Chapter 1

Introduction

1.1 Outline

1. Definition of a simple, typed lambda calculus
2. Context and definition of *effects* in programming languages
3. Examples of common effects
   (a) IO
   (b) mutable data, state
   (c) exception
   (d) nondeterminism
   (e) non-termination / non-totality
   (f) reader, writer, output
   (g) continuation
4. Explanation of how effects are handled in ML's style
   (a) call-by-value at runtime
   (b) examples of effects: IO, exceptions
   (c) effects are untyped
   (d) simple exception-handling system (type/catch)
5. Explain why this style is not purely functional
   (a) exposes *side-effects*, which are implicit effects not accessible by the programming language

lambda-calculus is a formal language for expressing computation. In this context, a *language* is defined to be a set of expression that are generated by syntactical rules. The lambda-calculus comes in many different variants, and here we will consider a basic variant of the *simply-typed lambda-calculus* in order to considering computation formally. Call our language Language-A.

1.2.1 Definition of Language-A

TODO: include let bindings for Language-A?

Syntax for Language-A

In Language-A, there are two forms used for constructing well-formed expressions: *terms* and *types*. They are expressed by the following syntax:

metavariable	constructors	name
$\langle term \rangle$	$\langle term\text{-}name \rangle$	atom
	$\langle name \rangle{:}\langle type \rangle \Rightarrow \langle term \rangle$	function
	$\langle term \rangle\ \langle term \rangle$	application
	$\mathsf{left}\ \langle term \rangle$	left
	$\mathsf{right}\ \langle term \rangle$	right
	$\mathsf{match}\ \langle term \rangle\ \{\ \mathsf{left}\ \langle name \rangle \Rightarrow \langle term \rangle\ \mid\ \mathsf{right}\ \langle name \rangle \Rightarrow \langle term \rangle\ \}$	split
	$(\langle term \rangle, \langle term \rangle)$	product
	$\mathsf{first}\ \langle term \rangle$	first
	$\mathsf{second}\ \langle term \rangle$	second
$\langle type \rangle$	$\langle type\text{-}name \rangle$	atom
	$\langle type \rangle \to \langle type \rangle$	arrow
	$\langle type \rangle + \langle type \rangle$	sum
	$\langle type \rangle \times \langle type \rangle$	product

(1.1)

where $\langle term\text{-}name \rangle$ and $\langle type\text{-}name \rangle$ are metavariables that range over an infinite collection of distinct names, and $\langle name \rangle$ ranges over all names. A given expression is well-formed with respect to Language-A if it is constructable by a series of applications of these rules. Note that this syntax does not specify any concrete terms or types. Language-A's syntax is defined abstractly in order for the definition its typing and semantics as simple as possible. For the sake of intuition however, these are some examples of the likes of which these meta-variables stand in for:

- $\langle term\text{-}name \rangle$: $0, 1, 2$, true, false, "hello world", $\bullet$.
- $\langle type\text{-}name \rangle$: natural, integer, boolean, string, void, unit.
- $\langle name \rangle$: x, y, this-is-a-constant

Typing Rules for Language-A

Terms and types are related by a *typing judgement*, by which a term is states to have a unique type. The judgement that a has type α is written as $a{:}\alpha$. In order to build typed terms using the constuctors presented in 1.1, the types of complex terms are infered from their sub-terms using inference rules making use of judgement contexts (i.e. collections of judgements). A statement of the form $\Gamma \vdash a{:}\alpha$ asserts that the context Γ entails that $a{:}\alpha$. The notation $\Gamma, J \vdash J'$ abbreviates $\Gamma \cup \{J\} \vdash J'$.

With judgements, we now can state *typing inferences rules*. Such inference rules have the form

$$\frac{P_1 \qquad \cdots \qquad P_n}{Q},$$

which asserts that the premises $P_1, \ldots, P_n$ entail the conclusion Q. For example,

$$\frac{\Gamma \text{ is a judgement context} \qquad a \text{ is a term} \qquad \alpha \text{ is a type} \qquad \Gamma \vdash a{:}\alpha}{\Gamma \vdash a{:}\alpha}$$

is a particularly uninteresting inference rule. Explicitly stating the domain of each variable as premises is cumbersome however, so the following are conventions for variable domains based on their name:

- $\Gamma, \Gamma_i, \ldots$ each range over judgement con texts,
- $a, b, c, \ldots$ each range over terms,
- $\alpha, \beta, \gamma, \ldots$ each range over types.

The following typing rules are given for Language-A:

$$\textbf{TODO}\quad \frac{}{\Gamma, a{:}\alpha \;\vdash\; a{:}\alpha}$$

$$\text{Function-Abstraction}\quad \frac{\Gamma, a{:}\alpha \;\vdash\; b{:}\beta}{\Gamma \;\vdash\; a{:}\alpha \Rightarrow b{:}\alpha \to \beta}$$

$$\text{Function-Concretization}\quad \frac{\Gamma \;\vdash\; a{:}\alpha \to \beta \qquad \Gamma \;\vdash\; b{:}\alpha}{\Gamma \;\vdash\; a\ b{:}\beta}$$

$$\text{Construct-Sum-Left}\quad \frac{\Gamma \;\vdash\; a{:}\alpha}{\Gamma \;\vdash\; (\mathsf{left}\ a){:}\alpha + \beta}$$

$$\text{Construct-Sum-Right}\quad \frac{\Gamma \;\vdash\; b{:}\beta}{\Gamma \;\vdash\; (\mathsf{right}\ b){:}\alpha + \beta} \tag{1.2}$$

$$\text{Construct-Product}\quad \frac{\Gamma \;\vdash\; a{:}\alpha \qquad \Gamma \;\vdash\; b{:}\beta}{\Gamma \;\vdash\; ((a, b)){:}\alpha \times \beta}$$

$$\text{Destruct-Sum}\quad \frac{\Gamma \;\vdash\; x{:}(\alpha + \beta) \qquad \Gamma, a{:}\alpha \;\vdash\; c_1{:}\gamma \qquad \Gamma, b{:}\beta \;\vdash\; c_2{:}\gamma}{\Gamma \;\vdash\; (\mathsf{match}\ x\ \{\ \mathsf{left}\ a \Rightarrow c_1\ \mid\ \mathsf{right}\ b \Rightarrow c_2\ \}){:}\gamma}$$

$$\text{Destruct-Product-First}\quad \frac{\Gamma \;\vdash\; x{:}(\alpha \times \beta)}{\Gamma \;\vdash\; (\mathsf{first}\ x){:}\alpha}$$

Reduction Rules for Language-A

Finally, the last step is to introduce *reduction rules*. So far we have outlined syntax and inference rules for building expressions in Language-A, but all these expressions are inert. Reduction rules describe how terms can be transformed, step by step, in a way that models computation. A series of these simple reductions may end in a term for which no reduction rule can apply. Call these terms *values*, and notate "v is a value" as "value v." The following reduction rules are given for Language-A:

$$\beta\text{-Reduction}\quad \frac{\Gamma \;\vdash\; (a{:}\alpha \Rightarrow b){:}(\alpha \to \beta) \qquad \Gamma \;\vdash\; v{:}\alpha \qquad \text{value } v}{(a{:}\alpha \Rightarrow b)\ v \;\twoheadrightarrow\; [v/a]b} \tag{1.3}$$

The most fundamental of these rules is β-Reduction, which is the way that function applications are resolved to the represented comptuation's output. The substitution notation $[v/a]b$ indicates to "replace with v each appearance of a in b." In this way, for a function $a{:}\alpha \Rightarrow b{:}(\alpha \to \beta)$ and an input $v{:}\alpha$, β-Reduction *substitutes* the input v for the appearances of the function parameter a in the function body b.

For example, consider the following terms: **TODO**: enlightening example of a series of β-reductions for some simple comptuation.

1.2.2 Other Structures in Language-A

TODO: decide, based on future developments, which terms and types should be included here. Things like:

- integer
- natural
- boolean
- unit

1.2.3 Properties of Language-A

With the syntax, typing rules, and reduction rules for Language-A, we now have a completed definition of the language. However, some of the design desicions may seem arbitrary even if intuitive. This particular framework is good because it maintains a few nice properties that make reasoning about Language-A intuitive and extendable.
TODO: how much detail do I want to go into this? come back after writing rest of chapter 1.

Type-Preserving Substitution

$$\Gamma, a{:}\alpha \vdash b{:}\beta \land \Gamma \vdash v{:}\alpha \land \text{value } v \implies \Gamma \vdash [v/a]b{:}\beta \tag{1.4}$$

Reduction Progress

$$\{\} \vdash a{:}\alpha \implies \text{value } a \lor \exists a'{:}a \twoheadrightarrow a' \tag{1.5}$$

Type Soundness

$$\Gamma \vdash a{:}\alpha \land a \twoheadrightarrow a' \implies \text{value } a' \lor \exists a''{:}\alpha \twoheadrightarrow a'' \tag{1.6}$$

Type Preservation

$$\Gamma \vdash a{:}\alpha \land a \twoheadrightarrow a' \implies \Gamma \vdash a'{:}\alpha \tag{1.7}$$

1.3 Computation with Effects

The definition of Language-A in section **??** gives a good general flavor for many similar functional programming languages. In terms of the such language's reductions from term to term, all the information relevant for deciding such reductions is explicit withing the term itself and the explicit context built up during reduction. In other words, there is no *implicit activity* that influences what a term's reduction will look like. The path of reductions is exactly the computation a term corresponds to, so this yields that the computation has the same property of not having access to or being affected by any implicit activity.

However nice a formalization of computation this is, it is immediately unrealistic. Actual computers, for which programming languages are abstractions of their activities, host multitudes of implicit processes while running a program. Even if a programming language modelled all such processes and incorporated them into the language so that each activity was made perfectly explicit as a term (a task that is certainly infeasable and likely impossible), the result would be an unuseful and inefficient language. The point of having layers of abstraction, in the form of high-and-higher level programming language, is to avoid this situtation in the first place. So there appears to be a dilemma:

The Dilemma of Implicit Activities

(1) Require fully explicit terms and reductions. This grants reasoning about programs is fully formalized and abstracted from the annoyances of hardware and lower-level-implementations, but restricts such programs from being applicable in almost all useful circumstances.

(2) Allow implicit activities that affect reductions. This grants many useful program applications and maintains some formal nature to the language's behavior, but reasoning about programs is now inescapably tainted by implicit activities.

A resolution to this apparent dilemma is to either choose one horn or to reject the dilemma. But first, let us consider what kinds of implicit activities are being considered here — in particular, what are computational *effects*.

1.3.1 Effects

The definition of an *effect* in the context of programming language is frequently debated, and there is no majority standard answer[1]. For the purposes of this thesis, the following definition is adopted.

Definition 1.3.1. A computational *effect* is a capability in a program that depends on factors outside of that capability's normal scope.

[1] **TODO**: source

The definition of *normal scope* will be left to intuitive interpretation, with the intent that what is the normal scope of a capability depends on many theoretical, design, and implementation factors.

Now for example, suppose we have a program P that computes the sum of two integers given as input. If P is designed and implemented exactly to this specification, then P has no effects; none of P's cababilities depend on factors outside of their usual scope. The only scope that P's capabilities depend on is that of the two inputs. No other factors influence what the correctly computed sum of the two integers is. Such a program with no effects is called a *pure* program.

Definition 1.3.2. A program is *pure* if it has no effects.

But if P is run, as abstractly as it is defined, not much use comes from it. P does not display its results, write the results to some P-specified memory, or give you an error if there is an overflow. These capabilities depend on factors outside of P's normal scope, per its specification, and so are examples of effects.

So as another example, let us consider a modified version of P that is effectual. Suppose that program P' takes as input two integers, computes the sum of the integers, throws an error if there is an overflow, writes the result into RAM, and prints the result to the console from which P' was run. These capabilities are examples of effects, since their behavior depends on factors outside of the normal scope of P' (the same scope as P). Such a program with effects is called an *impure* program.

Definition 1.3.3. A program is *impure* if it has effects.

There are a variety of common effects that are available in almost every programming language. These include:

- *input/output (IO),* e.g. printing to the console, accepting input from use, interfacing with other peripherals.
- *mutable data*, e.g. mutable variables, in-place arrays.
- *exception*, e.g. division by 0, out-of-bounds index of array.
- *nondeterminism*, e.g. **TODO**: what would be good examples for this...
- *partiality*, e.g. **TODO**
- *continuation*, e.g. **TODO**

TODO: go into detail here, or till after I've talked about comparing functional/imperative approaches to effects?

According to definition 1.3.1, there is an easy method for transforming an impure program into a pure program: add the factors depended upon by the program's effects to the program's scope. Using this intuition, we can reformulate the Dilemma of Implicit Activities with the new formal notion of computational effects:

The Dilemma of Scopes for Effects

(1) Require only pure programs. This grants reasoning about programs to depend only on the normal scope of the program.

(2) Allow impure as well as pure programs. This grants many useful programs, where the behavior of the programs depends on factors not entirely encapsulated by the program's normal scope.

The goal of resolving this dilemma is to make a choice about how programming languages should be designed. Are computational effects a feature to be avoided as much as possible? Or are they actually so necessary that it would be a mistake to restrict them? Unsurprisingly, no widely-adopted languages have chosen horn (1). But even in more generality, almost all languages have chosen horn (2), but with many varied approaches to incorporating effects. Overally languages have clustered into two groups.

- *Functional* programming language treat computation as the evaluation of mathematical functions. Such languages put varying degrees of emphasis on restricting effects, but in general much more emphasis than imperative languages.

- *Imperative* programming languages treat computation as a sequence of commands. E.g. the C family, Bash, Java, Python. Such languages put very little (if any) emphasis on restricting effects. E.g. Scheme, Lisp, SML, Clojure, Scala, Haskell, Agda, Gallina.

The perspective on effects is not the only way in which these groups differ, but nevertheless it is an important one. Among functional programming languages, we shall consider two in particular: SML and Haskell. SML takes a very relaxed approach to effects, and Haskell takes a more restrictive approach.

1.3.2 A Simple Approach to Effects

The SML (Standard Meta Language) language is most commonly used in academic settings for teaching about programming languages. However here, I shall avoid introducing an entire new syntax and language semantics. Instead, the SML effect impkementation strategy will be demonstrated using Language-B, a variant of Language-A with the following new features:

- *Sequence.* A collection of terms is arranged in the sequence of desired resolution. Reducing the sequence term will reduce the terms in order.

- *Mutable.* A new type that indicates mutable data. It must be explicitly read from and wrote to, rather than handled like the normal (immutable) value.

- *Exception.* A collection of new term constructs, allowing for programs to *throw* and *catch* exceptions that interrupt usual reduction.

Syntax for Language-B

metavariable	constructors	name
$\langle term \rangle$	$\langle term \rangle$; $\langle term \rangle$	sequence
	@$\langle term \rangle$	mutable
	!$\langle term \rangle$	read mutable
	$\langle term\text{-}name \rangle := \langle term \rangle$	write mutable
	throw $\langle term \rangle$	throw exception
	catch $\langle term \rangle$ { $\langle name \rangle \Rightarrow \langle term \rangle$ }	catch exception
$\langle type \rangle$	mutable $\langle type \rangle$	mutable

(1.8)

Typing Rules for Language-B

$$\text{Construct-Sequence} \quad \frac{\Gamma \vdash a{:}\alpha \qquad \Gamma \vdash b{:}\beta}{\Gamma \vdash (a \ ; \ b){:}\beta}$$

$$\text{Construct-Mutable} \quad \frac{\Gamma \vdash a{:}\alpha}{\Gamma \vdash (@a){:}\mathsf{mutable}\ \alpha}$$

$$\text{Construct-Mutable-Read} \quad \frac{\Gamma \vdash m{:}\mathsf{mutable}\ \alpha}{\Gamma \vdash (!m){:}\alpha}$$

$$\text{Construct-Mutable-Write} \quad \frac{\Gamma \vdash m{:}\mathsf{mutable}\ \alpha \qquad \Gamma \vdash a{:}\alpha}{\Gamma \vdash (m := a){:}\mathsf{unit}}$$

$$\text{Construct-Throw} \quad \frac{}{(\mathsf{throw}\ e){:}\epsilon}$$

$$\text{Construct-Catch} \quad \frac{\Gamma \vdash a{:}\alpha \qquad \Gamma, e{:}\epsilon \vdash a'{:}\alpha}{(\mathsf{catch}\ a\ \{\ e \Rightarrow a'\ \}){:}\alpha}$$

(1.9)

Reduction Rules for Language-B

TODO: formally explain how environments work (E). Should that go in this section, or the definition of Language-A? If I put it in Language-A, that would make defining let expressions much easier.

TODO: foramlly explain how mutable environments work (M). Models the storing of values for mutables.

$$\text{Sequence-Reduce}\quad \frac{a \twoheadrightarrow a'}{(a\ ;\ b) \twoheadrightarrow (a'\ ;\ b)}$$

$$\text{Truncate}\quad \frac{\text{value } a}{(a\ ;\ b) \twoheadrightarrow b}$$

$$\text{Initialize}\quad \frac{\text{value } v}{@v \twoheadrightarrow \bullet \qquad M[@v] = v}$$

$$\text{Mutable-Reduce}\quad \frac{a \twoheadrightarrow a'}{@a \twoheadrightarrow @a'}$$

$$\text{Read}\quad \frac{m \in M}{!m \twoheadrightarrow M[m]} \tag{1.10}$$

$$\text{Read-Reduce}\quad \frac{m \twoheadrightarrow m'}{!m \twoheadrightarrow !m'}$$

$$\text{Write}\quad \frac{m \in M \qquad \text{value } v}{m := v}$$

$$\text{Write-Source-Reduce}\quad \frac{m \twoheadrightarrow m'}{m := a \twoheadrightarrow m' := a}$$

$$\text{Write-Target-Reduce}\quad \frac{a \twoheadrightarrow a'}{m := a \twoheadrightarrow m := a'}$$

TODO: reduction rules for throw/catch.