NOTES ON TYPES AND PROGRAMMING LANGUAGES

NEAL PARIKH

Abstract. This document aims to serve as a brief outline and concise reference for *Types and Programming Languages* by B. Pierce. The sections roughly correspond to the chapters in *TAPL*, but all the ML implementation chapters are omitted.

1. Introduction

Type systems are the most popular and best established lightweight formal methods for helping ensure that a system behaves correctly with respect to some specification of its desired behavior.

Definition 1.1 (Type System)**.** A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.

A type system can be regarded as calculating a kind of static approximation to the run-time behaviors of the terms in a program. Type systems are good for detecting errors, abstraction, and documentation.

Languages can be statically (compile time) or dynamically (runtime) type checked. Languages can also be safe or unsafe; safe languages protect their own abstractions. Intuitively, if a program written in a safe language passes the typechecker, it will never screw up in an arbitrary way. Java, ML, etc. are safe; C, C++, etc. are unsafe (largely because of casting).

2. Mathematical Reference

Definition 2.1 (Relation)**.** An n-place relation on a collection of sets $S_1, S_2, ..., S_n$ is a set $R \subseteq S_1 \times S_2 \times ... \times S_n$ of n-tuples.

Definition 2.2 (Predicate)**.** A one-place relation on S is called a predicate on S. We can say that P is true of $s \in S$ if $s \in P$; we can also write $P(s)$ and think of P as a function mapping elements of S to truth values.

Definition 2.3 (Partial Function)**.** A relation R on sets S and T is a partial function from S to T if $(s, t_1) \in R$ and $(s, t_2) \in R \implies t_1 = t_2$. If $\text{dom}(R) = S$, then R is a total function.

Definition 2.4 (Preorder)**.** A reflexive and transitive relation R on S is called a preorder on S. A preorder that is also antisymmetric is a partial order. A partial order is a total order if $\forall s, t \in S$, either $t \leq s$ or $s \leq t$. We can also define joins and meets of elements of partially ordered sets.

Definition 2.5 (Reflexive Closure)**.** The reflexive closure of a binary relation R on S is the smallest reflexive relation R' that contains R. Transitive closures are defined similarly.

3. Untyped Arithmetic Expressions

3.1. **Introduction.** The language under consideration in this section is described by the following grammar (where t is a *metavariable* in that it is a placeholder for a term in the actual language):

```
t ::=
    true
    false
    if t then t else t
    0
    succ t
    pred t
```

Date: December 22, 2004.

`iszero t`

A program in the present language is just a term built from the forms given by the grammar above. We can use standard arabic numerals for numbers, which are represented formally as nested applications of `succ` to 0. The results of evaluation will always be booleans or numbers; such terms are called *values*.

The current syntax of terms permits the formation of nonsensical terms like `succ true`; these are exactly the kind of programs we want the type system to exclude.

3.2. **Syntax.** There are several ways of defining our language syntax in addition to the grammar given above; for example, inductively, concretely, or by inference rules.

Definition 3.1 (Terms, Inductively)**.** The set of terms is the smallest[1] set T such that

1. $\{$`true, false, 0`$\} \subseteq T$
2. if $t_1 \in T$, then $\{$`succ` t_1`, pred` t_1`, iszero` $t_1\} \subseteq T$
3. if $t_1, t_2, t_3 \in T$, then `if` t_1 `then` t_2 `then` $t_3 \in T$.

If we were to define terms by inference rules, we would start with a set of axioms like `true` $\in T$, `false` $\in T$, and $0 \in T$; then we could use these axioms to define rules like $t_1 \in T \implies$ `succ`$(t_1) \in T$. The other rules follow similarly.

The concrete definition of terms is simple. For each $i \in \mathbb{N}$, define a set S_i such that $S_0 = \varnothing$ and S_{i+1} contains `pred, succ, iszero` t_1 for $t_1 \in S_i$ (and similarly for the conditional statement). Then $S = \cup_i S_i$.

It can be proved that all of the above approaches are equally strong in the sense that they describe the same set. The outline of the proof is as follows: T was defined to be the smallest set satisfying certain conditions, so it suffices to show (a) that S satisfies these conditions and (b) that S is the smallest such set (i.e., that any set satisfying the conditions contains S).

3.3. **Induction on Terms.** If $t \in T$, one of three things must be true:

1. t is a constant.
2. t has the form `succ`$|$`pred`$|$`iszero` t_1 for some smaller term t_1
3. t has the form `if` t_1 `then` t_2 `else` t_3 for some smaller terms t_1, t_2, t_3.

(Above, by "smaller term," we use the following definition of size.) So we can give inductive definitions of functions over the set of terms and inductive proofs of the properties of terms.

Example 3.1. The size of a term t, written size(t), is defined as follows:

1. `size(true)` = `size(false)` = `size(0)` $= 1$;
2. `size(succ(`t_1`))` = `size(pred(`t_1`))` = `size(iszero(`t_1`)` = `size(`t_1`)` + 1;
3. `size(if` t_1 `then` t_2 `else` t_3`)` = `size(`t_1`)` + `size(`t_2`)` + `size(`t_3`)`.

In other words, the size of t is the number of nodes in its abstract syntax tree. One can define the depth of t very similarly. We can then prove various facts by induction on size or depth (which corresponds to strong induction on naturals) or by structural induction (which corresponds to the ordinary natural number induction principle). Structural induction is preferable, since it works on terms directly.

3.4. **Operational Semantics.** Having formulated our language syntax rigorously, we need a precise definition of how terms are evaluated. There are three main approaches to formalizing semantics:

1. *Operational semantics* specifies the behavior of a language by defining a simple abstract machine for it that uses the terms of the language as its machine code. A state of the machine is a term, and the machine's behavior is defined by a transition function that – for any state – either gives the next state by performing some simplification or declaring that the machine has halted. The meaning of a term t can be taken as the final value the machine reaches when started with t as its initial state.
2. *Denotational semantics* takes the meaning of a term to be a mathematical object such as a number or function. Giving denotational semantics for a language consists of finding a collection of semantic domains and then defining an interpretation function mapping terms into elements of the domains.
3. *Axiomatic semantics* takes the laws themselves as the definition of the language; the meaning of a term is just what can be proved about it.

[1]A smallest set satisfying certain properties can be thought of as the intersection of all sets satisfying these properties.

We'll work exclusively with operational semantics.

3.5. **Evaluation.**

Definition 3.2 (Satisfying a Rule)**.** A rule is *satisfied* by a relation if, for each instance of the rule, either the conclusion is in the relation or one of the premises is not.

Definition 3.3 (One Step Evaluation Relation)**.** The *one step evaluation relation* is the smallest binary relation on terms satisfying the given evaluation rules. When the pair (t, t') is in the evaluation relation, we say that $t \to t'$ is derivable.

One step evaluation is written $t \to t'$ (t evaluates to t' in one step).

For example, the evaluation rule E-IfTrue is defined as $\texttt{if true then } t_2 \texttt{ else } t_3 \to t_2$. E-IfFalse is similar. E-If is more interesting:

$$\frac{t_1 \to t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \to \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3}$$

We read this as saying that we need the statement above the horizontal line to deduce the statement below the line. E-If is simple enough; it means that if we know that the guard evaluates to something, then the whole conditional evaluates to a new conditional with this new guard. This implicitly specifies the order of evaluation; we cannot simplify a then or else subexpression before the if. Once the guard has been fully evaluated, E-IfTrue and E-IfFalse tell us what to do in either case.

The derivability of a given evaluation can be justified by showing a derivation tree whose leaves are labeled with instances of E-IfTrue or E-IfFalse and whose internal nodes are labeled with instances of E-If. [Give an example of a derivation tree.] (The trees will always be linear; since all evaluation rules have only one premise, there's no way to create a branching derivation tree.) The notion of a derivation tree gives rise to a proof technique called *induction on derivations.*

Theorem 3.1 (Determinacy of One-Step Evaluation)**.** *If $t \to t'$ and $t \to t''$, then $t' = t''$.*

One could think of evaluation as a partial function.

Definition 3.4 (Normal Form)**.** A term t is in normal form if there is no t' such that $t \to t'$.

Theorem 3.2. *Every value is in normal form.*

Even when we enrich the system with other constructs, we'll always arrange that 3.2 stays valid. Being in normal form is part of what it means to be a value, so any language definition in which this isn't the case is broken.

In the present system, the converse (if t is in normal form, then t is a value) is also true, but it isn't the case in general. Moreover, normal forms that are not values play a role in the analysis of runtime errors.

Definition 3.5 (The Multi-Step Evaluation Relation)**.** The multi-step evaluation relation $\to^*$ is the reflexive, transitive closure of one-step evaluation.

Theorem 3.3 (Uniqueness of Normal Forms)**.** *If $t \to^* u$ and $t \to^* u'$ where u and u' are normal forms, then $u = u'$.*

Proof. This follows from 3.1. □

Theorem 3.4 (Termination of Evaluation)**.** *For every term t there is a normal form n such that $t \to^* n$.*

Being able to evaluate every term to a value is a property that need not hold in richer languages, but clearly does in this one. Most termination proofs have the same basic form. First, we choose some well-founded[2] set S and give a function f mapping terms into S. Next, we show that whenever $t \to t'$, $f(t') < f(t)$. An infinite sequence of evaluation steps beginning from t can be mapped via f into an infinite descending chain of elements of S. Since S is well-founded, there can be no such chain. The function f is often called a *termination measure* for the evaluation relation.

[2]A well-founded set is a partially ordered set in which every nonempty subset has a minimal element (equivalently, it is a partially ordered set that contains no infinite descending chains).

To extend the definition of evaluation to arithmetic expressions, we need to add a new syntactic category of numeric values. This allows us to specify that the final result of evaluating an arithmetic expression is either 0 or `succ` nv; this forbids nonsense like `succ true`. We can then define evaluation rules that specify whether a value is numeric, such as E-PredSucc: `pred (succ` nv_1`)` $\to nv_1$. Since `pred 0` is not a numeric value, we have an evaluation rule that defines it to be 0. But something like `succ false` isn't defined to evaluate to anything (it's a normal form). Such terms are called stuck.

Definition 3.6 (Stuck Terms)**.** A closed term is stuck if it is in normal form but not a value.

Stuck terms give a simple notion of runtime error. They characterize situations where the program has reached a meaningless state. In more concrete implementations, such errors might correspond to failures like segmentation faults.

4. The Untyped Lambda Calculus

4.1. **Background.** The lambda calculus is a formal system invented by Church to investigate the Entscheidungsproblem. In 1928, Hilbert and Ackermann posed the Entscheidungsproblem (*decision problem*) in its modern form: find a general algorithm that decides whether given first-order statements are universally valid or not. Church answered in the negative, and to do so, he needed a formalization of decidability or computability. In 1936, he invented the lambda calculus and defined the notion of computable function within this framework. Computability can also be formalized by Turing machines as well as other formal systems.

In the lambda calculus, all computation is reduced to the basic operations of function definition and application. Lambda calculus is powerful because it can be viewed simultaneously as a simple programming language *in which* computations can be described and as a mathematical object *about which* rigorous statements can be proved.

4.2. **Basics.** In lambda calculus, everything is a function, including the arguments for and return values of other functions. The syntax of the lambda calculus comprises just three terms. A variable x by itself is a term; the abstraction of a variable x from a term t, written $\lambda x.t$, is a term; and the application of a term t_1 to a term t_2, written $t_1\ t_2$, is a term.

Application can be thought of as applying an algorithm to a piece of data: fa means that the function f is applied to the variable a; in other words, a is the argument "passed to" f. An abstraction $\lambda x.t(x)$ can be thought of as a function $x \mapsto t(x)$.

4.2.1. *Abstract and Concrete Syntax.* It is useful to distinguish two levels of structure in the syntax of programming languages. The concrete syntax refers to the strings of characters that programmers directly read and write. Abstract syntax is a simpler internal representation of programs as labeled trees; the tree representation renders the structure of the terms immediately obvious, making it suitable for rigorous language definitions and proofs about them as well as the internals of compilers and interpreters. Our focus is on abstract syntax.

We can express lambda terms as abstract syntax trees, but it's more convenient to write them out concretely; to avoid gratuitous parentheses, we establish some associativity rules. Application associates to the left: $stu = (st)u$. Abstraction associates to the right: $\lambda x.\lambda y.r = \lambda x.(\lambda y.r)$. Combining the two, $\lambda x.\lambda y.xyx = \lambda x.(\lambda y.((xy)x))$.

4.2.2. *Scope.*

Definition 4.1 (Bound variable)**.** An occurrence of a variable x is said to be bound when it occurs in the body t of an abstraction $\lambda x.t$.

We can also say that λx is a *binder* whose scope is t.

Definition 4.2 (Free variable)**.** An occurrence of a variable x is free if it appears in a position where it is not bound by an enclosing abstraction on x.

For example, x in xy or $\lambda y.xy$ is free. x in $\lambda x.x$ is bound. In $(\lambda x.x)x$, the first occurrence of x is bound and the second is free.

Definition 4.3 (Closed term)**.** A term with no free variables is closed. Closed terms are also called combinators.

The simplest combinator is the identity function, which returns its argument: $\lambda x.x$.

4.2.3. *Operational Semantics.* The sole means by which terms "compute" in the lambda calculus is the application of functions to arguments (which are also functions, since that's all we have). Each step consists of rewriting abstractions applied to terms as follows:

$$(\lambda x.t)s \to [x \mapsto s]t,$$

where $[x \mapsto s]t$ is the term obtained by replacing all free occurrences of x in t by s. Simply put, we're substituting the argument for the bound variable. A term of the form $(\lambda x.t)s$ is called a *redex* (reducible expression), and the substitution described above is called *beta reduction.*

Several different evaluation strategies for the lambda calculus have been studied over the years, but most languages use a call by value strategy, in which only outermost redexes are reduced and where a redex is reduced only when its right-hand side has already been reduced to a value.

Example 4.1. $(\lambda x.x)\underline{((\lambda x.x)(\lambda z.(\lambda x.x)z))} \to \underline{(\lambda x.x)(\lambda z.(\lambda x.x)z)} \to \lambda z.(\lambda x.x)z$. This cannot be reduced any further under call by value reduction. Under other reduction strategies, this could have been further reduced to $\lambda z.z$.

4.3. **Programming in the Lambda Calculus.**

4.3.1. *Multiple Arguments.* The lambda calculus provides no built-in support for multi-argument functions, but it's easy to achieve the same effect using higher order functions that return functions as results.

So instead of writing $f = \lambda(x, y).s$, we write $f = \lambda x.\lambda y.s$. In words, f is a function that, given a value v for x, yields a function that, given a value w for y, yields the desired result. We can now just apply f to its arguments one at a time: fvw reduces to $[y \mapsto w][x \mapsto v]s$. This transformation of multi-argument functions into higher-order functions is called *currying.*

4.3.2. *Church Booleans.* We can also encode boolean values and conditionals in the lambda calculus. Essentially, we come up with lambda abstractions that *behave* like booleans would. `tru` = $\lambda t.\lambda f.t$, and `fls` = $\lambda t.\lambda f.f$. These terms represent true and false in that we can use them to perform the operation of testing a boolean value.

We can use application to define a combinator `test` such that `test` bvw reduces to v when b is `tru` and w when b is `fls`: `test` = $\lambda l.\lambda m.\lambda n.lmn$. All `test` bvw does is reduce to bvw, but b picks its first argument if it's `tru` and its second if it's `fls`, which produces the desired effect.

Example 4.2. `tru` $vw = (\lambda t.\lambda f.t)vw \to (\lambda f.v)w \to v$. In the last step, since f is not bound, w is just discarded (under β-reduction, we'd replace the occurrences of the bound variable, but there aren't any). With `fls`, the first argument would have been discarded.

Really, `tru` and `fls` are doing all the work, but we define `test` for readability.

We can also define a combinator `and` that takes two boolean values b and c and returns c if b is `tru` and `fls` if b is `fls`: `and` = $\lambda b.\lambda c.bc$ `fls`. So if b is `tru` and c is `tru`, it would return `tru` (c); if b is `tru` and c is `fls`, it would return `fls` (again c).

Example 4.3. `and tru fls` = $(\lambda b.\lambda c.bcf)tf \to (\lambda c.tcf)f \to tff \to f$ where f is `fls` and t is `tru`. Remember from the previous example that `tru` just returns its first argument.

We could also define logical "or" and "not" combinators.

4.3.3. *Pairs.* Using booleans, we can encode pairs of values as terms. Essentially, we can define combinators `pair`, `fst`, and `snd` such that `fst (pair v w)` yields v and so on. `pair` = $\lambda f.\lambda s.\lambda b.bfs$; then `fst` and `snd` can be implemented by just supplying the appropriate boolean.

4.3.4. *Church Numerals.* In this case, we represent each number n by a combinator c_n that takes two arguments, s (for successor) and z (for zero), and applies s n times to z. Intuitively, a number n is represented by a function that does something n times.

Without going into excessive detail, $c_0 = \lambda s.\lambda z.z$, $c_1 = \lambda s.\lambda z.sz$, $c_2 = \lambda s.\lambda z.s(sz)$, and so on. Note that c_2 does not evaluate to 2; it *is* 2. We can also define a successor function, addition, multiplication, equality tests, and so on.

4.3.5. *Enriching the Calculus.* Although we can do all the programming we ever need without going outside the pure untyped lambda calculus, it's often useful to include booleans and numbers (and perhaps other data types) as well. λ stands for the pure lambda calculus, while λNB stands for the enriched system with booleans and arithmetic expressions.

If we do define primitive booleans, we can convert back and forth between Church booleans and primitive booleans without much trouble.

4.3.6. *Recursion.* Some terms cannot be evaluated to a normal form. Terms with no normal form are said to diverge. For example, $(\lambda x.xx)(\lambda x.xx)$ is divergent (it always recreates itself). This combinator has a useful generalization called the fixed point combinator, which produces the fixed point of any function it is applied to.

Theorem 4.1 (Fixed Point Theorem)**.** *Every function has a fixed point:* $\forall f\ \exists x\ fx = x$. *Additionally, there exists a (call-by-value) fixed point combinator* Y *such that* $\forall f\ f(Yf) = Yf$. *The combinator* $Y = \lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy))$.

The fixed point combinator is useful when we want to write recursive function definitions. The effect of "unrolling" the recursive definition where it occurs can be achieved by first defining $g = \lambda f.\langle$body containing $f\rangle$ and then $h = Yg$. Intuitively, g is a function that takes a function f and produces a "more accurate" f: in other words, it unrolls one more level of recursion. In the case of something like the factorial function, we have to apply g to `fact` a countably infinite number of times to produce the "true" factorial function defined for all $\mathbb{N}$. The details of how this works require too much detail for this document, but see *TAPL* 67.

4.3.7. *Representation.* In terms of their effects on the overall results of programs, there is no observable difference between the real numbers and their Church numeral representations.

4.4. **Formalities.** We now consider the syntax and operational semantics of the lambda calculus in more detail; the operation of substituting a term for a variable involves some interesting subtleties.

4.4.1. *Syntax.*

Definition 4.4 (Terms)**.** Let V be a countable set of variable names. The set of terms T is the smallest set such that

(1) $x \in T \quad \forall x \in V$;
(2) if $t_1 \in T$ and $x \in V$, then $\lambda x.t_1 \in T$;
(3) if $t_1, t_2 \in T$, then $t_1 t_2 \in T$.

Definition 4.5 (Free variables)**.** The set $FV(t)$ of free variables of a term t is defined as follows:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.t) &= FV(t) \setminus \{x\} \\ FV(ts) &= FV(t) \cup FV(s) \end{aligned}$$

4.4.2. *Substitution.* The operation of substitution is complicated because we need to make sure not to turn bound variables into free variables and vice versa. We use two different definitions: the first is compact and intuitive, and works well for mathematical definitions and proofs; the second (de Bruijn's) is notationally heavier but is more convenient for ML implementations.

We want to ensure that we don't change the state of bound/free variables, e.g. $[x \mapsto y](\lambda x.x) = \lambda x.y$ and $[x \mapsto z](\lambda z.x) = \lambda z.z$ shouldn't be allowed. We also want to make sure that our restrictions are not too restrictive in that they don't do anything at all for certain expressions. We decide to work with terms up to α-conversion, where α-conversion is the operation of consistently renaming a bound variable in a term. Terms that differ only in the names of bound variables are interchangeable in all contexts; in the definition below, we can always assume that the bound variable y is different from both x and the free variables of s.

Definition 4.6 (Substitution)**.**

$$\begin{cases} [x \mapsto s]\, x = s & \\ [x \mapsto s]\, y = y & \text{if } y \neq x \\ [x \mapsto s](\lambda y.t_1) = \lambda y.[x \mapsto s]t_1 & \text{if } y \neq x \text{ and } y \notin FV(s) \\ [x \mapsto s](t_1 t_2) = [x \mapsto s]t_1 [x \mapsto s]t_2 & \end{cases}$$

4.4.3. *Operational Semantics.* We define the following three evaluation rules:

$$\frac{t \to t'}{ts \to t's} \quad \text{E-App1}$$

$$\frac{s \to s'}{vs \to vs'} \quad \text{E-App2}$$

$$(\lambda x.t)v \to [x \mapsto v]t \quad \text{E-AppAbs}$$

Above, v is a value (lambda abstraction) and s, t are terms (variables, abstractions, or applications). These rules implicitly control the order of evaluation. For example, for an application ts, we first use E-App1 to reduce t to a value, then use E-App2 to reduce s to a value, and finally use E-AppAbs to perform the application itself.

5. Nameless Representation of Terms

In order to have a single representation for each term (for purposes of implementations), we can devise some representation of variables and terms that does not require α-conversion.

5.1. **Terms and Contexts.** We can represent terms more straightforwardly by making variable occurrences point directly to their binders (rather than naming them). This can be accomplished by replacing named variables by natural numbers, where k stands for the variable bound by the kth enclosing λ. The ordinary term $\lambda x.x$ corresponds to $\lambda.0$ and $\lambda x.\lambda y.x(yx)$ to $\lambda.\lambda.1(01)$. Nameless terms are also called *de Bruijn terms*, and their numeric values are called *de Bruijn indices* (also called *static distances* in compilers).

Definition 5.1 (Terms)**.** Let T be the smallest family of sets $\{T_0, T_1, ...\}$ such that

(1) $k \in T_n$ whenever $0 \leq k < n$;
(2) if $t_1 \in T_n$ and $n > 0$, then $\lambda.t_1 \in T_{n-1}$;
(3) if $t_1 \in T_n$ and $t_2 \in T_n$, then $(t_1 t_2) \in T_n$.

The elements of each T_n are called n-terms, which contain *at most* n free variables numbered between 0 and $n-1$.

Each closed ordinary term has just one de Bruijn representation, and two ordinary terms are equivalent up to α-conversion iff they have the same de Bruijn representation.

To deal with terms containing free variables, we need the notion of a *naming context.* If a variable is free, we cannot see its binder, so it's not clear what number to assign to it; the solution is to choose a naming context of de Bruijn indices to free variables, and use this assignment consistently when we need to choose numbers for free variables. For example, if $\Gamma = \{x \mapsto 4, y \mapsto 3, z \mapsto 2, a \mapsto 1, b \mapsto 0\}$, then $x(yz) = 4(32)$ and $\lambda w.yw = \lambda.40$. We can represent a naming context as a sequence.

Definition 5.2. Suppose x_0 through x_n are variable names from V. The naming context $\Gamma = x_n, x_{n-1}, ..., x_0$ assigns the de Bruijn index i to each x_i. We write $\text{dom}(\Gamma)$ for the set of variable names mentioned in Γ.

5.2. **Shifting and Substitution.** In order to define a substitution operation on de Bruijn terms, we need an auxiliary operation called *shifting* that renumbers the indices of the free variables in a term. The motivation for this is that we don't want the indices of free and bound variables to conflict; e.g. $\lambda w.wb \to \lambda.00$ is a conversion we want to avoid. Intuitively, if we have an expression with n binders, the bound variables could number up to $n-1$, so we effectively take our naming context and "shift" it by n to avoid ambiguity. We have to be careful, however, since we want to make sure not to shift bound variables.

Definition 5.3 (Shifting)**.** The d-place shift of a term t above cutoff c, written $\uparrow_c^d (t)$, is defined as follows:

$$\begin{aligned} \uparrow_c^d (k) &= \begin{cases} k & k < c \\ k + d & k \geq c \end{cases} \\ \uparrow_c^d (\lambda.t_1) &= \lambda. \uparrow_{c+1}^d (t_1) \\ \uparrow_c^d (t_1 t_2) &= \uparrow_c^d (t_1) \uparrow_c^d (t_2) \end{aligned}$$

We write $\uparrow^d (t)$ for $\uparrow_0^d (t)$.

Example 5.1. $\uparrow^2 (\lambda.\lambda.1(02)) \to \lambda. \uparrow_1^2 \lambda.1(02) \to \lambda.\lambda. \uparrow_2^2 1(02) \to \lambda.\lambda. \uparrow_2^2 1(\uparrow_2^2 0 \uparrow_2^2 2) \to \lambda.\lambda.1(04)$.

Definition 5.4 (Substitution)**.** The substitution of a term s for variable number j in a term t, written $[j \mapsto s]t$, is defined as follows:

$$\begin{aligned} [j \mapsto s] (k) &= \begin{cases} s & k < j \\ k & \text{otherwise} \end{cases} \\ [j \mapsto s] (\lambda.t_1) &= \lambda. \left[j + 1 \mapsto \uparrow^1 (s)\right] (t_1) \\ [j \mapsto s] (t_1 t_2) &= [j \mapsto s] (t_1) [j \mapsto s] (t_2) \end{aligned}$$

Example 5.2. We want to convert the following substitution to a nameless form assuming the context $\Gamma = \{a, b\}$ and then evaluate them using the above definition. $[b \mapsto a](b(\lambda x.\lambda y.b)) = [0 \mapsto 1](0(\lambda.\lambda.2)) \to 1(\lambda.\lambda.3) = a(\lambda x.\lambda y.a)$.

5.3. **Evaluation.** The only thing we need to change to define the evaluation relation on nameless terms is the β-reduction rule, which must now use our new nameless substitution operation. So the β-reduction rule looks like this:

$$(\lambda.t_{12})v_2 \to \uparrow^{-1} ([0 \mapsto \uparrow^1 (v_2)]t_{12}).$$

The other rules are identical.

It is not necessary to place too much emphasis on de Bruijn representations; the main idea to take away is that they are useful for implementations.

6. Typed Arithmetic Expressions

We return to the simple language of Section 3 and augment it with static types.

6.1. **Types.** Stuck terms correspond to meaningless or erroneous programs, so we would like to be able to tell that a term's evaluation will definitely not get stuck without actually doing the evaluation. To do this, we need to distinguish between terms that result in a numeric value and terms that result in a boolean, so we introduce the types `Nat` and `Bool`.

Saying that a term t has type T means that we can see this statically (without any evaluation of t). Our analysis will be conservative (making use only of static information), so we won't be able to conclude that terms like `if true then 0 else false` have any type at all, even though they are not stuck.

6.1.1. *The Typing Relation.* The typing relation for arithmetic expressions (written $t : T$) is defined by a set of inference rules assigning types to terms. The typing rules T-True[3], T-False, and T-If are defined only for booleans, and the rules T-Zero, T-Succ, T-Pred, and T-IsZero are defined only for numbers. The typing rules themselves are straightforward; for example, the most complicated one is T-If:

$$\frac{t_1 : \texttt{Bool} \quad t_2 : T \quad t_3 : T}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T}.$$

t_1 must be a `Bool`, and t_2 and t_3 must be of the same type. The rules for numbers are very simple, and T-True and T-False are just axioms.

Definition 6.1 (Typing Relation)**.** The typing relation for arithmetic expressions is the smallest binary relation between terms and types satisfying all instances of the typing rules above. A term t is *typable* (or *well typed*) if there is a T such that $t : T$.

[3]Note that we use T-* for typing rules and E-* for evaluation rules.

Lemma 6.1 (Inversion of the Typing Relation)**.** *This allows as to make statements like, "If this term has any type at all, it must have type t."*

(1) *If* $\texttt{true} : R$*, then* $R = \texttt{Bool}$*.*
(2) *If* $\texttt{false} : R$*, then* $R = \texttt{Bool}$*.*
(3) *If* $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$: R*, then* $t_1 : \texttt{Bool}$*,* $t_2 : R$*, and* $t_3 : R$*.*
(4) *If* $0 : R$*, then* $R = \texttt{Nat}$*.*
(5) *If* $\texttt{succ } t_1 : R$*, then* $R = \texttt{Nat}$ *and* $t_1 : \texttt{Nat}$*.*
(6) *If* $\texttt{pred } t_1 : R$*, then* $R = \texttt{Nat}$ *and* $t_1 : \texttt{Nat}$*.*
(7) *If* $\texttt{iszero } t_1 : R$*, then* $R = \texttt{Bool}$ *and* $t_1 : \texttt{Nat}$*.*

Proof. Follows from the definition of the typing relation. □

The Inversion Lemma is sometimes called the Generation Lemma, since it shows how a proof of a valid typing statement could have been generated. The Inversion Lemma leads directly to an algorithm for calculating the terms of types, since it tells us how to calculate the type of a term of each syntactic form given the types of its subterms. (We return to this in the next section.)

Like an evaluation derivation, a typing derivation is a tree of instances of the typing rules. Each pair (t, T) in the typing relation is justified by a derivation with conclusion $t : T$. *Statements* are formal assertions about the typing of programs, *typing rules* are implications between statements, and *derivations* are deductions based on typing rules.

Theorem 6.2 (Uniqueness of Types)**.** *If* t *is typable, then its type is unique. Moreover, there is just one derivation of this typing from the inference rules listed above.*

In our simple type system, every term has a single type (if any), so there is always only one derivation tree demonstrating this fact. When subtyping and other complexities are introduced, this will change.

6.2. **Safety = Progress + Preservation.** The most basic property of this type system is safety: we want to know that well-typed terms do not get stuck. This is shown in the progress and preservation theorems.

Lemma 6.3 (Canonical Forms)**.** *(1) If* v *is of type* `Bool`*, then* v *is either* `true` *or* `false`*. (2) If* v *is a value of type* `Nat`*, then* v *is a numeric value according to the grammar in Section 3.*

Proof. According to the grammar in Section 3, values can have four forms: `true`, `false`, 0, and `succ` *nv* (where *nv* is a numeric value). For part (1), the first two cases are the desired results, and by the Inversion Lemma, the last two cannot occur. The proof of part (2) is similar. □

The Canonical Forms Lemma is useful in the proof of the Progress Theorem, since it allows us to specify which typing rule might apply in certain situations.

Theorem 6.4 (Progress)**.** *If* t *is well-typed, then* t *is a value or there is some* t' *such that* $t \to t'$*.*

Theorem 6.5 (Preservation)**.** *If* $t : T$ *and* $t \to t'$*, then* $t' : T$*.*

Unlike uniqueness of types, progress and preservation will be basic requirements for all the type systems we consider.

7. Simply Typed Lambda Calculus

We introduce the most elementary member of the family of typed languages we will be studying in the rest of this document.

7.1. **Function Types.** We want to construct a type system similar to that of the previous section for a language combining booleans with the primitives of the pure lambda calculus. We want to introduce typing rules for variables, abstractions, and applications that (a) maintain type safety (satisfy the progress and preservation theorems) and (b) are not too conservative (they should assign types to most of the programs we care about).

To extend the type system for booleans to include functions, we need to add the infinite family of types $T_1 \to T_2$.

Definition 7.1. The set of simple types over the type `Bool` is generated by the following grammar:

`T ::= Bool | T → T`

The type constructor → is right associative.

Example 7.1. (`Bool → Bool`) → (`Bool → Bool`) is the type of functions that take boolean-to-boolean functions as arguments and return them as results.

7.2. **The Typing Relation.** In order to assign a type to an abstraction, we annotate the abstraction with the intended type of its arguments (so we can see what will happen when the abstraction is applied to some argument). Languages in which type annotations in terms are used to guide the typechecker are called explicitly typed; languages in which we ask the typechecker to infer this information are implicitly typed. This document will largely concentrate on explicit typing.

By providing the type of the argument to the abstraction, we can obtain a typing rule for abstractions. This changes the typing relation from a two-place relation ($t : T$) to a three-place relation ($\Gamma \vdash t : T$), where Γ is a set of assumptions about the types of the free variables in t.

Formally, a typing context Γ is a sequence of variables and their types, and the comma operator extends Γ by adding a new binding on the right. The empty context is sometimes written $\varnothing$, but is usually omitted. To avoid confusion between the new binding and any bindings already in Γ, we require that x be chosen so it is distinct from variables bound by Γ; Γ can thus be thought of as a finite function from variables to their types, and we can write dom(Γ) for the set of variables bound by Γ. We obtain the following general rule for typing abstractions (T-Abs):

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2}.$$

T-Var is simple. A variable has whatever type we're currently assuming it to have:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}.$$

Finally, our typing rule for applications, T-App, is the following:

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}}.$$

Our new typing rule for conditionals (T-If), however, now allows us to type conditionals whose branches are functions (because the metavariable T can now be instantiated to any function type):

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ then } t_3}.$$

Instances of the typing rules for the simply typed lambda calculus ($\lambda_\to$) can be combined into derivation trees, just as for arithmetic expressions.

7.3. **Properties of Typing.** We need a few basic lemmas before we can prove type safety; the only significant new requirement is a Substitution Lemma. First, we state an Inversion Lemma.

Lemma 7.1 (Inversion)**.**
1. *If* $\Gamma \vdash x : R$, *then* $x : R \in \Gamma$.
2. *If* $\Gamma \vdash \lambda x : T_1 . t_2 : R$, *then* $R = T_1 \to T_2$ *for some* R_2 *with* $\Gamma, x : T_1 \vdash t_2 : R$.
3. *If* $\Gamma \vdash t_1 t_2 : R$, *then there is some type* T_{11} *such that* $\Gamma \vdash t_1 : T_{11} \to R$ *and* $\Gamma \vdash t_2 : T_{11}$.
4. *If* $\Gamma \vdash \texttt{true} : R$, *then* $R = \texttt{Bool}$.
5. *If* $\Gamma \vdash \texttt{false} : R$, *then* $R = \texttt{Bool}$.
6. *If* $\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : R$, *then* $\Gamma \vdash t_1 : \texttt{Bool}$ *and* $\Gamma \vdash t_2, t_3 : R$.

The following theorem tells us that well-typed terms are in one-to-one correspondence with their typing derivations: the the typing derivation can be recovered uniquely from the term, and vice versa.

Theorem 7.2 (Uniqueness of Types)**.** *In a given typing context* Γ, *a term* t *(with free variables all in the domain of* Γ*) has at most one type: if a term is typable, its type is unique. Moreover, there is just one derivation of this typing built from the inference rules that generate the typing relation.*

Later on, this simple correspondence will not hold, and significant work may be necessary to show that typing derivations can be recovered effectively from terms.

Lemma 7.3 (Canonical Forms). *If v is a value of type* `Bool`*, then v is* `true` *or* `false`*. If v is a value of type $T_1 \to T_2$, then $v = \lambda x : T_1.t_2$.*

Using the Canonical Forms Lemma, we can prove a Progress Theorem similar to that of the previous section; the only difference is that we are interested only in closed terms. (For open terms, the progress theorem fails.)

Theorem 7.4 (Progress). *If t is a closed, well-typed term, then either t is a value or there is some t' such that $t \to t'$.*

We now want to show that evaluation preserves types. We state a few lemmas that will be useful to prove later theorems.

Lemma 7.5 (Permutation). *If $\Gamma \vdash t : T$ and Δ is a permutation of Γ, then $\Delta \vdash t : T$. Moreover, the latter derivation has the same depth as the former.*

Lemma 7.6 (Weakening). *If $\Gamma \vdash t : T$ and $x \notin dom(\Gamma)$, then $\Gamma, x : S \vdash t : T$. Moreover, the latter derivation has the same depth as the former.*

We can read the Weakening Lemma as saying that not all hypotheses need to be used; in other words, we can add arbitrary hypotheses to a derivation to weaken it. These lemmas are useful in proving that well-typedness is preserved when variables are substituted with terms of appropriate types. This is the Substitution Lemma mentioned earlier.

Lemma 7.7 (Preservation under Substitution). *If $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s]t : T$.*

Theorem 7.8 (Preservation). *If $\Gamma \vdash t : T$ and $t \to t'$, then $\Gamma \vdash t' : T$.*

7.4. **The Curry-Howard Isomorphism.** The Curry-Howard Isomorphism describes an interesting connection between logic and programming languages.[4] Some of the main observations of the isomorphism are (1) that propositions of logic correspond to types of a programming language; (2) that proofs in logic correspond to expressions in a programming language; (3) that verifying the correctness of a proof corresponds to type-checking its corresponding expression; and (4) that computation (β-reduction) corresponds to the logical operation of proof simplification by cut elimination. The isomorphism is often referred to as the "propositions as types" or "proofs as programs" analogy.

An in-depth discussion would be outside the scope of these notes, but briefly, a proof of a proposition P in a constructive logic requires concrete evidence for P (in other words, the law of the excluded middle, double negation, proof by contradiction, and other equivalent proof rules from classical logic are not allowed). Such evidence has a strongly computational feel. A proof of a proposition $P \supset Q$ can be viewed as a mechanical procedure that, given a proof of P, constructs a proof of Q. Similarly, a proof of $P \wedge Q$ consists of a proof of P together with a proof of Q. This observation gives rise to a correspondence between propositions as types; proposition $P \supset Q$ as the type $P \to Q$; proposition $P \wedge Q$ as the type $P \times Q$; proof of P as term t of type P; and proposition P is provable as type P is inhabited by some term.

The Curry-Howard isomorphism can be extended to a huge variety of type systems and logics, but initially we will be interested in the correspondence between simply typed lambda calculus and Gentzen's sequent calculus.

7.5. **Erasure and Typability.** In many full-scale programming languages, programs are converted back to an untyped form before they are evaluated. This can be formalized using an erasure function mapping simply typed terms to corresponding untyped terms.

Definition 7.2 (Erasure). The erasure of a simply typed term t is defined as follows:

$$\begin{aligned} erase(x) &= x \\ erase(\lambda x : T_1.t_2) &= \lambda x.erase(t_2) \\ erase(t_1 t_2) &= erase(t_1)erase(t_2) \end{aligned}$$

Evaluation commutes with erasure.

[4]Material in this section draws upon "Lecture notes on the Curry-Howard Isomorphism" by F. Pfenning.

Theorem 7.9. (1) *If $t \to t'$ under the typed evaluation relation, then $erase(t) \to erase(t')$.*

(2) *If $erase(t) \to m$ under the typed evaluation relation, then there is a simply typed term t' such that $t \to t'$ and $erase(t') = m$.*

Currently, this theorem is obvious, but for more interesting languages and compilers it becomes an important property: it tells us that a high-level semantics used by the programmer coincides with an alternative lower-level evaluation strategy used by an implementation of the language.

Definition 7.3 (Typable)**.** A term m in the untyped lambda calculus is said to be typable in $\lambda_{\to}$ if there are some simply typed term t, type T, and context Γ such that $erase(t) = m$ and $\Gamma \vdash t : T$.

8. Simple Extensions

So far, the simply typed lambda calculus is interesting as a theoretical object, but it is not yet much of a programming language. We close the gap with more familiar languages by introducing a number of familiar features. An important theme throughout the section is the concept of derived forms.

8.1. **Base Types.** Every programming language provides a variety of base types like `Bool`or `Nat`plus appropriate primitive operations for manipulating them; for theoretical purposes, it is useful to abstract away from the details of particular base types and instead simply suppose that our language comes equipped with some set $\mathscr{A}$ of unknown base types with no primitive operations at all. This is accomplished simply by including the elements of $\mathscr{A}$ (ranged over by the metavariable A) in the set of types. We will use A, B, C, etc. as the names of base types.

Example 8.1. $\lambda x : A.x$ is the identity function on the elements of A, whatever these may be.

8.2. **The Unit Type.** The singleton type `Unit` is a useful base type often found in languages in the ML family. This type is interpreted in the simplest possible way: we introduce a single element `unit` and a typing rule making `unit` an element of `Unit`. We also add `unit` to the set of possible result values of computations; indeed, `unit` is the only possible result of evaluating an expression of type `Unit`. `Unit`'s main application is in languages with side effects, and one of its uses is similar to the role of the void type in languages like C or Java.

Formally, our new syntactic forms include terms `t ::= unit`, values `v ::= unit`, types `T ::= Unit`; we also have a new typing rule $\Gamma \vdash$ `unit : Unit` and a new derived form $t_1; t_2 \overset{\triangle}{=} (\lambda x : \texttt{Unit}.t_2)t_1$ where $x \notin FV(t_2)$ (explained in more detail in the following section).

8.3. **Derived Forms: Sequencing and Wildcards.** In languages with side effects, it's often useful to evaluate two or more expressions in sequence. The sequencing notation $t_1; t_2$ evaluates t_1, throws away its trivial result, and then evaluates t_2.

There are two different ways to formalize sequencing. One is to follow the pattern we have used for other syntactic forms: add $t_1; t_2$ as a new option in the syntax of terms, and add evaluation rules (E-Seq and E-SeqNext) and a typing rule (T-Seq) capturing the intended behavior. An alternative way is simply to regard $t_1; t_2$ as an abbreviation for the term $(\lambda x : \texttt{Unit}.t_2)t_1$, where the variable x is chosen to be different from the free variables of t_2. (Because $x \notin FV(t_2)$, t_1 is effectively thrown out after being evaluated.) These two presentations are really the same thing: the high level typing and evaluation rules can be derived from the abbreviation. In other words, typing and evaluation commute with the expansion of the abbreviation.

Theorem 8.1 (Sequencing is a Derived Form)**.** *Write λ^E (external language) for for the simply typed lambda calculus with `Unit`, the sequencing construct, and the rules E-Seq, E-SeqNext, and T-Seq; write λ^I (internal language) for the simply typed lambda calculus with `Unit` only. Let $e \in \lambda^E \to \lambda^I$ be the* elaboration function *that translates from the external to the internal language by replacing every occurrence of $t_1; t_2$ with $(\lambda x : \texttt{Unit}.t_2)t_1$, where x is chosen fresh in each case. Now, for each term t of λ^E, we have*

$$t \to_E t' \text{ iff } e(t) \to_I e(t')$$

$$\Gamma \vdash^E t : T \text{ iff } \Gamma \vdash^I e(t) : T$$

where the evaluation and typing relations are annotated with I and E to show which is which.

Theorem 8.1 justifies the use of the term *derived form*, since it shows that the typing and evaluation behavior of sequencing can be derived from the more fundamental operations of abstraction and application. The advantage of introducing such features as derived forms rather than as full-fledged language constructs is that we can extend the programmer-level syntax without adding any complexity to the internal language about which theorems such as type safety must be proved.

Derived forms are often called syntactic sugar, and replacing a derived form with its lower-level definition is called desugaring.

8.4. **Ascription.** Another useful simple feature is the ability to explicitly ascribe a particular type to a given term. We write `t as T` to ascribe type T to the term t, and the typing rule T-Ascribe simply verifies that T is indeed the type of t. The evaluation rule E-Ascribe just throws away the ascription and leaves t to evaluate as usual.

Ascription can be useful both for documenting code and for controlling the printing of complex types. The former is simple; for the latter, we can introduce abbreviations for long or complex type expressions and then ascribe these abbreviations in abstractions. This helps the typechecker print more readable types for various expressions.

In a full-blown programming language, mechanisms for abbreviation for abbreviation and type printing will either be unnecessary (as in Java) or much more tightly integrated into the language (as in OCaml).

A final use of ascription that will be discussed in more detail later is as a mechanism for abstraction. In systems where a term t may have many types, ascription can be used to hide some of these types by telling the typechecker to treat t as if it only had a smaller set of types. Later, we will also discuss a relationship between ascription and casting.

8.5. **Let Bindings.** Being able to give names to some of the subexpressions in complex expressions is useful. In ML, we write $\texttt{let}\ x = t_1\ \texttt{in}\ t_2$ to mean "evaluate t_1 and bind x to the resulting value in t_2." Our let binder uses a call-by-value evaluation order, where the let-bound term must be fully evaluated before evaluation of the let-body can begin.

The typing rule T-Let tells us that the type of a `let` can be calculated by finding the type of the let-bound term, extending the context with a binding of this type, and in this enriched context calculating the type of the body, which is then the type of the whole let expression:

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : T_2}.$$

We also have the new evaluation rule E-Let, which is straightforward:

$$\frac{t_1 \to t_1'}{\texttt{let} x = t_1\ \texttt{in}\ t_2 \to \texttt{let} x = t_1'\ \texttt{in}\ t_2}.$$

We can define `let` as a derived form, but it is more subtle than the technique for sequencing or ascription. We can use a combination of abstraction and application to achieve the effect of let binding:

$$\texttt{let} x = t_1\ \texttt{in}\ t_2 \overset{\triangle}{=} (\lambda x : T_1.t_2)t_1;$$

however, the right-hand side of this abbreviation includes the type annotation T_1, which does not appear on the left-hand side. This information (which the parser needs to obtain from somewhere as it desugars this derived form) comes from the typechecker: we get the needed annotation simply by calculating the type of t_1. More formally, the let constructor is a "little less derived" than the other derived forms: we should regard it not as a desugaring transformation on terms but rather as a transformation on typing derivations that maps a derivation involving `let` to one using abstraction and application. We can derive its evaluation behavior by desugaring, but its typing behavior must be built into the internal language.

8.6. **Pairs.** The simplest kind of compound data structure is a pair (or more generally a tuple) of values. Pairs are very simple; we add two new forms of term–pairing, written $\{t_1, t_2\}$, and projection, written $t.1$ for the first projection from t and $t.2$ for the second projection–plus a product type $T_1 \times T_2$. We add a few simple evaluation rules like $\{v_1, v_2\}.1 \to v_1$ (E-PairBeta1), $t_1 \to t_1' \implies t_1.1 \to t_1'.1$ (E-Proj1), and

$t_1 \to t_1' \implies \{t_1, t_2\} \to \{t_1', t_2\}$ (E-Pair1). (Rules for second projections are similar.) We also add some simple typing rules like

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \{t_1, t_2\} : T_1 \times T_2} \quad \text{and} \quad \frac{\Gamma \vdash t_1 : T_{11} \times T_{12}}{\Gamma \vdash t_1.1 : T_{11}}$$

called T-Pair and T-Proj1 respectively. T-Proj2 is similar. The evaluation rules enforce a left-to-right evaluation strategy for pairs (E-Pair2 only fires if the first projection is already a value).

8.7. **Tuples.** It is easy to generalize the binary products of the previous section to tuples. We write $\{t_i^{i \in 1..n}\}$ for a tuple of n terms and $\{T_i^{i \in 1..n}\}$ for its type. When n is 0, we get the empty tuple. The evaluation and typing rules are straightforward generalizations of those in the previous section, but E-Tuple (which generalizes E-Pair1 and E-Pair2) deserves special comment, as it specifies the left-to-right evaluation order:

$$\frac{t_j \to t_j'}{\{v_i^{i \in 1..j-1}, t_j, t_k^{k \in j+1..n}\} \to \{v_i^{i \in 1..j-1}, t_j', t_k^{k \in j+1..n}\}}.$$

8.8. **Records.** The generalization from n-tuples to labeled records is equally straightforward. We annotate each field t_i with a label l_i drawn from some predetermined set $\mathcal{L}$. The syntactic form for records and projections are $\{l_i = t_i^{i \in 1..n}\}$ and $t.l$ respectively. The evaluation and typing rules are straightforward; as an example, E-ProjRcd says $\{l_i = v_i^{i \in 1..n}\}.l_j \to v_j$. E-Proj is identical to that for tuples, and E-Rcd is an obvious generalization of E-Tuple. We can obtain tuples as a special case of records simply by allowing the set of labels to include natural numbers; for example, we regard {`Bool`, `Nat`, `Bool`} as an abbreviation for {1:`Bool`, 2:`Nat`, 3:`Bool`}. Finally, note that our records are ordered: $\{x = 1, y = 2\} \neq \{y = 2, x = 1\}$.

8.9. **Sums.** Many programs need to deal with heterogeneous collections of values. For example, a node in a binary tree could be a leaf or an interior node with two children. The type-theoretic mechanism that supports this is variant types. Before introducing them in full generality, we look at the simpler case of sum types.

A sum type describes a set of values drawn from exactly two given types. Suppose we are using the record types PhysicalAddr and VirtualAddr; if we want to manipulate both uniformly (e.g. to make a list containing both kinds of records), we can introduce the sum type Addr = PhysicalAddr + VirtualAddr, each of whose elements is either a PhysicalAddr or a VirtualAddr.

We create elements of the this type by tagging elements of the component types; for example, if `pa` is a PhysicalAddr, then `inl pa` is an Addr; similarly for `va` and `inr va`. Think of `inl` (or `inr`) as a function from PhysicalAddr (or VirtualAddr) → PhysicalAddr + VirtualAddr.

To use elements of sum types, we introduce a `case` construct that allows us to distinguish which branch of the sum a given value comes from: `getName = λa:Addr.case a of inl x => x.fieldA | inr y = y.fieldB;`. The syntax is fairly intuitive.

The typing rules for tagging are straightforward: to show that `inl` t_1 has a sum type $T_1 + T_2$, it suffices to show that $t_1 : T_1$ and similarly for `inr`. The typing rule for the case construct is the most involved, but we will just look at the more general rule for variants.

8.9.1. *Sums and Uniqueness of Types.* When we extend the system with sums, the Uniqueness of Types Theorem (7.2) from pure $\lambda_\to$ fails. Once we have shown that $t_1 : T_1$, we can derive that `inl` t_1 is an element of $T_1 + T_2$ for *any* T_2. This means that we cannot build a typechecking algorithm simply by reading the rules from bottom to top as we have done so far. At this point, we have a few options, but the one we will use now is to demand that the programmer provide an explicit annotation to indicate which T_2 is intended.

8.10. **Variants.** Binary sums generalize to labeled variants just as products generalize to labeled records. Instead of $T_1 + T_2$, we write $< l_1 : T_1, l_2 : T_2 >$, where l_1 and l_2 are field labels. Instead of `inl` t `as`[5] $T_1 + T_2$, we write $< l_1 = t >$ `as` $< l_1 : T_1, l_2 : T_2 >$. Instead of labeling the branches of the `case` with `inl` and `inr`, we use the same labels as the corresponding sum type. With these generalizations, the getAddr example from the previous section becomes:

[5]The 'as' construct allows the kind of type annotation described in the previous section.

```
   Addr = <physical:PhysicalAddr, virtual:VirtualAddr>
a = <physical=pa> as Addr;
getName = λa:Addr.case a of <physical=x> => x.fieldA | <virtual=y> => y.fieldB;
```

We provide a few examples of evaluation and typing rules for variants below [fill in].

8.10.1. *Options.* One very useful idiom involving variants is optional values. For example, an element of the type `OptionalNat = <none:Unit, some:Nat>` is either the trivial `unit` value with the tag `none` or else a number with the tag `some`. In other words, OptionalNat is isomorphic to `Nat`extended with an additional distinguished value `none`.

Many languages support built-in support for options; OCaml predefines a type constructor `option`, and the `null` value in languages like C, C++, and Java is really an option in disguise.

8.10.2. *Enumerations.* Two degenerate cases of variant types are useful enough to deserve special mention: enumerated types and single-field variants. An enumeration is a variant type in which the field type associated with each label is `Unit`. For example, we could define a type like `Weekend = <saturday:Unit, sunday:Unit>`; the elements of this type are terms like `<sunday=unit> as Weekend`. Some languages, beginning with Pascal, provide special syntax for declaring and using enumerations; others, like ML, make enumerations a special case of variants.

8.10.3. *Single-Field Variants.* The other interesting special case is variants with just a single label: $V =< l : T >$. The important part is that the usual operations on T cannot be applied to elements of V without first unpacking them: a V cannot be mistaken for a T.

For example, if we wrote a program to do financial calculations, and wrote conversion functions like `dollars2euros` and `euros2dollars` of type `Float → Float`, we could write nonsense like `dollars2euros (dollars2euros mymoney)` because the amounts are represented simply as `Float`s. If we define dollars and euros as single-field variants (`DollarAmount = <dollars:Float>`; `EuroAmount = <euros:Float>`;) then we can define safe versions of the conversion functions that will only accept amounts in the correct currency. In this case, `euros2dollars` would have the type `EuroAmount → DollarAmount`.

Now the typechecker can track the currencies used in our calculations and even remind us how to interpret the final results.

8.10.4. *Variants vs. Datatypes.* A variant type T of the form $< l_i : T_i^{i \in 1..n} >$ is roughly analogous to the ML datatype defined by `type T =` l_1 `of` $T_1|l_2$ `of` $T_2|...|l_n$ `of` T_n, but there are several differences worth noticing.

(1) One trivial point is that in OCaml, types must begin with lowercase letters and datatype constructors (labels, in our terminology) with capital letters, so (strictly speaking) we should have written the datatype declaration above like `type t =` L_1 `of` $t_1|...|L_n$ `of` t_n, but we'll ignore this and use our convention to avoid confusion between terms t and types T.

(2) The most interesting difference is that OCaml does not require a type annotation when a constructor l_i is used to inject an element of T_i into the datatype T: we simply write $l_i(t)$. The way OCaml gets away with this is that T must be declared before it acn be used. Moreover, the labels in T cannot be used by any other datatype declared in the same scope. So when the typechecker sees $l_i(t)$, it knows that the annotation can only be T. The downside of this is that labels cannot be shared between different datatypes, but later on we will see ways of avoiding this.

(3) Another trick used by OCaml is that when the type associated with a label in a datatype definition is just `Unit`, it can be omitted altogether.

(4) OCaml datatypes bundle variants together with several additional features that we will examine in later sections: (a) a datatype definition may be recursive; (b) an OCaml datatype can be parameterized on a type variable.

8.10.5. *Variants as Disjoint Unions.* Sums and variant types are sometimes called disjoint unions. The phrase union type is also used to refer to untagged (non-disjoint) union types, described in a later section.

8.11. **General Recursion.** Another facility found in most programming languages is the ability to define recursive functions. In the untyped lambda calculus, such functions can be defined with the aid of the fixed point combinator. In a typed setting, recursive functions can be defined in a similar way. [Blah; come back to this.]

8.12. **Lists.** For every type T, the type `List` T describes finite-length lists whose elements are drawn from T. The empty list is written `nil[T]`; the list formed by appending t_1 to the front of a list t_2 is written `cons[T]` $t_1 t_2$; the head and tail of a list t are written `head[T] t` and `tail[T] t`; finally, the boolean predicate `isnil[T] t` returns true iff t is empty. Except for syntactic differences and the explicit type annotations on all our syntactic forms, these lists are essentially identical to ML's.

9. Normalization

We now consider another fundamental theoretical property of the pure simply typed lambda calculus: the evaluation of a well-typed program is guaranteed to halt in a finite number of steps — in other words, every well-typed term is *normalizable.*

Unlike the other type safety properties we've considered so far, the normalization property does not extend to full-blown languages, since they nearly always extend the simply typed lambda calculus with constructs such as general recursion or recursive types that can be used to write nonterminating programs.

9.1. **Normalization for Simple Types.** The calculus we consider is the simply typed lambda calculus over a single base type A. Normalization is not entirely trivial to prove, since each reduction of a term can duplicate redexes in subterms. The proof is done by the method of logical relations. The key issue is finding a strong enough induction hypothesis. (Though we will trace through some main points here, the proofs are largely omitted.) To this end, we begin by defining a set R_T of closed terms of type T for each type T. We regard these sets as predicates and write $R_T(t)$ for $t \in R_T$.

Definition 9.1. $R_A(t)$ iff t halts. $R_{T_1 \to T_2}(t)$ iff t halts and whenever $R_{T_1}(s)$, we have $R_{T_2}(s)$.

This definition gives us the induction hypothesis we need. Our main goal is to show that all closed terms of base type (in other words, all programs) halt. Since closed terms of base type can contain subterms of function type, we need to know something about these as well. For terms of function type, we need to know not only that they halt but also that they yield halting results when applied to halting arguments.

The form of Definition 9.1 is characteristic of the logical relations proof technique. The proof proceeds by using the following lemmas to show first that every element of every set R_T is normalizable, and then that every well-typed term of type T is in R_T. The first step is immediate from the definition:

Lemma 9.1. *If $R_T(t)$, then t halts.*

Next, we note that membership in R_T is invariant under evaluation:

Lemma 9.2. *If $t : T$ and $t \to t'$, then $R_T(t)$ iff $R_T(t')$.*

Next, we show that every term of type T belongs to R_T:

Lemma 9.3. *If $x_1 : T_1, ..., x_n : T_n \vdash t : T$ and $v_1, ..., v_n$ are closed values of types $T_1, ..., T_n$ with $R_{T_i}(v_i)$ for each i, then $R_T([x_1 \mapsto v_1]...[x_n \mapsto v_n]t)$.*

Finally, we obtain the normalization property as a corollary.

Theorem 9.4 (Normalization)**.** *If $\vdash t : T$, then t is normalizable.*

Proof. $R_T(t)$ by Lemma 9.3; t is therefore normalizable by Lemma 9.1. □

10. References

Most practical programming languages include various impure features that cannot be described in the simple semantic framework we have used so far; besides just yielding results, evaluation of terms may assign to mutable variables; perform input and output to files, displays, or network connections; make non-local transfers of control via exceptions, jumps, or continuations; and so on. Such side effects of computation are often referred to as *computational effects* in the programming languages literature.

In this section, we see how one kind of side effect – mutable references – can be added to our previous calculi. The main extension will be dealing explicitly with a memory store.

10.1. **Introduction.** Nearly every language provides some form of assignment that changes the contents of a previously allocated piece of storage. In some languages (notably the ML family), the mechanisms for name-binding and assignment are kept separate. We can have variables whose values are 5 or variables whose value is a pointer to a mutable cell whose current contents is 5.

10.1.1. *Basics.* The basic operations on references are allocation, dereferencing, and assignment. To allocate a reference, we use the `ref` operator, providing an initial value for the new cell. After running `r = ref 5`, r is of type `Ref Nat`. To read the current value of this cell, we use the dereferencing operator `!`: `!r` yields 5, which is of type `Nat`, as expected. To change the value in the cell, we use the assignment operator: `r := 7` yields `Unit`.

11. Exceptions

12. Subtyping

13. Metatheory of Subtyping