Simple Extensions

The simply typed lambda-calculus provides enough structure to make its theoretical properties interesting but lacks some features found in more familiar programming languages. Several familiar features can be added with straightforward treatments at the level of typing. A core concept is that of **derived forms**.

Base Types

Every programming language has a variety of **base types**, which are sets of simple, unstructured values like numbers, booleans, or characters. These types come with primitive operations.

Examples:

- `Bool`
- `Nat`
- `String` (e.g., `"hello"`)
- `Float` (e.g., `3.14159`)

For theoretical purposes, the details of base types and operations can be abstracted away by supposing the language has some set `A` of **uninterpreted base types** with no primitive operations. `A` can be thought of as standing for **atomic types**.

- We use `A`, `B`, `C`, etc., as the names of base types.
- A base type can still be useful even without named elements; you can bind variables that range over the elements of the base type.
  - `λx:A. x; <fun> : A → A` is the identity function on the elements of `A`.
  - `λf:A→A. λx:A. f(f(x)); <fun> : (A→A) → A → A` repeats the behavior of a given function `f` twice on an argument `x`.

The Unit Type

The **Unit type** is another useful base type, especially in languages in the ML family.

> The Unit type is interpreted in the simplest possible way: a single element, the term constant `unit`, is introduced, and a typing rule makes `unit` an element of `Unit`.

- `Unit` is the only possible result of evaluating an expression of type `Unit`.
- The main application of `Unit` is in languages with side effects, such as assignments to reference cells.
- The use of `Unit` is similar to the role of the `void` type in languages like C and Java.

Derived Forms: Sequencing and Wildcards

In languages with side effects, evaluating expressions in sequence is often useful. The sequencing notation `t1;t2` evaluates `t1`, discards the result, and evaluates `t2`.

Two ways to formalize sequencing:

1. Add `t1;t2` as a new alternative in the syntax of terms, with evaluation rules:

    - `t1 → t1'`
    - `t1;t2 → t1';t2 $E-Seq$`
    - `unit;t2 → t2 $E-SeqNext$`

    and a typing rule:

    - `Γ ⊢ t1 : Unit Γ ⊢ t2 : T2`
    - `Γ ⊢ t1;t2 : T2 $T-Seq$`

2. Regard `t1;t2` as an abbreviation for `(λx:Unit.t2) t1`, where `x` is fresh.

The high-level typing and evaluation rules for sequencing can be derived from the abbreviation of `t1;t2` as `(λx:Unit.t2) t1`.

> Theorem 11.3.1 [Sequencing is a derived form]:

Write `λE` for the simply typed lambda-calculus with the `Unit` type, the sequencing construct, and the rules `E-Seq`, `E-SeqNext`, and `T-Seq`, and `λI` for the simply typed lambda-calculus with `Unit` only. Let `e ∈ λE → λI` be the elaboration function that translates from the external to the internal language by replacing every occurrence of `t1;t2` with `(λx:Unit.t2) t1`, where `x` is chosen fresh in each case. Now, for each term `t` of `λE`, we have:

- `t →E t'` iff `e(t) →I e(t')`
- `Γ ⊢E t : T` iff `Γ ⊢I e(t) : T`

where the evaluation and typing relations of `λE` and `λI` are annotated with `E` and `I`, respectively, to show which is which.

> Derived form: A feature whose typing and evaluation behavior can be derived from more fundamental operations.
>
> Syntactic sugar: Derived forms are often called syntactic sugar.
>
> Desugaring: Replacing a derived form with its lower-level definition.

Another derived form is the wildcard convention for variable binders, where `λ_:S.t` abbreviates `λx:S.t`, where `x` is some variable not occurring in `t`.

Ascription

Ascription is the ability to explicitly ascribe a particular type to a given term.

> `t as T` means "the term `t`, to which we ascribe the type `T`."

The typing rule `$T-Ascribe$` verifies that the ascribed type `T` is the type of `t`. The evaluation rule `$E-Ascribe$` throws away the ascription, leaving `t` free to evaluate as usual.

Uses of Ascription:

- Documentation: Helps readers keep track of the types of subexpressions.
- Clarification: Clarifies the programmer's thinking in complex expressions.
- Type Error Pinpointing: Aids in finding the source of type errors.
- Controlling Printing of Complex Types: Influences how types are displayed by typecheckers.
- Abstraction: Hides some of the types of a term by treating it as if it had a smaller set of types.

Let Bindings

Let bindings are a way to name subexpressions in a complex expression to avoid repetition and increase readability.

> `let x=t1 in t2` means "evaluate the expression `t1` and bind the name `x` to the resulting value while evaluating `t2`."

The typing rule `$T-Let$` calculates the type of the let-bound term, extends the context with a binding with this type, and calculates the type of the body in the enriched context, which is the type of the whole let expression.

```
Γ ⊢ t1 : T1
Γ, x:T1 ⊢ t2 : T2
-----------------------
Γ ⊢ let x=t1 in t2 : T2
```

Let can also be defined as a derived form (Landin).

```
let x=t1 in t2 def= (λx:T1.t2) t1
```

Pairs

Pairs are the simplest form of compound data structures. The formalization of pairs involves:

- Pairing: `{t1,t2}`
- Projection: `t.1` (first projection) and `t.2` (second projection)
- Type Constructor: `T1 × T2` (product type)

Evaluation rules specify how pairs and projection behave:

- `{v1,v2}.1 → v1 $E-PairBeta1$`
- `{v1,v2}.2 → v2 $E-PairBeta2$`
- `t1 → t1'`
- `t1.1 → t1'.1 $E-Proj1$`
- `t1 → t1'`
- `t1.2 → t1'.2 $E-Proj2$`
- `t1 → t1'`
- `{t1,t2} → {t1',t2} $E-Pair1$`
- `t2 → t2'`
- `{v1,t2} → {v1,t2'} $E-Pair2$`

Typing rules:

- `Γ ⊢ t1 : T1 Γ ⊢ t2 : T2`
- `Γ ⊢ {t1,t2} : T1 × T2 $T-Pair$`
- `Γ ⊢ t1 : T11 × T12`
- `Γ ⊢ t1.1 : T11 $T-Proj1$`
- `Γ ⊢ t1 : T11 × T12`
- `Γ ⊢ t1.2 : T12 $T-Proj2$`

Tuples

Tuples are n-ary products, generalizing pairs.

- Example: `{1,2,true}` is a 3-tuple of type `{Nat,Nat,Bool}`.
- Notation: `{ti i∈1..n}` for a tuple of `n` terms, and `{Ti i∈1..n}` for its type.

Evaluation rule:

- `{vi i∈1..n}.j → vj $E-ProjTuple$`
- `t1 → t1'`
- `t1.i → t1'.i $E-Proj$`
- `tj → tj'`
- `{vi i∈1..j−1,tj,tk k∈j+1..n} → {vi i∈1..j−1,tj',tk k∈j+1..n} $E-Tuple$`

Typing Rules:

- `Γ ⊢ ti : Ti for each i`
- `Γ ⊢ {ti i∈1..n} : {Ti i∈1..n} $T-Tuple$`
- `Γ ⊢ t1 : {Ti i∈1..n}`
- `Γ ⊢ t1.j : Tj $T-Proj$`

Records

Records are a generalization of tuples where each field `ti` is annotated with a label `li` from a predetermined set `L`.

- Example: `{x=5}` and `{partno=5524,cost=30.27}` are record values with types `{x:Nat}` and `{partno:Nat,cost:Float}`, respectively.

Evaluation Rules:

- `{li=vi i∈1..n}.lj → vj $E-ProjRcd$`
- `t1 → t1'`
- `t1.l → t1'.l $E-Proj$`
- `tj → tj'`
- `{li=vi i∈1..j−1,lj=tj,lk=tk k∈j+1..n} → {li=vi i∈1..j−1,lj=tj',lk=tk k∈j+1..n} $E-Rcd$`

Typing Rules:

- `Γ ⊢ ti : Ti for each i`
- `Γ ⊢ {li=ti i∈1..n} : {li:Ti i∈1..n}` $T-Rcd$
- `Γ ⊢ t1 : {li:Ti i∈1..n}`
- `Γ ⊢ t1.lj : Tj` $T-Proj$

Records

In the simply typed lambda calculus, record values with different field orderings are considered to have different types. For example:

$ {partno=5524, cost=30.27} $ and $ {cost=30.27,partno=5524} $

...are different record values, with types $ {partno:Nat,cost:Float} $ and $ {cost:Float, partno:Nat} $, respectively.

However, in Chapter 15, a subtype relation will be introduced, where the types $ {partno:Nat,cost:Float} $ and $ {cost:Float,partno:Nat} $ are equivalent.

Exercise: Pattern Matching

Many high-level programming languages use pattern matching syntax to extract all fields from a record at once.

A simple pattern matching can be added to an untyped lambda calculus with records, by adding a new syntactic category of patterns and a new case to the syntax of terms.

Matching Rules

- `match(x, v) = [x v]` (M-Var)
- `match({li=pi i∈1..n}, {li=vi i∈1..n}) = σ1 ◦···◦ σn` (M-Rcd)

New evaluation rules

- `let p =v1 in t2 → match(p, v1) t2` (E-LetV)
- `let p =t1 in t2 → let p =t 1 in t2` (E-Let)

The computation rule for pattern matching generalizes the let-binding rule. It relies on a **matching function** that, given a pattern p and a value v, either fails (if v does not match p) or yields a substitution that maps variables appearing in p to the corresponding parts of v.

Sums

Many programs need to deal with **heterogeneous collections of values**. The type-theoretic mechanism that supports this kind of programming is **variant types**. Let's consider the simpler case of **binary sum types**.

> A **sum type** describes a set of values drawn exactly two given types.

For example, consider the following types:

- `PhysicalAddr = {firstlast:String, addr:String}`
- `VirtualAddr = {name:String, email:String}`

If we want to manipulate both sorts of records uniformly, we can introduce the sum type:

`Addr = PhysicalAddr + VirtualAddr`

Each of whose elements is either a `PhysicalAddr` or a `VirtualAddr`.

Tagging

We create elements of this type by tagging elements of the component types `PhysicalAddr` and `VirtualAddr`. For example, if `pa` is a `PhysicalAddr`, then `inl pa` is an `Addr`.

In general, the elements of a type $T_1 + T_2$ consist of the elements of $T_1$, tagged with the token `inl`, plus the elements of $T_2$, tagged with `inr`.

Case Construct

To use elements of sum types, we introduce a **case construct** that allows us to distinguish whether a given value comes from the left or right branch of a sum.

```
getName = λa:Addr. case a of inl x ⇒ x.firstlast | inr y ⇒ y.name;
```

When the parameter a is a `PhysicalAddr` tagged with `inl`, the case expression will take the first branch, binding the variable x to the `PhysicalAddr`; the body of the first branch then extracts the `firstlast` field from x and returns it. Similarly, if a is a `VirtualAddr` value tagged with `inr`, the second branch will be chosen and the name field of the `VirtualAddr` returned. Thus, the type of the whole `getName` function is `Addr→String`.

Evaluation Rules

- `case (inl v0 ) of inl x1⇒t1 | inr x2⇒t2 → [x1 v0]t1` (E-CaseInl)
- `case (inr v0 ) of inl x1⇒t1 | inr x2⇒t2 → [x2 v0]t2` (E-CaseInr)
- `case t0 of inl x1⇒t1 | inr x2⇒t2 → case t 0 of inl x1⇒t1 | inr x2⇒t2` (E-Case)
- `inl t1 → inl t 1` (E-Inl)
- `inr t1 → inr t 1` (E-Inr)

Typing Rules

- If $Γ |- t_1 : T_1$, then $Γ |- inl t_1 : T_1 + T_2$ (T-Inl)
- If $Γ |- t_1 : T_2$, then $Γ |- inr t_1 : T_1 + T_2$ (T-Inr)
- If $Γ |- t_0 : T_1 + T_2$, $Γ, x_1:T_1 |- t_1 : T$, and $Γ, x_2:T_2 |- t_2 : T$, then $Γ |- case t_0 of inl x_1⇒t_1 | inr x_2⇒t_2 : T$ (T-Case)

Sums and Uniqueness of Types

Most of the properties of the typing relation of pure $λ→$ extend to the system with sums, but one important one fails: the **Uniqueness of Types theorem**.

The difficulty arises from the tagging constructs `inl` and `inr`. The typing rule T-Inl, for example, says that, once we have shown that $t_1$ is an element of $T_1$, we can derive that $inl t_1$ is an element of $T_1 + T_2$ for any type $T_2$. For example, we can derive both $inl 5 : Nat+Nat$ and $inl 5 : Nat+Bool$.

Solutions to Type Uniqueness

1. Complicate the typechecking algorithm so that it somehow **"guesses"** a value for $T_2$.
2. Refine the language of types to allow all possible values for $T_2$ to somehow be **represented uniformly**.
3. Demand that the programmer provide an **explicit annotation** to indicate which type $T_2$ is intended.

Explicit Annotation

Instead of writing just `inl t` or `inr t`, we write `inl t as T` or `inr t as T`, where T specifies the whole sum type to which we want the injected element to belong.

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 T1+T2`, we write `<l1=t> as <l1:T1,l2:T2>`.

Example

```
Addr = <physical:PhysicalAddr, virtual:VirtualAddr>;
a = <physical=pa> as Addr;
a : Addr
getName = λa:Addr. case a of <physical=x> ⇒ x.firstlast | <virtual=y> ⇒ y.name
getName : Addr → String
```

Evaluation Rules

- `case (<lj=vj> as T) of <li=xi>⇒ti i∈1..n → [xj vj ]tj` (E-CaseVariant)
- `case t0 of <li=xi>⇒ti i∈1..n → case t 0 of <li=xi>⇒ti i∈1..n` (E-Case)
- `<li=ti> as T → <li=t i> as T` (E-Variant)

Typing Rules

- If $Γ |- t_j : T_j$, then $Γ |- <l_j=t_j> as <l_i:T_i i∈1..n> : <l_i:T_i i∈1..n>$ (T-Variant)
- If $Γ |- t_0 : <l_i:T_i i∈1..n>$ and for each i, $Γ, x_i:T_i |- t_i : T$, then $Γ |- case t_0 of <l_i=x_i>⇒t_i i∈1..n : T$ (T-Case)

Options

One 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.

The type `Table = Nat→OptionalNat` represents finite mappings from numbers to numbers.

```
emptyTable = λn:Nat. <none=unit> as OptionalNat;
emptyTable : Table
extendTable = λt:Table. λm:Nat. λv:Nat. λn:Nat. if equal n m then <some=v> as
extendTable : Table → Nat → Nat → Table
x = case t(5) of <none=u> ⇒ 999 | <some=v> ⇒ v;
```

Enumerations

An enumerated type is a variant type in which the field type associated with each label is Unit.

For example, a type representing the days of the working week might be defined as:

`Weekday = <monday:Unit, tuesday:Unit, wednesday:Unit, thursday:Unit, friday:Unit>;`

```
nextBusinessDay = λw:Weekday.
case w of
    <monday=x> ⇒ <tuesday=unit> as Weekday
  | <tuesday=x> ⇒ <wednesday=unit> as Weekday
  | <wednesday=x> ⇒ <thursday=unit> as Weekday
  | <thursday=x> ⇒ <friday=unit> as Weekday
  | <friday=x> ⇒ <monday=unit> as Weekday;
```

Single-Field Variants

A variant type with just a single label l: `V = <l:T>`. The elements of V will be in one-to-one correspondence with the elements of the field type T, since every member of V has precisely the form `<l=t>` for some $t : T$.

```
DollarAmount = <dollars:Float>;
EuroAmount = <euros:Float>;
dollars2euros = λd:DollarAmount. case d of <dollars=x> ⇒ <euros = timesfloat
dollars2euros : DollarAmount → EuroAmount
euros2dollars = λe:EuroAmount. case e of <euros=x> ⇒ <dollars = timesfloat x
euros2dollars : EuroAmount → DollarAmount
mybankbalance = <dollars=39.50> as DollarAmount;
euros2dollars (dollars2euros mybankbalance);
<dollars=39.49990125> as DollarAmount : DollarAmount
```

Variants vs. Datatypes

A variant type $T$ of the form $<l_i:T_i i∈1..n>$ is roughly analogous to the ML datatype defined by:

```
type T = l1 of T1 | l2 of T2 | ... | ln of Tn
```

Differences

Feature	Variant Types (ML Datatypes)
Capitalization	OCaml requires types to begin with lowercase letters and datatype constructors with capital letters.
Type Annotation	OCaml does not require a type annotation when a constructor $l_i$ is used to inject an element of $T_i$ into the datatype $T$.
Unit Types	When the type associated with a label in a datatype definition is just Unit, it can be omitted altogether.
Additional Features	OCaml datatypes bundle variant types together with several additional features: recursive definitions, parameterized types, etc.

Variants as Disjoint Unions

Sum and variant types are sometimes called disjoint unions. The type $T_1 + T_2$ is a "union" of $T_1$ and $T_2$ in the sense that its elements include all the elements from $T_1$ and $T_2$. This union is disjoint because the sets of elements of $T_1$ or $T_2$ are tagged with `inl` or `inr`, respectively, before they are combined.

Type Dynamic

To handle situations where the type of data cannot be determined at compile time safely, many languages offer facilities for inspecting the types of values at run time. One way of accomplishing this is to add a type Dynamic whose values are pairs of a value v and a type tag T where v has type T.

General Recursion

Recursive functions can be defined in a typed setting with the aid of the fix combinator. For example:

~~~
ff = λie:Nat→Bool. λx:Nat. if iszero x then true else if iszero (pred x) then
ff : (Nat→Bool) → Nat → Bool
iseven = fix ff;
iseven : Nat → Bool
iseven 7;
false : Bool
```## General Recursion

This section introduces the concept of general recursion using the **fix** co

### Fix Combinator

*   The **fix** combinator is used to achieve recursion in the simply typed la
*   It mimics the behavior of the untyped fix combinator.

#### Syntax
~~~

t ::= ... terms: fix t fixed point of t

```
#### Evaluation Rule
\$fix (λx:T1.t2) → [x (fix (λx:T1.t2))]t2\$ (E-FixBeta)

If \$t1 → t'1\$, then \$fix t1 → fix t'1\$ (E-Fix)

#### Typing Rule
If Γ |- t1 : T1→T1, then Γ |- fix t1 : T1 (T-Fix)

#### Example: Mutually Recursive Functions

The fix construct isn't restricted to function types, allowing the definition
```

ff = λieio:{iseven:Nat→Bool, isodd:Nat→Bool}. {iseven = λx:Nat. if iszero x then true else ieio.isodd (pred x), isodd = λx:Nat. if iszero x then false else ieio.iseven (pred x)}; ff : {iseven:Nat→Bool, isodd:Nat→Bool} → {iseven:Nat→Bool, isodd:Nat→Bool}

```
Forming the fixed point of the function `ff` gives a record of two functions:
```

r = fix ff; r : {iseven:Nat→Bool, isodd:Nat→Bool}

```
Projecting the first of these gives the `iseven` function itself:
```

iseven = r.iseven; iseven : Nat → Bool iseven 7; false : Bool

```
#### Divergence

The ability to form the fixed point of a function of type `T→T` for any `T` ir
```

divergeT = λ_:Unit. fix (λx:T.x); divergeT : Unit → T

```
`divergeT` applied to a unit argument results in a non-terminating evaluation

### Letrec

**Letrec** is a derived form used to bind a variable to the result of a recurs

> letrec x:T1=t1 in t2 def = let x = fix (λx:T1.t1) in t2

Example:
```

letrec iseven : Nat→Bool = λx:Nat. if iszero x then true else if iszero (pred x) then false else iseven (pred (pred x)) in iseven 7; false : Bool

```
##  Lists

This section introduces the **List** type constructor, which builds new types

### Syntax, Semantics, and Typing

Lists are similar to those found in ML and other functional languages, with ex

#### Syntax

| Term              | Description                |
| ----------------- | -------------------------- |
| `nil[T]`          | Empty list                 |
| `cons[T] t1 t2`   | List constructor           |
| `isnil[T] t`       | Test for empty list        |
| `head[T] t`       | Head of a list             |
| `tail[T] t`       | Tail of a list             |

#### Values

| Value           | Description    |
| --------------- | -------------- |
| `nil[T]`        | Empty list     |
| `cons[T] v1 v2`| List constructor |

#### Types
The type of lists are defined as: `List T`

#### Evaluation Rules

| Rule                                      | Description
| ----------------------------------------- | ---------------------------------
| `isnil[S] (nil[T]) → true`              | Checks if nil is empty         |
| `isnil[S] (cons[T] v1 v2) → false`              | Checks if cons is empty
| `head[S] (cons[T] v1 v2) → v1`              | Returns the head of the list
| `tail[S] (cons[T] v1 v2) → v2`              | Returns the tail of the list
| If `t1 → t'1`, then `cons[T] t1 t2 → cons[T] t'1 t2` | Evaluates the first
| If `t2 → t'2`, then `cons[T] v1 t2 → cons[T] v1 t'2` | Evaluates the second
| If `t1 → t'1`, then `isnil[T] t1 → isnil[T] t'1` | Evaluates the term in is
| If `t1 → t'1`, then `head[T] t1 → head[T] t'1` | Evaluates the term in head
| If `t1 → t'1`, then `tail[T] t1 → tail[T] t'1` | Evaluates the term in tail

#### Typing Rules

| Rule                                    |
| ------------------------------------    |
| Γ |- `nil[T1]` : `List T1`              |
| Γ |- `t1` : `T1`                          |
| Γ |- `t2` : `List T1`                   |
| Γ |- `cons[T1] t1 t2` : `List T1`       |
| Γ |- `t1` : `List T11`                  |
| Γ |- `isnil[T11] t1` : `Bool`           |
| Γ |- `t1` : `List T11`                  |
| Γ |- `head[T11] t1` : `T11`            |
| Γ |- `t1` : `List T11`                  |
| Γ |- `tail[T11] t1` : `List T11`      |
<script type="text/javascript">
  window.MathJax = {
    tex: {
      inlineMath: [['$', '$'], ['\\(', '\\)']],
      displayMath: [['$$', '$$'], ['\\[', '\\]']]
    }
  };
</script>
<script type="text/javascript" async
  src="https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-mml-chtml.js">
</script>
```