# CS3110 Fall 2013 Lecture 2: Evaluation of Expressions, Types, Recursion (9/3)

Robert Constable

September 3, 2013

## 1 Lecture Plan and Overview

1. Lecture Plan and Overview
2. Review: syntax, evaluation
3. Evaluation by reduction rules, substitution model
4. OCaml types and type inference
5. Typing rules and type inference
6. Recursion and induction
7. Summary

The first topic of the lecture will be *evaluation* and the role of *types* in OCaml evaluation. OCaml's theory of computation requires a strong understanding of types, so we mention briefly the wider mathematical theory of types. We also look more closely at *recursive functions* as preparation for Problem Set 1, and we discuss *induction* as well which is part of the problem set. We start with a brief review of the technical elements of Lecture 1 and mention good questions from students.

**Suggested Reading**: This lecture is closely related to Lecture 2 by Professor Kozen in Spring 2011. Also, the textbook by Jason Hickey, http://courses.cms.caltech.edu/cs134/cs134b/book.pdf. Chapter 2 has an excellent account of types. You should glance at the *OCaml Reference Manual* on types as well, http://caml.inria.fr/pub/docs/manual-ocaml/.

# 2 Review: syntax, evaluation

We have seen the elements of OCaml syntax and have started discussing the evaluation rules in the form of reduction rules or rewrite rules. We discussed the definition of *canonical values* in an evaluation system based on reduction rules.

There was a good question after class about why $fun\ x \rightarrow x * x$ is a canonical value when inside the function body we have the multiplication operator which we know is a non-canonical form, e.g. 3*3 reduces to 9. The point is that there is no value for the multiplication operator to work on until the function is applied, so the inside of the body can't be reduced. This is canonical in the same sense that the operator expression $\star$ is canonical, as is $+$.[1]

This leads to a more subtle example, what about the function

$$fun\ x \rightarrow 3 * 5$$

is this a canonical value?

The answer is yes and you can test it by writing the function in the OCaml top loop. OCaml will respond with exactly this message.

$- :' a \rightarrow int =< fun >.$

The fact that this constant function is canonical is determined by looking

[1] A person might also think that $fun\ x \rightarrow fun\ y \rightarrow x \star y$ is just like the $\star$ multiplication operation. What do you think? Might $\star$ be equal to $fun\ p \rightarrow\ let\ (x_1, x_2) = p\ in\ x_1 * x_2$ or if we wrote $let\ times\ =\ fun\ p\ \rightarrow\ let\ (x_1, x_2) = p\ in\ x_1 * x_2$, then would $times\ =\ \star$?

at the *outer most constructor* of the expression, the key word $fun$. This operator name tells us that the expression is canonical *without looking inside at all.*

## 2.1 historical aside: the lambda calculus, Lisp, Classic ML, and abstract syntax

**lambda notation** In the language Classic ML, function values are written as $\backslash x.body$. This is the closest ascii representation of the function notation in the lambda calculus [3] created by Alonzo Church.[2] In the lambda calculus this would be written $\lambda x.body$ and the identity function is just $\lambda x.x$ a very compact notation. The outer operator is $\lambda$. From time to time we'll use the lambda notation in an informal way to remind you of the connection. Mentioning the lambda calculus from time to time is an *historical and cultural* side note for the lectures.

Church's lambda calculus provided a way to write functions precisely compared to the standard notation in mathematics. We still see that most mathematics books would write a function like $\lambda x.\ x \times x$ as $y = x \times x$. They would write the *sine* function as $y = sine(x)$ and a general function $f$ as $y = f(x)$. His colleagues in the mathematics department at Princeton did not find Church's function notation very appealing and did not use it. Probably even 74 years later most of them still wonder what "all the fuss is about." You can see that in the Cornell Mathematics Department as well. Mathematics books are divided about whether a function is a rule or a set of ordered pairs. For this course, a function means a *partial computable function.*

**Lisp from the lambda calculus** By 1962 Church's lambda calculus from 1940 inspired one of the most innovative and influential programming languages, Lisp [7], still heavily used today. Lisp treated functions as first class objects and introduced many other fundamental concepts in modern

[2] American Standard Code for Information Interchange is abbreviated as ascii or ASCII. In Lecture 1 we discussed briefly the ISO8859-1 character set, an ASCII extension.

computer science such as *abstract syntax trees*, garbage collection, reflection of programs as data, formal rules of evaluation, rules for proving properties of programs, and the idea of proof checking as well as other fundamental notions. In one of the most influential foundational articles of the field, A Basis for a Mathematical Theory of Computation [6] by John McCarthy sketched a theory of computation on which the OCaml theory is based.[3] McCarthy was a graduate student at Princeton where he learned about Church's lambda calculus. A key concept from Lisp's abstract syntax is helpful right at this point in thinking about OCaml.

The Lisp programming language is a signature accomplishment of the MIT computer science department. A company was spun off from MIT that made Lispmachines. In these machines even the operating system was written in Lisp, it was "Lisp all the way down." MIT also developed the forerunner of programming courses such as CS3110, it was based on Lisp. The textbook for the MIT course is still widely known and respected: *Structure and Interpretation of Computer Programs* [1].

**abstract syntax** Note, for the pairing operator, as in $(0, true)$ the syntax does not reveal an outer operator as clearly and simply as it does for functions where the outer operator is $fun$ (or $\lambda$). Abstractly a pair has an outer operator, and we can think of pairs as $pair(0, true)$, also internally, in the compiler, the outer operator of its abstract syntax tree is $pair$.

For conceptual clarity, we will "make up" outer operators from time to time when they help clarify an OCaml concept, but they are not official OCaml syntax. It is very convenient to have an outer operator when we discuss canonical and non-canonical expressions. Indeed, this approach to syntax

[3] One of the most inspiring paragraphs in any of the early papers on computer science is this opening by McCarthy. "Computation is sure to become one of the most important of the sciences. This is because it is the science of how machines can be made to carry out intellectual processes. We know that any intellectual process that can be carried out mechanically can be performed by a general purpose digital computer. Moreover, the limitations on what we have been able to make computers do so far clearly come far more from our weakness as programmers than from intrinsic limitations of machines. We hope that these limitations can be greatly reduced by developing a mathematical science of computation."

from Lisp is called *abstract syntax*.

Notice that Alonzo Church of the Princeton mathematics department was featured in our "short history" about the lambda calculus. He was one of the first major American logicians, one of the first mathematicians to discover a provably unsolvable problem in mathematics. He was also the host for Alan Turing's visit to America based on the fact that Turing had independently discovered unsolvability at about the same time. One could say that "Church brought computer science to America" in the person of Alan Turing in 1937 and though his influence on McCarthy and other pioneering computer scientists whom we will mention later. I don't think anyone realized at the time how important these influences were. We already mentioned Church's connection to Lisp.

**Church's Simple Type Theory** Church was also one of the founders of modern type theory [2] which is key to OCaml. For some reason McCarthy did not think of adopting Church's type theory to Lisp. It would be very interesting to understand why that happened – can we say that a class of non-events happened? I think one reason is that McCarthy believed that the computation system should come first, and that it was fundamentally untyped. There is a great deal of merit in thinking that computation comes before types, but also a great deal of merit in realizing that types are conceptually essential to reasoning – something Aristotle understood; he called them *categories*.

Church's type theory is implemented in the family of proof assistants based on Higher Order Logic, these are called HOL provers. HOL-light is used by Intel to verify floating point operations on its chips. HOL is widely used in Europe and it has a feature to produce its proofs in a natural language format.

# 3 Evaluation by Reduction Rules

We refer back to the last section of the written Lecture 1 notes; there reduction rules are discussed briefly and some examples are given. Here we repeat the rule for evaluating function application.

**Function Application Rule**

$$exp_2 \downarrow v_2, exp_1 \downarrow fun\ x\ ->\ body(x), body(v_2/x) \downarrow v_3 \vdash (exp_1\ exp_2) \downarrow v_3.$$

Notice the *order of evaluation*, we evaluate the argument, $exp_2$ first. If that expression has a value, then we evaluate $exp_1$ and if that evaluates to a function $fun\ x\ ->\ body(x)$, then we substitute the value $v_2$ for the variable $x$ in *body* and evaluate that expression. This is called eager evaluation or call by value reduction because we eagerly look for the input to the function, even before we really know that $exp_1$ evaluates to a function.

There is another order of function evaluation in programming languages called *lazy evaluation*. To evaluate $exp_1\ exp_2$ lazily, first evaluate $exp_1$ to make sure it is a function, then substitute $exp_2$ for the variable in the body. Whether $exp_2$ is evaluated depends on the evaluation of the body. For example, a constant function such as $fun\ x\ ->\ 0$ does not use the value of the input to construct its result. Likewise the identity function, $fun\ x\ ->\ x$ can produce output without evaluating the input first – nevertheless, OCaml will evaluate it. OCaml supports lazy evaluation as well, and we will discuss that topic later.

These simple rules might seem tedious, but they are the basis for a precise semantics of the language that both people and machines can use to understand programs. By writing down all these rules formally, we create a *shared knowledge base with proof assistants* that can help us reason about evaluation.

**divergence** In all of the evaluation rules for OCaml it is possible that the expression we try to evaluate will *diverge*, meaning "fail to terminate".

That is, the evaluation process runs on indefinitely until memory is exhausted or until you get tired of waiting and intervene to stop the evaluation process which is diverging. We can write very simple programs that will loop forever without using up memory (and thus is not caught by the OCaml stack overflow detector).

**Question:** Could OCaml support a compiler feature that detected whether evaluation of any given expression *exp* would diverge?

**exceptions** Evaluation might also just "get stuck" as when we try to apply a number to another number, as in 5 7 or take the first element of a function value, e.g. $fst\ fun\ x -> (x, x)$ or divide by 0. Such attempts to evaluate an expression do not make sense and would get stuck if we tried to evaluate them. The run time system can detect these bad states of the evaluation process. When this happens OCaml can report an error called an *exception*. We discuss exceptions later.

We will see that the type system helps us avoid expressions whose attempted evaluation would "get stuck", but we cannot avoid all such situations by type checking. Later we will discuss computations that cause exceptions, which is a response to evaluations that become stuck even though they are *well formed* (pass the type checker).

As written in Lecture 1, reduction rules can be presented in several styles. We are using a style that is easy to write down in a single linear expression called a *sequent*.

Sequents have the form

$$hyp_1, ..., hyp_n \vdash concl.$$

Recall the following rule from Lecture 1.

**Rule Boolean-or** $exp_1 \downarrow true \vdash (exp_1 \mid\mid exp_2) \downarrow true$

The symbol $\vdash$ is called a *turnstile* because it resembles a very simple turnstile device that might be used to count people entering a store. Note

that the parentheses around $exp_1$ || $exp_2$ are to make the scope of the reduction operation $\downarrow$ clear, the OCaml expression might not have these out parentheses.

This method is called *structured operational semantics* or *small step semantics* and was developed by Gordon Plotkin at Edinburgh University. Gordon has made numerous important contributions to theoretical computer science from his base in Scotland. His idea for semantics is now used in logic and type theory as well. A similar method was developed by Gilles Kahn, a French computer scientist at INRIA (Institut National de Recherche en Informatique et en Automatique). Kahn's method is called *big step semantics* or natural semantics [5, 4].

This semantic method is designed to present semantics in a style that is easy for people to understand and reason about. It does not mimic what the computer actually does step by step, but it is faithful to the computer's operation. We say that the computer's operations are a *refinement* of this reduction rule semantics. Using this formal semantic model, we can easily prove properties of OCaml's computation system. For example, we can easily see that if an expression involving *boolean or*, $bexp_1$ || $bexp_2$, has the property that $bexp1$ reduces to the value *true*, then it will do so even if $bexp_2$ fails to terminate. On the other hand, if $bexp_1$ fails to terminate, then even if $bexp_2$ terminates with value *true*, the disjunction fails to produce a value. We will see that this failure to terminate is a possibility for Boolean expressions defined by recursive procedures and by loops.

There are too many rules for OCaml evaluation to include them all in these notes or even in the OCaml User's Manual. On the other hand, the class could easily assemble a complete set of rules as a crowdsourcing project. In this lecture we examine more rules and sketch some properties of the OCaml evaluation system. Then we reflect on the method as applied to function evaluation and talk about the scope of the declaration of the function's inputs and about how reduction is done by substitution of values for arguments in the function body. This leads us to examine types, especially the types of functions.

# 4 OCaml types and type inference

In order to define OCaml evaluation of expressions, we first need to understand elements of the OCaml type system. This is because before OCaml evaluates an expression *exp*, it first checks to see if it has a type. Expressions that have a type make sense at some level as either programs or data. OCaml tries to *infer a type* for *exp*. If it succeeds, then the term is evaluated. Moreover, the evaluator can take advantage of the information gathered by type inference. If OCaml cannot infer a type, then it will not attempt to evaluate the expression because the result won't make sense or the computation process might get stuck or might go on forever.

**Definition:** To define a *computational type* $T$ we provide a canonical name for it, and we say how to form expressions that are *canonical values* of that type. Each OCaml type is the partial type of an underlying computational type in mathematics. The partial type includes all expressions that are in the "mathematical type" if they converge.

**Design Invariant:** OCaml is designed to have the property that if an expression *exp* is assigned the type $t$ by the typing rules or the type inference algorithm, then if the expression *exp* converges to a value, it will be a canonical value of the underlying mathematical type of $t$.

I do not know of a proof that OCaml satisfies this design invariant.

The type inference algorithm will catch *some but not all* instances of expressions that will not evaluate properly because they do not make sense, e.g. attempting to add an integer to a Boolean or multiplying a function by a character string. In these cases the operation is not defined, and in some cases we can't even imagine what kind of value might result if any.

**undefined operations** The type checker/inferencer cannot check against all the reasons that computation does not make sense. For example, OCaml does not allow division by 0, so 17 divided by 0, 17/0, does not make sense. But the type checker will not be able to determine whether a divisor, say d in the expression n/d, is 0 or not. Likewise, when we are defining the

rational numbers, such as $1/2$, we use ordered pairs $(1, 2)$ and we will not be able to rule out $(1, 0)$ as a rational number using only the typing rules.

**diverging computations** No type checker for OCaml can decide in advance whether the evaluation of an expression halts. This is called the *halting problem*, and it is a fundamental result of computing theory that for any *universal programming language*, i.e. one that is capable of defining all partial computable functions, it is not possible to recognize by an algorithm all cases where the algorithm will fail to halt. OCaml is a universal language in this sense.[4]

**advantages of type checking** We do not see the advantages of type information at the granularity of reduction rules, but the OCaml compiler can use type information to improve the way modern computer hardware implements evaluation. For instance, the type checker can provide information that lays out expressions efficiently in memory, e.g. we know that we need only one bit of information to store a Boolean value.

The collection of all OCaml types is defined inductively, that is, we start with the *atomic types* and define how to build new types using operators called *type constructors*. OCaml does not have a type that includes types as values. There are programming languages that do, the Agda, Coq, and Nuprl languages all treat *types as values*, e.g. they can say $int \in Type_i$.

## 4.1 atomic types

Please read Kozen's Lecture 2 in 2009 and his Lecture 3, pages 1 and 2. Also see Recitation 1 from 2009. You will also find Hickey's chapter on types very clear. Here are some of the OCaml atomic type expressions.

[4]Universal programming languages are also called *Turing complete programming languages* in honor of the first computer scientist, Alan Turing. His computation model is now called a Turing machine.

1. int, the type of OCaml positive and negative integers.[5]
2. bool, the type of Boolean values $true$ and $false$.
3. float, the type of finite precision real numbers.
4. char, the type of individual characters.
5. string, the type of strings of characters.
6. unit, the type with one value denoted ().

## 4.2 computational types versus OCaml types

These OCaml type expressions are meant to relate to types that are well known and precisely defined in computational mathematics. For example, *int* is suggestive of the mathematical type $\mathbb{Z}$ of the integers, of which the unbounded type of positive natural numbers $\mathbb{N}$ = 0,1,2,3,... is a subtype.

This computational type is *unbounded*, that is, there is no bound on the size of number we can imagine. Sometimes we say that $\mathbb{N}$ is an infinite type or an infinite set, but we do not need to go that far. It suffices to think of the type as *unbounded*. In computational type theory, when we talk about the natural numbers, we also know what we mean to add, subtract, and multiply two numbers. In addition we know that it means to say that two numbers are equal, $n =_{\mathbb{N}} m$ or $n = m\ in\ \mathbb{N}$. OCaml types do not always come with a well defined notion of equality on them. This makes them less mathematically complete.

We write $\mathbb{N}^+$ for the positive numbers, 1,2,3,4,..., a *subtype* of the natural numbers that can be defined as $\{x : \mathbb{Z} | x \geq 1\}$. In OCaml, we do not have this notion of subtyping, but we use it in the computational theory of types that OCaml approximates.

[5]These integers are of bounded precision, either 32 bit or 64 bit depending on the hardware.

## 4.3 type constructors and compound types

Given types already constructed, starting with the atomic types, we define compound types using constructors. These will build product types, union types, function types, record types and more. We list a few and give their syntax. The Users Manual provides a complete list and the Lecture notes by Kozen from 2009 and 2011 cover most of the types in detail.

Note, we only know from the type inference system that if $t$ terminates in a value $v$, that is $t \downarrow v$, then value $v$ has type $T$.

**Definition** Product Type

See Hickey Chapter 5 on Tuples and the OCaml Reference Manual.

If $T_1$, ..., $T_n$ are types, then so is their product $T_1 \star T_2 \star ... \star T_n$. The elements of this type are *ordered tuples, e.g. pairs, triples, etc.* $(t_1, t_2, ..., t_n)$ where $t_i$ belongs to $T_i$. In the course notes we sometime write $t \epsilon T$ to assert the judgment that $t$ belongs to the type $T$.

Notice that the product type is a partial type corresponding to a basic mathematical type. If $\mathbb{R}$ is the type of computable mathematical real numbers, then $\mathbb{R} \times \mathbb{R}$ is the Cartesian plane so important in calculus.

**Definition** Function Type

See Hickey Chapter 3.

If $T_1$ and $T_2$ are types, then $T_1 \rightarrow T_2$ is the type of all *OCaml computable functions* $f$ such that if $t_1$ is an expression of type $T_1$, and if the evaluation of $t_1$ terminates in a value $v_1$, then if $f(v_1)$ terminates, its value will be of type $T_2$.

Values of type $T_1 -> T_2$ are of the form $fun\ x -> body(x)$ where $body(x)$ is an expression that possibly has an occurrence of the identifier $x$. We say that the scope of $x$ in this function value is the body $body(x)$, and we say that the occurrence of $x$ in $fun\ x...$ is the *binding occurrence* of $x$ and that

$x$ is *bound* in *body*$(x)$. The function expression $fun\ x{-}{>}\ fun\ y{-}{>}\ y$ is a canonical function value whose body *body*$(x)$ is the function constant $fun\ y{-}{>}\ y$ which is the identity function. In this *body*$(x)$ the bound variable $x$ does not occur just as in the constant function $fun\ x\ {-}{>}\ 0$ it does not appear.

The function expression can also display its explicit type, so $fun\ (x : int)\ {-}{>}\ x + 1$ is the successor function on integers. It can be written without the types, as $fun\ x\ {-}{>}\ x + 1$, and OCaml will provide the type based on knowing that + is a function on integers, saying $- : int \rightarrow int\ =\ <fun>$.

For every rule that *constructs* an object in type theory, there is a corresponding rule for *deconstructing* it or *using* it. For functions the rule for using it is called *application*, and we write $f\ a$ to denote the application of function $f$ to its argument, $a$. We can also write $f(a)$ using parentheses in the standard mathematical way. We can also define an operator *ap* to apply a function which we'll explain later.

The function type constructor is also familiar from mathematics. In computational type theory, the type $\mathbb{Z} \rightarrow \mathbb{Z}$ is the type of all computable functions from integers to integers. This is an *uncountable type*, that is, it's "larger" than $\mathbb{Z}$. The OCaml functions denote elements of this type, and the OCaml type is a subtype of $\mathbb{Z} \rightarrow \mathbb{Z}$.

**Definition** Disjoint Union Type

This type is also called *tagged unions* or *variant records* (or less commonly algebraic data types). See Hickey Chapter 6.

If $T_1$, ..., $T_n$ are types and $Id_1$, ..., $Id_n$ are distinct *identifiers*, then

*type typename* $= |Id_1\ of\ T_1|Id_2\ of\ T_2|...|Id_n\ of\ T_n$ is a closed disjoint union. The first | is optional. The $Id_i$ are constructor names. They must be capitalized.

Values are created by applying a constructor name to an element of the corresponding type, e.g. $Id_1(t_1)$.

# 5 Typing Rules and Type Inference

Just as we have specific formal rules for evaluating OCaml expressions, we also provide rules for assigning types to expressions. We call these typing rules, and we can use the same rule format that we have used for evaluation. These rules define the canonical values of the OCaml types.

## 5.1 typing constants

Integers

$\vdash n \; \epsilon \; int$ for $n$ an OCaml integer constant

Booleans

$\vdash true \; \epsilon \; bool$

$\vdash false \; \epsilon \; bool$

Unit

$\vdash () \; \epsilon \; unit$

## 5.2 typing operators

Integers

$exp_1 : int, exp_2 : int \vdash exp_1 \; intop \; exp_2 \; \epsilon \; int$

Booleans

$exp_1 : bool, exp_2 : bool \vdash (exp_1 \; booleanop \; exp_2) \; \epsilon \; bool$

See the previous lecture notes for typing rules as well as the OCaml Manual and Jason Hickey's book.

# 6 Recursion and Induction

The first problem set is our initial effort to explain in detail the deep connection between recursion and induction. They are two aspects of the same idea. Recursion focuses on the *computational aspect* of "*induction-recursion*" and induction focuses on the *logical aspects*. Recursion seems much easier for computer science students to understand, so we use it as a bridge to induction.

Why is recursion easier to grasp? We are not sure, but here are some thoughts. Let's look at a recursive program in PS1. We want to sum the natural numbers from 1 to n. We write this OCaml recursive program.

let rec sumto (n:int) : int = if n = 0 then 0 else n + sumto (n-1).

For a computer science student after the first year in college or even before, the fact that *sumto* adds up the numbers from 0 to $n$ is clear. It is perhaps even clearer than the mathematical formulas used to capture the same idea, formulas such as

$$\Sigma_{i=0}^{n} i.$$

We can even think of $\Sigma_{i=0}^{n} i$ as the recursive program we wrote above. We can use a suggestive notation for the *sumto* function, let's call it *sigma*, and say that it is the precise definition of the above $\Sigma$ operator:
let rec sigma (n:int) : int = if n = 0 then 0 else n + sigma (n-1)

Here are formulas that state mathematical facts that we can prove about the summation operator $\Sigma$.

$$0 + 1 + 2 + ... + n = (n \times (n + 1))/2.$$

or the formula

$$(\Sigma_{i=0}^{n} i) = (n \times (n+1))/2.$$

We could imagine a theorem of this form.

For all natural numbers $n$, there is a natural number $m$, such that $(\Sigma_{i=0}^{n} i) = m$. Symbolically this is written as follows, using typed "quantifiers" related to what you saw in CS2800.

$\forall n : \mathbb{N}.\exists m : \mathbb{N}.((\Sigma_{i=0}^{n} i) \ = \ m)$.

We can prove this by induction as follows.

**Base case**: for $n = 0$ the summation $\Sigma_{i=0}^{n} i$ has value 0, so take $m = 0$.

**Induction case**: Assume for $n - 1$ we have a natural number $m$ such that $(\Sigma_{i=0}^{n-1} i) = m$. We need to find a natural number $m'$ such that $(\Sigma_{i=0}^{n} i) = m'$. By the definition of the formula, $(\Sigma_{i=0}^{n} i) = n + (\Sigma_{i=0}^{n-1} i)$, and by substitution of $m$ for $(\Sigma_{i=0}^{n-1} i)$, from the induction hypothesis, we thus have $(\Sigma_{i=0}^{n} i) \ = \ n + m$. So $m' \ = \ n + m$.

**Qed**

Notice that this inductive proof is really just writing out the algorithm *sigma* using other words, where the induction hypothesis is not a "mysterious assumption", but the *recursive call* in the proof! The base case is the non-recursive case of the recursive definition. The correspondence could hardly be clearer. If we wrote the induction constructor as a function it would look like this.

let rec induction (n:nat) = if n = 0 then 0 else n + m where induction (n-1) = m compare that to
let rec induction (x:int) = if x = 0 then 0 else x + induction (n-1).

**a conclusion worth thinking about** This form of recursion is an instance of *primitive recursion*, and we know "by induction" that it

terminates. Indeed we know that all primitive recursive function definitions are known to terminate. So induction is implemented by a recursive procedure which we know (or assume by an axiom that we believe) terminates on natural number inputs. This is the key difference between recursion and induction, and it is something that we cannot express in OCaml, but we can express in mathematics.

So in one sense, recursion is easier, we don't always think carefully about termination. Induction is harder because we need to think carefully about termination. The deepest mathematical fact here, that is beyond the reach of this course, is that all induction schemes are really just recursive programs that we believe terminate. But what is the basis of this belief? That is a deep philosophical question which modern logicians know a great deal about.

We see that recursion is expressed in a program, a mathematical construct that we can execute. Learning recursion provides a powerful tool for solving computational problems, so there is a practical reason to know it. Second, we can execute these programs by hand and come to understand the process by examples. The recursive execution model can be presented in a simple symbolic manner which we will later capture using a single construct called $fix$.

On the other hand, induction seems mysterious. It is stated as a logical principle, but you can't directly test it and apply it and see it working.

**the computational meaning of induction** We have seen in detail that the standard form of induction on natural numbers matches a particular form of recursion – *primitive recursion*. So induction on natural numbers is primitive recursion. This makes perfectly good sense if we understand that all inductive proofs on the natural numbers $\mathbb{N}$ are means of assigning values or evidence to each natural number $n$. The induction axiom is a terminating recursive procedure for building evidence.

We can find this kind of explanation for induction in textbooks. Some authors say that "induction is not a single proof but a method for creating

an *arbitrarily large number of proofs of the same form.*" Our problem set looks at this approach to understanding induction. For the above example the sequence of *simple proofs* might look like this: $sumto(0) = 0$, the base case, $(sumto(0) = 0) \Rightarrow (sumto(1) = 0 + 1 = 1)$, thus $sumto(1) = 1$, if $sumto(1) = 1 \Rightarrow sumto(2) = 2 + sumto(1) = 3$, thus $sumto(2) = 3$, etc.

If we look at the evidence needed at each step it is a number, to witness that a number exists, and in addition, *evidence that the number equals the summation.* The equality evidence can be given by a Boolean value that checks the sequence of equalities and relies on the *transitivity property* of equality. So in this case, the evidence constructed by the inductive proof is the pair $(m, true)$ where $true$ is the value of the boolean expression $sumto(n) = m$ at each stage of the construction. So we have reduced the "evidence chain" to exactly this sequence of pairs $(0, sum(0) = 0), (1, sum(1) = 1), (3, sum(2) = 3), (6, sum(3) = 6), ...$ which reduces by computation to $(0, true), (1, true), (3, true), (6, true), ..., (n, true)$ for any specific $n$.

The conclusion from this analysis is that induction on natural numbers is a proof method that uses primitive recursion to build evidence step by step starting from 0 to provide evidence of the proposition for any natural number $n$. *Under most circumstances of interest to us, we can execute the inductive proofs to actually produce data, and we know that the computation will terminate.* In Problem Set 1, we ask you to examine this analysis of induction on a proof that Gauss' summation formula is correct.

This connection between induction and recursion is used in modern proof assistants such as Agda, Coq, and Nuprl, but those are not yet common tools in undergraduate courses.

**Theorem (Gauss)**: $\forall n : \mathbb{N}.\ (\Sigma_{i=1}^{n} i)\ =\ (n \times (n + 1))/2$.

# 7 Summary of the Lecture

The first key point we covered is that in the abstract syntax for expressions, we can identify an outer operator such as $fun$, $ap$, $pair$, $fst$, $snd$, $plus$, $times$, etc. It helps to think of constants such as 0, 0.0, as having outer operators such as $integer$, $float$, $character$, and so forth. The outer operator tells us whether an expression is canonical or not, e.g. $fun$ indicates a canonical value, $ap$, for application, indicates a non-canonical one.

The second key point we examined is that the OCaml evaluator uses type information before it attempts to evaluate an expression. If the type inference algorithm cannot find a type, then the evaluator is not engaged. If it can assign a type, this does not mean that the expression will terminate. Much of the work in writing an OCaml program is getting it to be type correct. Experience has shown that type correctness with respect to an expressive type system is one of the best ways to prevent errors.

The third key point is that OCaml types are only an approximation to mathematical types, they are only partial types and they have limits on the size of canonical values.

The fourth point is that recursion and induction have the same computational meaning, that is, induction is a recursive procedure that is typically used to compute values and compute evidence. The evidence shows why a proposition is true. The reason this identity between recursion and induction is a bit hard to see is that most students have not been taught to understand evidence in a computational manner. We will gradually explain this advanced idea that underlies the effectiveness of modern tools for checking program correctness.

# References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[2] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:55–68, 1940.

[3] Alonzo Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, Princeton, 1941.

[4] Gérard Huet, Gilles Kahn, and Christine Paulin-Mohring. The Coq proof assistant : A tutorial : Version 6.1. Technical report, INRIA-Rocquencourt, CNRS and ENS Lyon, August 1997.

[5] G. Kahn. Natural semantics. In G. Vidal-Naquet F. Brandenburg and M. Wirsing, editors, *STACS '87*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, Berlin, 1987.

[6] J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[7] J. McCarthy et al. *Lisp 1.5 Users Manual*. MIT Press, Cambridge, MA, 1962.