\Delta| \leq 1$, and we write $\Gamma \vdash A$ when $\Delta = \{A\}$, and $\Gamma \vdash \perp$ when $\Delta = \emptyset$. We keep the base sequents *Axiom*. The basic idea is to keep the introduction rules, but to replace the elimination rules by rules of introduction to the left of $\vdash$. The structural rules are:

$$\textit{Struct right} \quad : \quad \frac{\Gamma \vdash \perp}{\Gamma \vdash A}$$

$$\textit{Struct left} \quad : \quad \frac{\Gamma \vdash B}{\Gamma \cup \{A\} \vdash B}$$

and minimal logic is obtained with:

$$\Rightarrow\ \textit{right} \quad : \quad \frac{\Gamma \cup \{A\} \vdash B}{\Gamma \vdash A \Rightarrow B}$$

$$\Rightarrow \textit{ left} \quad : \quad \frac{\Gamma \cup \{B\} \vdash C \qquad \Delta \vdash A}{\Gamma \cup \Delta \cup \{A \Rightarrow B\} \vdash C}.$$

Conjunction is then obtained by adding:

$$\wedge \textit{ right} \quad : \quad \frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma \cup \Delta \vdash A \wedge B}$$

$$\wedge \textit{ left} \quad : \quad \frac{\Gamma \cup \{A, B\} \vdash C}{\Gamma \cup \{A \wedge B\} \vdash C}.$$

This last rule may be replaced by two simpler rules:

$$\frac{\Gamma \cup \{A\} \vdash C}{\Gamma \cup \{A \wedge B\} \vdash C}$$

$$\frac{\Gamma \cup \{B\} \vdash C}{\Gamma \cup \{A \wedge B\} \vdash C},$$

with the help of *Struct left*. We get positive logic by adding the rules:

$$\vee \textit{ right}_1 \quad : \quad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}$$

$$\vee \textit{ right}_2 \quad : \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

$$\vee \textit{ left} \quad : \quad \frac{\Gamma \cup \{A\} \vdash C \qquad \Delta \cup \{B\} \vdash C}{\Gamma \cup \Delta \cup \{A \vee B\} \vdash C}.$$

Finally, propositional calculus is obtained with:

$$\neg \textit{ right} \quad : \quad \frac{\Gamma \cup \{A\} \vdash \bot}{\Gamma \vdash \neg A}$$

$$\neg \textit{ left} \quad : \quad \frac{\Gamma \vdash A}{\Gamma \cup \{\neg A\} \vdash \bot}.$$

The classical system L_K is obtained by adding supplementary sets of propositions to the right of $\vdash$.

Facts. The system L has the subformula property, i.e. every formula appearing in the numerator of every rule is a subformula of some formula appearing in the denumerator. Furthermore, it is consistent, in the sense that the sequent $\emptyset \vdash \bot$ is not derivable, as is obvious since no rule applies. Finally, it is easy to see that if $\Gamma \vdash_L A$ then $\Gamma \vdash_N A$, by showing that every rule of L is a derived rule of N.

The converse of the last statement is not obvious, since the elimination rules of N do not satisfy the subformula property. Let us consider an extended system L^+, obtained by adding to L the following:

$$\textit{Cut} \quad : \quad \frac{\Gamma \vdash A \qquad \Delta \cup \{A\} \vdash B}{\Gamma \cup \Delta \vdash B}.$$

It is now easy to show that the rules of N are derived rules of L^+. All that is left to show in order to prove that L and N are equivalent is thus:

Gentzen's Theorem. $\Gamma \vdash_{L^+} A$ implies $\Gamma \vdash_L A$.
Proof. We look at the derivation tree of A in L^+, and we eliminate the occurrences of the cut rule by appropriate reduction rules, in a bottom-up fashion. Details are given in [10, 12, 19, 3].

This historically fundamental theorem establishes the consistency of the corresponding theories in a completely finitist way, except for the termination proof (the argument showing that the proof normalization terminates). Rather than giving a completely detailed proof, we shall now show how it is possible to normalize directly the proofs in N, using λ-calculus techniques.

9.3 A λ-calculus formalization

9.3.1 The minimal system with proofs

The judgements of the Gentzen systems admit implicit structural rules from the structure of finite set of propositions on the left of $\vdash$. For a meta-mathematical study it is more convenient to be more explicit and represent these sets as lists of propositions. The judgements become thus sequents of the kind we encountered in our study of λ-calculus. Furthermore, the idempotence of set union on the left of $\vdash$ is not justifiable intuitionistically, since it really corresponds to a principle of proof irrelevance. (Whereas commutativity and associativity are justifiable categorically since the corresponding laws correspond to natural isomorphisms)

For instance, we may formalize minimal logic (with its proofs) along exactly the formation rules of (typed) λ-calculus, in the following system Min:

$$
\begin{array}{rcl}
Var_1 & : & \dfrac{}{\Gamma \times A \vdash Var(1) : A} \\[2ex]
Var_{n+1} & : & \dfrac{\Gamma \vdash Var(n) : B}{\Gamma \times A \vdash Var(S(n)) : B} \\[2ex]
Abstr & : & \dfrac{\Gamma \times A \vdash M : B}{\Gamma \vdash [A]M : A \Rightarrow B} \\[2ex]
Appl & : & \dfrac{\Gamma \vdash M : A \Rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) : B}.
\end{array}
$$

Now we define a cut as an application of the β-conversion rule. Cut-elimination is thus strong normalization of (typed) λ-calculus. It proves the equivalence of the above system with a system describing the structure of normal terms.

9.3.2 The normal minimal system with proofs

Such a system may be obtained by adding to judgements $\Gamma \vdash M : A$ another sort of judgement: $\Gamma \vdash M :: A$, meaning that M is a non-abstraction term of type A in context Γ. We thus obtain the *normal* system $Norm(Min)$ of minimal logic:

$$
\begin{array}{rcl}
Var_1 & : & \dfrac{}{\Gamma \times A \vdash Var(1) :: A} \\[2ex]
Var_{n+1} & : & \dfrac{\Gamma \vdash Var(n) :: B}{\Gamma \times A \vdash Var(S(n)) :: B} \\[2ex]
Abstr & : & \dfrac{\Gamma \times A \vdash M : B}{\Gamma \vdash [A]M : A \Rightarrow B}
\end{array}
$$

$$Appl \quad : \quad \frac{\Gamma \vdash M :: A \Rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) :: B}$$

$$Coerce \quad : \quad \frac{\Gamma \vdash M :: A}{\Gamma \vdash M : A}.$$

Discussion. Our judgements $\Gamma \vdash M : A$ may be taken as propositions of a meta-system admitting a ternary functor (i.e. predicate) for each form of judgement. Thus such judgements could be verified using a Prolog-like mechanism. This is essentially the method used by G. Kahn and his colleagues in their natural semantics system [2]. However, if one is interested in using this inference system as the specification of a proof checker, we remark that this specification is functional in nature, in that $\Gamma \vdash M : A$ may be interpreted as determining A uniquely from Γ and M, leading to a direct implementation in a functional language with pattern-matching such as `ML` . Furthermore if the meta-language has a mechanism for type inclusion then the rule $Coerce$ vanishes completely, the coercion from :: to : being automatic from the shape of M. This mechanism has been proposed by Ait-Kaci [1]. Of course, we could also dispense with $Coerce$ by duplicating rule *Appl* with two cases for M (corresponding to variable and application), but this would not be as elegant.

Remark 1. The η-rule can be accomodated as a maximum η-expansion normalization. This amounts to requiring type A to be atomic in rule $Coerce$ above.

Remark 2. Adding conjunction amounts to considering a λ-calculus with products. The CCC presentation **Exp** gives a simple axiomatization, with 1 standing for the truth-value $True$.

Problem. Compare this axiomatization with the one derived from the Hilbert presentation in Chapter 3. Quantify precisely in which way it is more efficient. Is it possible to give a complete restriction to Prolog which will make it a decision procedure for Minimal Logic in either axiomatization?

9.3.3 The proof-irrelevant minimal system

If one is interested in the inference system above for generating proofs, then the power of Prolog is needed for dealing with rule *Appl*, since A does not appear in the denumerator. If one is interested only in the existence of a proof, and not in the proof object itself, the M arguments may be dropped from the rules. It is convenient to formulate such a proof-irrelevance system with Gentzen-like sequents $\Gamma \vdash M$, where Γ is again a finite set of propositions:

$$\frac{A \in \Gamma}{\Gamma \vdash A}$$

$$\frac{\Gamma \cup \{A\} \vdash B}{\Gamma \vdash A \Rightarrow B}$$

$$\frac{\Gamma \vdash A_1, \ldots, \Gamma \vdash A_n \quad A_1 \Rightarrow (A_2 \Rightarrow \cdots (A_n \Rightarrow B) \cdots) \in \Gamma}{\Gamma \vdash B}.$$

This infinite, but simple, system expresses exactly in what way the normal system above has the subformula property. Using remark 1 above, we may restrict B to be atomic in the third rule.

Exercise. Derive from the system above a decision procedure for Minimal Logic.

Problem. Derive from the above considerations a generalization of Prolog where clauses and literals are generalized to a common abstraction.

9.4 Tait's computability method

We give in this section a short overview of Tait's method for proving strong normalization of typed λ-terms. We write TSN for the set of strongly normalizable (typed) λ-terms.

Definition. Let $\Gamma \vdash M : A$. We say that M is *computable* iff:

- either A is atomic and $M \in TSN$
- or $A = B \Rightarrow C$ and for every computable N, with $\Gamma \vdash N : B$, we have $(M\ N)$ computable.

From now on, we assume that Γ is such that it contains an initial prefix containing every primitive type. Thus we assume the primitive types to be non-empty. This will ensure that for every type A there exists a computable term C_A such that $\Gamma \vdash C_A : A$.

Remark. If M is computable, and M reduces to N, then N is computable, by an easy induction on the type of M.

Convention. In the following, when we write $(M\ M_1 \ldots M_n)$, this notation includes the term M in case $n = 0$.

Lemma 1. $\Gamma \vdash (n\ N_1\ \ldots\ N_k) : A$ is computable whenever $N_i \in TSN \quad (0 \leq k)$, and $\Gamma \vdash M : A$ and M computable implies $M \in TSN$.
Proof. Induction on A.

Corollary. Every computable term is strongly normalizable.

Lemma 2. If $M\{N\}$ is computable, then $(\texttt{[}A\texttt{]}M\ N)$ is computable provided N is computable when M is constant (i.e. the variable 1 does not appear in M).

Lemma 3. Let $\Gamma \times A \vdash M : B$ and $\Gamma \vdash N : A$, with N computable. Then $M\{N\}$ is computable.

Corollary. Every term is computable.
Hint: Induction on M.

We may now conclude:

Strong Normalization Theorem. Every (typed) λ-term is in TSN.

The complete proof of the lemmas above is given in [19]. Extensions of this method to calculi with more functors is given in [20].

9.5 A syntactic interpretation

We shall here give a more abstract treatment of Tait's method, due originally to Girard, and modified by Coquand. This method extends to stronger calculi, as we shall see in the next chapter.

The idea is to build an interpretation $\mathcal{I}$ of typed λ-terms as follows. We denote by $\nu(M)$ the pure λ-term obtained by erasing the type information of bound variables. Let SN denotes the set of strongly normalizable closed *pure* λ-terms built on one constant Ω.

Definition. A subset S of SN is said to be *saturated* iff

- $\forall N, M_1, \ldots, M_n \in SN \quad (M\{N\}\ M_1\ \ldots\ M_n) \in S \Rightarrow (\texttt{[]}M\ N\ M_1\ \ldots\ M_n) \in S$

- $N_1, \ldots, N_k \in SN \Rightarrow (\Omega\ N_1 \ldots N_k) \in S$.

We write Sat for the set of saturated subsets of SN.

Now assume the interpretation $\mathcal{I}$ associates with every typed term M its corresponding pure term $\mathcal{I}(M) = \nu(M)$, and with every type A a set $\mathcal{I}(A) \subseteq Sat$, in such a way that $\Gamma \vdash M : A$ implies $\mathcal{I}(M) \in \mathcal{I}(A)$. This clearly shows that every M is strongly normalizable, since $M \rhd N$ iff $\nu(M) \rhd \nu(N)$.

We may now immediately place Tait's method in this framework, by defining:

$$\mathcal{I}(A) = SN \quad \text{if } A \text{ is atomic},$$

$$\mathcal{I}(B \Rightarrow C) = \{M \mid \forall N \in \mathcal{I}(B) \quad (M\ N) \in \mathcal{I}(C)\}.$$

Exercise. Show that the lemmas above prove that for every proposition P, $\mathcal{I}(P)$ has the required properties. Actually $\mathcal{I}(A)$ may be interpreted as an arbitrary element of Sat, for A an atomic type.

Remark 1. The "saturated" condition is directly inspired from the result one wants to prove, here strong normalization. Other desired properties may be obtained using other definitions of saturation. For instance, if one only wants to show normalizability, it is enough to replace the first closure condition by the simpler condition: $M \rhd N$ and $N \in S$ implies $M \in S$. The special constant Ω is needed in order to have a non-trivial interpretation for the non-atomic propositions.

Remark 2. The interpretation above is syntactic in nature. It is possible to transform such an interpretation into an extensional one, using Gandy's hull technique, generalized by Plotkin and others with the technique of the so-called *logical relations*. (Note that the interpretation above is typically a logical predicate). More generally, logical mappings are (typed) λ-calculus morphisms. The state of the art on logical relations is described in [18].

9.6 Realizability

The interpretation above may be considered a realizability interpretation, in the spirit of Kleene [9]. However, here the derivation of $\Gamma \vdash M : A$ gives directly a realization of proposition A as a λ-term $\nu(M)$, without going to the detour of computing Gödel numbers as codes of recursive functions. We leave the development of this recursion-theoretic view of intuitionistic proofs for a future version of these notes.

References

[1] H. Ait-Kaci. "A lattice theoretic approach to computation based on a calculus of partially ordered type structures." Ph. D. thesis, University of Pennsylvania (1984).

[2] D. Clément, J. Despeyroux, T. Despeyroux, G. Kahn. "Natural semantics on the computer." Research Report 416, Inria, June 1985.

[3] Dummett. "Elements of Intuitionism." Clarendon Press, Oxford (1977).

[4] G. Gentzen. "The Collected Papers of Gerhard Gentzen." Ed. E. Szabo, North-Holland, Amsterdam (1969).

[5] J.Y. Girard. "Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. Proceedings of the Second Scandinavian Logic Symposium, Ed. J.E. Fenstad, North Holland (1970) 63–92.

[6] J.Y. Girard. "Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieure." Thèse d'Etat, Université Paris VII (1972).

[7] K. Gödel. "Uber eine bisher noch nicht benutze Erweitrung des finiten Standpunktes." Dialectica, **12** (1958).

[8] W. A. Howard. "The formulæ-as-types notion of construction." Unpublished manuscript (1969). Reprinted in to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Eds Seldin J. P. and Hindley J. R., Academic Press (1980).

[9] S.C. Kleene. "On the interpretation of intuitionistic number theory." J. Symbolic Logic **31** (1945).

[10] S.C. Kleene. "Introduction to Meta-mathematics." North Holland (1952).

[11] G. Kreisel. "On the interpretation of nonfinitist proofs, Part I, II." JSL **16** (1952, 1953).

[12] D. Prawitz. "Natural Deduction." Almqist and Wiskell, Stockolm (1965).

[13] D. Prawitz. "Ideas and results in proof theory." Proceedings of the Second Scandinavian Logic Symposium (1971).

[14] H. Rasiowa, R. Sikorski "The Mathematics of Metamathematics." Monografie Matematyczne tom **41**, PWN, Polish Scientific Publishers, Warszawa (1963).

[15] J.R. Shoenfield. "Mathematical Logic." Addison-Wesley (1967).

[16] R. Statman. "Intuitionistic Propositional Logic is Polynomial-space Complete." Theoretical Computer Science **9** (1979) 67–72, North-Holland.

[17] R. Statman. "The typed Lambda-Calculus is not Elementary Recursive." Theoretical Computer Science **9** (1979) 73–81.

[18] R. Statman. "Logical relations and models of λ-calculus." Unpublished course notes, CMU, Fall 1985.

[19] S. Stenlund. "Combinators λ-terms, and proof theory." Reidel (1972).

[20] M.E. Szabo. "Algebra of Proofs." North-Holland (1978).

[21] W. Tait. "Intensional interpretations of functionals of finite type I." J. of Symbolic Logic **32** (1967) 198–212.

[22] W. Tait. "A Realizability Interpretation of the Theory of Species." Logic Colloquium, Ed. R. Parikh, Springer Verlag Lecture Notes **453** (1975).

[23] G. Takeuti. "Proof theory." Studies in Logic **81** Amsterdam (1975).

Chapter 10

Polymorphism

10.1 ML's polymorphism

We saw that formal systems could be pleasantly presented using polymorphic operators (inference rules) at the meta level. It seems a good idea to push polymorphism to the object level, for functions defined by the user as λ-expressions. To this end, we introduce bindings for type variables. This idea of type quantification corresponds to allowing proposition quantifiers in our propositional logic. First we allow a universal quantifier in prenex position. That is, with $T_0 = T(\Phi, V)$, we now introduce *type schemas* in $T_1 = T_0 \cup \forall\alpha \cdot T_1$, $\alpha \in V$. A (type) term in T_1 has thus both free and bound variables, and we write $FV(M)$ and $BV(M)$ for the sets of free (respectively bound) variables.

We now define *generic instantiation*.
Let $\tau = \forall\alpha_1 ... \alpha_m \cdot \tau_0 \in T_1$ and $\tau' = \forall\beta_1 ... \beta_n \cdot \tau'_0 \in T_1$. We define $\tau' \geq_G \tau$ iff $\tau'_0 = \sigma(\tau_0)$ with $D(\sigma) \subseteq \{\alpha_1, ..., \alpha_m\}$ and $\beta_i \notin FV(\tau)$ $(1 \leq i \leq n)$. Note that $\geq$ acts on FV whereas $\geq_G$ acts on BV. Also note

$$\tau' \geq_G \tau \quad \Rightarrow \quad \sigma(\tau') \geq_G \sigma(\tau).$$

We now present the Damas-Milner inference system for polymorphic λ-calculus [11]. In what follows, a sequent hypothesis A is assumed to be a list of specifications $x_i : \tau_i$, with $\tau_i \in T_1$, and we write $FV(A) = \bigcup_i FV(\tau_i)$.

$$TAUT \;:\; A \;\vdash\; x \;:\; \alpha \quad (x : \alpha \in A)$$

$$INST \;:\; \frac{A \;\vdash M \;:\; \alpha}{A \;\vdash M \;:\; \beta} \quad (\alpha \leq_G \beta)$$

$$GEN \;:\; \frac{A \;\vdash M \;:\; \tau}{A \;\vdash M \;:\; \forall\alpha \cdot \tau} \quad (\alpha \notin FV(A))$$

$$APP \;:\; \frac{A \;\vdash M \;:\; \tau' \rightarrow \tau \quad A \;\vdash N \;:\; \tau'}{A \;\vdash (M \; N) \;:\; \tau}$$

$$ABS \;:\; \frac{A \cup \{x : \tau'\} \;\vdash M \;:\; \tau}{A \;\vdash [x]M \;:\; \tau' \rightarrow \tau}$$

$$LET \;:\; \frac{A \;\vdash M \;:\; \tau' \quad A \cup \{x : \tau'\} \;\vdash N \;:\; \tau}{A \;\vdash let\; x = M \; in \; N \;:\; \tau}.$$

For instance, it is an easy exercise to show that

$$\vdash \mathit{let}\ i\ =\ \texttt{[}x\texttt{]}x\ \mathit{in}\ (i\ i)\ :\ \alpha \rightarrow \alpha.$$

The above system may be extended without difficulty by other functors such as product, and by other `ML` constructions such as letrec. Actually every `ML` compiler contains a type-checker implementing implicitly the above inference system. For instance, with the unary functor *list* and the following `ML` primitives: `[]` : $(\mathit{list}\ \alpha)$, $\mathit{cons} : \alpha \times (\mathit{list}\ \alpha)$ (written infix as a dot), $\mathit{hd} : (\mathit{list}\ \alpha) \rightarrow \alpha$ and $\mathit{tl} : (\mathit{list}\ \alpha) \rightarrow (\mathit{list}\ \alpha)$, we may define recursively the map functional as:

$$\mathit{letrec}\ \mathit{map}\ f\ l\ =\ \mathit{if}\ l = \texttt{[]}\ \mathit{then}\ \texttt{[]}\ \mathit{else}\ (f\ (\mathit{hd}\ l))\ \cdot\ \mathit{map}\ f\ (\mathit{tl}\ l)$$

and we get as its type:

$$\vdash\ \mathit{map} : (\alpha \rightarrow \beta) \rightarrow (\mathit{list}\ \alpha) \rightarrow (\mathit{list}\ \beta).$$

Of course the `ML` compiler is not implemented directly from the inference system above, which is non-deterministic because of rules $INST$ and GEN. It uses unification instead, and thus computes deterministically a principal type, which is minimum with respect to $\leq_G$:

Milner's Theorem. Every typable expression of the polymorphic λ-calculus possesses a principal type, minimum with respect to generic instantiation [18].

`ML` is a strongly typed programming language, where type inference is possible because of the above theorem: the user need not write type specifications. The compiler of the language does more than type-checking, since it actually performs a proof synthesis. Types disappear at run time, but because of the type analysis no dynamic checks are needed to enforce the consistency of data operations, and this allows fast execution of `ML` programs. `ML` is a generic name for languages of the `ML` family. For instance, by adding exceptions, abstract data types (permitting in particular user-defined functors) and references, one gets approximately the meta-language of the LCF proof assistant [15]. By adding record type declarations (i.e. labeled sums and products) one gets L. Cardelli's `ML` [2]. By adding constructor types, pattern-matching and concrete syntax, we get the `ML` presented in Chapter 1. A more complete language, including modules, is under design as Standard `ML` [19]. Current research topics on the design of `ML` -like languages are the incorporation of object-oriented features allowing subtypes, remanent data structures and bitmap operations [3], and "lazy evaluation" permitting streams and ZF expressions [28, 20].

Note on the relationship between `ML` and λ-calculus. First, `ML` uses so-called call by value implementation of procedure call, corresponding to innermost reduction, as opposed to the outermost regime of the standard reduction. Lazy evaluation permits standard reductions, but closures (i.e. objects of a functional type $\alpha \rightarrow \beta$) are *not* evaluated. Finally, types in `ML` serve for ensuring the integrity of data operations, but still allow infinite computations by non-terminating recursions.

10.1.1 The limits of `ML` 's polymorphism

Consider the following `ML` definition:

$$\mathit{letrec}\ \mathit{power}\ n\ f\ u\ =\ \mathit{if}\ n = 0\ \mathit{then}\ u\ \mathit{else}\ f\ (\mathit{power}\ (n-1)\ f\ u)$$

of type $\mathit{nat} \rightarrow (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$. This function, which associates to natural n the polymorphic iterator mapping function f to the n-th power of f, may be considered a coercion operator between `ML` 's internal naturals and Church's representation of naturals in pure λ-calculus [6]. Let us recall briefly this representation. Integer 0 is represented as the projection term $[f][u]u$. Integer 1 is $[f][u](f\ u)$. More generally, n is represented as the functional $\bar{n}$ iterating a function f to its n-th power:

$$\bar{n} = [f][u](f\ (f\ ...(f\ u)...))$$

and the arithmetic operators may be coded respectively as:

$$n + m = [f][u](n\ f\ (m\ f\ u))$$

$$n \times m = [f](n\ (m\ f))$$

$$n^m = (m\ n).$$

For instance, with $\bar{2} = [f][u](f\ (f\ u))$, we check that $\bar{2} \times \bar{2}$ converts to its normal form $\bar{4}$.

We would like to consider a type

$$NAT = \forall\alpha \cdot (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$$

and be able to type the operations above as functions of type $NAT \rightarrow NAT \rightarrow NAT$. However the notion of polymorphism found in `ML` does not support such a type, it allows only the weaker

$$\forall\alpha \cdot ((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)) \rightarrow ((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)) \rightarrow ((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))$$

which is inadequate, since it forces the *same* generic instantiation of NAT in the two arguments.

10.2 Second-order λ-calculus

The example above suggests using the universal type quantifier *inside* type formulas. We thus consider a functor alphabet based on one binary $\rightarrow$ constructor and one quantifier $\forall$. We shall now consider a λ-calculus with such types, which we shall call *second-order λ-calculus*, owing to the fact that the type language is now a second-order propositional logic, with propositional variables explicitly quantified. In order to emphasize this connection, we actually write $\Rightarrow$ instead of $\rightarrow$. In this calculus, we shall be able to form types (propositions) such as:

$$(\forall A \cdot A \Rightarrow A) \Rightarrow (\forall A \cdot A \Rightarrow A).$$

Such a calculus was proposed by J.Y. Girard [13, 14], and independently discovered by J. Reynolds [22].

10.2.1 The inference system

We now have two kinds of variables, the variables bound by λ-abstraction, and the propositional variables. Each kind will have its own de Bruijn indexing scheme, but we put both kinds of bindings in one context sequence, in order to ensure that in a λ-binding $[x : P]$ the free propositional variables of P are correctly scoped. A *context* Γ is thus a sequence of bindings $[x : P]$ and of bindings $[A : Prop]$. We use de Bruijn indexes $V(n)$ and $P(n)$ to reference respectively the two kinds of variables. However, there is a slight difficulty if one tries to adhere too strictly to de Bruijn's notation. Consider the context $\Gamma = [A : Prop][x : A][B : Prop]$. In concrete syntax, we write $\Gamma \vdash x : A$. But if we use de Bruijn's indexes for propositional names, we get in the abstract syntax $\Gamma \vdash V(1) : P(2)$, i.e. the propositions have to be relocated.

In order to remedy this notational difficulty, we shall assume a mixed naming scheme, allowing concrete names for free variables of expressions as well as integers for bound variables. The binding operation $[x:P]M$ denotes now the abstract $[P]M'$, where M' is M where every occurrence of x is replaced by the correct de Bruijn's index. Similarly we provide a binding operation $\forall A \cdot P$ for propositional variables. Finally an operation $\Lambda A \cdot M$ binds a propositional variable in a term.

A context Γ is said to be *valid* if it binds variables with well-formed propositions. Thus the empty context is valid, if Γ is valid and does not bind A then $\Gamma[A:Prop]$ is valid, and finally if Γ is valid and does not bind x then $\Gamma[x:P]$ is valid provided $\Gamma \vdash P : Prop$. This last judgement (propositional formation) is defined recursively as follows:

$$\frac{[A:Prop] \in \Gamma}{\Gamma \vdash A : Prop}$$

$$\frac{\Gamma \vdash P : Prop \qquad \Gamma \vdash Q : Prop}{\Gamma \vdash P \Rightarrow Q : Prop}$$

$$\frac{\Gamma[A:Prop] \vdash P : Prop}{\Gamma \vdash \forall A \cdot P : Prop}.$$

Let us now give the term-formation rules. We have two more constructors: $\Lambda A \cdot M$ which makes a term polymorphic, by $\forall$-introduction, and $<M\ P>$, which instantiates the polymorphic term M over the type corresponding to proposition P, by $\forall$-elimination.

$$Var \quad : \quad \frac{[x:P] \in \Gamma}{\Gamma \vdash x : P}$$

$$Abstr \quad : \quad \frac{\Gamma \vdash P : Prop \qquad \Gamma[x:P] \vdash M : Q}{\Gamma \vdash [x:P]M : P \Rightarrow Q}$$

$$Appl \quad : \quad \frac{\Gamma \vdash M : P \Rightarrow Q \qquad \Gamma \vdash N : P}{\Gamma \vdash (M\ N) : Q}$$

$$Gen \quad : \quad \frac{\Gamma[A:Prop] \vdash M : P}{\Gamma \vdash \Lambda A \cdot M : \forall A \cdot P}$$

$$Inst \quad : \quad \frac{\Gamma \vdash M : \forall A \cdot P \qquad \Gamma \vdash Q : Prop}{\Gamma \vdash <M\ Q> : P\{Q\}_P}.$$

We do not make explicit the propositional substitution operation $P\{Q\}_P$, which is defined similarly to the λ-calculus substitution $M\{N\}$ seen previously. The latter will be denoted here by $M\{N\}_V$.

Proposition 1. If Γ is valid, then $\Gamma \vdash M : P$ implies $\Gamma \vdash P : Prop$.

We leave the proof of such easy (but tedious) lemmas to the patient reader.

Let us now give an example of a derivation. Let $Id := \Lambda A \cdot [x:A]x$. Id is the polymorphic identity algorithm, and we check easily that $\vdash Id : One$, where $One := \forall A \cdot A \Rightarrow A$. Note that indeed One is well-formed in the empty context. Now we may instantiate Id over its own type One, yielding: $\vdash <Id\ One> : One \Rightarrow One$. The resulting term may thus be applied to Id, yielding: $\vdash (<Id\ One>\ \ Id) : One$.

Similarly, we can define a composition operator for proofs, whose type is the analogue of the cut, or detachment rule:
$[P:Prop][Q:Prop][R:Prop] \vdash [f:P \Rightarrow Q][g:Q \Rightarrow R][x:P](g\ (f\ x)) : ((P \Rightarrow Q) \Rightarrow$

$(Q \Rightarrow R) \Rightarrow (P \Rightarrow R))$.
We shall use the notation $f;g$ as a shorthand for the too cumbersome $(<Compose\ P\ Q\ R>\ f\ g)$, since the type arguments P, Q and R can be retrieved as subparts of the types of f and g.

10.2.2 The conversion rules

The calculus admits two conversion rules. The first one is just β:

$$\beta \quad : \quad \frac{}{\Gamma \vdash ([x:P]M\ N) \rhd M\{N\}_V}.$$

The second one eliminates the cut formed by introducing and eliminating a quantification:

$$\beta' \quad : \quad \frac{}{\Gamma \vdash <\Lambda A \cdot M\ P> \rhd M\{P\}_P}.$$

Of course, we assume all other rules extending $\rhd$ as a term congruence, as usual. We may also consider analogues of the η rule.

Proposition 2. If Γ is valid, $\Gamma \vdash M : P$ and $\Gamma \vdash M \rhd N$ then $\Gamma \vdash N : P$.

10.2.3 The syntactic interpretation

We proceed as in the last chapter. However, here there are no primitive types. In order to have a non-trivial interpretation, we introduce a supplementary constant Ω to our untyped λ-terms. Let λ_Ω be the set of such terms, and SN be the set of strongly normalizable terms of λ_Ω.

Definition. A subset S of SN is said to be *saturated* iff

- $\forall N \in SN \quad (M\{N\}\ M_1\ \ldots\ M_n) \in S \Rightarrow ([\,]M\ N\ M_1\ \ldots\ M_n) \in S$
- $\Omega \in S$
- $N_1, \ldots, N_k \in SN \Rightarrow (\Omega\ N_1 \ldots N_k) \in S$.

Note that in the first clause, we may limit ourselves to considering M and the M_i's in SN. We write Sat for the set of saturated subsets of SN.

We now define the interpretation $\mathcal{I}$ by defining for every term M its corresponding pure term $\mathcal{I}(M) = \nu(M)$, where

- $\nu(V(n)) = n$
- $\nu([x:P]M) = [\,]\nu(M)$
- $\nu((M\ N)) = (\nu(M)\ \nu(N))$
- $\nu(\Lambda A \cdot M) = \nu(M)$
- $\nu(<M\ P>) = \nu(M)$.

Note that $\nu(M)$ is a pure λ-term constructed over the list of free variables $\{x \mid [x:P] \in \Gamma\}$.

Finally, to every A such that $[A : Prop] \in \Gamma$ we associate an arbitrary saturated set $\mathcal{I}(A)$. Let $I(\Gamma)$ be the product of all such $\mathcal{I}(A)$'s. We define recursively the interpretation $\mathcal{I}_{\mathcal{I}(\Gamma)}(P)$ of a proposition P, such that $\Gamma \vdash P : Prop$, as follows:

- $\mathcal{I}_G(P \Rightarrow Q) = \{M \mid \forall N \in \mathcal{I}_G(P) \quad (M\ N) \in \mathcal{I}_G(Q)\}$
- $\mathcal{I}_G(\forall A \cdot P) = \bigcap_{S \in Sat} \mathcal{I}_{G \times S}(P)$
- $\mathcal{I}_G(A) = G_A$.

Example. $\mathcal{I}(Id) = \texttt{[]}1$. $\mathcal{I}(One)$ contains all strongly normalizable terms whose canonical form is $\texttt{[]}1$, plus strongly normalizable terms whose canonical form has head variable Ω.

10.2.4 Basic meta-mathematical properties

The main use of the interpretation above is to prove:

Girard's theorem. If Γ is valid and $\Gamma \vdash M : P$, then $\mathcal{I}(M) \in \mathcal{I}(P) \in Sat$.
Corollary 1. $\nu(M) \in SN$.

Corollary 2: Strong normalization. The conversion $\triangleright$ on typed terms is Nœtherian.

(Note that β' alone is Nœtherian).

Definition. Let Γ be a valid context, with $\Gamma \vdash P : Prop$. We say that P is *inhabited* in Γ iff $\mathcal{I}_{\mathcal{I}(\Gamma)}(P)$ contains a term without Ω's.

Note that if $\Gamma \vdash M : P$, then P is inhabited (by $\mathcal{I}(M)$). We know obtain the consistency of the logical system as:

Soundness Theorem. The type $\nabla = \forall A \cdot A$ is not inhabited.
Corollary. There is no term M which proves ∇.

Undecidability Theorem. The following problem is recursively unsolvable: Given a valid context Γ and a proposition P, with $\Gamma \vdash P : Prop$, find whether or not there exists an M such that $\Gamma \vdash M : P$.

The second-order λ-calculus does not admit principal types. For instance, we shall show below that combinator K may be typed in several incompatible manners. We may still wonder whether it is decidable whether an arbitrary pure λ-term is typable in the system or not. This is an important open problem:

Problem. Give a procedure which, given a pure λ-term T, decides whether or not there exist M and P such that $\vdash M : P$, with $T = \nu(M)$. Alternatively, show that the problem is undecidable.

10.3 Examples of polymorphic proofs

In this section, we demonstrate the power of expression of the second-order calculus by way of examples.

10.3.1 Intuitionistic connectives

We first show that the other propositional connectives are definable in the calculus. It is well known that the intuitionistic connectives are definable in the second-order propositional calculus. The encoding of conjunction was already proposed by Russell, as explained in Prawitz [21].

Let P and Q be two propositions. We define $P \wedge Q$ as the proposition:

$$P \wedge Q \ := \ \forall A \cdot (P \Rightarrow Q \Rightarrow A) \Rightarrow A.$$

As usual, we associate implications to the right, and applications to the left. The definition above is a correct encoding of $\wedge$, as can be seen from the derivation of the standard rules of conjunction:

$$[P : Prop]\,[Q : Prop]\,[x : P]\,[y : Q] \vdash \Lambda A \cdot [h : P \Rightarrow Q \Rightarrow A](h\ x\ y) : P \wedge Q$$

$$[P : Prop]\,[Q : Prop]\,[x : P \wedge Q] \vdash (<x\ P>\ \ [u : P]\,[v : Q]u) : P$$

$$[P : Prop]\,[Q : Prop]\,[x : P \wedge Q] \vdash (<x\ Q>\ \ [u : P]\,[v : Q]v) : Q.$$

In order to understand this sort of definition, it is best to wonder what is the *operational* use of the concept one is trying to define. Once this is clear, the concept can be easily *programmed*. This *procedural interpretation* is faithful to the intuitionistic semantics. For instance, $P \wedge Q$ is a method for proving any proposition A, provided one has a proof that A follows from P and Q. Note that the proof of $\wedge$-introduction above is a pairing algorithm, the two projections being the proofs of $\wedge$-elimination on the left and on the right.

We may similarly "program" the (intuitionistic) sum $P + Q$ of two propositions P and Q:

$$P + Q \ := \ \forall A \cdot (P \Rightarrow A) \Rightarrow (Q \Rightarrow A) \Rightarrow A.$$

Sum elimination is proved by the conditional, or *case* expression:

$$[P : Prop]\,[Q : Prop] \vdash \Lambda A \cdot [u : P \Rightarrow A]\,[v : Q \Rightarrow A]\,[x : P + Q](<x\ A>\ \ u\ v)$$

$$: \forall A \cdot (P \Rightarrow A) \Rightarrow (Q \Rightarrow A) \Rightarrow (P + Q) \Rightarrow A.$$

The two sum introductions correspond to the two injections:

$$[P : Prop]\,[Q : Prop] \vdash [x : P]\Lambda A \cdot [u : P \Rightarrow A]\,[v : Q \Rightarrow A](u\ x) : P \Rightarrow (P + Q)$$

$$[P : Prop]\,[Q : Prop] \vdash [y : Q]\Lambda A \cdot [u : P \Rightarrow A]\,[v : Q \Rightarrow A](v\ y) : Q \Rightarrow (P + Q).$$

10.3.2 Classical logic

Classical reasoning is reasoning by contradiction. The contradiction, or absurd proposition, proves every proposition A by mere application:

$\nabla \ := \ \forall A \cdot A.$

∇ has no proof, and may thus play the rôle of the truth-value *False*. Negating a proposition amounts to asserting that it implies ∇, whence the concept of negation:
$\neg\, [A : Prop] \ := \ A \Rightarrow \nabla.$

The Sheffer's stroke $A \mid B$ (read "A contradictory with B") may be defined as:
$[A : Prop] \mid [B : Prop] \ := \ A \Rightarrow B \Rightarrow \nabla.$

It is easy to show $\forall A \cdot \forall B \cdot (A \mid B) \iff \neg(A \wedge B)$. The other classical connectives may be simply expressed in term of $\mid$:
$[A : Prop] \supset [B : Prop] \ := \ A \mid \neg B$
$[A : Prop] \vee [B : Prop] \ := \ (\neg A) \mid (\neg B)$
$[A : Prop] \equiv [B : Prop] \ := \ (A \supset B) \wedge (B \supset A).$

Let us call *classical closure* of proposition A its double negation:
$\mathcal{C}([A:Prop]) \ := \ \neg(\neg A)$.

Every proposition denies its negation:

$$[A:Prop] \vdash [p:A][q:\neg A](q\ p) : A \Rightarrow \mathcal{C}(A).$$

The reverse implication holds only of classical propositions:
$Classical([A:Prop]) \ := \ \mathcal{C}(A) \Rightarrow A$.

We can show that $\nabla, \neg\ , \mid$ construct only classical propositions, and thus so do $\vee$ and $\supset$. Finally, $\wedge$ preserves the property of being classical, and thus $\equiv$ constructs also classical propositions.

Actually, classical reasoning consists in general in showing that a set of propositions $\{A_1, ..., A_n\}$ is contradictory. The connectives $\nabla, \neg\ , \mid$ express this notion for $n = 0, 1, 2$ respectively.

Let us remark that it is easy to prove the principe of the excluded middle:
$[A:Prop] \vdash < Id\ \mathcal{C}(A) >: \neg A \vee A$.

Remark. Many other encodings of the propositional connectives may be used. Let us give two alternate definitions for classical disjunction:
$[A:Prop] \vee' [B:Prop] \ := \ \mathcal{C}(A+B)$
$[A:Prop] \vee'' [B:Prop] \ := \ \forall C \cdot Classical(C) \Rightarrow (A \Rightarrow C) \Rightarrow (B \Rightarrow C) \Rightarrow C$.

10.3.3 Universal algebra and abstract data types

Initial algebras

We first show how to formalize the elementary notions from Algebra, in particular the notion of free algebra over a given signature. We start with the homogeneous case, that is we assume in the following that contexts start with a proposition letter taken as unique sort: $[A:Prop]$.

For every $n \geq 0$, we define the *A-cardinal* $\overline{n}$ associated to n by induction:

$\overline{0} = A$

$\overline{n+1} = A \Rightarrow \overline{n}$.

We define now the *functionality* $\varphi(\Sigma)$ associated to a signature Σ represented as a list of operators given with their arity, by:
$\varphi(\emptyset) = A$
$\varphi([F:n]\ \Sigma) = \overline{n} \Rightarrow \varphi(\Sigma)$.

Such definitions are easily programmable in the meta-language.

We now obtain the *initial algebra* associated to signature Σ by abstracting over the type given as carrier of the algebra:
$I(\Sigma) = \forall A \cdot \varphi(\Sigma)$.

Let us now consider an arbitrary Σ-algebra. That is, we assume we place ourselves in context Γ :

$$\Gamma = [A:Prop][F_1:\overline{n_1}]\cdots[F_s:\overline{n_s}].$$

If $M : I(\Sigma)$ is an arbitrary construction of an element of the initial Σ-algebra, we call *image* of M in the Σ-algebra Γ the term $M^\Gamma = (< M\ \ A >\ \ F_1\ \cdots\ F_s)$. We remark that this term is well-formed, with type A. This notion of image corresponds, classically, to taking the image of M by the unique Σ-morphism from $I(\Sigma)$ to Γ. For instance, when $M_1 : I(\Sigma)$, $M_2 : I(\Sigma)$, ... , $M_{n_k} : I(\Sigma)$, we get $(F_k\ M_1^\Gamma\ \cdots\ M_{n_k}^\Gamma) : A$. We define thus a F_k operator of arity n_k over $I(\Sigma)$, that we call the F_k-constructor, obtained in discharging Γ, and a list of n_k variables of type $I(\Sigma)$.

Definition. Let Σ be an arbitrary signature of length s:

$$\Sigma = [F_1 : \overline{n_1}] \cdots [F_s : \overline{n_s}].$$

We define the set $Dat(\Sigma)$ of *data elements* of Σ as the set:

$$\{\nu(M) \mid M = \Lambda A \cdot [F_1 : \overline{n_1}] \cdots [F_s : \overline{n_s}] N \ \text{ with } N \text{ canonical}\}.$$

Remark. The set of canonical elements in $\mathcal{I}(\Sigma)$ has too much redundancy if we do not assume the η rule of conversion. The data elements restrict consideration to the λ-terms in η-expanded normal form:

The Representation Theorem. $Dat(\Sigma)$, structured with the constructors, is isomorphic to the initial algebra in the class of all Σ-algebras.

Problem. Prove the theorem above.

Examples of data types

Let us now give a few examples. When $\Sigma = \emptyset$, we get $I(\Sigma) = \nabla$, the empty algebra. When $\Sigma = [i : 0]$, we get $I(\Sigma) = One\ :=\ \forall A \cdot A \Rightarrow A$, and the i-constructor is $Id = \Lambda A \cdot [i : A] i$.

With $\Sigma = [t : 0][f : 0]$, we get: $I(\Sigma) = Bool\ :=\ \forall A \cdot A \Rightarrow A \Rightarrow A$, and the two constructors are the Booleans of Church [6]:
$True\ :=\ \Lambda A \cdot [t : A][f : A]\ v$
$False\ :=\ \Lambda A \cdot [t : A][f : A]\ f.$

When $\Sigma = [s : 1][z : 0]$, we get $I(\Sigma) = Nat$, Church's naturals :
$Nat\ :=\ \forall A \cdot (A \Rightarrow A) \Rightarrow A \Rightarrow A$
$S\ :=\ [n : Nat] \Lambda A \cdot [s : A \Rightarrow A][z : A]\ (s\ (<n\ A>\ \ s\ z))$
$0\ :=\ \Lambda A \cdot [s : A \Rightarrow A][z : A]\ z.$

When $\Sigma = [c : 2][n : 0]$, we get $I(\Sigma) = Bin$, the binary trees:
$Bin\ :=\ \forall A \cdot (A \Rightarrow A \Rightarrow A) \Rightarrow A \Rightarrow A$
$Cons\ :=\ [a_1 : Bin][a_2 : Bin] \Lambda A \cdot [c : A \Rightarrow A \Rightarrow A][n : A]\ (c\ (<a_1\ A>\ \ c\ n)\ (<a_2\ A>\ \ c\ n))$
$Nil\ :=\ [A : Prop][c : A \Rightarrow A \Rightarrow A][n : A]\ n.$

Generalization to non-homogeneous algebras

It is straightforward to generalize these notions to the non-homogeneous case, introducing as many sorts as necessary. For instance, the list structure is axiomatized on two sorts A and B as follows:
$List\ :=\ \forall A, B \cdot (A \Rightarrow B \Rightarrow B) \Rightarrow B \Rightarrow B.$

The operation of adding an element to a list is polymorphic. Let us consider the list schema, over proposition A:
$List\ A\ :=\ \forall B \cdot (A \Rightarrow B \Rightarrow B) \Rightarrow B \Rightarrow B.$
We now define, in context $\Gamma = [A : Prop]$:

$Add \ := \ [x:A]\,[L:(List\ A)]\Lambda B\cdot[c:A\Rightarrow B\Rightarrow B]\,[e:B]\ (c\ x\ (<L\ B>\ \ c\ e))$
$:\forall A\cdot A\Rightarrow(List\ A)\Rightarrow(List\ A)$.

We remark the analogy with `ML` 's list constructor. Here the empty list is doubly polymorphic:
$Empty \ := \ \Lambda A\cdot\Lambda B\cdot[c:A\Rightarrow B\Rightarrow B]\,[e:B]\,e \ \ : List$.

More generally, we may define all the data structures corresponding to free algebras. We remark that the corresponding propositions are restricted to degree 2, with the degree δ defined as:

- $\delta(A)=0 \qquad (A\ variable)$
- $\delta(\forall A\cdot M)=\delta(M)$
- $\delta(P\Rightarrow Q)=max\{1+\delta(P),\delta(Q)\}$.

Problem. Generalize the Representation theorem above to the non-homogeneous case.

Second-order arithmetic

Let us give a few examples of programs over naturals. Addition is obtained by iterating successor:
$Plus \ := \ [m:Nat]\,[n:Nat](<n\ Nat>\ \ S\ m)$.
Other definitions are possible. Multiplication is similarly obtained by iterating addition:
$Times \ := \ [m:Nat]\,[n:Nat](<n\ Nat>\ \ (Plus\ m)\ 0)$.

We may also "see" our naturals as polymorphic iterators. Another possible definition of multiplication of m and n would thus be the composition $m;n$.

Exponentiation is obtained by iterating multiplication:
$Exp \ := \ [m:Nat]\,[n:Nat](<n\ Nat>\ \ (Times\ m)\ (S\ 0))$.

Iterating a natural on a functional type may produce non-primitive recursive functions; for instance we get Ackermann's function by diagonalization:
$Ack \ := \ [n:Nat](<n\ (Nat\Rightarrow Nat)>\ \ ([f:Nat\Rightarrow Nat]\,[m:Nat]\ (<m\ Nat>\ \ f\ m))\ S)$.

Indeed, most (total) recursive functions are definable as proofs in this formal system:

Theorem. (Girard [14]. See also [26]). Every recursive function provably total in second-order arithmetic is definable as a proof of type $Nat\Rightarrow Nat$ in the polymorphic λ-calculus.

10.3.4 Algebraic Programming

We may consider the polymorphic λ-calculus a powerful applicative programming language. It is both poorer than `ML` , in that no universal recursion operator is available, and richer, in that it provides a more complicated type structure. The price to pay is that there is no algorithm for synthesizing a principal type.

This language is revolutionary, in that it confuses *data structures* and *control structures*. Here, a data structure is but an unfulfilled control structure, waiting for more arguments to be able to "compute itself out". Thus to each of the data types seen above corresponds naturally a control structure. For *One* it is just the identity algorithm. For *Bool* it is the notion of conditional; that is, if $b:Bool$ and $M:A$, $N:A$ are the two branches of the conditional, the expression $If\ b\ Then\ M\ Else\ N$ may be implemented as $(<b\ A>\ \ M\ N):A$. For Nat, the polymorphic natural $n:Nat$ may be thought of as the construction `for i:= 1 to n do`. Compare this with `iterate n`, as defined in 1.1.1. Note that equality to zero is easily defined as:
$EqZero \ := \ [n:Nat](<n\ Bool>\ \ [b:Bool]\,False\ True)$.

As remarked above, the conjunction connective builds in product. Writing alternatively $A \times B$ for $A \wedge B$ as defined above, we get the pairing and projection algorithms as proofs of respectively $\wedge$-intro and $\wedge$-elim:
$Pair\ :=\ \Lambda A, B \cdot [x : A]\,[y : B]\Lambda C \cdot [h : A \Rightarrow B \Rightarrow C]\,(h\ x\ y)$
$Fst\ :=\ \Lambda A, B \cdot [x : A \times B](<x\ A>\ \ [u : A]\,[v : B]u)$
$Snd\ :=\ \Lambda A, B \cdot [x : A \times B](<x\ B>\ \ [u : A]\,[v : B]v)$
Thus, for instance, for any types A and B, $<Fst\ A\ B>: A \times B \Rightarrow A$, just as in `ML` .

However, the sum constructor is different: there is no analogue of the operators `outl` and `outr` here, since all the functions we may define are total:
$Case\ :=\ \Lambda A, B \cdot [x : A + B]\Lambda C \cdot [u : A \Rightarrow C]\,[v : B \Rightarrow C](<x\ C>\ \ u\ v)$
$Inl\ :=\ \Lambda A, B \cdot [x : A]\Lambda C \cdot [u : A \Rightarrow C]\,[v : B \Rightarrow C](u\ x)$
$Inr\ :=\ \Lambda A, B \cdot [x : B]\Lambda C \cdot [u : A \Rightarrow C]\,[v : B \Rightarrow C](v\ x)$

Primitive recursion

It is possible to represent standard program schemas by combinators. For instance, it is shown in [8] how to define simple primitive recursive schemes.

10.3.5 Ordinals

All the propositions (types) considered above are very simple, since they are restricted to degree 2.

With more complex types, we may define richer data structures. For instance, Th. Coquand [7] has shown how to define ordinal notations, as an extension of the naturals above. We just enrich Nat with a limit operation, which associates an ordinal to a sequence of ordinals, represented as a function of domain Nat. We define thus:
$Ord\ :=\ \forall A \cdot ((Nat \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow A) \Rightarrow A \Rightarrow A$
$Olim := [\sigma{:}Nat{\Rightarrow}Ord]\Lambda A{\cdot}[li{:}(Nat{\Rightarrow}A){\Rightarrow}A]\,[s{:}A{\Rightarrow}A]\,[z{:}A](li\ [n{:}Nat](<(\sigma\ n)\ A>\ \ li\ s\ z))$
$Osucc\ :=\ [\alpha : Ord]\Lambda A \cdot [li : (Nat \Rightarrow A) \Rightarrow A]\,[s : A \Rightarrow A]\,[z : A]\ (s\ (<\alpha\ A>\ \ s\ z))$
$Ozero\ :=\ \Lambda A \cdot [li : (Nat \Rightarrow A) \Rightarrow A]\,[s : A \Rightarrow A]\,[z : A]\ z.$
It is straightforward to coerce a natural into the corresponding ordinal, which defines the sequence of finite ordinals:
$Finite\ :=\ [n : Nat]\ (<n\ Ord>\ \ Osucc\ Ozero).$

Note that we instantiate the polymorphic natural n over type Ord. Thus the meaning of type quantification is to quantify over an arbitrary proposition definable in the calculus, and not simply over some totality circumscribed to the construction at hand. In other words, the calculus is inherently *non predicative*, and we are using this feature in an essential way.

The first transfinite ordinal, ω, may be simply obtained as limit of finite ordinals:
$\omega\ :=\ (Olim\ Finite).$

We may program over ordinals the same way we do with naturals:
$Oplus\ :=\ [\alpha : Ord]\,[\beta : Ord](<\beta\ Ord>\ \ Olim\ Osucc\ \alpha)$
$Otimes\ :=\ [\alpha : Ord]\,[\beta : Ord](<\beta\ Ord>\ \ Olim\ (Oplus\ \alpha)\ Ozero)$
$Oexp\ :=\ [\alpha : Ord]\,[\beta : Ord](<\beta\ Ord>\ \ Olim\ (Otimes\ \alpha)\ (Osucc\ Ozero)).$

Our ordinals are in fact *ordinal notations*, i.e. ordinals presented by fundamental sequences. In particular, $(Oplus\ (Osucc\ Ozero)\ \omega)$ and ω are two distinct constructions.

We may get the ordinal ϵ_0 as the iteration $(Oexp\ \omega\ (Oexp\ \omega \cdots))$:
$\epsilon_0\ :=\ (<\omega\ Ord>\ \ Olim\ (Oexp\ \omega)\ Ozero)$.

We may now use ordinals to define functional hierarchies. First, we give preliminary definitions concerning integer functions:
$Incr\ :=\ [f : Nat \Rightarrow Nat][n : Nat]\ (S\ (f\ n))$
$Iter\ :=\ [f : Nat \Rightarrow Nat][n : Nat]\ (n\ Nat\ f\ n)$
$Diag\ :=\ [\sigma : Nat \Rightarrow Nat \Rightarrow Nat][n : Nat]\ (\sigma\ n\ n)$.

Schwichtenberg's fast hierarchy may be defined as:
$Fast\ :=\ [\alpha : Ord](<\alpha\ (Nat \Rightarrow Nat)>\ \ Diag\ Iter\ Osucc)$
and the slow hierarchy is defined similarly (note that we just change the successor argument):
$Slow\ :=\ [\alpha : Ord](<\alpha\ (Nat \Rightarrow Nat)>\ \ Diag\ Incr\ Osucc)$.

It is to be noted that $(Fast\ \epsilon_0)$ is a total recursive function, but this fact is independent (i.e. undecidable) from Peano's arithmetic [12, 16].

Note that more powerful ordinals may be defined. For instance:
$BigOrd\ :=\ \forall A \cdot ((Ord \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow A) \Rightarrow A \Rightarrow A$
We may then use $BigOrd$'s in turn to define still more powerful sequences, defining $HugeOrd$, etc...
Let us call this family of ordinal notations the Scott family. Now consider the Plotkin ordinals:
$ORD\ :=\ \forall A \cdot (\forall B \cdot (B \Rightarrow A) \Rightarrow A) \Rightarrow (A \Rightarrow A) \Rightarrow A \Rightarrow A$.

Problem. Define the limit constructor :
$LimORD : \forall A \cdot (A \Rightarrow ORD) \Rightarrow ORD$. Consider now:
$RetractORD\ \ := <LimORDORD>: (ORD \Rightarrow ORD) \Rightarrow ORD$. Investigate the properties of $(RetractORD\ \ <Id\ ORD>) : ORD$.

References

[1] C. Böhm, A. Berarducci. "Automatic Synthesis of Typed Lambda-Programs on Term Algebras." Unpublished manuscript, (June 1984).

[2] L. Cardelli. "ML under UNIX." Bell Laboratories, Murray Hill, New Jersey (1982).

[3] L. Cardelli. "Amber." Bell Laboratories Technical Memorandum TM 11271-840924-10 (1984).

[4] L. Cardelli. "The Amber Machine." Bell Laboratories, Murray Hill, New Jersey (1985).

[5] L. Cardelli. "Basic Polymorphism Type-checking." Polymorphism, Jan. 1985

[6] A. Church "The Calculi of Lambda-Conversion." Princeton U. Press, Princeton N.J. (1941).

[7] Th. Coquand. "Une théorie des constructions." Thèse de troisième cycle, Université Paris VII (Jan. 85).

[8] Th. Coquand, G. Huet. "Constructions: A Higher Order Proof System for Mechanizing Mathematics." EUROCAL85, Linz, Springer-Verlag LNCS 203 (1985).

[9] D. Clément, J. Despeyroux, T. Despeyroux, G. Kahn. "Natural semantics on the computer." Research Report 416, Inria, June 1985.

[10] D. Clément, J. Despeyroux, T. Despeyroux, G. Kahn. "A Simple Applicative Language: mini-ML." Research Report to appear, Inria, 1986.

[11] L. Damas, R. Milner. "Principal type-schemes for functional programs." Proceedings of the ACM Conference on Principles of Programming Languages (1982) 207–212.

[12] S. Fortune, D. Leivant, M. O'Donnell. "The Expressiveness of Simple and Second-Order Type Structures." Journal of the Assoc. for Comp. Mach., **30,1**, (Jan. 1983) 151–185.

[13] J.Y. Girard. "Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. Proceedings of the Second Scandinavian Logic Symposium, Ed. J.E. Fenstad, North Holland (1970) 63–92.

[14] J.Y. Girard. "Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieure." Thèse d'Etat, Université Paris VII (1972).

[15] M. J. Gordon, A. J. Milner, C. P. Wadsworth. "Edinburgh LCF" Springer-Verlag LNCS **78** (1979).

[16] G. Kreisel "On the interpretation of nonfinitist proofs, Part I, II." JSL **16** (1952, 1953).

[17] N. McCracken. "An investigation of a programming language with a polymorphic type structure." Ph.D. Dissertation, Syracuse University (1979).

[18] R. Milner. "A Theory of Type Polymorphism in Programming." Journal of Computer and System Sciences **17** (1978) 348–375.

[19] R. Milner. "A proposal for Standard ML." Report CSR-157-83, Computer Science Dept., University of Edinburgh (1983).

[20] B. Nordström. "Description of a Simple Programming Language." Report 1, Programming Methodology Group, University of Goteborg (Apr. 1984).

[21] D. Prawitz. "Natural Deduction." Almqist and Wiskell, Stockolm (1965).

[22] J. C. Reynolds. "Towards a Theory of Type Structure." Programming Symposium, Paris. Springer Verlag LNCS **19** (1974) 408–425.

[23] J. C. Reynolds. "Types, abstraction, and parametric polymorphism." IFIP Congress'83, Paris (Sept. 1983).

[24] J. C. Reynolds. "Polymorphism is not set-theoretic." International Symposium on Semantics of Data Types, Sophia-Antipolis (June 1984).

[25] J. C. Reynolds. "Three approaches to type structure." TAPSOFT Advanced Seminar on the Role of Semantics in Software Development, Berlin (March 1985).

[26] R. Statman. "Number theoretic functions computable by polymorphic programs." 22nd IEE Symp. on Foundations of Computer Science (1981) 279–282.

[27] W. Tait. "A non constructive proof of Gentzen's Hauptsatz for second order predicate logic." Bull. Amer. Math. Soc. **72** (1966).

[28] D.A. Turner. "Miranda: A non-strict functional language with polymorphic types." In Functional Programming Languages and Computer Architecture, Ed. J. P. Jouannaud, Springer-Verlag LNCS 201 (1985) 1–16.

Chapter 11

Constructive Type Theory

11.1 The Calculus of Constructions

11.1.1 Designing a higher-order system

The first step consists in extending the polymorphic λ-calculus in order to allow the binding of proposition *schemas*. This permits the definition of propositional connectives inside the formalism. For instance, in polymorphic λ-calculus, we defined $\wedge$ at the level of the meta-notation: $\wedge$ was just a macro of the meta language expanding into a proposition of the formal system. Now we want to be able to write $\wedge$ as a combinator internally.

Next we abstract on such propositional connectives, leading to a higher-order propositional calculus. The first problem we encounter is a notational one. We shall have to distinguish between the proposition schemas, where some variable is functionally abstracted, and the propositions where the same variable is universally quantified.

Convention. We shall keep the square brackets for functional abstraction, and use parentheses for universal quantification, using the traditional notation $(x : A)M$.

The second extension consists in adding a first-order part, allowing quantification and abstraction on "elements". The natural question to investigate is: what are we going to choose as the types of the elements? The simplest decision is to follow once more the Curry-Howard paradigm: we already have the proofs, as elements of the types the propositions. This gives us not only 1st-order logic, but higher-order logic as well, since an implication will play the role of a functional type, and thus we encompass Church's theory of types just because we shall have intuitionistic propositional calculus as a sub-system of the propositions. We may wonder why it is legitimate to use the proofs as elements: aren't we pre-supposing some structure of our domains? Actually not, since the proofs are the bare bones of a functional type system: they are nothing more and nothing less than the λ-expressions of the right type.

Let us thus assume that we have propositions closed under quantification $(x : P)Q$ and abstraction $[x : P]Q$. The first remark is that implication becomes a derived notion: $P \Rightarrow Q$ is just a notational variant for $(x : P)Q$ in the special case when x does not occur in Q.

11.1.2 A first formalization

Let us now introduce explicitly a constant $Prop$ for the type of propositions. At the level of proofs $[P : Prop]M$ gives us what we wrote previously $\Lambda P \cdot M$. Similarly, quantifying a proposition over

$Prop$, as in $(P : Prop)Q$, gives us what we wrote previously $\forall P \cdot Q$. This suggests unifying also the notation $< M\ P >$ with $(M\ N)$. We thus arrive at a very simple calculus.

The types of proposition schemas are formed by quantification over the constant $Prop$. Let us use the constant $Type$ for denoting all the types of the system, that is the propositions plus the quantifications over $Prop$.

In all the following rules, Γ is assumed to be a valid context, where the rules for valid contexts are:

- The *initial context* $\Gamma_0 = [Prop : Type]$ is valid.
- If Γ is a valid context which does not bind variable x and $\Gamma \vdash T : Type$ then $\Gamma[x : T]$ is a valid context.
- If Γ is a valid context which does not bind variable t then $\Gamma[t : Type]$ is a valid context.

The first rule concerns accessing variables in a context:

$$Var : \frac{[x : T] \in \Gamma}{\Gamma \vdash x : T}.$$

The above rule is shorthand for $\Gamma \vdash Var(k) : T^{+(k-1)}$ when $\Gamma_k = [x : T]$.

We now generalize proposition formation to type formation, as follows:

$$Inclusion : \frac{\Gamma \vdash P : Prop}{\Gamma \vdash P : Type}$$

$$Product : \frac{\Gamma \vdash P : Type \quad \Gamma[A : P] \vdash M : Type}{\Gamma \vdash (A : P)M : Type}$$

$$Quant : \frac{\Gamma \vdash P : Type \quad \Gamma[A : P] \vdash M : Prop}{\Gamma \vdash (A : P)M : Prop}.$$

Finally, we have term formation rules:

$$Abstr : \frac{\Gamma \vdash T : Type \quad \Gamma[x : T] \vdash P : Type \quad \Gamma[x : T] \vdash M : P}{\Gamma \vdash [x : T]M : (x : T)P}$$

$$Appl : \frac{\Gamma \vdash M : (x : T)P \quad \Gamma \vdash N : T}{\Gamma \vdash (M\ N) : P\{N\}}.$$

Remarks. The constant $Type$ is a "type of all types". However, it is not itself of type $Type$. Note also that there is a redundancy between the rules *Product* and *Quant*. We could formulate the system without this redundancy, but it is convenient to keep this presentation because it is compatible with further extensions.

Definitions. Let $\Gamma \vdash M : N$, with Γ a valid context. When $N = Type$, we say that M is a valid Γ-*type*. When $N = Prop$, we say that M is a valid Γ-*proposition*. The valid Γ-types of the form $(A_1 : P_1)(A_2 : P_2) \cdots (A_n : P_n)Prop$ are called Γ-*products*. Finally, when $\Gamma \vdash M : N$, with N a valid Γ-proposition, we say that M is a Γ-*element*. The *pure* system of Constructions is obtained by deleting the third rule of context formation, which allows the introduction of $Type$ variables. In the pure system, the only primitive type is $Prop$, and thus the only valid types are the products and the propositions.

We shall use a number of abbreviations. First, we write $\vdash M : P$ for $\Gamma_0 \vdash M : P$. Then, we give notations for the non-dependent products, that is for terms $(u : P)Q$ in the case where u does not occur in Q. When both P and Q are propositions, we write $P \Rightarrow Q$. In other cases we write rather $P \rightarrow Q$. Finally, we abbreviate $(A : Prop)M$ into $\forall A \cdot M$ and $[A : Prop]M$ into $\Lambda A \cdot M$.

11.1.3 Adding type conversion

In the polymorphic λ-calculus seen in the last chapter, we defined propositional connectives as abbreviations. Thus for propositions P and Q, the notation $P \wedge Q$ was just a meta-linguistic notation for the appropriate proposition. In the new calculus under consideration, connectives are indeed definable as expressions, and propositions are formed using the general rules of λ-calculus. We should therefore expect to need internal reduction rules for playing the rôle of macro-expansion.

It is indeed the case that such rules are necessary for type-checking. For instance, let us assume we define conjunction along the ideas of the previous chapter:

$$\wedge \ := \ [P : Prop]\,[Q : Prop]\,(R : Prop)(P \Rightarrow Q \Rightarrow R) \Rightarrow R.$$

Now if we try to define the first projection, in a context

$$\Gamma = [P : Prop]\,[Q : Prop]\,[x : (\wedge\ P\ Q)],$$

we shall be unable to form the term $(x\ P)$, unless we are able to recognize that the type $(\wedge\ P\ Q)$ is equal (by β-conversion) to $(R : Prop)\cdots$.

The above discussion shows that some amount of type equality rules must be provided in a higher-order calculus. To what extent such rules should be explicit (from the point of view of a user checking a derivation using inference rules) is unclear. For instance, we may profit from meta-theoretical results (confluence, strong normalization) and convert all types to normal form using λ-calculus reduction rules. Now type equality is just identity of such canonical forms. But there is an obvious drawback here: we may spend useless time converting to normal form some types which could be recognized as different immediately by inspection of their head normal form. Thus $[u : A]\,[v : A](u\ \cdots)$ and $[u : A]\,[v : A](v\ \cdots)$ need not be reduced any further. This problem is aggravated by the fact that the higher-order nature of the calculus makes it possible to have subparts of types which are elements. For instance, if P is a predicate over a propositional type A, then for any element $p : A$ we may have to convert p to q in order to apply $x : (P\ p)$ as argument to a proof of some lemma of type $(P\ q) \Rightarrow \cdots$.

We now present various rules of conversion which may be used to axiomatize type equality $\cong$. Various sub-calculi are obtainable by taking a subset of these rules, together with the rule of type conversion:

$$Type\ Equality : \frac{\Gamma \vdash M : P \qquad \Gamma \vdash P \ \cong\ Q}{\Gamma \vdash M : Q}$$

$$Refl : \frac{\Gamma \vdash M : N}{\Gamma \vdash M \ \cong\ M}$$

$$Sym : \frac{\Gamma \vdash M \ \cong\ N}{\Gamma \vdash N \ \cong\ M}$$

$$Trans : \frac{\Gamma \vdash M \ \cong\ N \qquad \Gamma \vdash N \ \cong\ P}{\Gamma \vdash M \ \cong\ P}$$

$$Abseq : \frac{\Gamma \vdash P_1 \ \cong\ P_2 \qquad \Gamma\,[x : P_1] \vdash M_1 \ \cong\ M_2}{\Gamma \vdash [x : P_1]\,M_1 \ \cong\ [x : P_2]\,M_2}$$

$$Quanteq : \frac{\Gamma \vdash P_1 \ \cong\ P_2 \qquad \Gamma\,[x : P_1] \vdash M_1 \ \cong\ M_2}{\Gamma \vdash (x : P_1)M_1 \ \cong\ (x : P_2)M_2}$$

$$Appleq : \frac{\Gamma \vdash (M\ N) : P \qquad \Gamma \vdash M\ \cong\ M_1 \qquad \Gamma \vdash N\ \cong\ N_1}{\Gamma \vdash (M\ N)\ \cong\ (M_1\ N_1)}$$

$$Beta : \frac{\Gamma\,[x:A] \vdash M : P \qquad \Gamma \vdash N : A}{\Gamma \vdash ([x:A]M\ N)\ \cong\ M\{N\}}$$

$$Eta : \frac{\Gamma \vdash M : P}{\Gamma \vdash [x:A](M^+\ x)\ \cong\ M}.$$

Various subsystems can now be discussed. First, the rule *Eta* may be omitted. Then the rule *Abseq* (corresponding to the ξ rule of λ-calculus) may be deleted, yielding a weak conversion system corresponding to combinatory conversion.

Finally, when the conversions at the level of the elements (i.e. the terms of type a proposition) are omitted, we get the *restricted* calculus of constructions.

The calculus of constructions presented above was defined in Th. Coquand's thesis [18], who proved the main meta-theoretic properties. Variations on the basic calculus are presented in [22, 23].

11.1.4 Example

We want to define the intersection of a class of classes on a given type A. A natural attempt is to take

$$Inter \quad := \quad [\mathcal{C} : (A \rightarrow Prop) \rightarrow Prop]\,[x : A](P : A \rightarrow Prop)(\mathcal{C}\ P) \rightarrow (P\ x).$$

Let us place ourselves in the context

$$\Gamma = [\mathcal{C}_0 : (A \rightarrow Prop) \rightarrow Prop]\,[P_0 : A \rightarrow Prop]\,[p_0 : (\mathcal{C}_0\ P_0)].$$

We shall build a proof of the inclusion of the predicate $(Inter\ \mathcal{C}_0)$ in the predicate P_0. Let us consider

$$\Delta = \Gamma\,[x : A]\,[h : (Inter\ \mathcal{C}_0\ x)].$$

We want to build with p_0, x, h, P_0, $\mathcal{C}_0$ a term of type $(P_0\ x)$.

Intuitively, h which is of type $(Inter\ \mathcal{C}_0\ x)$ is also (by logical conversion using the definition of $Inter$) of type $(P : A \rightarrow Prop)(\mathcal{C}_0\ P) \Rightarrow (P\ x)$, and thus we may take the term $(h\ P_0\ p_0)$. Now, taking:

$$Subset := [P : A \rightarrow Prop]\,[Q : A \rightarrow Prop](x : A)P(x) \Rightarrow Q(x)\ :\ (P : A \rightarrow Prop)(Q : A \rightarrow Prop)Prop,$$

we get

$$\Gamma \vdash [x : A]\,[h : (Inter\ \mathcal{C}_0\ x)](h\ P_0\ p_0) : (Subset\ (Inter\ \mathcal{C}_0)\ P_0).$$

This example shows that the conversion of types rules are absolutely needed as soon as one wants to develop mathematical proofs (note that this example can be developed in the restricted calculus as well as in the full calculus). The need for conversion rules is equally emphasized in [46] and [62].

11.1.5 A few meta-theoretic properties

In the following statements, the meta-variable E denotes an arbitrary judgement (which may be of the form $M : P$ or $M \cong N$). We omit the proofs, which proceed by induction on the derivation.

Lemma 1. If $\Gamma \vdash E$, then every prefix of Γ is valid.

Lemma 2. If $\Gamma[x : P]\Delta \vdash E$ and $\Gamma \vdash M : P$, then $\Gamma[M/x]\Delta \vdash [M/x]E$.

Lemma 3. If $\Gamma \vdash M : P$ and $\Gamma \vdash M \cong N$, then $\Gamma \vdash N : P$.

Lemma 4. If $\Gamma \vdash M : N$ in the pure calculus, then N is a Γ-product or a Γ-proposition.

Thus, the only types are propositions and products, and the type of a valid term is a valid term. Finally, we may show that types are unique, up to conversion:

Lemma 5. If $\Gamma \vdash M : N_1$ and $\Gamma \vdash M : N_2$, then $\Gamma \vdash N_1 \cong N_2$.

For the **restricted** calculus, we can show directly that the relation $\Gamma \vdash E$ between contexts and judgements is decidable in the restricted pure calculus of constructions.

The proof in all details is rather long, but the main idea is simple, and its development straightforward. One defines first the notion of *reduction* $\triangleright$ associated to our notion of conversion, similarly to what we saw in the previous chapters. Then the usual argument of normalisation for the (simply) typed λ-calculus applies, with the notion of *complexity* of a term defined as follows.

Definition. The logical *rank* $\delta(M)$ of a term M is defined by the inductive rules

1. $\delta(M) \;=\; 0$, if M is not a product type
2. $\delta(Prop) \;=\; 1$
3. $\delta((x : M)N) \;=\; max(\delta(M) + 1, \delta(N))$.

Lemma 6. If $\Gamma \vdash M \cong N$, then $\delta(M) = \delta(N)$.

This lemma shows that all the types of a constructed object have the same rank, and it allows the definition:

Definition. Let $\Gamma \vdash (([x : P]\ M\ N) : \mathcal{T}$, with T a product type. We say that we have a *logical redex*, of *complexity* the rank $\delta(P)$.

The complexity of a valid term of the restricted calculus is then defined as the multiset of complexities of all its logical redexes. This complexity decreases by innermost reduction, whence the existence of a normal form, and the decidability of the conversion relation. The normalisation property of $\triangleright$ entails the decidability of $\Gamma \vdash E$.

Decidability Theorem. Given Γ and M, it is decidable in the restricted calculus whether or not there exists a term N such that $\Gamma \vdash M : N$. Furthermore, if the answer is positive, we can compute effectively such an N.

The proof is an induction on the sum of the length of M and the length of Γ, as in [43].

The reduction rules for logical redexes correspond to the notion of instantiation for predicate variables (see [65] for a more traditional presentation). Strong normalisation also holds, and this is also provable analogously to the simply typed λ-calculus.

For the full calculus, the decidability property still holds, but its proof is harder, since we need the normalisation property for all constructed terms, since arbitrary elements can appear in the types.

11.2 A realisability interpretation

11.2.1 Stripping

We shall now show how to extract from a given proof (i.e. a given functional) its associated pure (non-typed) λ-term which represents in some way its computational contents. All this is a generalisation of the usual realisability concept, but we use λ-terms instead of Gödel's codes for recursive functions. This can be done for the full calculus as well as for the restricted calculus.

11.2.2 The context contraction map

Let Γ be a valid context. We shall distinguish in Γ the quantifications over products from the quantifications over propositions, since only the latter will be considered free variables of stripped formulas. The quantifications over products are used solely at compile-time, for polymorphic type-checking.

Definition. The number of parameters, or *arity*, α_Γ and the canonical injection $j_\Gamma : \alpha_\Gamma \to |\Gamma|$ of a context Γ are determined by the following inductive rules (confusing n with $\{1, ..., n\}$):

$$\alpha_{\Gamma_0} = 0 \quad j_{\Gamma_0} = Id_0.$$

If $\Gamma = \Delta[x : M]$, then if M is a product, we take

$$\alpha_\Gamma = \alpha_\Delta, \quad j_\Gamma(k) = j_\Delta(k) + 1,$$

and if M is a proposition, we take

$$\alpha_\Gamma = \alpha_\Delta + 1, \quad j_\Gamma(1) = 1, \quad j_\Gamma(k+1) = j_\Delta(k) + 1.$$

11.2.3 Untyping

Definition. if $\Gamma \vdash M : N$, and N is a proposition, we define the *stripped* algorithm $\nu_\Gamma(M) \in \lambda^{\alpha_\Gamma}$ by induction on M:

1. If $M = Var(k)$, we take $\nu_\Gamma(M) = j_\Gamma^{-1}(k)$.

2. If $M = (M_1\ M_2)$, we know that $\Gamma \vdash M_1 : P_1$ and $\Gamma \vdash M_2 : P_2$. If P_2 is a proposition, we take $\nu_\Gamma(M) = (\nu_\Gamma(M_1)\ \nu_\Gamma(M_2))$, and if P_2 is a product, we take $\nu_\Gamma(M) = \nu_\Gamma(M_1)$. That is, we simply forget all type information, which is now viewed as a comment in the algorithm.

3. If $M = [x : P]N$, we know that $\Delta \vdash N : Q$, with $\Delta = \Gamma[x : P]$, and Q a Δ-proposition. Now if P is a proposition, we take $\nu_\Gamma(M) = \lambda\ x.\nu_\Delta(N)$, and if P is a product we take $\nu_\Gamma(M) = \nu_\Delta(N)$.

We shall usually write $\nu(M)$ instead of $\nu_\Gamma(M)$ when the context Γ is clear.

This λ-term $\nu(M)$ may be thought of as the computational contents of the proof M. The intuitive meaning of the above translation rules is that the propositions are comments of programs, and that those programs behave in a uniform way with respect to these comments.

11.2.4 A syntactic interpretation

We first need some notations: let $\mathcal{I}$ the set of all closed λ-terms, built on a special constant named Ω. Let SN be the set of strongly normalizable terms of $\mathcal{I}$. We define the notion of saturated subset of SN as in the previous chapter, and we denote by $\mathcal{U}$ the set of such saturated subsets.

Definition. If $A \in \mathcal{U}$ and $F \in \mathcal{I} \to \mathcal{U}$, then the *dependent product* $\Pi(A, F)$ of A and F is the set $\{M \in \mathcal{I} \mid \forall N \in A\ (M\ N) \in F(x)\}$.

Intuitively, the elements of $\mathcal{I}$ are the programs and the elements of $\mathcal{U}$ the types. In the previous definition, F is a dependent type. We may then check that $\mathcal{U}$ has the following closure properties:

Lemma 7. $\mathcal{U}$ is closed under intersection of non empty families and under dependent product.

What follows is a realisability interpretation similar to the one defined in the previous chapter.

11.2.5 The functionality of a type

Definition. We define the *functionality* $\varphi(M)$ of a type M as follows. If M is a proposition, we take $\varphi(M) = \mathcal{I}$. For products, we take $\varphi(Prop) = \mathcal{U}$, and $\varphi([x : P]Q) = \varphi(P) \to \varphi(Q)$, the set of all applications from $\varphi(P)$ to $\varphi(Q)$.

This definition holds for the restricted calculus. In the full calculus, we would define of a product $\varphi([x : P]Q)$ as $\varphi(P) \to \varphi(Q)$, if P is a product, and if P is a proposition, as the set of all applications f from $\mathcal{I}$ to $\varphi(Q)$ such that $f(M) = f(N)$ if M and N are β-convertible.

The following lemma is true for both the restricted and the full calculus:

Lemma 8. If $\Gamma \vdash M \cong N$ then $\varphi(M) = \varphi(N)$.

Definition. If Γ is any valid context, $\Gamma = [x_n : A_n] \dots [x_1 : A_1]$, then the *environment* associated with Γ is the product $\vec{\varphi}(\Gamma) = \varphi(A_n) \times \cdots \times \varphi(A_1)$.

11.2.6 Interpretation of sequents

Let $\Gamma \vdash M : N$ be a derived sequent, with N a type. We shall interpret it as an application $\rho_\Gamma(M) : \vec{\varphi}(\Gamma) \to \varphi(N)$.

There are two cases, according to whether N is a proposition or a product.

When N is a proposition, let us consider the pure λ-term $\nu_\Gamma(M)$. It has α_Γ free variables, and may thus be interpreted as a function $\overline{\nu_\Gamma(M)} : \mathcal{I}^{\alpha_\Gamma} \to \mathcal{I}$, which simply substitutes its actual arguments to the corresponding free variables. Furthermore, to the previously defined type forgetting operation j_Γ corresponds the projection $\pi_\Gamma : \vec{\varphi}(\Gamma) \to \mathcal{I}^{\alpha_\Gamma}$. We then define $\rho_\Gamma(M)$ as $\overline{\nu_\Gamma(M)} \circ \pi_\Gamma$.

When N is a product, we define $\rho_\Gamma(M)$ by induction on the derivation of the sequent $\Gamma \vdash M : N$ as follows.

- **product formation**: $\Gamma \vdash [x : M_1]M_2 : Prop$ results from $\Gamma[x : M_1] \vdash M_2 : Prop$. Let $\Delta = \Gamma[x : M_1]$. We have two subcases according to whether M_1 is a proposition or a product

or not:
subcase 1: M_1 is a proposition. Then by induction we can compute $f = \rho_\Gamma(M_1)$ and $g = \rho_\Delta(M_2)$. Thus $f : \vec{\varphi}(\Gamma) \to \mathcal{U}$ and $g : \vec{\varphi}(\Gamma) \times \mathcal{I} \to \mathcal{U}$. We define $\rho_\Gamma([x : M_1]M_2)$ as the function from $\vec{\varphi}(\Gamma)$ mapping a to $\Pi(f(a), g(a))$.
subcase 2: M_1 is a product, then by induction we can compute $f = \rho_\Delta(M_2)$ so that $f : \vec{\varphi}(\Gamma) \times \varphi(M_1) \to \mathcal{U}$. We define then $\rho_\Gamma([x : M_1]M_2)$ as the function from $\vec{\varphi}(\Gamma)$ mapping a to $\cap\{f(a,x) \mid x \in \varphi(M_1)\}$.

- **variable**: we have $\Gamma \vdash Var(k) : \Gamma_k$, with $k \leq |\Gamma| = n$. Then, $\rho_\Gamma(M)$ is simply the projection mapping $(x_n, ..., x_1)$ to x_k.

- **abstraction**: $\Gamma \vdash [x : M_1]M_2 : (x : M_1)P$ results from $\Gamma[x : M_1] \vdash M_2 : P$ by abstraction. Let $\Delta = \Gamma[x : M_1]$. By induction, we can compute $\rho_\Delta(M_2)$, which is an application from $\vec{\varphi}(\Gamma) \times \varphi(M_1)$ to $\varphi(P)$. We then define $\rho_\Gamma((\lambda x : M_1)M_2)$ as the application from $\vec{\varphi}(\Gamma)$ to $\varphi(M_1) \to \varphi(P)$ mapping a to the function mapping x to $f(a,x)$.

- **application**: $\Gamma \vdash (M\ N) : Q\{N\}$ results from $\Gamma \vdash M : [x : P]Q$ and $\Gamma \vdash N : P$. By induction, we have defined $\rho_\Gamma(M) : \vec{\varphi}(\Gamma) \to (\varphi(P) \to \varphi(Q))$ and $\rho_\Gamma(N) : \vec{\varphi}(\Gamma) \to \varphi(P)$. We then define $\rho_\Gamma((M\ N))$ as the application from $\vec{\varphi}(\Gamma)$ to $\varphi(Q)$ mapping x to $\rho_\Gamma(M)(x, \rho_\Gamma(N,x))$.

We have not taken into account the conversion rules, and this is justified by lemma 8.

Examples. The sequent $\vdash [A : Prop][x : A]A : Prop$ is interpreted as the set:

$$\{M \in \mathcal{I} \mid \forall A \in \mathcal{U}\ \forall N \in A\ (M\ N) \in A\}.$$

The sequent $\vdash [A : Prop][x : A]x : \forall A \cdot A \Rightarrow A$ is interpreted as the untyped λ-term $[x]x$.

Lemma 9. Let M and N be terms such that $\Gamma \vdash M \cong N$. We have $\rho_\Gamma(M) = \rho_\Gamma(N)$.

This lemma holds for the restricted calculus. Similarly, in the full calculus, if $\Gamma \vdash M \cong N$, $\rho_\Gamma(M)$ and $\rho_\Gamma(N)$ are two β-convertible λ-terms.

11.2.7 Interpretation of contexts

With each valid context Γ, we shall associate an inclusion $D(\Gamma) \hookrightarrow \vec{\varphi}(\Gamma)$ by induction on the formation of Γ:

- **case 1** : $D(\Gamma_0) = \vec{\varphi}(\Gamma_0) = 1$.

- **case 2** : $\Gamma = \Delta[x : M]$, with M a proposition. Since $\Delta \vdash M : Prop$, we have already defined $\rho_\Delta(M)$. By induction, we have already an inclusion $D(\Delta) \hookrightarrow \vec{\varphi}(\Delta)$. We take $D(\Gamma) = \{(a,x) \mid a \in D(\Delta)\ \wedge\ x \in \rho_\Delta(M)(a)\}$.

- **case 3** : $\Gamma = \Delta[x : M]$, with M a product. By induction, we have an inclusion $D(\Delta) \hookrightarrow \vec{\varphi}(\Delta)$ and we take $D(\Gamma) = \{(a,x) \mid a \in D(\Delta)\ \wedge\ x \in \varphi(M)\}$.

Example : $\Gamma_0[A : Prop][x : A]$ is interpreted as $\{(A,x) \in \mathcal{U} \times \mathcal{I} \mid x \in A\}$.

11.2.8 Consistency

We can now state the principal theorem, whose proof is simply a straightforward (but somehow tedious) structural induction and which holds in the restricted and in the full calculus.

Consistency Theorem : If $\Gamma \vdash M : P$, and $\Gamma \vdash P : Prop$ then for all x in $D(\Gamma)$, the pure λ-term $\rho_\Gamma(M,x)$ is an element of the saturated set $\rho_\Gamma(P,x)$.

Example : We have $[A : Prop][x : A] \vdash x : A$, then, with $\Gamma = \Gamma_0[A : Prop][x : A]$ we have $D(\Gamma) = \{(A,x) \in \mathcal{U} \times \mathcal{I} \mid x \in A\}$ and $[A : Prop][x : A] \vdash x : A$ is interpreted as $f : \mathcal{U} \times \mathcal{I} \to \mathcal{I}$ mapping (A,x) to x. Similarly $[A : Prop][x : A] \vdash A : Prop$ is interpreted as $g : \mathcal{U} \times \mathcal{I} \to U$ mapping (A,x) to A, and we see that $f(A,x) \in g(A,x)$ if $(A,x) \in D(\Gamma)$.

Corollary 1. If $\vdash M : N$ and N is a proposition, then $\rho_{\Gamma_0}(M)$ is a strongly normalisable closed pure λ-term.

Let $\rho = \rho_{\Gamma_0}$. It is sufficient to note that $\rho(M)$ is an element of $\rho(N)$, by the previous theorem, and $\rho(N)$ belongs to $\mathcal{U}$ by construction. By the definition of $\mathcal{U}$, we see that $\rho(M)$ is strongly normalisable.

Definition. A proposition $\vdash P : Prop$ is *inhabited* if, and only if there is an element term M such that $\vdash M : P$.

Corollary 2. The Calculus of Constructions is consistent, in the sense that there exists a proposition which is not inhabited.

The intuitive meaning of this statement is that the calculus does not prove all its well-formed propositions. Indeed, the term $\perp := (A : Prop)A$ is a valid Γ_0-proposition, but the special constant Ω appears in all the terms of $\rho(\perp)$. But if $\vdash M : N$ then Ω does not appear in the term $\rho(M)$, hence the corollary.

The realisability interpretation we have presented is syntactic in nature. However, it is consistent with the set-theoretical intuition of interpreting $M : P$ as $M \in P$. Still, the functional spaces $M \to N$ are not interpreted as the full function space, but only as sets of definable algorithms, closed by the operations corresponding to the syntactic operators. We know from Reynolds' work that a complete set-theoretic semantics cannot exist in extensions of the polymorphic λ-calculus [59].

Other interpretations of the calculus are possible. For instance, the Boolean interpretation, where each proposition is mapped to 0 or $1 = \{0\}$, and the elements are mapped to 0, is simpler and suffices for proving the consistency. In some sense, this is the "proof-irrelevance" interpretation of classical logic.

It is also possible to interpret the calculus in domains such as $P\omega$, where each object (proposition or element) is mapped to an element of $P\omega$, in such a way that propositions become closures [48]. However, such models also provide an interpretation for logically inconsistent systems (with $Type{:}Type$) [12, 4]. Thus, such interpretations fail to capture the essential feature of the calculus.

11.2.9 Extracting programs from proofs

Every proof construction $\Delta \vdash M : P$ corresponds to an algorithm $\nu_\Delta(M)$. Intuitively, this algorithm obeys proposition P considered as its specification, under the hypothesis on its α_Δ inputs described by Δ. This algorithm, a pure λ-expression in λ^{α_Δ}, always terminates *for well-typed values of its inputs*. This is the main limitation of our calculus as far as its programming language character goes. However, almost all partial recursive functions are definable in the calculus. For instance, all total recursive functions which are provably total in higher order arithmetic are definable, as shown in Girard [26]. They correspond to the stripped proofs of the proposition $Nat \Rightarrow Nat$, with $Nat = \forall A \cdot (A \Rightarrow A) \Rightarrow (A \Rightarrow A)$.

As another example, we may consider the partial recursive function defined as:

$$f(n) = if\ n = 0\ or\ n = 1\ then\ 0\ else\ if\ even(n)\ then\ f(n/2)\ else\ f(3n+1).$$

This function is easily definable in the calculus, as a proof of $(n : Nat)(D\ n) \Rightarrow Nat$, with the domain D defined as the proper smallest predicate preserving termination of f, that is $(D\ n)$ is:

$$(P : Set_{Nat})(P\ 0) \Rightarrow (P\ 1) \Rightarrow ((u : Nat)(P\ u) \Rightarrow (P\ 2u)) \Rightarrow ((u : Nat)(P\ 3u{+}2) \Rightarrow (P\ 2u{+}1)) \Rightarrow (P\ n),$$

where Set_{Nat} is an abbreviation for $Nat \rightarrow Prop$. Note that here nothing tells us that f is total on non-negative integers. If some day a proof of that fact is known, we shall get f as an algorithm in $Nat \rightarrow Nat$ by feeding it this proof as the $(D\ n)$ argument. This example is especially simple, since the domain argument is redundant for the computation. For more complicated examples, the domain argument may be needed, since its proof may describe the recursion structure.

Of course the above discussion on recursion extends to inductive definitions on any data type.

We may thus consider this calculus as a general formalism in which to develop programs consistently with their specifications. The logic is strong enough to articulate arbitrarily complex algorithmic specifications, as well as the more mundane standard data-types found in usual programming languages [21].

11.3 Examples of constructions

All the examples discussed in the previous chapter can be developed without modification in this new calculus, which incorporates in a natural way the polymorphic λ-calculus as a subcase. Let us now show how quantifiers can be expressed in the calculus.

11.3.1 Universal Quantification

Universal quantification, or general product, is implicit from the notation:

$$\Pi \ := \ \Lambda A \cdot [P : A \rightarrow Prop]\ (x : A)(P\ x).$$

Π-introduction, i.e. universal generalization, is proved by abstraction:

$$Gen \ := \ \Lambda A \cdot [P : A \rightarrow Prop] \Lambda B \cdot [f : (x : A)\ B \Rightarrow (P\ x)]\,[y : B]\,[x : A](f\ x\ y)$$

$$: \forall A \cdot (P : A \rightarrow Prop) \forall B \cdot ((x : A)\ B \Rightarrow (P\ x)) \Rightarrow (B \Rightarrow (\Pi\ A\ P)).$$

Similarly, Π-elimination is proved by instantiation, i.e. application:

$$Inst \ := \ \Lambda A \cdot [P : A \rightarrow Prop]\,[x : A]\,[p : (\Pi\ A\ P)](p\ x)$$

$$: \forall A \cdot (P : A \rightarrow Prop)(x : A)(\Pi\ A\ P) \Rightarrow (P\ x).$$

11.3.2 Existential Quantification

Existential quantification, or general sum, can be defined by a generalization of the binary sum:

$$\Sigma \ := \ \Lambda A \cdot [P : A \rightarrow Prop]\forall B \cdot ((x : A)\ (P\ x) \Rightarrow B) \Rightarrow B.$$

We leave if as exercise to the reader to prove existential introduction and elimination:

$$Exist \ := \ \forall A \cdot (P : A \rightarrow Prop)(x : A)\ (P\ x) \Rightarrow (\Sigma\ A\ P)$$

$$Witness \ := \ \forall A \cdot (P : A \rightarrow Prop)\ (\Sigma\ A\ P) \Rightarrow A.$$

Note that in a certain sense existential quantification is an abstraction mechanism: from $(\Sigma\ A\ P)$ it is possible to get some $a : A$ such that $(P\ a)$, but *not* the proof $p : (P\ A)$ that it indeed satisfies predicate P. Thus the existential quantification of the calculus of constructions is fundamentally different from the sum in Martin-Löf's calculus [47].

11.3.3 Equality

Leibniz' equality is definable in the calculus:

$$Equal \ := \ \Lambda A \cdot [x : A]\,[y : A](P : A \rightarrow Prop)(P\ x) \Rightarrow (P\ y).$$

Exercise. Define the properties for a polymorphic relation to be reflexive, symmetric and transitive. Give the three proofs that $Equal$ verifies these properties.

11.3.4 Tarski's theorem

Let us now present a simple example of a higher-order proof. The goal is to prove Tarski's theorem [66]:

Tarski's Theorem. A function monotonous over a complete partial ordering admits a fixpoint.

The first difficulty in formalizing Tarski's theorem is to give it in as abstract a setting as possible, in order to get the most direct proof. Let us try the following. Let A be a set, R a transitive relation over A which is complete, in the sense that every subset of A has a least upper bound. Let $f : A \rightarrow A$ be monotonously increasing. Then f admits a fixpoint.

We must now formalize the notions of set, subset, and fixpoint. A simple attempt at axiomatizing sets consists in assuming some type A given with an equality relation $=$, and to represent sets in the "universe" A by their characteristic predicate, i.e. as elements of type $A \rightarrow Prop$. As for fixpoint, it turns out that all we need to require is that for some X we have $(R\ (f\ X)\ X)$ and $(R\ X\ (f\ X))$. That is, the only property of equality that is needed here is the that R is anti-symmetric.

We thus assume that we are in a context Γ, containing the following hypotheses:
$[A : Type]$
$[= : A \rightarrow A \rightarrow Prop]$
$[R : A \rightarrow A \rightarrow Prop]$
$[Rtrans : (x : A)(y : A)(z : A)(R\ x\ y) \Rightarrow (R\ y\ z) \Rightarrow (R\ x\ z)]$
$[Rantisym : (x : A)(y : A)(R\ x\ y) \Rightarrow (R\ y\ x) \Rightarrow (=\ x\ y)]$
$[lim : (A \rightarrow Prop) \rightarrow A]$
$[Upperb : (P : A \rightarrow Prop)(y : A)(P\ y) \Rightarrow (R\ y\ (lim\ P))]$

[$Least : (P : A \rightarrow Prop)(y : A)((z : A)(P\ z) \Rightarrow (R\ z\ y)) \Rightarrow (R\ (lim\ P)\ y)$]
[$f : A \rightarrow A$]
[$Incr : (x : A)(y : A)(R\ x\ y) \Rightarrow (R\ (f\ x)\ (f\ y))$]

Now we consider the predicate Q defined as:

$$Q \ := \ [u : A](R\ u\ (f\ u))$$

(that is, Q is the set of pre-fixpoints of f) and the element $X : A$ defined as:

$$X \ := \ (lim\ Q).$$

The first part of the proof consists in showing a proof of $(R\ X\ (f\ X))$ in context Γ. Let us first consider $\Delta = \Gamma[y : A][h : (Q\ y)]$, and terms $M = (Upperb\ Q\ y)$ and $N = (Incr\ y\ X)$. We get:
$\Delta \vdash M : (R\ y\ (f\ y)) \Rightarrow (R\ y\ X)$, and:
$\Delta \vdash N : (R\ y\ X) \Rightarrow (R\ (f\ y)\ (f\ X))$. Composing the two proofs we get:
$\Delta \vdash M; N : (R\ y\ (f\ y)) \Rightarrow (R\ (f\ y)\ (f\ X))$.
Thus, taking $p = (M; N\ h)$, we obtain:
$\Delta \vdash (Rtrans\ y\ (f\ y)\ (f\ X)\ h\ p) : (R\ y\ (f\ X))$.
Discharging the hypotheses h and y, we get $T = [y : A][h : (Q\ y)](Rtrans\ y\ (f\ y)\ (f\ X)\ h\ p)$ such that:
$\Gamma \vdash T : \forall y \in Q \cdot (R\ y\ (f\ X))$.
The proof is completed by constructing $U = (Least\ Q\ (f\ X)\ T)$, since:
$\Gamma \vdash U : (R\ X\ (f\ X))$.

The second part of the proof is the converse. Taking $Z = (Incr\ X\ (f\ X)\ U)$, we get:
$\Gamma \vdash Z : (R\ (f\ X)\ (f\ (f\ X)))$
but since this last proposition converts to $(Q\ (f\ X))$, we get:
$\Gamma \vdash (Upperb\ Q\ (f\ X)\ Z) : (R\ (f\ X)\ X)$.

The proof of Tarski's theorem is thus obtained as:
$\Gamma \vdash (Rantisym\ (f\ X)\ X\ (Upperb\ Q\ (f\ X)\ Z)\ U) : (=\ (f\ X)\ X)$.

Exercise. Use the above argument and the quantifier manipulation combinators above to prove Tarski's theorem as a fully quantified statement.

Numerous examples of proofs verified on machine are presented in [21]. A general discussion on the formalization of mathematical arguments in higher order intuitionistic logic is given in [61].

11.4 A constructive theory of types

Let us now augment the calculus with rules allowing for the abstraction over all types. The first natural attempt is to allow $Type : Type$. We would thus get a system of rules very close to the one considered by P. Martin-Löf in [43]. However, this was shown to be inconsistent by Girard, who showed that it was possible to encode the paradox of Burali-Forti in such a system. An abstract analysis of such paradoxes is given by Coquand in [19]. Coquand showed that it was possible to quantify propositions over all types, but *not* other types such as product types. Such a system is presented below.

11.4.1 A system for uniform proofs

First, two rules provide for abstraction over all types:

$$TypeQuant : \frac{\Gamma\,[t : Type] \vdash P : Prop}{\Gamma \vdash (t : Type)P : Prop}$$

$$TypeAbstr : \frac{\Gamma\,[t : Type] \vdash P : Prop \quad \Gamma\,[t : Type] \vdash M : P}{\Gamma \vdash [t : Type]\,M : (t : Type)P}.$$

Finally, we give one more type convertion rule:

$$TypeEq : \frac{\Gamma\,[t : Type] \vdash P : Prop \qquad \Gamma\,[t : Type] \vdash P \;\cong\; Q}{\Gamma \vdash (t : Type)P \;\cong\; (t : Type)Q}.$$

In such a system, we may now abstract the above proof of Tarski's theorem.

11.4.2 A system with a hierarchy of universes

It is even possible to iterate the idea of a type gathering all the types obtained so far. One thus gets a system with a hierarchy of universes like in Martin-Löf's system [47]. See [19] for details.

11.5 An ML implementation

11.5.1 A system with normal types

It is important to clearly distinguish between the presentation of the construction calculus for a metamathematical study and its presentation for an implementation and the development of proofs and programs in this calculus. The presentation we have chosen here is the best suited for the proofs of the mathematical property of the calculus of constructions. But once we have these properties, it is possible to derive other presentations of the system. For example, since we have the normalisation property, it is possible to present the full calculus in such a way that all types appear in normal form. This essentially amounts to writing the rule for application as:

$$\frac{\Gamma \;\vdash\; M : (x : P)Q \quad \Gamma \;\vdash\; N : P}{\Gamma \;\vdash\; (M\ N) : \mathcal{N}(Q\{N\})},$$

where $\mathcal{N}(M)$ denotes the normal form of the term M.

This presentation avoids the conversion rules and thus seems a bit simpler (and it is the one used in [22]). But this system does not seem to be well suited for a metamathematical study.

11.5.2 Introducing constants

The first step toward providing a usable system consists in defining combinators which abbreviate definitions. These constants are given, in a context Γ, with a definition consisting of term M, and a unique name. We check before entering the constant in the theory that $\Gamma \vdash M : T$ for some type T. We may define propositional constants (when T is a product) or element constants (when T is a proposition). Later on the type checker retrieves the type of each constant by looking it up in the theory tables. This saves space (by sharing commonly used constructions) and time (by not re-checking similar constructions). These constant definitions can be internalized in the language by the "let" construct, where $let\ \ x\ =\ M_1\ \ in\ \ M_2$ abbreviates the redex $([x:P]M_2\ \ M_1)$ (with P the type of M_1). We can thus get "local" constants at any context depth.

No extension of the theory is required to explain the calculus with constants. The only problem is to implement an absolute naming scheme, orthogonal to de Bruijn's indexes considered so far, while preserving a notion of static scope. This problem is the logical analogue of the problem of linking separately compiled modules in a programming language. We do not comment further on this issue, but we remark that from a practical point of view this facility is crucial, since it would be impossible to effectively realize any significant proof without constants.

11.5.3 Synthesis of implicit arguments

The next step in providing the user with a realistic system in which to develop proofs is to reduce the burden of polymorphic instantiation. Many propositional arguments are redundant, since they may be inferred automatically as sub-components of types of further arguments. Thus a certain amount of type synthesis is possible without any non-deterministic search. Let us give a trivial example. In the following discussion, we shall confuse abstractions with products.

If one wants to define composition (i.e. the cut rule of propositional logic) in the basic calculus, we have to define the constant:

$$Comp\ :=\ [A:Prop][B:Prop][C:Prop][f:A\Rightarrow B][g:B\Rightarrow C][x:A](g\ (f\ x)).$$

This is very cumbersome, and if one assumes that $Comp$ is always used with all arguments up to g there is a lot of redundancy, since the actual arguments corresponding to A,B and C are necessary parts of the types of the actual arguments corresponding to f and g. The crucial observation is that certain parts of the terms will always have residuals in every reduction of every substitution instance of the term. This determines in the normal forms of types rigid skeletons in which one may access sub-components by pattern-matching. For instance, in $A \to B$, i.e. $(u:A)B$, we can use the whole term as a pattern in the free variables A and B. This method relies on the variant explained above of keeping types in normal form.

The notion of rigid skeleton was defined in [35] in the context of a unification algorithm for typed λ-calculus. Let us recall this notion. Let

$$M = [u_1:P_1]\cdots[u_n:P_n](x\ N_1\ \cdots\ N_p)$$

be a term in normal form. The variables of the head prefix may be bound by product rather than abstraction for the purpose of this discussion. Let V be a set of variables. We call *rigid occurrence* of M relative to V the set of following positions in M. First, we take the rigid occurrences in P_i relative to $V \cup \{u_1,...,u_{i-1}\}$, for $i=1,...,n$. Then, if $p=0$, the occurrence of the head variable x, and if $p>0$, and when $x \in W = V \cup \{u_1,...,u_n\}$, the rigid occurrences in N_j relative to W, for $j=1,...,p$. Now let z be any variable. We say that M *determines* z iff z appears in M at a rigid occurrence relative to $\emptyset$.

We are now able to explain how to declare combinators of the calculus given with an arity of *explicit* arguments, whose types determine automatically *implicit* arguments which will be automatically synthesized. In our example above, we would write:

$$Comp[A|Prop][B|Prop][C|Prop][f:A\Rightarrow B][g:B\Rightarrow C]\ :=\ [x:A](g\ (f\ x)),$$

where the bar | instead of the colon : indicate the implicit arguments. Now the combinator $Comp$ may be invoked with only its explicit actual arguments, as in $Comp(F,G)$. In the general situation, a declaration of a combinator with arguments $u_i : P_i$ will be legal iff for every implicit i there exists an argument $j > i$ such that P_j determines u_i. It is not mandatory that j be itself explicit, since the synthesis of implicit arguments may be iterated (from right to left).

Remark. It is possible to generalize this method, by computing recursively whether some functional argument determines some of its parameters. For instance, consider:

$$C \ := \ \texttt{[}P : A \rightarrow Prop\texttt{]}\,\texttt{[}x : A\texttt{]}\,\texttt{[}h : (P\ x)\texttt{]} \cdots$$

The occurrence of x in $(P\ x)$ is *not* rigid. However, if the actual first argument P_0 of a given application $(C\ P_0\ x_0\ h_0)$ is of the form $\texttt{[}u : A\texttt{]}M$ such that M has a rigid occurrence of u, then x_0 may be synthesized from the type of h_0; i.e. rigidity may be inherited. However, it is not yet clear how to specify such iterated synthesis in a clearly understandable way, since the notion of implicit argument is not bound to the definition of combinator C anymore, but rather varies dynamically with every use of C. A possibly useful restriction would be to impose in the definition of C that certain arguments ought to determine certain of their own parameters, using a syntax such as:

$$C\,\texttt{[}P : (u|A)Prop\texttt{]}\,\texttt{[}x|A\texttt{]}\,\texttt{[}h : (P\ x)\texttt{]} \ := \ \cdots$$

Now in an invocation $C(P_0, h_0)$, the analysis of P_0 will yield a rigid occurrence of its argument. The argument x may then be retrieved at the corresponding position in the type of h_0. This is in a way a natural extension of restrictions of λ-calculus expressibility at the proposition level, such as Church's use of λI-calculus, or relevance logic.

Note that the synthesis of implicit arguments corresponds exactly to the mathematical practice. For instance, in category theory, one writes Id_A, but $f \circ g$ is not annotated with objects, since the arrows f and g determine the proper composition from their domains and co-domains.

Finally, we stress that a certain sophistication in concrete syntax, i.e. in the way new notations may be associated to concepts by the user in the course of the development of a theory, is crucial if one wants to mechanize mathematical concepts beyond the attempts of Frege, the Principia and even Automath. Hopefully modern computer technology will help, and dynamically extendable parsers and complex window managers seem to be necessary components of user interfaces to programming and proving environments [16]. Let us just mention one proposal [22] for concrete syntax definition of combinators given with arities, which fits nicely with the above algorithm for synthesis of implicit arguments.

11.5.4 Concrete syntax

Since we now accept combinators with arities, we might as well endow them with concrete syntax. A straightforward device for declaring arbitrary mixfix notation is to allow the declaration of combinators by *patterns*:

$$pattern \ := \ term,$$

where *pattern* is an arbitrary sequence of concrete strings, implicit argument declarations $\{x : M\}$, and explicit argument declarations $\texttt{[}x : M\texttt{]}$. Standard methods such as precedence declarations may complement this basic mechanism to resolve ambiguities. For instance, we would now allow the declaration:

$$\texttt{[}A|Prop\texttt{]}\,\texttt{[}B|Prop\texttt{]}\,\texttt{[}C|Prop\texttt{]}\,\texttt{[}f : A \Rightarrow B\texttt{]} \circ \texttt{[}g : B \Rightarrow C\texttt{]} \ := \ \texttt{[}x : A\texttt{]}(g\ (f\ x)),$$

and be able to write in the usual manner $F \circ G$. Examples of development of mathematical notions along those lines are presented in [22].

More ambitiously, we may imagine incorporating theorem-proving capabilities progressively in what is initially an interactive proof-checker. We may synthesize whole constructions by systematic search of possible combinations of given sets of combinators. Such tacticals may be programmed in the meta language of the system, in the tradition of LCF [30] or Pearl [16]. This will offer a powerful help to the mathematician, who will be able to concentrate on the global proof strategy, i.e. on the proper ordering of lemmas, without losing time over the combinatory headaches of the technical proofs.

References

[1] P. B. Andrews. "Resolution in Type Theory." Journal of Symbolic Logic **36,3** (1971), 414–432.

[2] P. B. Andrews, D. A. Miller, E. L. Cohen, F. Pfenning. "Automating higher-order logic." Dept of Math, University Carnegie-Mellon, (Jan. 1983).

[3] H. Barendregt. "The Lambda-Calculus: Its Syntax and Semantics." North-Holland (1980).

[4] H. Barendregt and A. Rezus. "Semantics for Classical AUTOMATH and Related Systems." Information and Control **59** (1983) 127–147.

[5] E. Bishop. "Foundations of Constructive Analysis." McGraw-Hill, New-York (1967).

[6] E. Bishop. "Mathematics as a numerical language." Intuitionism and Proof Theory, Eds. J. Myhill, A.Kino and R.E.Vesley, North-Holland, Amsterdam, (1970) 53–71.

[7] N.G. de Bruijn. "The mathematical language AUTOMATH, its usage and some of its extensions." Symposium on Automatic Demonstration, IRIA, Versailles, 1968. Printed as Springer-Verlag Lecture Notes in Mathematics **125**, (1970) 29–61.

[8] N.G. de Bruijn. "Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem." Indag. Math. **34,5** (1972), 381–392.

[9] N.G. de Bruijn. "Automath a language for mathematics." Les Presses de l'Université de Montréal, (1973).

[10] N.G. de Bruijn. "Some extensions of Automath: the AUT-4 family." Internal Automath memo M10 (Jan. 1974).

[11] N.G. de Bruijn. "A survey of the project Automath." (1980) in to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, Eds Seldin J. P. and Hindley J. R., Academic Press (1980).

[12] L. Cardelli. "A Polymorphic λ-calculus with Type:Type." Private communication (1986).

[13] A. Church. "A formulation of the simple theory of types." Journal of Symbolic Logic **5,1** (1940) 56–68.

[14] A. Church. "The Calculi of Lambda-Conversion." Princeton U. Press, Princeton N.J. (1941).

[15] R.L. Constable, J.L. Bates. "Proofs as Programs." Dept. of Computer Science, Cornell University. (Feb. 1983).

[16] R.L. Constable, J.L. Bates. "The Nearly Ultimate Pearl." Dept. of Computer Science, Cornell University. (Dec. 1983).

[17] R.L. Constable, N.P. Mendler. "Recursive Definitions in Type Theory." Private Communication (1985).

[18] Th. Coquand. "Une théorie des constructions." Thèse de troisième cycle, Université Paris VII (Jan. 85).

[19] Th. Coquand. "An analysis of Girard's paradox." First Conference on Logic in Computer Science, Boston (June 1986).

[20] Th. Coquand, G. Huet. "A Theory of Constructions." Preliminary version, presented at the International Symposium on Semantics of Data Types, Sophia-Antipolis (June 84).

[21] Th. Coquand, G. Huet. "Constructions: A Higher Order Proof System for Mechanizing Mathematics." EUROCAL85, Linz, Springer-Verlag LNCS 203 (1985).

[22] Th. Coquand, G. Huet. "Concepts Mathématiques et Informatiques Formalisés dans le Calcul des Constructions." Colloque de Logique, Orsay (Juil. 1985).

[23] Th. Coquand, G. Huet. "The Calculus of Constructions." To appear, Information and Control (1986).

[24] D. Van Daalen. "The language theory of Automath." Ph. D. Dissertation, Technological Univ. Eindhoven (1980).

[25] G. Frege. "Begriffschrift, a formula language, modeled upon that of arithmetic, for pure thought." (1879). Reprinted in From Frege to Gödel, J. van Heijenoort, Harvard University Press, 1967.

[26] J.Y. Girard. "Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. Proceedings of the Second Scandinavian Logic Symposium, Ed. J.E. Fenstad, North Holland (1970) 63–92.

[27] J.Y. Girard. "Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieure." Thèse d'Etat, Université Paris VII (1972).

[28] K. Gödel. "Uber eine bisher noch nicht benutze Erweitrung des finiten Standpunktes." Dialectica, **12** (1958).

[29] W. D. Goldfarb. "The Undecidability of the Second-order Unification Problem." Theoretical Computer Science, **13**, (1981) 225–230.

[30] M. J. Gordon, A. J. Milner, C. P. Wadsworth. "Edinburgh LCF" Springer-Verlag LNCS **78** (1979).

[31] W. E. Gould. "A Matching Procedure for Omega Order Logic." Scientific Report 1, AFCRL 66-781, contract AF19 (628)-3250 (1966).

[32] G. Huet. "Constrained Resolution: a Complete Method for Type Theory." Ph.D. Thesis, Jennings Computing Center Report 1117, Case Western Reserve University (1972).

[33] G. Huet. "A Mechanization of Type Theory." Proceedings, 3rd IJCAI, Stanford (Aug. 1973).

[34] G. Huet. "The Undecidability of Unification in Third Order Logic." Information and Control **22** (1973) 257–267.

[35] G. Huet. "A Unification Algorithm for Typed Lambda Calculus." Theoretical Computer Science, **1.1** (1975) 27–57.

[36] L.S. Jutting. "A translation of Landau's "Grundlagen" in AUTOMATH." Eindhoven University of Technology, Dept of Mathematics (Oct. 1976).

[37] L.S. van Benthem Jutting. "The language theory of Λ_∞, a typed λ-calculus where terms are types." Unpublished manuscript (1984).

[38] J. Ketonen, J. S. Weening. "The language of an interactive proof checker." Stanford University (1984).

[39] J. Ketonen. "EKL-A Mathematically Oriented Proof Checker." 7th International Conference on Automated Deduction, Napa, California (May 1984). Springer-Verlag LNCS **170**.

[40] J. Ketonen. "A mechanical proof of Ramsey theorem." Stanford Univ. (1983).

[41] D. MacQueen, G. Plotkin, R. Sethi. "An ideal model for recursive polymorphic types." Proceedings, Principles of Programming Languages Symposium, Jan. 1984, 165–174.

[42] D. B. MacQueen, R. Sethi. "A semantic model of types for applicative languages." ACM Symposium on Lisp and Functional Programming (Aug. 1982).

[43] P. Martin-Löf. "A theory of types." Report 71-3, Dept. of Mathematics, University of Stockholm, Feb. 1971, revised (Oct. 1971).

[44] P. Martin-Löf. "About models for intuitionistic type theories and the notion of definitional equality." Paper read at the Orléans Logic Conference (1972).

[45] P. Martin-Löf. "An intuitionistic Theory of Types: predicative part." Logic Colloquium 73, Eds. H. Rose and J. Sepherdson, North-Holland, (1974) 73–118.

[46] P. Martin-Löf. "Constructive Mathematics and Computer Programming." In Logic, Methodology and Philosophy of Science **6** (1980) 153–175, North-Holland.

[47] P. Martin-Löf. "Intuitionistic Type Theory." Studies in Proof Theory, Bibliopolis (1984).

[48] N. McCracken. "An investigation of a programming language with a polymorphic type structure." Ph.D. Dissertation, Syracuse University (1979).

[49] D.A. Miller. "Proofs in Higher-order Logic." Ph. D. Dissertation, Carnegie-Mellon University (Aug. 1983).

[50] R.P. Nederpelt. "Strong normalization in a typed λ calculus with λ structured types." Ph. D. Thesis, Eindhoven University of Technology (1973).

[51] R.P. Nederpelt. "An approach to theorem proving on the basis of a typed λ-calculus." 5th Conference on Automated Deduction, Les Arcs, France. Springer-Verlag LNCS **87** (1980).

[52] B. Nordström. "Programming in Constructive Set Theory: Some Examples." Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture, Portmouth, New Hampshire (Oct. 1981) 141–154.

[53] B. Nordström, K. Petersson. "Types and Specifications." Information Processing 83, Ed. R. Mason, North-Holland, (1983) 915–920.

[54] B. Nordström, J. Smith. "Propositions and Specifications of Programs in Martin-Löf's Type Theory." BIT **24**, (1984) 288–301.

[55] L. C. Paulson. "Constructing Recursion Operators in Intuitionistic Type Theory." Tech. Report 57, Computer Laboratory, University of Cambridge (Oct. 1984).

[56] T. Pietrzykowski, D.C. Jensen. "A complete mechanization of ω-order type theory." Proceedings of ACM Annual Conference (1972).

[57] T. Pietrzykowski. "A Complete Mechanization of Second-Order Type Theory." JACM **20** (1973) 333–364.

[58] PRL staff. "Implementing Mathematics with the NUPRL Proof Development System." Computer Science Department, Cornell University (May 1985).

[59] J. C. Reynolds. "Polymorphism is not set-theoretic." International Symposium on Semantics of Data Types, Sophia-Antipolis (June 1984).

[60] D. Scott. "Constructive validity." Symposium on Automatic Demonstration, Springer-Verlag Lecture Notes in Mathematics, **125** (1970).

[61] D. Scott. "Identity and existence in intuitionistic logic." Proceedings of the Research Symposium on Applications of Sheaf Theory to Logic, Algebra and Analysis, Durham (July 1977), Eds. M.P. Fourman, C.J. Mulvey and D.S. Scott. Lecture Notes in Mathematics 753, Springer-Verlag (1979) 660–696.

[62] J.P. Seldin. "Progress report on generalized functionality." Ann. Math. Logic. 17 (1979).

[63] M. Takahashi. "A proof of cut-elimination theorem in simple type theory." J. Math. Soc. Japan **19** (1967).

[64] G. Takeuti. "On a generalized logic calculus." Japan J. Math. **23** (1953).

[65] G. Takeuti. "Proof theory." Studies in Logic **81** Amsterdam (1975).

[66] A. Tarski. "A Lattice-Theoretical Fixpoint Theorem and its Applications." *Pacific J. Math.* **5** (1955), 285–309.

[67] R. de Vrijer "Big Trees in a λ-calculus with λ-expressions as types." Conference on λ-calculus and Computer Science Theory, Rome, Springer-Verlag LNCS **37** (1975) 252–271.

Chapter 12

Induction

We shall now present various methods of proofs by induction, formalized in the calculus of constructions.

12.1 A constructive set theory

We assume a global context where we have declared: $[U : Type]$. We may think of U as the current *universe*. *Sets* defined over U are represented as predicates of type $U \rightarrow Prop$, which we abbreviate from now on as Set_U, or even as Set when U is clear from the context. This may be formally justified by the type synthesis algorithm sketched in the last chapter. If $A : Set_U$ is a U-set, we define $x \in A$ as the proposition $(A\ x)$. That is, the *elements* of U-sets are of type U. We abbreviate the quantification $(x : U)E$ as $\forall x \cdot E$, (there should be no confusion with our previous use of this notation for type quantification), and the abstraction $[x : U]E$ as $\{x \mid E\}$. For successive bindings, we use respectively $\forall x, y \cdot E$ and $\{x, y \mid E\}$.

We define inclusion of sets A and B by:

$$A \subseteq B \ := \ \forall x \cdot x \in A \Rightarrow x \in B$$

and set equality by:

$$A = B \ := \ A \subseteq B \wedge B \subseteq A.$$

Note that this extensional equality is distinct from Leibniz' intensional equality. Intensional U-equality may be defined, between two elements x and y of type U, as:

$$x \equiv y \ := \ (A : Set)\, x \in A \Rightarrow y \in A.$$

If we decided to give a primitive equality $=$ on type U this would complicate matters quite a bit, since we would have to state that sets are predicates compatible with this equality, i.e. such that $(P\ x)$ and $x = y$ imply $(P\ y)$, and iterate this condition with classes, etc...

The empty U-set is defined as:

$$\emptyset \ := \ \{x \mid \nabla\}.$$

The usual set operations are available through the corresponding logical connectives:

$$A \cap B \ := \ \{x \mid x \in A \wedge x \in B\}$$

$$A \cup B \ := \ \{x \mid x \in A \vee x \in B\}$$

$$\sim A \;:=\; \{x \mid \neg x \in A\}.$$

Remark. If we were completely formal, we should index all our notations with U, and write for instance $x \in_U A$, $\emptyset_U$, etc. We assume here no ambiguity arises as to which universe we are into.

Our sets resemble ordinary sets, except that the inclusion relation is defined constructively. Thus, we have $A \subseteq \sim\sim A$, but the converse in not true in general. That is, our sets behave more like open sets of a topological space, and classical sets are the analogue of closed sets, i.e. double-negation closed. The complement $\sim A$ of A is closed, and for every A we get its closure as $\sim\sim A$.

We now define classes as set predicates. That is, a U-class is of type $(Set_U \rightarrow Prop)$, abbreviated $Class_U$ or simply $Class$. For instance, the class of subsets of A is defined as:

$$(\mathcal{P}\ A) \;:=\; [B : Set]\ B \subseteq A.$$

Class inclusion may be defined in the same way as set inclusion. Actually, all sets operations above extend to class operations, since U may be instantiated with Set_U.

If C is a U-class, we define the *intersection* of C as the U-set defined as follows:

$$\cap C \;:=\; \{x \mid (A : Set)\ (C\ A) \Rightarrow x \in A\}.$$

For instance, we may define the singleton $\{x\}$ as follows:

$$\{x\} \;:=\; \cap([A : Set]\ x \in A).$$

We say that set A is *universal* if it contains all the objects of the universe:

$$(Universal\ A) \;:=\; \forall x \cdot x \in A.$$

A *mapping* maps a set to a set. More precisely, a U-map has type $Set_U \rightarrow Set_U$, abbreviated Map_U or simply Map. If φ is a U-map, we define:

$$(Stable\ \varphi) \;:=\; [A : Set](\varphi\ A) \subseteq A$$

and

$$(Fixpt\ \varphi) \;:=\; [A : Set](\varphi\ A) = A.$$

Note that these two constructions are of type $Class_U$. We now define the standard interpretation of map φ as the intersection of the class of sets for which φ is stable:

$$(Initial\ \varphi) \;:=\; \cap(Stable\ \varphi)$$

that is, in expanded form:

$$\{u \mid (A : Set)(\forall x \cdot x \in (\varphi\ A) \Rightarrow x \in A) \Rightarrow u \in A\}.$$

12.2 Induction

We get an induction principle by restricting ourselves to the standard interpretation:

$$(Induction\ \varphi) \;:=\; (A : Set)(Stable\ \varphi\ A) \Rightarrow (Universal\ A),$$

or, in an equivalent expanded formulation:

$$(A : Set)(\forall x \cdot x \in (\varphi\ A) \Rightarrow x \in A) \Rightarrow \forall u \cdot u \in A.$$

Note that $Initial$ and $Induction$ are really the same construction, up to permutation of independent hypotheses: the binding on u migrated from the outermost abstraction in $(Initial\ \varphi)$ to the innermost quantification in $(Induction\ \varphi)$.

This notion is especially important when φ is an *increasing* map:

$$(Incr\ \varphi)\ :=\ (A : Set)(B : Set)\ A \subseteq B \Rightarrow (\varphi\ A) \subseteq (\varphi\ B)$$

since then we may apply Tarski's theorem above, and thus consider $(Initial\ \varphi)$ as the least solution to φ considered as a recursive definition. Let us check the details.

In Tarski's theorem as presented in last chapter, we instantiate the type variable A with Set and $(R\ x\ y)$ becomes $y \subseteq x$. The hypotheses $Rtrans$ and $Rantisym$ are easy. What was called a set is now a class, and we take for lim the intersection operation, for which it is immediate to show $Upperb$ and $Least$. Note that it is essential here that Tarski's theorem be expressed over an arbitrary type. This allows us to instantiate it over the type of sets given with the inclusion relation, obtaining thus what is usually called the Knaster-Tarski's theorem.

Hence we get:

$$(\varphi : Map)(Incr\ \varphi) \Rightarrow (Fixpt\ \varphi\ (Initial\ \varphi)). \qquad (FIX)$$

Actually it is possible to refine Tarski's theorem and prove that the fixpoint obtained in the proof is actually the limit of all fixpoints. Here this shows that $(Initial\ \varphi)$ is the smallest fixpoint:

$$(\varphi : Map)(Incr\ \varphi) \Rightarrow (Initial\ \varphi) = \cap(Fixpt\ \varphi). \qquad (MIN)$$

Note the similarity with the treatment in Park[5], where $Induction$ is called a *convergence* formula.

12.3 Nœtherian induction

Let R be a binary relation on the universe, that is $R : U \rightarrow U \rightarrow Prop$, abbreviated below as Rel_U, or simply Rel. Note that every relation may be seen as an indexed family of sets. Thus if $\geq$ is a preorder, $(\geq\ x)$ is the set of elements below x.

We define the *adjoint* map associated with R as the U-map:

$$(Adjoint\ R)\ :=\ [A : Set]\ \{x \mid (R\ x) \subseteq A\}.$$

It is a simple exercise (left to the reader) to prove that this map is always increasing:

$$(R : Rel)\ (Incr\ (Adjoint\ R)) \qquad (Adjoint\_Incr).$$

The class of *R-inductive* sets is defined as:

$$(Inductive\ R)\ :=\ (Stable\ (Adjoint\ R)).$$

The induction associated with the adjoint map states that the inductive sets are universal. This is just what is usually called Nœtherian induction[2]:

$$(Noetherian\ R)\ :=\ (Induction\ (Adjoint\ R))$$

or, in expanded form:

$$(A : Set)\ (\forall x \cdot (\forall y \cdot (R\ x\ y) \Rightarrow y \in A) \Rightarrow x \in A) \Rightarrow \forall u \cdot u \in A.$$

We recognize the definition used in [3] to prove Newman's lemma:

$$(R : Rel)\ (Noetherian\ R) \Rightarrow (Loc\_Confluent\ R) \Rightarrow (Confluent\ R).$$

Thus we see that this very powerful transfinite induction principle is but a special case of the very general *Induction* above. Usual complete induction principles are in turn obtained by further specialization. For instance, we shall see below that complete induction over the naturals is simply $(Noetherian\ \ >)$.

12.4 Structural Induction

It is now time to introduce some further notation. Let A be a U-set, and E be any construction expression. We let $\forall x \in A \cdot E$ to stand for an abbreviation of $\forall x \cdot x \in A \Rightarrow E$, and similarly we let $\{x \in A \mid E\}$ stand for $\{x \mid x \in A \wedge E\}$. We shall also use the notation $\exists x \in A \cdot E$ to stand for $\sum_U [x : U](x \in A \wedge E)$.

We shall now show how to express structural induction [1] in the calculus. First we define the relation "f preserves A", when f is a U-function (i.e. $f : U \to U$) and A is a U-set:

$$(Preserve\ f\ A)\ :=\ \forall x \in A \cdot (f\ x) \in A.$$

Now we define what it means for the element $y : U$ to be *reachable* from element $x : U$ using function f. This notion is axiomatized by the relation:

$$(Iter\ f)\ :=\ \{x, y \mid (A : Set)\,(Preserve\ f\ A) \Rightarrow x \in A \Rightarrow y \in A\}.$$

The general method consists in defining, in the structure under consideration, the suitable generalization of reachability expressing what are the elements *expressible* using the operations of the structure. The expressibility proposition may then be seen as a set (the initial algebra, or standard model). Similarly to above we get an induction principle by postulating that every element of type U is in this set.

Let us consider for instance arithmetic. The structure is given here by a successor operation $S : U \to U$ and a zero constant $0 : U$. A universe presented with this structure we call a *Peano algebra*. On any Peano algebra, we may define a relation

$$\leq\ :=\ (Iter\ S)$$

and it is easy to show that $\leq$ is reflexive and transitive (using as proofs respectively identity and composition of the proper type). Now the set

$$\{n \mid 0 \leq n\}$$

is the characteristic predicate of the standard model $\mathcal{N}$:

$$\mathcal{N}\ :=\ \{n \mid (A : Set)\ (\forall m \in A \cdot (S\ m) \in A) \Rightarrow 0 \in A \Rightarrow n \in A\}$$

The corresponding universally quantified sentence is Peano's induction principle:

$$Peano \ := \ (A : Set)\ (\forall m \in A \cdot (S\ m) \in A) \Rightarrow 0 \in A \Rightarrow \forall n \cdot n \in A.$$

Note how the binding on n migrated from $\mathcal{N}$ to $Peano$, similarly to the transformation between $Initial$ and $Induction$.

Let us now indicate the relationship with our general induction principle above. The map φ needed here may be defined as sending A to: $\{n \mid \exists m \in A \cdot n \equiv (S\ m) + n \equiv 0\}$, or equivalently, we define:

$$(Nat\_map\ A) \ := \ \{n \mid (P : Set)\ \forall u \cdot (u \in A \Rightarrow (S\ u) \in P) \Rightarrow 0 \in P \Rightarrow n \in P\}.$$

It is easy to prove that Nat_map is increasing, and that:

$$(Stable\ Nat\_map\ A) \Leftrightarrow (Nat\_stable\ A)$$

with

$$(Nat\_stable\ A) \ := \ (\forall n \in A \cdot (S\ n) \in A) \wedge (0 \in A)$$

and thus we get that

$$(Induction\ Nat\_map) \Leftrightarrow Peano.$$

Indeed, a simple Curryfication suffices to show that:

$$\mathcal{N} = \cap Nat\_stable.$$

Remark. The equivalence between $(Stable\ Nat\_map)$ and Nat_stable boils down to recognizing the following propositional equivalence:

$$\forall Q \cdot ((P \Rightarrow Q) \Rightarrow Q) \Leftrightarrow P.$$

Intuitively, it means that P is equivalent to its operational contents.

Actually $Peano$ is only one half of initiality. What it says is that the universe contains only elements which are definable with the algebra operators. The other half is to postulate that different operators give rise to distinct elements. In the case of arithmetic, for instance, this amounts to adding the following two postulates:

$$Peano1 \ := \ \forall n \cdot \neg(S\ n) \equiv 0$$

$$Peano2 \ := \ \forall m, n \cdot (S\ m) \equiv (S\ n) \Rightarrow m \equiv n.$$

A standard model of arithmetic is thus any universe verifying $Peano$, $Peano1$ and $Peano2$.

Remark. We recall that the natural numbers may be expressed by the second-order proposition:

$$Nat \ := \ (X : Prop)(X \Rightarrow X) \Rightarrow X \Rightarrow X,$$

with the successor function, of type $Nat \Rightarrow Nat$, defined as:

$$S \ := \ [n : Nat]\,[X : Prop]\,[s : X \Rightarrow X]\,[z : X](s\ (n\ X\ s\ z))$$

and the zero, of type Nat, defined as:

$$0 \ := \ [X : Prop]\,[s : X \Rightarrow X]\,[z : X]\,z.$$

It is possible to apply the whole theory above, with Nat standing for the universe U. However, even in Nat we need to postulate the *Peano* axioms. This is a bit puzzling, since we know that the normal forms of constructions (with η conversion allowed) of type Nat are isomorphic to the standard model of natural numbers. But this knowledge is from meta-theoretic analysis, and cannot be internalized in the system. However, it is a simple matter to define in the meta-language of constructions appropriate macros, so that the Peano axioms are automatically generated from the signature Nat.

The method above is of course generalizable in a straightforward way to any algebraic type, leading to structural induction over a wide variety of structures.

Finally, complete induction is easily seen a direct application of Nœtherian induction. For instance, over integers, with

$$x > y \ := \ (S\ y) \leq x$$

we get complete induction as $(Noetherian\ \ >)$.

12.5 Computational Induction

We now show how to imbed in Constructions Scott's computational induction method, as presented for instance in LCF [4].

12.5.1 The domain postulates

We assume axioms on the universe U giving it the structure of a pre-order:

$$[\ \sqsubseteq : Rel]$$

$$[Refl : \forall u \cdot u \sqsubseteq u]$$

$$[Trans : \forall u, v, w \cdot u \sqsubseteq v \Rightarrow v \sqsubseteq w \Rightarrow u \sqsubseteq w].$$

We define $\doteq$ as the associated equivalence:

$$\doteq \ := \ \forall u, v \cdot u \sqsubseteq v \wedge v \sqsubseteq u.$$

We say that U-set A is *directed* whenever:

$$(Directed\ A) \ := \ \forall x \in A \cdot \forall y \in A \cdot \exists z \in A \cdot x \sqsubseteq z \wedge y \sqsubseteq z.$$

Now we postulate the partial order U to be *complete*, in the sense that every directed set possesses a limit, its lub: $(A : Set)\ (Directed\ A) \Rightarrow \exists u \cdot u \in (Lub\ A)$, with:

$$(Lub\ A) \ := \ \{u \mid \forall x \in A \cdot x \sqsubseteq u \wedge \forall v \cdot (\forall x \in A \cdot x \sqsubseteq v) \Rightarrow u \sqsubseteq v\}.$$

For ease of application, we shall Skolemize the limit u as a function $(lim\ A)$. We could have lim depend on an extra argument of type $(Directed\ A)$, but this extra generality is not needed; this is an application of the principle of "proof irrelevance". Thus we postulate:

$$[lim : Set_U \rightarrow U]$$

$$[Complete : (A : Set)\ (Directed\ A) \Rightarrow (lim\ A) \in (Lub\ A)].$$

It is easy to show that the elements of $(Lub\ A)$ are equivalent:

$$\forall u \in (Lub\ A) \cdot \forall v \in (Lub\ A) \cdot u \doteq v.$$

The empty set $\emptyset$ is directed, and thus every complete pre-order possesses a minimum element:

$$\perp \ := \ (lim\ \emptyset).$$

It is straightforward to prove that $\perp$ is indeed minimum:

$$\forall u \cdot \perp \sqsubseteq u. \qquad (Bot)$$

12.5.2 Computational induction

Let $f : U \to U$. We define the set of (finite) f-*approximants* as:

$$(Approx\ f) \ := \ (Iter\ f\ \perp)$$

that is:

$$\{u \mid (A : Set)\ (Preserve\ f\ A) \Rightarrow \perp \in A \Rightarrow u \in A\}.$$

Remark the similarity with the definition of the standard model $\mathcal{N}$ above. Similarly to maps above, we define the notion of *increasing* function:

$$(Increasing\ f) \ := \ \forall u, v \cdot u \sqsubseteq v \Rightarrow (f\ u) \sqsubseteq (f\ v)$$

and we may show that:

$$(f : U \to U)\ (Increasing\ f) \Rightarrow (Directed\ (Approx\ f)). \qquad (Dir\_Approx)$$

The proof of this proposition, left as an exercise, is analogous to defining inductively the function computing the maximum of two natural numbers. We may now define, for any increasing f:

$$(Y\ f) \ := \ (lim\ (Approx\ f)).$$

The limit of finite approximants $(Y\ f)$ is intuitively $\sqcup_n\ f^n(\perp)$.

We now define an *admissible* U-set as one which contains all the limits of its directed subsets:

$$(Adm\ A) \ := \ (B : Set)\ B \subseteq A \Rightarrow (Directed\ B) \Rightarrow (lim\ B) \in A.$$

The restriction of A to admissible sets in the definition of approximant permits to iterate f in the transfinite, which gives the notion of transfinite f-approximation:

$$(\infty\ f) \ := \ \{u \mid (A : Set)\ (Adm\ A) \Rightarrow (\forall x \in A \cdot (f\ x) \in A) \Rightarrow u \in A\}.$$

Note that $(\infty\ f)$ is the intersection of the class of admissible sets preserved by f, whereas $(Approx\ f)$ is the intersection of the class of sets containing $\perp$ and preserved by f. In some sense $(\infty\ f)$ is to $(Approx\ f)$ what ordinals are to natural numbers.

Let us now show that $(\infty\ f)$ is admissible:
$[f : U \to U]\ [B : Set]\ [h_1 : B \subseteq (\infty\ f)]\ [h_2 : (Directed\ B)] \vdash l_1 : (lim\ B) \in (\infty f)$

where l_1 is proved by:
$[C : Set]\,[h_3 : (Adm\ C)]\,[h_4 : (Stable\ C)](h_3\ B\ l_2\ h_2)$
where $l_2 : B \subseteq C$ is proved by:
$[u : U]\,[h_5 : u \in B](h_1\ u\ h_5\ C\ h_3\ h_4)$.
Discharging all this temporary context, we get:

$$(f : U \to U)(Adm\ (\infty\ f)). \qquad (Adm\_\infty)$$

Now it is a simple matter to prove:
$[f : U \to U]\,[i : (Increasing\ f)] \vdash (Adm\_\infty\ f\ (Approx\ f)\ incl\ (Dir\_Approx\ f)) : (Y\ f) \in (\infty\ f)$,
where the proof of $incl : (Approx\ f) \subseteq (\infty\ f)$ is left to the reader. Thus we get finally:

$$(f : U \to U)(Increasing\ f) \Rightarrow (Y\ f) \in (\infty\ f).$$

By unwinding this proposition it is easy to see that this is precisely Scott's computational induction principle. Writing it in long form:

$$(f : U \to U)(Increasing\ f) \Rightarrow (A : Set)\ (Adm\ A) \Rightarrow (\forall x \in A \cdot (f\ x) \in A) \Rightarrow (Y\ f) \in A. \qquad (Comp\_Ind)$$

Two remarks are in order. Firstly, note that this principle is *provable* from our postulates on the domain U (i.e., the complete partial ordering axioms). Secondly, the notion of admissible set is axiomatized inside the calculus, and thus we can use all the power of the logical system to prove that a given set is indeed admissible, whereas in LCF [4] the notion of admissible predicate is weakened to a syntactic check of the meta-linguistic support. Finally note that the hypothesis $\bot \in A$ is not needed above, since it is implicit from the hypothesis $(Adm\ A)$.

12.5.3 Continuity and fixpoints

It may seem curious that it is not necessary in the justification of computational induction to assume that f is continuous. But this assumption is indeed needed for recursion. Let us now make this point precise.

First, let us define the image by f of a U-set A:

$$(Image\ f\ A)\ :=\ \{y \mid \exists x \in A \cdot y \doteq (f\ x)\}.$$

It is easy to show that:

$$(Directed\ A) \Rightarrow (Increasing\ f) \Rightarrow (Directed\ (Image\ f\ A)). \qquad (Dir\_Im)$$

Thus, for every increasing f and directed set A, we may define:

$$(Lim\ f\ A)\ :=\ (lim\ (Image\ f\ A)).$$

Now let us call *diagram* any non-empty directed set:

$$(Diagram\ A)\ :=\ \exists u \in A \cdot (Directed\ A).$$

Next we define what it means for an increasing f to be continuous:

$$(Continuous\ f)\ :=\ (A : Set)\ (Diagram\ A) \Rightarrow (Lim\ f\ A) \doteq (f\ (lim\ A)).$$

Note that we must restrict A to be a non-empty directed set, since we do not demand our functions to be strict.

Exercise. Prove that for all f, $(Continuous\ f) \Rightarrow (Increasing f)$.

Now, defining the fixpoints of f in a similar way as for maps:

$$(Fixpoints\ f)\ :=\ \{u \mid (f\ u) \doteq u\},$$

we can prove:

$$(f : U \to U)(Continuous\ f) \Rightarrow (Y\ f) \in (Fixpoints\ f)$$

and:

$$(f : U \to U)(Continuous\ f) \Rightarrow \forall x \in (Fixpoints\ f) \cdot (Y\ f) \sqsubseteq x.$$

In other words, $(Y\ f) \doteq \cap(Fixpoints\ f)$. This is analogous to Tarski's theorem, but still significantly different.

A variant of Tarski's theorem would say here is that if f is increasing (and not necessarily continuous), then $(Z\ f)$ is the minimum fixpoint of f, where

$$(Z\ f)\ :=\ (lim\ (\infty\ f)).$$

Problem. Show the above statement. In particular, you will need to prove that $(\infty\ f)$ is itself directed.

Thus, continuity is needed for finiteness, i.e. computability. This concludes our incursion into domain theory.

12.6 Nœtherian as a well-foundedness principle

We are going to show in this section that $(Noetherian\ R)$ implies that there are no infinite R-chains, relating induction to well-foundedness.

Let A be a U-set. We say that A is *R-eternal* iff:

$$(Eternal\ R\ A)\ :=\ \exists x \in A \land \forall x \in A \cdot \exists y \in A \cdot (R\ x\ y).$$

It is straightforward to show, with the definition of Nœtherian given above, that:

$$(Eternal\ R\ A) \mid (Noetherian\ R\ \sim A)$$

where the incompatibility connective $\mid$ is Sheffer's stroke.

Thus $(Noetherian\ R)$ implies (classically, but constructively) that R is *well-founded*, in the sense that there are no infinite R-chains:

$$(WFR)\ :=\ (A : Set)\ \neg(Eternal\ R\ A).$$

Intuitively, the set $(Initial\ (Adjoint\ R))$ contains all elements which have only finite R-chains issued from them, and $(Noetherian\ R)$ says that this set is universal.

References

[1] R. Burstall. “Proving Properties of Programs by Structural Induction.” Comp. J. **12** (1969), 41–48.

[2] P. M. Cohn. “Universal Algebra.” Reidel, 1965.

[3] Th. Coquand, G. Huet. “Constructions: A Higher Order Proof System for Mechanizing Mathematics.” EUROCAL85, Linz, Springer-Verlag LNCS 203 (1985).

[4] M. Gordon, R. Milner and C. Wadsworth. “Edinburgh LCF.” Springer-Verlag LNCS 78 (1979).

[5] D. Park. “Fixpoint Induction and Proofs of Program Properties.” *Machine Intelligence 5*, Eds. B. Meltzer & D. Michie, 59–77, Edinburgh University Press.

[6] L. C. Paulson. “Constructing Recursion Operators in Intuitionistic Type Theory.” Tech. Report 57, Computer Laboratory, University of Cambridge (Oct. 1984).

[7] D. Scott. “Data Types as Lattices.” SIAM Journal of Computing **5** (1976) 522–587.