COMPSCI 3MI3 : Project 3 - A Mini Language

Fall 2021

Nicholas Moore

1 The Task

Throughout this course, we have been developing language components and type checking rules which evaluate the terms of a simple programming language. The time has come to tie it all together, and Build! That! Language!

You are tasked with developing a Haskell program which implements the following language features covered in this course.

- Booleans (As defined in Typed Arithmetic Expression language)
- Naturals (As defined in Typed Arithmetic Expression language)
- Simply Typed Pure λ-Calculus
- The Unit data type and value
- Let bindings
- The Sequencing Operator
- Reference Operations

You will develop a Haskell function, `ssos`, which implements the semantics of the above language features. You will also develop a function `typeCheck`, which determines whether or not a term is well typed. You have been provided with data types in the file `template.hs`. There are likely to be minor variations with the datatypes you used in previous projects. For this project, you will be required to use the data types provided.

The recommended approach to this project is to go through the language constructs in the order provided, testing as you go, and making sure that one language feature works before the next one is attempted. It is highly recommended that you take a backup after completing each language feature, or store your solutions in a (**private!**) repository. Foir more information on how your project is to be submitted and scored, see §2.

1.1 Test Cases

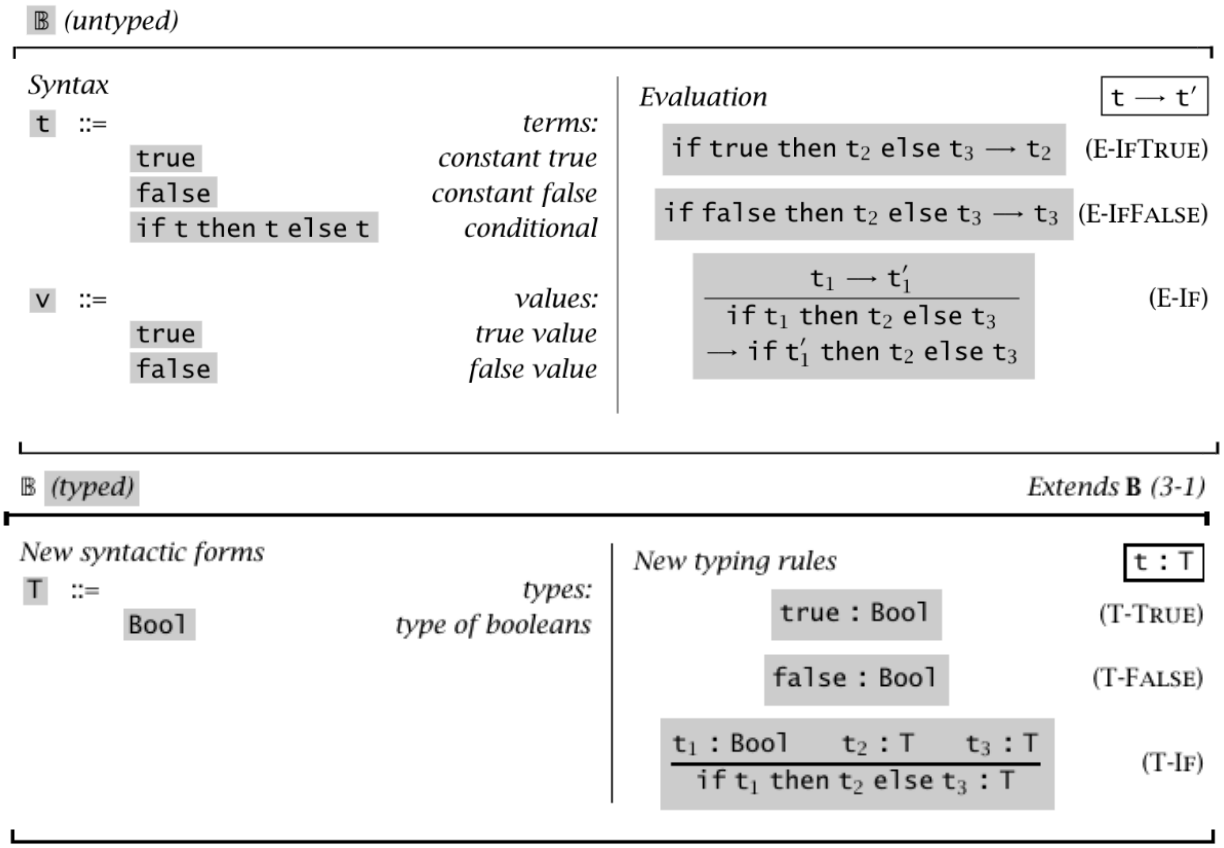
A file called `Test_Cases.txt` has been provided. This plain-text file contains numerous test cases, with both evaluated results and the found type. Step-by-step evaluations, including intermediate memory states, have been provided.

1.2 `eval` and `run`

As with previous projects in this course, you will construct an `eval` function which repeatedly applies the term you are evaluating to `ssos` until a normal form is reached (regardless of correctness).

Due to the addition of a typechecker, we can finally verify the correctness of our programs **before** executing them. You will need to write a function, `run`, which first typechecks the given term, and only evaluates it if the term typechecks.

1.3 Booleans (Challenge Rating ⋆⋆)

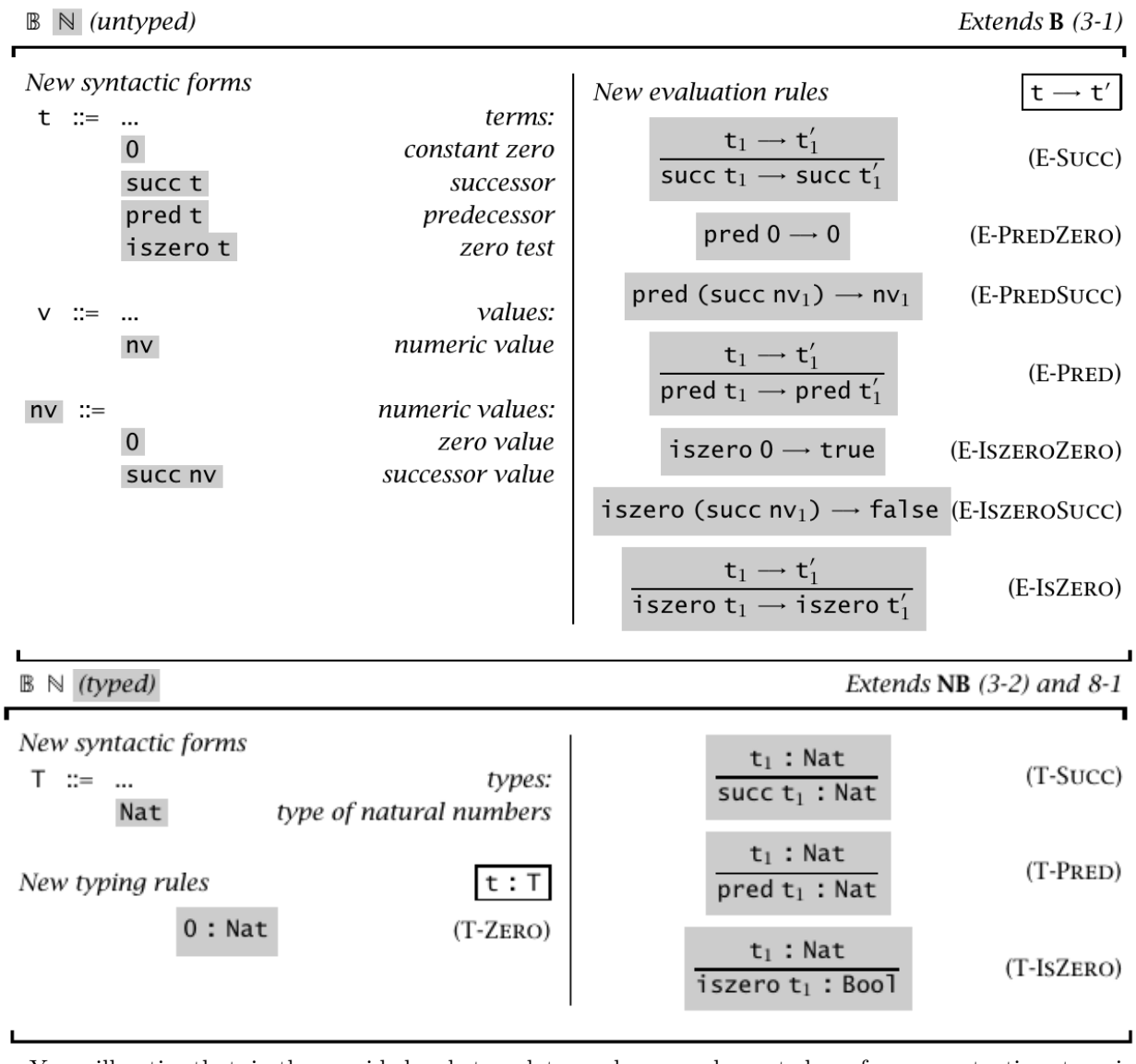


Implementation of Boolean semantics was a large component of Project 1, so our main task with the Booleans in Project 3 will be creating the typechecker. Recall that typechecking is a process that may fail to find a type for a term. While single step semantics, augmented with reflexivity is a total function over terms, typing is a **partial function** over terms. The way that Haskell handles partial functions is using the `Maybe` monad.

- You can apply the `Maybe` monad to a type to indicate that it may or may not exist.
  - For example, a value of the type `Maybe Bool` might be a Boolean, or it might not exist.
- `Maybe`'s values are either `Just x` or `Nothing`, with x being a term of the type within the `Maybe`.

So, rather than our typechecker directly yielding a type as a value, we want it to return a value wrapped in the `Maybe` monad. This way, any time our type inference rules fail to determine the type of an expression, we return `Nothing`. This is reflected in the project template.

1.4 Naturals (⋆⋆)



You will notice that, in the provided code template, we have amalgamated our former syntactic categories of values and numeric values (and a few more besides!) Before we had a type system, we needed two categories of value to correctly restrict the construction of successor values. With the addition of typechecking, this happily becomes unnecessary. Since the input term to the successor function is required to be typed Nat, we can be confident that a well-typed term does not need to be restricted in this manner at the semantic level.

1.5 λ-Calculus ($\star\star\star\star$)

→ *(typed)* — *Based on λ (5-3)*

Syntax

t	::=		*terms:*
		x	*variable*
		λx:T.t	*abstraction*
		t t	*application*
v	::=		*values:*
		λx:T.t	*abstraction value*
T	::=		*types:*
		T→T	*type of functions*
Γ	::=		*contexts:*
		∅	*empty context*
		Γ, x:T	*term variable binding*

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-App2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-AppAbs)}$$

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-Var)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-App)}$$

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

Typechecking λ-Calculus requires the introduction of Γ as a parameter of our typechecking function. In the provided template, pairs in the Γ relation are expressed as a list of two element tuples. Naturally, Γ must be passed down through recursive calls to the typechecking function, with modifications any time we are typechecking through a λ abstraction (or let binding).

You will notice that, in the provided template, both variables and abstractions have been included in the set of values, rather than in the set of terms. This is intentional. λ abstractions in particular must be values in order for us to be able to store them in memory.

Finally, on the subject of substitution. The above substitution rules apply for the terms of pure λ-Calculus, but no rules are provided for the other terms. You may assume that substitutions are passed through all other terms with subterms, and that a substitution applied to a non-variable value does nothing. You may assume similar behaviour for relabelling operations.

1.6 The Unit Type ($\star$)

→ `Unit` *Extends $\lambda_{\to}$ (9-1)*

New syntactic forms

t	::=	...	*terms:*
		`unit`	*constant* `unit`
v	::=	...	*values:*
		`unit`	*constant* ***`unit`***
T	::=	...	*types:*
		`Unit`	*unit type*

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\Gamma \vdash \texttt{unit} : \texttt{Unit} \qquad \text{(T-Unit)}$$

There's not much to say about the unit type and value. Most of the work has been provided for you in the code template, and unit has no evaluation rules.

1.7 Let Bindings ($\star\star$)

→ `let` *Extends $\lambda_{\to}$ (9-1)*

New syntactic forms

t	::=	...	*terms:*
		`let x=t in t`	*let binding*

New evaluation rules $\boxed{t \longrightarrow t'}$

$$\texttt{let x=}v_1\texttt{ in }t_2 \longrightarrow [x \mapsto v_1]t_2 \qquad \text{(E-LetV)}$$

$$\frac{t_1 \longrightarrow t_1'}{\texttt{let x=}t_1\texttt{ in }t_2 \longrightarrow \texttt{let x=}t_1'\texttt{ in }t_2} \qquad \text{(E-Let)}$$

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \texttt{let x=}t_1\texttt{ in }t_2 : T_2} \qquad \text{(T-Let)}$$

Let bindings in this form are not substantially different from λ abstractions with application, except for the fact that we use a form of type inference to type the variable.

The only tricky part is, in order for type checking to work, let bindings must also contribute a variable-type pair to Γ. As with λ abstraction, typing information for the variable bound by a let binding must also be made available to sub-calls to the typechecking function.

Let bindings are included in this project because we will be using them for a special purpose, when we implement reference operations. Essentially, we will use let bindings to simulate the algebraic variable manipulation used to construct programs in the textbook and lectures.

1.8 The Sequencing Operator ($\star\star$)

$$\frac{t_1 \to t_1'}{t_1; t_2 \to t_1'; t_2} \qquad \text{(E-Seq)}$$

$$\texttt{unit}; t_2 \to t_2 \qquad \text{(E-SeqNext)}$$

$$\frac{\Gamma \vdash t_1 : Unit \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1; t_2) : T_2} \qquad \text{(T-Seq)}$$

On it's own, the sequencing operator should be reasonably straightforward to implement. We have chosen here to model sequencing in the manner of our external language, with sequencing as a full term of the language with its own evaluation and typing rules.

1.9 Reference Operations (⋆⋆⋆⋆⋆)

Syntax

t	::=		*terms:*
		x	*variable*
		λx:T.t	*abstraction*
		t t	*application*
		unit	*constant* unit
		ref t	*reference creation*
		!t	*dereference*
		t:=t	*assignment*
		l	*store location*
v	::=		*values:*
		λx:T.t	*abstraction value*
		unit	*constant unit*
		l	*store location*
T	::=		*types:*
		T→T	*type of functions*
		Unit	*unit type*
		Ref T	*type of reference cells*
Γ	::=		*contexts:*
		∅	*empty context*
		Γ, x:T	*term variable binding*
μ	::=		*stores:*
		∅	*empty store*
		$\mu, l = v$	*location binding*
Σ	::=		*store typings:*
		∅	*empty store typing*
		Σ, l:T	*location typing*

Evaluation $\boxed{t \mid \mu \longrightarrow t' \mid \mu'}$

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1\ t_2 \mid \mu \longrightarrow t_1'\ t_2 \mid \mu'} \quad \text{(E-App1)}$$

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1\ t_2 \mid \mu \longrightarrow v_1\ t_2' \mid \mu'} \quad \text{(E-App2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \mid \mu \longrightarrow [x \mapsto v_2]t_{12} \mid \mu \quad \text{(E-AppAbs)}$$

$$\frac{l \notin dom(\mu)}{\text{ref } v_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto v_1)} \quad \text{(E-RefV)}$$

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{\text{ref } t_1 \mid \mu \longrightarrow \text{ref } t_1' \mid \mu'} \quad \text{(E-Ref)}$$

$$\frac{\mu(l) = v}{!l \mid \mu \longrightarrow v \mid \mu} \quad \text{(E-DerefLoc)}$$

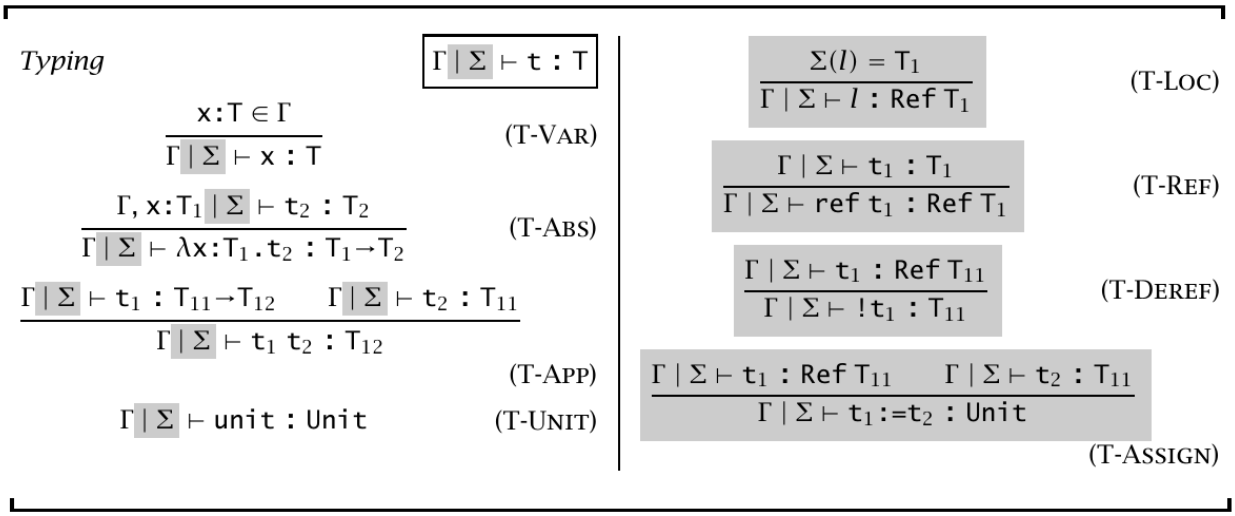
$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{!t_1 \mid \mu \longrightarrow !t_1' \mid \mu'} \quad \text{(E-Deref)}$$

$$l{:=}v_2 \mid \mu \longrightarrow \text{unit} \mid [l \mapsto v_2]\mu \quad \text{(E-Assign)}$$

$$\frac{t_1 \mid \mu \longrightarrow t_1' \mid \mu'}{t_1{:=}t_2 \mid \mu \longrightarrow t_1'{:=}t_2 \mid \mu'} \quad \text{(E-Assign1)}$$

$$\frac{t_2 \mid \mu \longrightarrow t_2' \mid \mu'}{v_1{:=}t_2 \mid \mu \longrightarrow v_1{:=}t_2' \mid \mu'} \quad \text{(E-Assign2)}$$

continued...



Reference operations are the most difficult portion of this project. The grading scheme has been set such that completing the entire project up to section §1.8 is worth 90% (an A+). A complete and correct submission with a complete and operational implementation of referencing is worth 110% (An A++?) Getting the semantics working is easier than getting the typing operational, but both are challenging.

1.9.1 Semantics (⋆⋆⋆⋆)

Implementing the evaluation rules for referencing has a few tricky parts:

- For (E-RefV), an unused location must be obtained from the available locations. It doesn't matter which location is chosen, so long as it is fresh. It is acceptable, if no more locations are available and another is requested, that the program fails with whatever error your implementation would naturally trip. The process greatly resembles selection of a fresh variable name during relabelling.
- Note that *all* evaluation rules must be augmented with a proper handling of μ. Remember, reflexivity rules need to pass on changes that may have been made to μ during the handling of the subterm.
- Another place μ will need to be factored in is in your `eval` function. Under multi-step single step operational semantics, it is correct for the state of μ to be pass from one call to `ssos` to the next. This is the reason that, in the code template, `ssos` returns a tuple containing both a term and some μ.

Aside from these points, implementing the referencing evaluation rules should be relatively straightforward. The difficulty of this part is largely the refactoring of `ssos` to handle μ correctly.

1.9.2 Typing (⋆⋆⋆⋆⋆)

An observant student may have noticed that, while the code template contains data structures for Γ and μ, the typing store Σ is noticeably absent. For this project, we will modify our semantics, so as to eliminate the use of Σ.

So far, our implementation has managed to keep typing and evaluation rules completely separate. The population of Σ with location-type pairs is, however, strictly a run-time phenomenon. It is dependent on locations found during memory allocation. In theory, if the new memory location selection mechanism you coded in §1.9.1 was deterministic, and if the typechecker and the single step semantics were persuaded to execute subterms in the same order, a typing traversal would reliably yield the same locations as evaluation traversal in the same order. This is, however, a very bad model of how real memory works, and so we won't reconstruct Σ during pure typechecking.

To solve this problem, we must first **restrict the typing rule for memory allocation so that it is only well typed when it occurs in a let binding**. Free occurrences of memory instantiation outside of let bindings will not be considered well typed. Thus, the type of any allocated locations will be added as a

type for the bound variable to Γ. From there, the typing rule for variables will pick the reference type out of Γ later on in the program.

While this approach is somewhat restrictive, it makes the construction of a separate typing store completely unnecessary. Our typing and evaluation rules remain separate and distinct, and nobody has to completely, painfully and laboriously rewrite `eval`, `typeCheck` and `ssos` to account for data sharing between execution and typechecking.

2 Scoring

Scoring will be progressive, based on the number of language features you correctly implement.

Feature Set	Semantics	Type Checking	Total	Cumulative
$\mathbb{B}$	2	13	15	15 / 100
$\mathbb{B} \cup \mathbb{N}$	2	13	15	30 / 100
$\mathbb{B} \cup \mathbb{N} \cup \lambda$	4	21	25	55 / 100
$\mathbb{B} \cup \mathbb{N} \cup \lambda \cup \mathbb{U}$	3	2	5	60 / 100
$\mathbb{B} \cup \mathbb{N} \cup \lambda \cup \mathbb{U} \cup \mathbb{L}$	8	7	15	75 / 100
$\mathbb{B} \cup \mathbb{N} \cup \lambda \cup \mathbb{U} \cup \mathbb{L} \cup \mathbb{S}$	8	7	15	90 / 100
$\mathbb{B} \cup \mathbb{N} \cup \lambda \cup \mathbb{U} \cup \mathbb{L} \cup \mathbb{S} \cup \mathbb{R}$	8	12	20	110 / 100

Booleans	$\mathbb{B}$
Naturals	$\mathbb{N}$
λ-Calculus	λ
Unit Type	$\mathbb{U}$
Let bindings	$\mathbb{L}$
Sequencing	$\mathbb{S}$
References	$\mathbb{R}$

- According to the above, the maximum grade available for this project is 110%.
- Only the above feature sets are valid feature sets. If a feature is skipped, the project will be graded with whichever of the above feature sets is the largest subset of the provided features.
  - For example, an implementation containing $\mathbb{B} \cup \mathbb{N} \cup \mathbb{U}$ will be graded as $\mathbb{B} \cup \mathbb{N}$, because it skipped λ.
- Full marks will only be available for solutions in which lines of the `ssos` and `typeCheck` functions have been commented with the evaluation and typing rule names implemented in that line. For the most part, each typing or evaluation rule only requires one line in either function. Lines which do not correspond to an explicit rule should have a comment indicating the purpose of the line.
  - For example, lines giving values reflexivity in `ssos` should be indicated with a comment.
- Your final submission should be a single Haskell source file, named `<Your MacID>.hs`. Submission is through the Avenue dropbox.