K 1

Introduction

[**TODO**] is this entire chapter, change concrete programs to be entirely monospace font

[**TODO**] explain goal of making it extend A to make it more expressive, but express C-like programs in a way that preserves formal reasoning ability

[**TODO**] include note or something explaining what the convention is for reduction and other rules that have the same name (e.g. SIMPLIFY)

[**TODO**] introduce terms **compile-time** and **run-time** somwhere. Or, come up with new terms, like **typing** and **evaluation** perhaps? But doesn't exactly capture everthing. Probably better: **compile-time** and **evaluation-time**.

[**TODO**] fix style for colon, I think its too far apart (especially around commas): $a : \alpha$

1.1 Introduction

[**TODO**] Introduce my thesis.

1.2 Language Func

[**TODO**] reformat this section to be more of a narrative introduction to the language rather than a manual. Then I can put the manual in an appendix.

[**TODO**] Points to cover:

1. whitespace doesn't matter
2. function definitions

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 Func.

1.2.1 Syntax for Func

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

Π 1.1: Syntax for Func

metavariable	**constructor**	**name**
«*program*»	⟦ «*declaration*» ⟧	program
«*declaration*»	`type` «*type-name*» : «*kind*» := «*kind*».	constructed type
	`primitive type` «*type-name*» : «*kind*».	primitive type
	`term` «*term-name*» : «*type*» := «*term*».	constructed term
	`basic term` «*term-name*» : «*type*» := «*term*».	basic term
	`primitive term` «*term-name*» : «*type*».	primitive term
«*kind*»	`kind`	atom
	`kind` → «*kind*»	arrow
«*type*»	«*type-name*»	atom
	(«*type-param*» : «*kind*») ⇒ «*kind*»	function
	«*kind*» «*kind*»	application
«*term*»	«*term-name*»	atom
	(«*term-param*» : «*kind*») ⇒ «*term*»	function
	«*term*» «*term*»	application

1.2.1.1 Metavariables and Names

The metavariables «*term-name*», «*type-name*», and «*term-param*» range over, for a given program, a fixed and collection of names. There are two kinds of names included in this collection:

- **constructed names**: Require a construction written in the language; establish a *functional equality* between name and construction — the name universally can be substituted *with* its construction.

- **basic names**: Require a construction written in the language; establish a *definitional equality* between name and construction — the name universally can be substituted *with* or *for* its construction.
- **primitive names**: Included a priori. Terms and types provided this way are provided along with their a priori type or kind respectively, and primitive terms along with a priori reduction rules (if any). Primitive names are also trivially basic.

This syntax schema is formatted as a *generative context-free grammar*, with

- *non-terminals*: The meta-variables «*progam*», «*declaration*», «*kind*», «*kind*», «*term*».
- *terminals*: Ranged over by the meta-variables «*term-name*», «*type-name*».
- *repeated*: Expressions of the form `[` «*meta-var*» `]` indicate that any number of «*meta-var*» can be repeated in its place. Note also that metavariables inside of repeated expressions may be indexed by i or j, indicating that they are the i-th repeated sub-expression.

The following are some examples of the sorts of formal items that these meta-variables stand range over:

- «*term-name*»: `1`, `12309`, `true`, `false`, `"hello world"`, `•`.
- «*type-name*»: `natural`, `integer`, `boolean`, `string`, `void`, `unit`.
- «*term-param*»: `x`, `y`, `this-is-a-parameter`.

[**TODO**] include some function terms and (and maybe types?)

A given expression is well-formed with respect to Func if it is constructible by a series of applications of these rules. Note that this syntax does not specify any concrete terms or types. In this way, Func's syntax is defined so abstractly in order to make the definitions for its typing and semantics as concise as possible. For example, we shall reference to the type `int`, some «*term-name*»s that have type `int` such as `0`, `1`, `-1`, `2`, `-2`, `3`, `-3`, and so on. Such types and terms as these are called **atoms** because they do not have any internal; they are simply posited as primitives in the language.

1.2.2 Notations for Func

[**TODO**] change this section to be more of a narrative introduction to what Func programs look like, rather than a manual. Use some examples for each thing.

Notations establish a convenient shorthand for certain structures that establish a **syntactical equivalency** — the notation and the structure is abbreviates are universally substitutable for each other. In the programming languages community such these notations are commonly referred to as **syntax sugar**, since they are addictive and can concisely and readable express otherwise very dense code. Notations posit *new* structures in the syntax of Func, but they can completely reduce to the core set of syntactical structures as defined by table **??**. Since there are some code structures that will be used very commonly throughout this work, we shall adopt a variety of notations. This subsection starts us off with with the most common and low-level notations.

> [**TODO**] describe notational convention for lists of joined items i.e. having "[[i ;]]" means that the ";" goes between each of the "i"s in the list.

1.2.2.1 Declarations

> [**TODO**] define how declarations work, and are not part of evaluation exactly...

> [**TODO**] give typing rules for declarations

1.2.2.2 Parameters

For parametrized terms and types, a convenient notation is to accumulate the parameters to the left side of a single " ⇒ " as follows:

Listing 1.1: Notation for multiple parameters

```
(«term-param»₁ : «kind»₁) ⋯ («term-param»ₙ : «kind»ₙ) ⇒ «term»*
  ::=
    («term-param»₁ : «kind»₁) ⇒ ⋯ ⇒ («term-param»ₙ : «kind»ₙ) ⇒ «term»*

(«type-param»₁ : «kind»₁) ⋯ («type-param»ₙ : «kind»ₙ) ⇒ «kind»*
  ::=
    («type-param»₁ : «kind»₁) ⇒ ⋯ ⇒ («type-param»ₙ : «kind»ₙ) ⇒ «kind»*
```

When defining a term or type in a declaration it is convenient to write the names of parameters immediately to the right of the name being defined, resembling the syntax for applying the new term or type to its given arguments. The following notations implement this.

Listing 1.2: Notation for declaration parameters

```
term «term-name» ⟦ («type-param»:«kind») ⟧ ⟦ («term-param»:«type») ⟧
  : «type» := «term».
    ::=
      term «term-name»
        : ⟦ («type-param»:«kind») ⟧ ⇒ «type»
        := ⟦ («term-param»:«type») ⟧ ⇒ «term».
```

```
type «type-name» ⟦ («type-param»:«kind») ⟧ : «kind» := «type».
  ::=
    type «type-name» : «kind» := ⟦ («type-param»:«kind») ⟧ ⇒ «type».
```

When two consecutive parameters have the same type or kind, the following notation allows a reduction in redundancy:

Listing 1.3: Notations for multiple shared-type and shared-kind parameters

```
(⟦ «term-param» ⟧ : «type») ⇒   ::=   ⟦ («term-param»:«type») ⟧ ⇒

(⟦ «type-param» ⟧ : «kind») ⇒   ::=   ⟦ («type-param»:«kind») ⟧ ⇒
```

1.2.2.3 Local bindings

This is a core feature in all programming languages, and is expressed in Func with this notation:

Listing 1.4: Notation for local binding.

```
let «term-param»₁ : «kind»₁ := «term»₂ in «term»₃
  ::=
    ((«term-name»₁:«term»₁) ⇒ «term»₃) «term»₂
```

Such a binding allows for the scoped binding of a name to a value. The instance of the name introduced is only available from inside «*term*»$_3$ — the *body* of the local binding.

1.2.2.4 Omitted types

The types of «*term-param*»s may sometimes be omitted when they are unambiguous and obvious from their context. For example, in

```
term twice : (α → α) → α → α :=
  f a ⇒
    let a' = f a in
    f a'.
```

the types of `f`, `a`, and `a'` are obvious from the immediately-previous type of `twice`.

1.2.3 Primitives in Func

The primitive names for a program are defined by its **`primitive term`** and **`primitive type`** declarations, and the constructed names for a program are defined by the other **`term`** and **`type`** declarations. There are three particularly prevalent types, along with some primitive terms, to have defined as part of the core of Func. They are the arrow type, sum type, and product type.

1.2.3.1 Arrow Type

The first notable primitive type is the *arrow type*, where `arrow` α β is the type of terms that are functions with input type α and output type β. All we need is the following declaration, since functions are a syntactic structure already introduced by table **??**.

Listing 1.5: Primitive for the arrow type

```
primitive type arrow : kind → kind → kind.
```

Notation for infixed type arrow.

[TODO] english

Listing 1.6: Notation for infixed type arrow

```
«kind»₁ → «kind»₂   ::=   arrow «kind»₁ «kind»₂
```

This notation is right-associative. For example, the types $\alpha \to \beta \to \gamma$ and $\alpha \to (\beta \to \gamma)$ are equal. This right-associativity corresponds to the idea of *currying* functions (also called *partial application*). For example consider what the type of a function, that takes two inputs of types α and β respectively and has output of type γ, should be. The type could be written `product` α $\beta \to \gamma$ (the `product` type is defined in the soon subsubsection "Product Type"), where `product` α β contains the two inputs. It could also be written $\alpha \to \beta \to \gamma$, which associates to $\alpha \to (\beta \to \gamma)$,

1.2.3.2 Sum Type

The second notable primitive type is the *sum type*, where `sum` α β is the type of terms that are either of type α (exclusively) or of type β. This property is structured by the following declarations:

Listing 1.7: Primitives for the sum type

```
primitive type sum : kind → kind → kind.

// constructors
primitive term left  (α β : kind) : α → sum α b.
primitive term right (α β : kind) : β → sum α b.

// destructor
primitive term split (α β γ : kind)
  : sum α β → (α → γ) → (β → γ) → γ.
```

Notation for infixed type sum.

[**TODO**] english

Listing 1.8: Notation for infixed type sum

```
«kind»₁ ⊕ «kind»₂    ::=    sum «kind»₁ «kind»₂
```

This notation is right-associative. For example, the types $\alpha \oplus \beta \oplus \gamma$ and $\alpha \oplus (\beta \oplus \gamma)$ are equal.

Notation for cases of a sum term.

[**TODO**] more english

We also introduce the following syntax sugar for `split` — the following `cases` notation:

Listing 1.9: Notation for `case`.

```
cases «term»* to «kind»*
  { left («term-param»₁:«kind»₁) ⇒ «term»₁
  ; right («term-param»₂:«kind»₂) ⇒ «term»₂ }

::=

split «kind»₁ «kind»₂ «kind»*
  «term»*
  ((«term-param»₁:«kind»₁) ⇒ «term»₁)
  ((«term-param»₂:«kind»₂) ⇒ «term»₂)
```

Note however that the `to` «*kind*»$_*$ phrase in this notation will often omitted and leave the type of the case expression implicit (in the same way as described by paragraph **??**).

***n*-ary sums.** The definition of `sum` can be considered as a special case of n-ary sum types — the binary sum type. However only the binary case need be introduced primitively, since any n-ary sum type can be constructed by a nesting of binary sum types. For example, the sum of types α, β, γ can be written `sum` α `(sum` β γ`)`. For example, the product of types α, β, γ can be written `α ⊕ (β ⊕ γ)`. To streamline representation, we adopt that $\times$ is right-associative; the product `α ⊕ (β ⊕ γ)` can be written simply as `α ⊕ β ⊕ γ`. Observe that providing a $b : \beta$ as a case of this sum is cumbersomely written `right (left` b`)`. The following notation specifies a family of terms of the form `case-`i where $0 < i \in \mathbb{Z}$, which construct the i-th component of a sum term.

```
case-i   ::=   (right ∘ ⋯ ∘ right ∘ left)
```

$$\texttt{case-}i \quad ::= \quad (\texttt{right} \circ \overset{i-1}{\cdots} \circ \texttt{right} \circ \texttt{left})$$

Finally, destructing an n-ary product involves many levels of `cases` where each level handles just two cases at a time. The following notation flattens this structure by allowing the top-level `cases` to have more than two branches.

Listing 1.10: Notation for destructing n-ary sum types

```
cases «term»_* to «kind»_*
  { case-1 («term-param»_1:«kind»_1) ⇒ «term»_1
  ; ⋯
  ; case-n («term-param»_n:«kind»_n) ⇒ «term»_n }

::=

split «kind»_1 «kind»_2 «kind»_*
  «term»_*
  («term-param»_1:«kind»_1) ⇒ «term»_1)
  (x ⇒ cases x to «kind»_*
          { case-1 («term-param»_2:«kind»_2) ⇒ «term»_2
          ; ⋯
          ; case-(n − 1) («term-param»_n:«kind»_n) ⇒ «term»_n })
```

This notation is defined inductively for the sake of simplicity. Note that the parameter x introduced is a **fresh** name i.e. unique and not able to be referenced from outside this notation definition.

Named sums. The sum type is widely applicable for modeling data types which have terms that must be exclusively one from a delimited set of cases. Such data types can be defined as the sum of the types of each case. In such constructions, it is convenient to name the components as basic terms. The following notation provides a conceptually-fluid way of defining them.

Listing 1.11: Notation for defining named sum types and constructing named sum terms

```
type «type-name»_* : «kind»_*
  := «term-name»_1:«type»_1 | ⋯ | «term-name»_n:«type»_n.

::=

type «type-name»_* : «kind»_* := «type»_1 ⊕ ⋯ ⊕ «type»_n.

// constructors
basic term «term-name»_1 : «type»_1 → «type-name»_* := case-1.
⋮
basic term «term-name»_n : «type»_n → «type-name»_* := case-n.
```

There is an additional requirement for this notation that kinds $«kind»_1, \ldots, «kind»_n$ are each (independently) one of the following: either $«type»_*$ or an application of $«type»_*$, or, an n-ary arrow type ending with either $«type»_*$ or an application of $«type»_*$.

As for destruction, the notation given in listing **??** is compatible with replacing the `case-`i with the constructors named by the named sum type definition since the constructors are basic terms.

1.2.3.3 Product Type

The third notable primitive type to define here is the product type, where `product` α β is the type of terms that are a term of type α joined with a term of type β. This property is structured by the following declarations:

Listing 1.12: Primitives for `product`.

```
primitive type product : kind → kind → kind.

// constructor
primitive term pair (α β : kind) : α → β → product α β.

// destructors
primitive term first (α β : kind) : product α β → α.
primitive term second (α β : kind) : product α β → β.
```

Infixed type product.

[TODO] english

Listing 1.13: Notation for infixed type product

```
«kind»₁ × «kind»₂    ::=    product «kind»₁ «kind»₂
```

This notation is right-associative. For example, the types $\alpha \times \beta \times \gamma$ and $\alpha \times (\beta \times \gamma)$ are equal.

n-ary products. The definition of `product` can be considered as a special case of n-ary product types — the binary product type. However only the binary case need be introduced primitively, since any n-ary product type can be constructed by a nesting of binary product types (this is the same strategy as for n-ary sum types before). For example, the product of types α, β, γ can be written `α × (β × γ)`. However, constructing a product term from $a{:}\alpha$, $b{:}\beta$, $c{:}\gamma$ is cumbersomely written as `pair a (pair b c)`. A common mathematical and computer science notation for product terms, often called *tuples*, is the following.

Listing 1.14: Notation for constructing n-ary product terms

```
(«term»₁, …, «term»ₙ)
  ::=
    pair «term»₁ (pair ⋯ (pair «term»ₙ₋₁ «term»ₙ))
```

Additionally, the projecting of an n-ary product term onto one of its components is verbose. For example, the function that projects a term of type $\alpha \times \beta \times \gamma \times \delta$ onto its second component is constructed `second ∘ second ∘ first`. The following notation specifies a family of terms of the form `part-i` where $0 < i \in \mathbb{Z}$, which destruct a product term by projecting onto its i-th component.

Listing 1.15: Notations for destructing n-ary product terms

```
part-i   ::=   (second ∘ ⋯ ∘ second ∘ first)
```

(with $\overset{i-1}{\cdots}$ over the ellipsis)

One other useful utility is the ability to apply a function to a particular component of the product, called *mapping* a function over that component. The following notation defines a collection of such mapping utilities.

Listing 1.16: Notations for mapping over n-ary product terms

```
map-i   ::=   f (a₁, …, aₙ) ⇒ (a₁, …, aᵢ₋₁, f aᵢ, aᵢ₊₁, …, aₙ)
```

Named products. The product type is widely applicable for modeling data types which are composed of several required parts. Such data types can be defined as the product of the types of each other their parts. In such constructions, it is convenient to name the parts (in a similar way to the cases of named sums). These constructions are often called *record types*, since they liken to a record holding named data entries. The following notation provides a concise way of defining the type along with its appropriately named constructors and component maps.

Listing 1.17: Notation for defining named products types and utilities

```
type «type-name»_* : «kind»_*
  := { «term-name»_1:«type»_1 ; ⋯ ; «term-name»_n:«type-name»_n }.

::=

type «type-name»_* : «kind»_*
  := «type»_1 × ⋯ × «type»_n.

// component destructors
term «term-name»_1 : «type-name»_* → «type»_1 := part-1.
⋮
term «term-name»_n : «type-name»_* → «type»_n := part-n.

// components maps ∀i
term map-«term-name»_1 : («type»_1 → «type»_1) → «type-name»_* → «type-name»_*
  := map-1.
⋮
term map-«term-name»_n : («type»_n → «type»_n) → «type-name»_* → «type-name»_*
  := map-n.
```

Lastly, the following notation provides an intuitive way to construct named product terms without having to remember the order of the components:

Listing 1.18: Notation for constructing named product terms

```
{ «term-name»_1 := «term»_1 ; ⋯ ; «term-name»_n := «term»_n }
  ::=
    («term»_name_1, …, «term»_name_n)
```

where $name_1, \ldots, name_n$ are the components named in the order intended for the underlying product type.

1.2.3.4 Other Common Primitives

[**TODO**] mention other common primitives that will not be explicitly defined, based on the following:

- integer
- natural
- boolean
- unit

[**TODO**] note that primitive terms need to have reduction rules defined outside of Func (if they have any reductions), and that these are not guaranteed to not break the rest of Func. So separate proofs need to be provided for complicated primitive stuff.

1.2.3.5 Infixed-notation Associativity Levels

Having multiple infixed notations for «*kind*»s introduce inter-notational ambiguity. For example, the type $\alpha \rightarrow \beta \times \gamma \oplus \delta$ has not yet been determined to associate in any one of many possible ways. We adopt the following precedence order of increasing tightness to eliminate this ambiguity: $\rightarrow$, $\oplus$, $\times$. So, the type $\alpha \rightarrow \beta \times \gamma \oplus \delta$ associates as $(\alpha \rightarrow \beta) \times (\gamma \oplus \delta)$

1.2.4 Type application

The notation for «*type-name*» «*kind*» ⋯ «*kind*» is read as «*type-name*» acting as a kind of function that takes some number of type parameters. So, `arrow` is a type-constructor with type type parameters, say α and β, and forms the type `arrow α β`. Note that this is a different kind of function than the **function** syntactical construct, but the intuitive similarity will be addressed later in chapter **??**.

1.2.5 Typing Rules for Func

[**TODO**] fix code in math mode

[**TODO**] write Typing rules for types

Terms and types are related by a **typing judgement**, by which a term is stated to have a type. The judgement that a has type α is written as $a : \alpha$. In order to build typed terms using the constructors presented in **??**, the types of complex terms are inferred 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 $\Gamma \cup \{J\} \vdash J'$. Keep in mind that these propositions (e.g.

judgements) and inferences are in an explicitly-defined language that has no implicit rules. The terms "inference" and "judgement" are called such in order to have an intuitive sense, but rules about judgement and inference in general (outside of this context) are not implied to also apply here.

With judgements, we now can state **typing inferences rules**. Such inference rules have the form which asserts that the premises $P_1, \dots, P_n$ entail the conclusion Q. For example, could be 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:

- $\Gamma, \Gamma_i, \dots$ each range over judgement contexts,
- $a, b, c, \dots$ each range over terms,
- $\alpha, \beta, \gamma, \dots$ each range over types.

The following typing rules are given for Func:

[**TODO**] describe kinding rules for kinds (of types)

Π 1.2: Typing in Alge: Resource

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

Function-Abstraction
$$\frac{\Gamma, a{:}\alpha \vdash b{:}\beta}{\Gamma \vdash ((a{:}\alpha) \Rightarrow b){:}(\alpha \rightarrow \beta)}$$

Function-Application
$$\frac{\Gamma \vdash a{:}(\alpha \rightarrow \beta) \qquad \Gamma \vdash b{:}\alpha}{\Gamma \vdash (a\ b){:}\beta}$$

1.2.6 Reduction Rules for Func

Finally, the last step is to introduce **reduction rules**. So far we have outlined syntax and inference rules for building expressions in Func, 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 Func:

Π 1.3: Reduction in Func

$$\textsc{Simplify}\quad \frac{a \twoheadrightarrow a'}{a\ b \twoheadrightarrow a'\ b}$$

$$\textsc{Simplify}\quad \frac{b \twoheadrightarrow b' \qquad \text{value } v}{v\ b \twoheadrightarrow v\ b'}$$

$$\textsc{Apply}\quad \frac{\text{value } v}{((a : \alpha) \Rightarrow b)\ v \twoheadrightarrow [v/a]\ b}$$

$$\textsc{Split-Right}\quad \texttt{split}\ \alpha\ \beta\ \gamma\ (\texttt{left}\ a)\ f\ g \twoheadrightarrow f\ a$$

$$\textsc{Split-Left}\quad \texttt{split}\ \alpha\ \beta\ \gamma\ (\texttt{right}\ a)\ f\ g \twoheadrightarrow f\ b$$

$$\textsc{Project-First}\quad \texttt{first}\ \alpha\ \beta\ (a,\ b) \twoheadrightarrow a$$

$$\textsc{Project-Second}\quad \texttt{second}\ \alpha\ \beta\ (a,\ b) \twoheadrightarrow b$$

[**TODO**] note that Apply is usually referred to as β-reduction.

The most fundamental of these rules is Apply (also known as β-reduction), which is the way that function applications are resolved to the represented computation'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 $((a{:}\alpha) \Rightarrow b) : \alpha \rightarrow \beta$ 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 computation.

1.2.7 Properties of Func

With the syntax, typing rules, and reduction rules for Func, we now have a completed definition of the language. However, some of the design decisions may seem arbitrary even if intuitive. This particular framework is good because it maintains a few nice properties that make reasoning about Func intuitive and extendable.

[**TODO**] define these properties in English; want to keep prog-language and meta-language separate.

Theorem 1.1. (Type-Preserving Substitution in Func**).** If $\Gamma, a{:}\alpha \vdash b{:}\beta$, $\Gamma \vdash v{:}\alpha$, and value v, then $\Gamma \vdash ([v/a]b){:}\beta$.

Theorem 1.2. (Reduction Progress in Func**).** If $\{\} \vdash a{:}\alpha$, then either value a or there exists a term a' such that $a \twoheadrightarrow a'$.

Theorem 1.3. (Type Preservation in Func**).** If $\Gamma \vdash a{:}\alpha$ and $a \twoheadrightarrow a'$, then $\Gamma \vdash a'{:}\alpha$.

Theorem 1.4. (Type Soundness in Func**).** If $\Gamma \vdash a{:}\alpha$ and $a \twoheadrightarrow a'$, then either value a' or there exists a term a'' such that $\Gamma \vdash a''{:}\alpha$ and $a' \twoheadrightarrow a''$.

Theorem 1.5. (Strong Normalization in Func**).** For any term a, either value a or there is a sequence of reductions that ends in a term a' such that value a.