Osazone Overview of the Artifact

This is an overview of the artifact for the OOPSLA'24 paper entitled "Semantics Lifting for Syntactic Sugar". This document is written with Typst. And the source code is located in `doc/README.typ`.

Contents

*We will use **§3.2** to refer to the section of this document and use **Sec. 3.2** for the section of our paper.*

1. Introduction

1.1. The Artifact

Our paper (Semantics Lifting for Syntactic Sugar) proposes a systematic framework to lift semantics for syntactic sugars. Based on the semantics of the host language and DSL language construct definitions as syntactic sugars, we can obtain a standalone, host-independent semantics of the DSL for free.

The two most important properties of our semantics-lifting algorithm are correctness and abstraction, mentioned in Sec. 2 and Sec 3.4 in our paper. As a concrete example, consider the example given in Sec. 2. Taking LC (Lambda calculus) as the host language, we can define boolean operations using syntactic sugars. For instance, the `and` operation can be defined as:

$$e_1 \text{ and } e_2 \to_d \text{if } e_1 \text{ then } e_2 \text{ else false}$$

where the left-hand side (LHS) and right-hand side (RHS) are separated by $\to_d$. The LHS defines a new language construct of DSL Bool with meta-variables and the RHS is composed of constructs in the host language and these meta-variables. Given the semantics of the host language (the evaluation rule of `if` is given below, while other rules can be found in Fig. 2 of the paper),

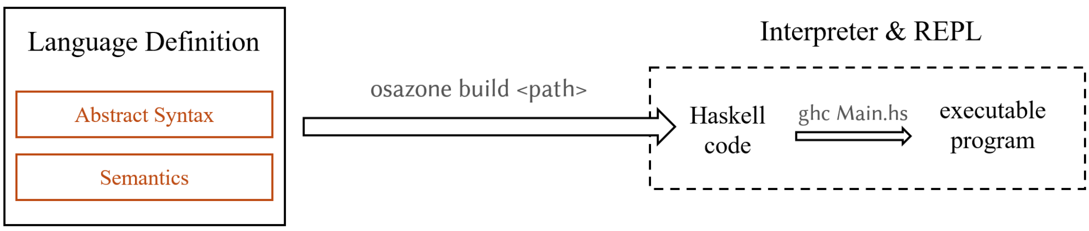
$$\frac{e_1 \Downarrow \text{true} \qquad e_2 \Downarrow v_2}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v_2} \quad \text{and} \quad \frac{e_1 \Downarrow \text{false} \qquad e_3 \Downarrow v_3}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v_3}$$

we can obtain the following host-independent semantics for DSL, without mentioning host-language construct `if`.

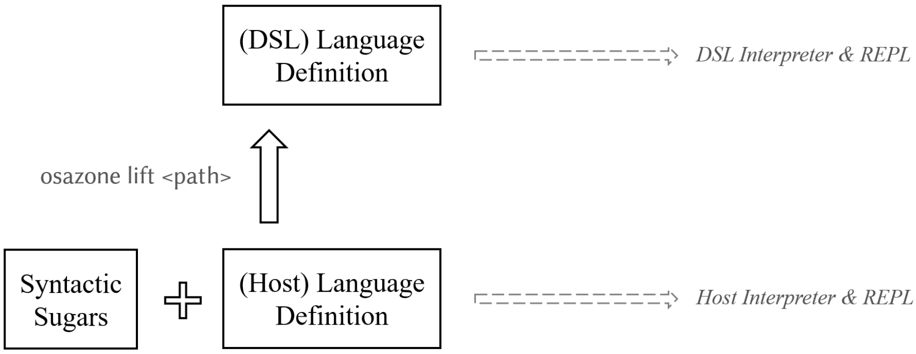
$$\frac{e_1 \Downarrow \text{true} \qquad e_2 \Downarrow v_2}{e_1 \text{ and } e_2 \Downarrow v_2} \quad \text{and} \quad \frac{e_1 \Downarrow \text{false}}{e_1 \text{ and } e_2 \Downarrow \text{false}}$$

This artifact, *Osazone* is a prototype of the DSL development framework, implementing the semantics-lifting algorithm. As mentioned in Sec. 4 of our paper, *Osazone* provides **two aspects of functionality:** (1) to define a (host) language using *Osazone*'s meta-language; and (2) to define a (domain-specific) language with syntactic sugars on top of an existing host language. The latter is the focus of our paper, while the former is the basis for making DSL programs executable.

- **Define a (host) language.** The input is a language definition (abstract syntax and semantics); and the output is an interpreter of the language (note: our system will generate a set of Haskell files for further compilation).



- **Define DSL via syntactic sugars.** The input is the definition of an existing language and syntactic sugars; and the output is the DSL definition, which can be used to generate an interpreter.



We are applying for the **Reusable** badge. We believe that the *Osazone* system can support the main claims of the paper and the code structure is clear and reusable. If reviewers have any problems, please contact us on the artifact submission system and we are always available to resolve all problems.

1.2. Claims made by the paper

In Sec. 5 of our paper, taking MiniML as the host language, we implement a series of DSLs in *Osazone*, as shown in the Table 1 and Table 2. As mentioned in Sec. 5.2, we use these examples to cover various scenario of DSL design to illustrate that *Osazone* is expressive enough to develop DSLs (RQ1). Also, we show the number of syntactic sugars as a metric for measuring development efficiency to show *Osazone* can simplify DSL development, and the lifted semantics are abstract for DSL users (RQ2). We have given definitions of all these languages to demonstrate the functionality of the system and to verify our claims.

Table 1. DSLs implemented in Osazone

ID	Example	Extension	#Sugars	Description
1	Calculator	IO(2)	4	Output computed value according to input
2	Expressions	–	2/3	A simple expression language (deep/shallow embedding)
3	Chemical Equation	–	12	Verify if the chemical equation is balanced
4	Complex	–	6	Calculate complex numbers
5	XML	–	7	Describe XML structures
6	Robot	Ref(4) + IO(1)	18	Control robot movement
7	State Machine	–	8	Simulate finite-state machine
8	Bool	–	6	Calculate Boolean expressions
9	IMP	Ref(4)	9	An imperative language
10	Forth	Stack(2)	11	Stack-oriented programming language
11	Pretty Printer	Strings(3)	13	Format text in a flexible and convenient way
12	List Comprehension	Strings(1)	7	Produce lists from generators (Haskell)

Extension: Extension on MiniML and number of language constructs introduced; **#Sugars**: Number of language constructs introduced by syntactic sugars. We implement the expression language using two methods: deep and shallow embedding [5], with different numbers of sugars.

Table 2. Sample Programs in our DSLs (Program Entrances Omitted)

Bool `(not false) xor true`	Expressions $4 + 6 * 8$	Chemical Equation $2H_2 + O_2 \rightarrow 2H_2O$	Complex $(2 +_c 3i) *_c (5 -_c 4i)$

XML
 	
`<birds>`
`<bird name="Sparrow">`
`<food>seed</food>`
`<food>insects</food>`
`</bird>`
`</birds>`
	Robot
	
	`routine turnAround`
	`repeat 2 times`
	`turnLeft`
	`end`
	`end`
	`turnAround`
	`step`
	`printPos`
		State Machine
		
		`Rules Begin`
		`close [force] open`
		`close [request] wait`
		`wait [request] wait`
		`wait [permit] open`
		`wait [force] open`
		`End`
		`Initial state: close`
		`Run [request]`
		`Run [request]`
		`Run [permit]`
			Calculator
			
			`calculator` *area*
			`input` w
			`input` h
			`output` $w * h$

Imp
 	
`var i = 1;`
`var sum = 0;`
`while (i <= 10) do`
`sum += i; i += 1;`
`end;`
`sum`
	Forth
	
	`: SQUARE DUP * ;`
	`3 SQUARE 4 SQUARE + .`
		Pretty
		
		`text "bbbbb[" <>`
		`nest 2 line <>`
		`text "ccc," <>`
		`nest 2 line <>`
		`text "dd" <>`
		`nest 0 line <>`
		`text "]"`
			List Comprehension
			
			`let xs = [1, 2, 3] in`
			`[ x * x | x <- xs,`
			`y = x - 2, y > 0 ]`

2. Hardware Dependencies

Our system has loose hardware requirements, as long as it can run Haskell's environment. The experiment is conducted on a laptop with Windows 11, AMD Ryzen 7 4800HS with Radeon Graphics @ 2.90GHz, and 16 GB RAM.

3. Getting Started Guide

3.1. Installation

There are two ways for complete installation of *Osazone*: build from the source; or use in a docker container (recommended).

3.1.1. Build From Source

1. Install dependencies. The *Osazone* system is written in Haskell, which requires the following toolchains (tested on Windows 11). All these tools can be installed using GHCup (https://www.haskell.org/ghcup/).
   - stack (tested on 2.15.3, resolver: LTS 22.6)
   - cabal (tested on 3.10.2.1)
   - ghc (tested on 9.6.3)
2. Clone *Osazone*.

```
$ git clone https://github.com/vbcpascal/Osazone-oopsla24 Osazone
$ cd Osazone
```

3. Build the whole project under the root directory of the project.

```
Osazone$ stack build
```

4. Test whether *Osazone* is successfully installed:

```
Osazone$ stack run version
```

The expected output is `Osazone 0.1.3.0`.

3.1.2. Download Docker Image

We both upload the Dockerfile to the *Osazone* repository, and release a docker image of *Osazone* (about 10GB) on Docker Hub.

- **Build from Dockerfile.** Download the Dockerfile from https://github.com/vbcpascal/Osazone-oopsla24/blob/master/Dockerfile and run it using the following commands.

```
$ docker build -t osazone .
$ docker run -it osazone /bin/bash
Osazone$ stack run version
```

The expected output is `Osazone 0.1.3.0`.

- **Pull from Docker Hub.** Download and run it using the following commands.

```
$ docker pull vbcpascal/osazone
$ docker run -it vbcpascal/osazone /bin/bash
Osazone$ stack run version
```

The expected output is `Osazone 0.1.3.0`.

3.2. Quick Start

All the languages that appeared in the paper are implemented, which can be found in the `examples/` of the project. The paths to these languages are given as follows:

	Language	Location in Paper	Path in Project
Host	LC	Sec. 2	examples/Host/LC/
	Func	Sec. 3	examples/Host/Func/
	MiniML	Sec. 5	examples/Host/MiniML/
	MiniML + Ref	Sec. 5	examples/Host/MiniMLRef/
	MiniML + Stack	Sec. 5	examples/Host/MiniMLStack/
DSL	Bool	Sec. 2	examples/DSLs/Bool/
	Sugar examples in Sec. 3	Sec. 3	examples/DSLs/Funcex/
	DSLs in Table 1	Sec. 5	examples/DSLs/<name of DSL>

We provide a command to lift and build all these languages automatically[1].

```
Osazone$ stack run review crazy-build
```

The expected output is

[1]We use `Osazone$` to represent that you should be at the root directory of the project.

```
Build all the host languages

[ 1 of 5 ] host examples/Host/Func
Building language from examples/Host/Func
...
Generating interface files
Done.
...
```

In the path of these languages, a `build/` directory will be generated, with lots of Haskell files. These Haskell codes will be used to be compiled as interpreters later. A total of 5 host languages and 15 DSLs will be processed. We will show the instructions to build and execute these languages in the next section.

For docker users, we provide a method to compile all these generated files by the following command.

```
Osazone$ Osazone-exe review compile  # do not use stack run
```

The excepted output is

```
Try to compile interpreters of all the host languages

This will use command `stack ghc`. If one of them failes, try to run `cd <path>/
build; stack ghc Main.hs` and read the error messages.
[ 1 of 5 ] host examples/Host/Func
...
```

And executable files `<path>/build/Main` will be generated. Users can try to use the interpreter as follows (using Bool as an example):

```
Osazone$ cd examples/DSLs/Bool/build/
build$ ./Main ../test.l
```

Some notes about the commands:
- Using `stack run review compile` may result in an unexpected error. This is due to some special behavior of stack.
- We have installed `Osazone-exe` in the **docker** (copying the binary file to local bin directory[2]). This means that the vast majority of `stack run` and `Osazone-exe` in commands can be replaced with each other in use, EXCEPT `review compile`.
- For those who build *Osazone* from source, our command didn't install `Osazone-exe` by default. You must use `stack run` in commands, and `Osazone-exe` is invalid. You can install `Osazone-exe` binary file by running `stack install`.
- The principle of `review compile` is actually to call `stack ghc` in the path of each language. Whether the compilation is successful depends on the ghc version and package path found by stack. We can only guarantee that the correct environment is provided in the Docker.

4. Step-by-Step Instructions

In this section, we will take LC as the host language and lift the semantics of Bool using *Osazone* as an example. They are also suitable for building and running other languages.

You can remove all the generated files after `crazy-build` by using the following command:

[2]https://docs.haskellstack.org/en/stable/GUIDE/#the-stack-install-command-and-copy-bins-option

```
Osazone$ stack run review clean
```

4.1. Define the Host Language

Check the definition. The definition of LC (lambda calculus) is given in `examples/Host/LC`, containing the following files:

1. `language.yaml`: provides basic information of the language;
2. `src/Lang.osa`: declares the abstract syntax of LC;
3. `src/Eval.osa`: defines the semantics of LC;
4. `src/Subst.osa`: defines substitution, a meta-function;
5. `test.l`: sample programs of LC.

The `.osa` files in `src` use the Osazone's meta-language, which has a similar syntax to Haskell. At a glance, the semantics of if-then-else is defined as follows (in `src/Eval.osa:10-13`), straightforwardly:

```
eval (EIf e1 e2 e3) =
  case eval e1 of
    ETrue  -> eval e2
    EFalse -> eval e3
```

Build the language. We will use the following command to generate an interpreter (source code written in Haskell).

```
Osazone$ stack run build examples/Host/LC
```

You will find a `build` directory is generated in `example/Host/LC`, consisting of lots of Haskell files.

Then, compile these Haskell codes. This requires that you must have a global Haskell environment, and make sure that the following packages have been installed globally: `prettyprinter`, `mtl`, `prettyprinter-ansi-terminal`.

- If you are running a docker, the environment has been prepared.
- If you are in the `examples` directory of the project, you can use stack to compile them. You don't need to specify additional dependency information because the stack will find the dependency of the *Osazone* project (`stack.yaml`) as the dependency.
- If you have a system GHC and you miss one or more of the packages, use `cabal install <package-name> --lib` to install them.

```
Osazone$ cd examples/Host/LC/build

# if using stack or docker
build$ stack ghc Main.hs -- -Wno-overlapping-patterns

# if using system GHC
build$ ghc Main.hs -package containers -package prettyprinter -package mtl -package
prettyprinter-ansi-terminal -Wno-overlapping-patterns
```

After that, an executable program is generated.

Test the language. Now, you should be in the `build/` directory. Type

```
build$ ./Main
```

and you will see an REPL interpreter. You can have some interaction with this interpreter. Use `:e` or `:eval` to evaluate an expression and use `:q` or `quit` to quit.

```
LC interpreter: generated by Osazone-0.1.3.0
LC>
```

Note that we don't implement parser and printer for the language (which is not our focus), so all the programs are presented as S-expressions based on Haskell's `deriving (Read, Show)`. Type `:eval EIf ETrue EFalse ETrue`, and it will return `EFalse`, which means the program is evaluated successfully and the return value is `EFalse`.

```
LC interpreter: generated by Osazone-0.1.3.0
LC> :eval EIf ETrue EFalse ETrue
EFalse
LC> :eval EApp (EAbs "x" (EIf (EVar "x") EFalse ETrue)) ETrue
EFalse
LC> :quit
```

Also, a set of sample programs have been already defined in `test.l`. Use the following commands to evaluate all the programs in `test.l`

```
build$ ./Main ../test.l
```

You will obtain the following result. There are some error messages. We purposely wrote some buggy programs in the test file.

```
└ ./Main ../test.l
Line 10: if true false true
 [:e] EFalse
Line 13: \x. if x false true
 [:e] EAbs "x" (EIf (EVar "x") EFalse ETrue)
Line 16: (\x. if x false true) true
 [:e] EFalse
Line 19: (\f. \x. f x) (\x. if x false true)
 [:e] EAbs "x" (EApp (EAbs "x" (EIf (EVar "x") EFalse ETrue)) (EVar "x"))
Line 22: Hello, oopsla!
 [:e] ParseError: no parse.
Line 25: true false
 [:e] MatchError: match ETrue failed. Expected patterns:
  (Lib.Lang.EAbs x e)
when
  eval EApp ETrue EFalse
```

Don't forget to go back to the root directory of the project after testing.

```
build$ cd ../../../../
Osazone$
```

4.1.1. Lift and Use the DSL

Check the definition. The definition of Bool is given in `examples/DSLs/Bool`, containing the following files:

1. `lifting.yaml`: provides configurations of the language. In this file, LC is used as host language is specified.
2. `sugars/Bool.ext`: defines syntactic sugars for Bool language.

A syntactic sugar is defined as follows:

```
And Exp Exp :: Exp
And e1 e2 = EIf e1 e2 EFalse
```

The first line declares the program sort of this new construct (an `Exp`) and the sorts of its arguments (both `Exp`s). The right-hand side is written as S-expression using host language constructs (`EIf` and `EFalse` in LC) and variables. This definition equals to the following format in our paper:

$$e_1 \text{ and } e_2 \to_d \text{if } e_1 \text{ then } e_2 \text{ else false}$$

The extension file contains some other declarations, well documented, which are used for advanced control in lifting and can be ignored for now. The complete explanation of these declarations can be found in § 5.2.

Lift the semantics. We will use the following command to get a standalone DSL definition. Note that you should be in the root directory of project.

```
Osazone$ stack run lift examples/DSLs/Bool
```

You will find `language.yaml` and `src/` are generated in `examples/DSLs/Bool`, which has similar structure to LC. For example, in `src/Lang.osa`, `And` is appended as a constructor of `Exp`; and in `src/Eval.osa`, the evaluation rule of `And` is lifted as follows:

```
eval (And e1 e2) =
  case eval e1 of
    ETrue -> eval e2
    EFalse -> EFalse
```

not mentioning host-language construct `EIf` (abstraction property).

Generate and compile the interpreter. Then, according to the generated DSL definition, we can generate and compile an interpreter for Bool, just like what we did for LC.

```
Osazone$ stack run build examples/DSLs/Bool
Osazone$ cd examples/DSLs/Bool/build
# replace the following command if using system GHC
build$ stack ghc Main.hs -- -Wno-overlapping-patterns
build$ ./Main ../test.l
```

4.1.2. Summary

Each DSL should be evaluated in the following steps:
1. Check the syntactic sugars in `<path-to-DSL>/sugars/<file>.ext`
2. Lift the language and get a standalone DSL definition (using `stack run lift`)
3. Build the language and get a set of Haskell source code (using `stack run build`)
4. Compile the Haskell code and run test file (using `stack ghc Main.hs`)

To verify our claim, we provide some supplementary notes:
- To obtain the result of Table 1, just count the number of syntactic sugars in `.ext` file.
- All the programs in Table 2 are defined in related test file (`test.l`).
- We introduce the concept of admissible set S in Sec. 3.4, which corresponds to the filters in `.ext` file.

What's more, we provide some snippets for convenience.

Lift and build. We always need to build a language after lifting, by adding an option `--build` when lifting. Note that when using stack, an extra `--` is necessary.

```
Osazone$ stack run lift ./example/DSLs/Bool -- --build
```

Build all languages. We hard-code all the languages mentioned in the paper, and automate the above steps as much as possible. However, compiling the generated Haskell code and running sample programs still need to be done manually.

```
Osazone$ stack run review crazy-build
```

Clean the generated files. To remove generated files, i.e. `build/` for host language, `build/` and `src/` for DSLs, the following command is useful.

```
Osazone$ stack run review clean
```

Complete documentation of these command line arguments are given in § 5.4.

5. Reusability Guide

5.1. Design a Language in *Osazone* from Scratch

Osazone is a platform to development language, readers can define their own language for further research. In this section, we will define a simple language named `Num` as an example step by step. We use `osa` icon to represent the limitation of our framework in current version in terms of practicality. These issues do not affect the thesis statement, and they are just not yet perfect in user-friendliness. A sample program of `Num` is like `x = 1; y = 2; x = x + y; x`.

The definition of a language contains at least the following parts:
- `language.yaml`: configurations
- `src/`: language definition
  - `Lang.osa`: abstract syntax
  - `Eval.osa`: evaluation rules

Configuration file. We need to specify the name, version, extension name, the path to the standard library, language modules and REPL configurations in the configuration file.

```
# language.yaml
name:          Num
version:       1.0
extension:     .l
lib:           # path to Osazone/lib
modules:
  ast:         Lang
  execution:   Eval
repl:
  # todo
```

The value of `lib` specifies the path of the standard library, which locates at the `lib` directory of *Osazone* project. And the value of `repl` declares the commands for REPL. It contains a mapping from a command to a string, which should be a function call containing `$input$`. For example, if we want to print the input integer when running `:test 42` in REPL, use the following declaration:

```
repl:
  test: "$input$ :: Int"
```

which will be compiled as a program segment just like `\str -> print ((read str) :: Int)`. Note that you should never use `q`, `quit`, `h` and `help` as the command name. For Num, we will define two commands: `eval` for evaluation and `show` to print input expression. We will provide these REPL declarations later.

osa *We require that the name of AST module must be* `Lang`*, and that of semantics must be* `Eval`*.*

The first progress. The abstract syntax of a language is defined as a data structure. In language definition, we call the type *program sort* like expressions, values and types; and we call the constructors of the type *language constructs*. Let us begin with integers and their operations, as expressions. Create a file named `Lang.osa` in `src`, and write the following code.

```
module Lang where

data Exp              -- the program sort
  = EInt Int          -- the language constructs
  | EAdd Exp Exp
  | ENeg Exp
```

To define the semantics of this language, we just need to write the big-step semantics using *Osazone*'s syntax. Create a file named `Eval.osa` with the following code.

```
module Eval where

import Lang
import Meta.Monad.Trans

eval for Exp :: Int
eval (EInt i) = i
eval (EAdd e1 e2) =
  let i1 = eval e1
      i2 = eval e2
  in  i1 + i2
eval (ENeg e) =
  let i = eval e
  in  0 - i
```

Here `eval` has a different signature from general function signatures. We use such declaration to distinguish it is a semantics or a meta-function of the language. The type after `for` is the input type, or the type to be matched; and the type after `::` is the output type. The other parts have no difference from Haskell.

osa *We use* `0-i` *here because our parser doesn't support negation currently, too bad.*

osa *Please always import* `Meta.Monad.Trans` *in* `Eval` *because our compiler is not clever enough.*

And now let's back to `language.yaml`. For a given parsed string, we just want it be evaluated. We can also define a command to show the input expression directly.

```
repl:
  show: "$input$ :: Exp"
  eval: "ll (rr (eval $input$))"
```

To invoke a semantics definition like `eval`, include the call with `ll` and `rr`. It is a limitation of our current system.

And now, we have finished a simplest definition of Num. Build and compile it.

```
Osazone$ stack run build ./examples/Host/Num
Osazone$ cd ./examples/Host/Num/build
build$ stack ghc Main.hs -Wno-overlapping-patterns
build$ ./Main
```

Try to run `:show EAdd (EInt 1) (EInt 2)`, `:eval EAdd (EInt 1) (EInt 2)` and `:quit` in REPL.

Side effects and meta-functions. We would like to support mutable variables in NUM, which requires us to add side effects in semantics. Before that, we update the syntax of NUM and insert new language constructs into `Exp`.

```
import Meta.Identifier

data Exp
    ...
  | EVar Id
  | EAsgn Id Exp   -- assignment
  | ESeq Exp Exp
```

Osazone uses monads to capture side effects. In particular, for NUM, we need a global state to record the value of each variable, and a state monad is introduced.

```
module Eval where

import Data.Map
import Meta.Monad.Trans
import Meta.Monad.State
import Meta.Semantics

type St = Map Id Int

eval for Exp :: Int
  monad State St
  as S
```

Note the new signature of `eval`. We append an effect declaration to `eval` making it impure now, and name the effect after `S`. If you are familiar with Haskell, you may have decided to replace the original semantics definitions with do-notations. Fortunately, however, this is not needed in *Osazone* — the original semantics don't require any change, we just need to add the semantics for new constructs. That is what we call a *monad extension*.

In order to define the evaluation rules of variable assignment and reading, we need methods for manipulating side effects. Meta-functions work for this.

Add the following function definitions to the end of `Eval.osa`.

```
[#monadic]
insertVar :: Id -> Int -> S ()
insertVar x i =
  let st = get
      st' = insert x i st
      _ = put
  in  ()

[#monadic]
readVar :: Id -> S ()
readVar x =
  let st = get
  in  st ! x
```

Let's have a more detailed look at the definitions. Each function has a similar definition as Haskell, except that we don't use the do-notation syntax. How can we do this? The key point is the `[#monadic]` annotation, which is used to specify that this function has side effects.

Another annotation is `pure` which specifies that the meta-function has no side effects. Each meta-function must be defined with one of these two annotations.

Next up it is the concrete definition. There are `get` and `put` introduced by state monad. Both of them are monadic functions. And `insert` and `(!)` introduced by `Data.Map` are pure functions.

Based on these two functions, we can define the evaluation rules of `EVar` and `EAsgn`, as well as `ESeq`.

```
eval (EVar x) = readVar x
eval (EAsgn x e) =
  let i = eval e
      _ = put e
  in  i
eval (ESeq e1 e2) =
  let _ = eval e1
  in  eval e2
```

The configuration file after introducing side effects. Let us come back to the configuration of REPL, where we have to specify the initial state. Add these information between `ll` and `rr`. We haven't provided a more user-friendly interface temporarily. Also, because `Data.Map.empty` is used in the invocation, we need to add `Data.Map` to the `extra` key, to specify that the *main* module of interpreter should import module `Data.Map`.

```yaml
repl:
  show: "$input$ :: Exp"
  eval: "ll (evalStateT (rr (eval $input$)) empty)"

extra:
  - Data.Map
```

Now build and compile the interpreter again and try to write some programs of NUM.

5.2. Define a DSL using Syntactic Sugars

It's time to define a DSL based on our NUM. Here we just extend the language with some more user-friendly constructs: `x - y`, `x += y`. A DSL definition contains the following parts:

- `lifting.yaml`: configurations

- `sugars/`: extension definitions
  - <module>.ext

Configuration file. Let's write the following declarations in `lifting.yaml`.

```yaml
name:   Numex
core:   # path to the Num language
extension:
  - Numex
```

The first line specifies the name of the language. The second line delegates to the host language, i.e., the path to the Num we just defined. Last but not least, the last line describes a list of extension modules used. These modules will be passed one by one.

Extension file. An extension file consists of a set of sugars and filters. Let's create `sugars/Numex.ext` and define these syntactic sugars directly.

```
module Numex where

sugar newconstructs where

  ESub Exp Exp :: Exp
  ESub e1 e2 = EAdd e1 (ENeg e2)

  EAddAsgn Id Exp :: Exp
  EAddAsgn x e = EAsgn x (EAdd (EVar x) e)
```

The first line of each sugar definition is the signature, consisting of the program sorts of its parameters and itself. For example, for `ESub`, it will be added as a construct of `Exp` in the lifted language, i.e., `data Exp = .. | ESub Exp Exp`. And the second line is the desugaring rule.

Try to lift the language using stack run lift <path> -- --build and compile the interpreter. Write some programs with `ESub` and `EAddAsgn`.

So far, we have been extending the host language. Sometimes, we want to *remove* some host languages, making them invalid to DSL users. We also provide a tool for such purpose. For instance, if we disallow the appearance of `EAsgn` in the Numex, we just need to append the following declarations in `Numex.ext`:

```
filter where
  use  newconstructs (..)
  use  host (..)
  hide host (EAsgn)
```

When the system meets a filter, it begins to record the constructs that will be retained in DSL, getting started with an empty list. Then each filters will be processed one by one. `use` will insert constructs to the list and `hide` will delete some of them. They are followed by a name, which can be `host` or the extension name of sugars; and a list of constructs, where `(..)` means all the constructs in the scope.

Then lift the language and check `src/Lang.osa`, `EAsgn` is not a member of `Exp` any more.

5.3. File Structure

We will roughly explain the structure of the code to provide reference for future researchers to modify the code. We use `source` to demonstrate it is the source code of the *Osazone* library (i.e. a part of the stack project).

- `app/`: source the `Main` module
- `docs/`: documentation, source code of this file
- `examples/`: examples of the languages
  - `Host/`: host languages implemented
  - `DSLs/`: DSL implemented
- `lib/`: the standard library of *Osazone*, providing some commonly used data types and functions
- `src/`: source source code of *Osazone* library
  - `Config/`: interfaces of file IO
  - `Language/`: definition of *Osazone* meta-languages
  - `Lifting/`: the semantics-lifting algorithm
  - `Target/`: the interpreter generator
  - `Utils/`: utilities
- `test/`: source tests (There's no tests actually.)
- Other files: source stack configurations of *Osazone* library

5.4. Complete List of Command-line Parameters

All the commands should be started with `stack run` or `Osazone-exe` (if installed). We will omit it in the following list. Options passed to `stack run` need an extra `--`. For example, `stack run lift <path> -- --build` or `Osazone-exe lift <path> --build`.

Main Commands

`build <path>`

Build the language in `<path>`, generating a set of Haskell code to `<path>/build`.

`lift <path> [--build]`

Lift the semantics of DSL in `<path>`, generating the definition of the DSL. The `--build` option is to build the definition after lifting. It has the same behavior as running `lift` followed by running `build`. Note that to pass the option, stack requires an extra `--` in command, i.e., `stack run lift <path> -- --build`.

`version`

Show the version information of *Osazone* system.

Review Snippets

`review crazy-build [--host|--dsl]`

Build all the pre-defined languages. Note that to pass the option, stack requires an extra `--` in command, i.e., `stack run review crazy-build -- --host`.

`review clean [--host|--dsl]`

Clean all the generated files of pre-defined languages. Note that to pass the option, stack requires an extra `--` in command, i.e., `stack run review clean -- --host`.

`review compile [--host|--dsl]`

Compile all the generated Haskell codes and generate interpreters for languages. Its behavior is to enter the `build/` path of languages and execute `stack ghc Main.hs`. If you use `stack run` to execute this command, it will cause conflicts in the use of `stack.yaml` and may result in compilation failures.

Utilities

```
util parse scan|layout|cst|ast <module-path>
```

This command is designed to test the parser or check the parsing result. The `<module-path>` should refer to an individual `.osa` module, like `./examples/Host/LC/src/Lang.osa`.

```
util parse scan|layout|ext <extension-file-path>
```

This command is designed to test the parser or check the parsing result. The `<extension-file-path>` should refer to an individual `.ext` module, like `./examples/DSLs/Bool/sugars/Bool.ext`.

```
util lang info|modules <path>
```

This command is used to check the basic information of a language. The `info` command will show version of the language, and a list of modules imported by the language. The `modules` command is more detailed, providing concrete dependencies on modules, as well as imported Haskell modules.