Final Project of Compiler Class

Yanchong Peng, Hongwei Xi

{ycpeng, hwxi}@bu.edu

1 Abstract

Program compilation is a popular and indispensable topic in the programming language field. This paper introduces a regular compilation procedure that an ATS-like language is complied to LAMBDA, A-normal form, and C language in order. Moreover, we developed an interpreter and type-checker for LAMBDA.

2 Introduction

One exciting thing about lambda calculus is that we can represent almost anything as functions, such as numbers and strings. Moreover, it is a universally popular model to simulate Turing Machine.(Wikipedia 2022) Consequently, compiling to LAMBDA term will be a necessary and beneficial in-progress design for a language to be compiled to other favored language, such as C.

While every code block and data structure is translated into a syntax tree of LAMBDA, it still preserves linguistic information that is proper for human interpretation but abstract for machine processing. At this crossroad, A-normalization undertakes the task of detaching programming language from linguistic complexity.(Might 2022) A-normal form partitions programming expressions into atomic expressions and complex expressions. The former is represented as *t2box* type, while the latter as *t2ins* type in our project. Due to such simplicity, the expressions are much easier to be interpreted.

C is a classical and the most popular programming language. After translating expressions into A-normal, we constructed an emitter to print out the corresponding C code. Because C doesn't support nested functions, we made several rules to interpret A-normal expressions and lift the nested function to a global one.

We also developed an interpreter and type-checker. The input of both of them is the LAMBDA term. The interpreter directly reads what every term means and outputs the expected results, like what a compiled program should do. The type-checker checks if the types of LAMBDA terms are appropriate. If passed, the type-checker will output a "Type Checking Pass!!!" message. If not, the terminal will raise an error referring to the bug location in the type-checker. This project doesn't include providing users with type error references toward the users' test file. This feature is easy and will be implemented in the future.

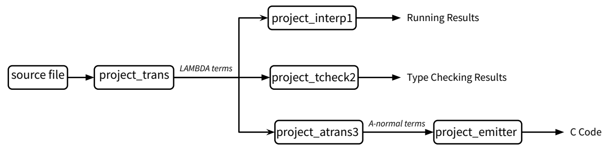

Figure 1: The structure of our compiler. Firstly, project_trans translates the source file into LAMBDA terms, which are the input of project_interp1, project_tcheck2, and project_atrans3. Then project_atrans3 feeds project_emitter with its output A-normal terms. Finally, project_emitter outputs the corresponding C code.

3 Design

3.1 Compiler Structure

Our compiler is composed mainly of

project.sats	*project_trans.dats*
project_interp1.dats	*project_tcheck2.dats*
project_atrans3.dats	*project_emiter.dats*

Figure1 demonstrates the workflow of this compiler.

project.sats defines essential data types of LAMBDA terms, LAMBDA types, and A-normal terms.

project_trans.dats contains the parser for translating ATS-like language into LAMBDA. This file is provided by (Xi 2022). We will not explain this file in the paper.

project_interp1.dats implements the interpreter of LAMBDA. Running it results in the intended output results.

project_tcheck2.dats type check the input LAMBDA.

project_atrans3.dats translate LAMBDA into A-normal form.

project_emiter.dats emits C code from A-normal form.

3.2 Support Operations

Our compiler supports operations as follows:

$+, -, *, /, \%, <, >, =, \leq, \geq, !=$
show, showval, print, list_nil(), list_cons(x,list(x))
list_nilq(list(x)), list_consq(list(x))
list_uncons1(list(x)), list_uncons2(list(x))
strm_nil(), strm_cons(x,lazy(strm(x)))
strm_nilq(strm(x)), strm_consq(strm(x))

t1erm	T1Mnil	()
	T1Mint	int
	T1Mbtf	bool
	T1Mstr	string
	T1Mvar	t1var
	T1Mlam	(t1var, t1ypeopt, t1erm)
	T1Mapp	(t1erm, t1erm)
	T1Mopr	(t1opr, t1ermlst)
	T1Mif0	(t1erm, t1erm, t1ermopt)
	T1Mfst	(t1erm)
	T1Msnd	(t1erm)
	T1Mtup	(t1erm, t1erm)
	T1Mseq	(t1ermlst)
	T1Mfix	(t1var, t1var, t1ypeopt, t1erm, t1ypeopt)
	T1Mlazy	(t1erm)
	T1Manno	(t1erm, t1ype)
	T1Mnone	(d1exp(*unsupported*))
t1dcl	T1DCLbind	(t1var, t1erm)
t1val	T1Vnil	()
	T1Vint	int
	T1Vbtf	bool
	T1Vstr	string
	T1Vtup	(t1val, t1val)
	T1Vlam	(t1erm, t1env)
	T1Vfix	(t1erm, t1env)
	T1Vlazy	(t1erm, t1env)
	T1Vcons	(int, t1valist)
env	t1env	mylist(@(t1var, t1val))
list	t1dclist	mylist(t1dcl)
	t1ermlst	mylist(t1erm)
	t1ermopt	myoptn(t1erm)
	t1valist	mylist(t1val)
	t1var	string
	t1opr	string

Table 1: LAMBDA terms datatype

strm_uncons1(strm(x)), strm_uncons2(strm(x)), $eval

3.3 Interpreter

LAMBDA terms We define LAMBDA terms as Table1.

Declaration Interpretation Declaration list is a sequence of declarations of variables in syntax. We use the function *t1dclist_interp1()* to interpret the declaration list. The inputs of this function are a t1dclist and t1env. If the t1dclist is nil, then the function returns the environment t1env. If not, the function interprets the first declaration (t1var, t1erm) in the list, appends t1var and its t1val to the t1env, and interprets the left declaration list with the newest t1env.

T1Mseq(tmlst) Interpretation We interpret every t1erm in the tmlst in order, and return the t1val of the last t1erm.

T1Mlet(dcls, tm1) Interpretation Firstly, we use *t1dclist_interp1* to interpret dcls and obtain a new t1env with the information from dcls. Then, with the new t1env, tm1 is interpreted.

T1Mlazy(tm1) Interpretation The procedure of dealing with T1Mlazy(tm1) is uncomplicated. T1Mlazy can be seen as the counterpart of $lazy{} operation in the functional language. Thus, there's no need to process tm1 so early. The interpretation will be executed in lazy evaluation operations. Consequently, T1Mlazy(tm1) is interpreted just as T1Vlazy(tm1, t1env).

Operation "print" Interpretation "print" operation supports three data types: integer, boolean, string, and list. When processing list value "T1Vcons(tag, tvlst)", the interpreter will first check the "tag" value. If $tag = 0$, then the program prints "list_nil()" to demonstrate nil list value. If not, we will call a cyclically printing function *interp_print_lst(tvlst)* to print the list like "(_, _, ...)".

List Operations Interpretation All the list operations will return either a boolean or a T1Vcons(tag, t1valist). $tag = 0$ represents the list is nil. $tag = 1$ means the list is not nil, and the later procedure can "safely" interpret t1valist.

- **list_nil()** returns T1Vcons(0, mylist_nil())
- **list_cons(x,list(x))** appends x to list(x) and returns the list result in T1Vcons with $tag = 1$.
- **list_nilq(list(x))** checks if $tag = 0$. If so, it returns T1Vbtf(true), otherwise, T1Vbtf(false).
- **list_consq(list(x))** checks if $tag = 1$. If so, it returns T1Vbtf(true); otherwise, T1Vbtf(false).
- **list_uncons1(list(x))** uses the function "mylist_head()" to return the first element in mylist_cons().
- **list_uncons2(list(x))** uses the function "mylist_tail()" to obtain the second element tv1_tail in mylist_cons(). If the list is nil, the interpreter returns T1Vcons(0, mylist_nil()); otherwise, T1Vcons(1, tv1_tail).

Lazy Operations Interpretation As we discussed before, T1Vlazy() can be seen as the counterpart of $lazy{} in the functional language. Meanwhile, T1Vcons() is a great "substitution" for strm(). Thus, the interpretation of lazy operations focuses on these two t1val types.

- **strm_nil()** returns T1Vcons(0, mylist_nil())
- **strm_cons(x,lazy(strm(x)))** appends x to strm(x) and returns the list result in T1Vcons with $tag = 1$.
- **strm_nilq(strm(x))** checks if $tag = 0$. If so, it returns T1Vbtf(true), otherwise, T1Vbtf(false).
- **strm_consq(strm(x))** checks if $tag = 1$. If so, it returns T1Vbtf(true); otherwise, T1Vbtf(false).
- **strm_uncons1(strm(x))** uses the function "mylist_head()" to return the first element in strm().
- **strm_uncons2(strm(x))** uses the function "mylist_tail()" to obtain the second element tv1_tail in strm(). It should be noticed that the data structure of tv1_tail is mylist_cons(T1Vlazy(),_). Thus, T1Vlazy() will be returned.
- **$eval** extracts tm1 from T1Vlazy(). Then it returns the result of the interpretation of tm1.

t1ype	T1Pnil	()
	T1Pbas	tpbas
	T1Pext	tpVar
	T1Pfun	(t1ype, t1ype)
	T1Ptup	(t1ype, t1ype)
	T1Plist	(t1ype)
	T1Plazy	(t1ype)
	T1Pstrm	(t1ype)
	T1Pnone	(s1exp(*unsupported*))
env	t1ctx	mylist(@(t1var, t1ype))
list	t1ypelst	mylist(t1ype)
	tpbas	string
	tpVar	ref(myoptn(t1ype))

Table 2: LAMBDA types definition

Running Instruction In *project.dats*, after obtaining LAMBDA syntax tree **t1ds**, we pass it to the function *t1dclist_interp0()*. The running output will appear on the terminal following the "process_fpath: t1ds = ..." output.

3.4 Type Checker

LAMBDA types We define LAMBDA types as Table2.

Declaration Type Checking We use the function *t1dclist_oftype1()* to type-check the declaration list. The inputs of this function are a t1dclist and t1ctx. If the t1dclist is nil, then the function returns the environment t1ctx. If not, the function type-checks the first declaration (t1var, t1erm) in the list, appends t1var and its t1ype to the t1ctx, and type-checks the left declaration list with the newest t1ctx.

T1Mlam(xnm, topt, tm1) Type Checking There're two conditions for topt: the type of argument is given (myoptn_cons(tp1)) or not (myoptn_nil()). If the user doesn't provide the argument's type, then the type-checker will create a new existential variable type T1Pext for tp1, obtain tm1's type tp2 with the newest t1ctx including tp1, and return T1Pfun(tp1, tp2). Otherwise, the type-checker directly gets tp2 with the t1ctx including the given tp1, and returns T1Pfun(tp1, tp2).

T1Mfix(fnm, xnm, tpo_arg, tm1, tpo_res) Type Checking There're four conditions in T1Mfix:

- The type of argument tpo_arg is **not** given. The type of result tpo_res is **not** given.
- The type of argument tpo_arg is given. The type of result tpo_res is **not** given.
- The type of argument tpo_arg is **not** given. The type of result tpo_res is given.
- The type of argument tpo_arg is given. The type of result tpo_res is given.

It should be mentioned that "tpo_arg" tp1 is actually a function type "T1Pfun(tp11, tp12)", where tp11 refers to the argument x's type and tp12 refers to lambda function's output type. Meanwhile, "tpo_res" tp2 is the function's output type as well.

The overall type-checking design for every condition is the same. If the argument's type is not given, then the type-checker will create a T1Pfun(_, tp_res) for tp1. After obtaining the type tp2 of tm1, we examine if tp2 matches tp_res. If not, an unmatched error will be raised; otherwise, tp1 will be returned.

T1Mif0(tm1, tm2, tmopt) Type Checking The type-checker firstly obtains tp1 and tp2 correspondingly from tm1 and tm2. Then, it follows rules below to examine the type of each term:

- Check if tp1 matches T1Pbool.
- If tm3 in tmopt is **not** given, directly return tp2.
- If tm3 is given, check if tp2 matches tp3. If passed, return tp2; otherwise, raise an error.

T1Mfst(tm) Type Checking The type-checker gets tm's type tp firstly. Then, it checks if tp matches T1Ptup or T1Pext. If passed, there will be two separate roads. If tp is T1Ptup(tp1, tp2), then the type-checker returns tp1. If T1Pext(_), then it returns a new existential type T1Pext.

T1Mseq(tmlst) Type Checking The type-checker checks the type of each term and returns the type of the last term.

T1Mlet(dcls, tm1) Type Checking With the function *t1dclist_oftype1()*, the type-checker checks the type of each declaration and obtains a new t1ctx including every declaration's type. Using the new t1ctx, we type-check tm1 and return its type.

T1Mlazy(tm1) Type Checking After type-checking tm1 and getting its type tp1, the type-checker returns T1Plazy(tp1).

T1Manno(tm1, tp1) Type Checking After type-checking tm1 and getting its type tp2, the type-checker examines if tp1 matches tp2. If so, return tp1.

Operation "print" Type Checking Because "print" supports integer, boolean, string, and list, the type-checking will pass only when the type of the term is T1Pint, T1Pbool, T1Pstring, or T1Plist. Moreover, this operation returns T1Pnil.

List Operations Type Checking

- **list_nil()** The type-checker creates a new existential type T1Pext as tp1, and returns T1Plist(tp1).
- **list_cons(x,list(x))** After obtaining tp1 and tp2 from two parameters, the type-checker firstly creates a new existential type T1Pext as tp3. Then, we check if tp2 matches T1Plist(tp3). Next, we check if tp1 matches tp3. Finally, the type-checker outputs T1Plist(tp1).
- **list_nilq(list(x))** The type-checker examines if tp1 of the parameter matches T1Plist(_). If so, it returns T1Pbool.
- **list_consq(list(x))** The type-checker examines if tp1 of the parameter matches T1Plist(_). If so, it returns T1Pbool.
- **list_uncons1(list(x))** The type-checker examines if tp1 of the parameter matches T1Plist(tp2). If so, it returns tp2.
- **list_uncons2(list(x))** The type-checker examines if tp1 of the parameter matches T1Plist(tp2). If so, it returns tp1.

Lazy Operations Type Checking

- **strm_nil()** The type-checker creates a new existential type T1Pext as tp1, and returns T1Pstrm(tp1).
- **strm_cons(x,lazy(strm(x)))** After obtaining tp1 and tp2 from two parameters, the type-checker firstly creates a new existential type T1Pext as tp3. Then, we check if tp2 matches T1Plazy(T1Pstrm(tp3)). Next, we check if tp1 matches tp3. Finally, the type-checker outputs T1Pstrm(tp1).
- **strm_nilq(strm(x))** The type-checker examines if tp1 of the parameter matches T1Pstrm(_). If so, it returns T1Pbool.
- **strm_consq(strm(x))** The type-checker examines if tp1 of the parameter matches T1Pstrm(_). If so, it returns T1Pbool.
- **strm_uncons1(strm(x))** The type-checker creates a new existential type T1Pext as tp2, and examines if tp1 of the parameter matches T1Pstrm(tp2). If so, it returns tp2.
- **strm_uncons2(strm(x))** The type-checker creates a new existential type T1Pext as tp2, and examines if tp1 of the parameter matches T1Pstrm(tp2). If so, it returns T1Plazy(T1Pstrm(tp2)).
- **$eval** The type-checker creates a new existential type T1Pext as tp2, and examines if tp1 of the parameter matches T1Plazy(T1Pstrm(tp2)). If so, it returns T1Pstrm(tp2).

Running Instruction In *project.dats*, after obtaining LAMBDA syntax tree **t1ds**, we pass it to the function *t1dclist_oftype0()*. If **t1ds** passes the type-checking, then the terminal will output ”Type Checking Pass!!!”.

3.5 A-normal Form Translator (atrans)

A-normal Form Data Types We define A-normal form as Table3.

Closure Conversion Because C doesn't support nested functions, the translator and emitter must lift them to global functions.

However, for some inner functions, their definition relies on the variables of the outside function they nest. For example, in the pseudocode listing1, the inner function $lam(r)$ needs tv1 as its variable, but tv1 may vary due to different calls for func1. Thus, we need to create a new environment individually to convey the variables that inner functions need.

This is the reason why we add a new t2ins type: ”T2Icfp(t2box, t2env)”. t2box is T2Vlam(t2cmp of function body), and t2env is the environment including the outside variables.

Declaration Translation We use the function *t1dclist_atrans1()* to translate the declaration list. There're three inputs:

dcls: Declaration list containing unprocessed LAMBDA term
dcls_atrans: A-normal expression list conveying translated declaration expressions

t2cmp	T2CMP	(t2bndlst, t2box)
t2box	T2Vnil	()
	T2Vint	int
	T2Vbtf	bool
	T2Vstr	string
	T2Vvar	t2var
	T2Vfix	string
	T2Varg	(t2arg)
	T2Vreg	(t2reg)
	T2Vlam	(t2cmp)
t2ins	T2Imov	(t2box)
	T2Ical	(t2box, t2box)
	T2Iopr	(t1opr, t2boxlst)
	T2Ifst	(t2box)
	T2Isnd	(t2box)
	T2Itup	(t2box, t2box)
	T2Iif0	(t2box, t2bndlst, t2bndlst)
	T2Ilet	(t2env)
	T2Icfp	(t2box, t2env)
t2bnd	T2BND	(t2reg, t2ins)
env	t2env	mylist(@(t1var, t2cmp))
	t2arg	int
	t2reg	int
	t2var	string
list	t2boxlst	mylist(t2box)
	t2bndlst	mylist(t2bnd)
	t2inslst	mylist(t2ins)

Table 3: A-normal Form Data Types

Listing 1: Example pseudocode for the demonstration of closure conversion

```
fun func1(tuple(arg1, arg2))
{
var tv1 = ...
var tv2 = ...

return func1(tuple(1, lam(r) => (tv1 +
    1)))
}
```

t2env: Environment containing variables and their t2cmp

For every declaration (t1var, tm1) in the list, the translator translates tm1 into t2cmp first and appends this (t1var, t2cmp) to the dcls_atrans. Then, if t1var is a variable's name, (t1var, t2cmp_val(T2Vvar(t1var))) is appended to t2env, representing the translation system already has the information of the corresponding t1var for later translation. If t1var is a function's name, then (t1var, t2cmp_val(T2Vfix(t1var))) is appended to t2env. Note that the function *t2cmp_val(t2box)* creates a t2cmp for t2box with empty t2bndlst.

T1Mtup(t1m1, t1m2) Translation

- **Normal Translation** Normally, when translating a tuple term, after obtaining the corresponding t2bndlst (bds1, bds2) and t2box (t2x1, t2x2) of two terms t1m1 and t1m2, our translator creates a new t2bndlst named tbnd as T2BND(new_treg, T2Itup(t2x1, t2x2)). Then, the translator returns T2CMP(bds1+bds2+tbnd, T2Vreg(new_treg)).

- **Closure Conversion Modification** Due to the introduction of the closure conversion feature, there are some modifications for the translation of T1Mtup.

  One of the tuple's activities is to convey the multiple arguments of functions. One of the rules of our compiled program is that if users want to implement closure conversion, they must put the inner function in the second place of the argument's tuple. Consequently, the second element of T1Mtup may be T1Mlam and we need to translate it into T2Icfp.

  The translation of T1Mlam for closure conversion here is much simple.

  (1) We regularly translate t1m2 and get T2BND(empty_t2bndlst, T2Vlam(t2cmp_of_body)).

  (2) By interpreting t2cmp_of_body, we could get its t2bndlst "lam_bds2" containing all the variables T2Vvar()s of the inner function.

  (3) The function *find_cfp_env()* can find all dependent variables in the inner function by fed with lam_bds2 and the global environment t2env. The output of this function is represented as cfp_env.

  (4) Establish t2bnd named tbnd_cfp by T2BND(new_treg_cfp, T2Icfp(T2Vlam(t2cmp_of_body), cfp_env)).

  (5) Finally, we can add tbnd_cfp to tuple's t2bndlst, and select T2Vreg(new_treg_cfp) as the second t2box of T2Itup.

T1Mlam(targ, topt, t1m1) Translation

- **Normal Translation** The translator creates a new argument t2box: T2Varg(0) and its t2cmp: t2cmp_val(T2Varg(0)). After adding the argument pair (t1var, t2cmp) to the environment t2env, we translate the function's body t1m1 with the newest t2env and get the result t2cmp named "body". Finally, the translator returns T2CMP(mylist_nil(), T2Vlam(body)).
- **Closure Conversion Modification** Most parts of this version are the same as normal translation, except what we return. Figure2 refers to the data structure of the returned t2cmp. The compiled program only allows users to use the nested function, having outside variables, in the argument of functions, such as Listing1. Consequently, for other T1Mlams, there's no need to include outside variables in their environment. That's the reason why cfp_env here is nil.

T1Mif0(tm1, tm2, tmopt) Translation For convenience, we assume that T1Mif0 doesn't have tm3 in tmopt. Only the translation of tm1 and tm2 will be discussed. Processing tm3 is similar to tm2.

The overall procedure can be explained as follows:

(1) The translator translates tm1 and tm2 into two t2cmps: T2CMP(bds1, t2x1) and T2CMP(bds2, t2x2).

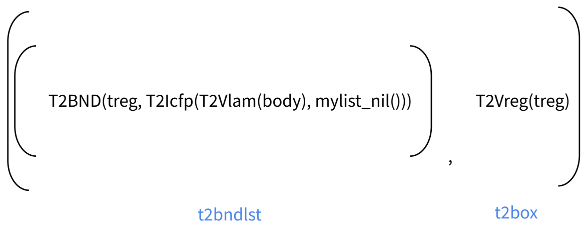

Figure 2: The data structure of the A-normal expression of T1Mlam in closure conversion

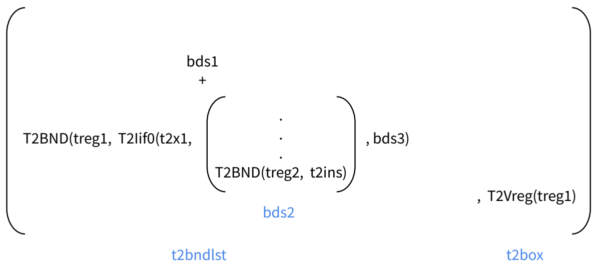

Figure 3: The final A-normal expression of T1Mif0. treg1 and treg2 should be the same.

(2) We check if t2x2 matches T2Vreg(treg). If so, the translator doesn't do anything in this step. If not, that means $< then >$ block in T1Mif0 only returns a value without any other operations, such as

$$if(condition)then\{8\}$$

When this happens, the translator will modify tm2's t2cmp to T2CMP(list(T2BND(treg, T2Imov(t2x2))), T2Vreg(treg)). The intention of T2Imov is to convey t2x2 in t2cmp with empty t2bndlst. After this processing, we ensure that bds2 and bds3 are both not empty.

(3) Because the returned last treg numbers from tm2 should be the same as T1Mif0, we need to unify treg2 in the last element of bds2 with treg1 of T2Iif0. This procedure is implemented in the function *reg_if_if()*. Figure3 refers to the translated result.

T1Mfst(tm) Translation The translator translates tm to obtain T2CMP(bds,t2x). Then, we create T1Mfst's bnd: T2BND(new_treg,T2Ifst(t2x)). Finally, the translator returns T2CMP(bds+tbnd,T2Vreg(new_treg)).

T1Mseq(tmlst) Translation We translate every term into A-normal form, append each bnd together as a whole bndlst, and return T2CMP(bndlst, t2x), where t2x is the t2box of the last term.

T1Mfix(fnm, xnm, tpo_arg, tm1, tpo_res) Translation The procedure is almost the same as T1Mlam, except constructing new t2env. In this part, we add not only (xnm, t2cmp_val(T2Varg(0))) to t2env, but also (fnm, t2cmp_val(T2Vfix(fnm))).

T1Mlet(dcls, tm1) Translation By applying the function *t1dclist_atrans1()* to dcls, the translator obtains translated dcls1 and the newest env1. Then, we create T1Mlet's bnd named tbnd: T2BND(new_treg,T2Ilet(dcls1)). Next, the translator translates tm1 with env1 and gets T2CMP(bndl1,t2x1). Finally, T2CMP(tbnd+bndl1, t2x1) is returned.

T1Mlazy(tm1) Translation Because we construct a novel data structure in the emitter and *runtime.h*, we don't do much in this part and just return the translated tm1.

The brief introduction is that, in C, we only have one data structure "lamval1_strm" for lazy evaluation. Except for the tag, this structure has three data entries: *ele*, *func*, and *args*. If in functional language there is a line of code

$$strm\_cons(pair, generate(((0, sum + 1), sum + 1)))$$

then *ele* represents the next generated output data "pair", *func* represents the generator "generate", and *args* represents arguments "((0, sum+1), sum+1))".

Thus, the whole lamval1_strm can be seen as the representation of $lazy() without evaluation. After evaluation, a tuple data lamval1_tup(ele, lamval1_strm) will be returned. This tuple can be logically regarded as the counterpart of strm().

More detail will be discussed in the **runtime.h** part.

Operation "strm_cons" Translation Almost all the operations' translations follow the same pattern, except "strm_cons".

The translator primarily gets t2cmps of tm1 and tm2: T2CMP(bds1,t2x1) and T2CMP(bds2,t2x2). As we discussed in **T1Mlazy(tm1) Translation**, we should detach the generator from its argument. "strm_cons" in output C code will not have two but three arguments. With the function *find_args()*, the translator finds the individual generator and argument from T2CMP(bds2,t2x2), separates them, and merges them with t2x1 into t2boxlst. That is, the resulting t2boxlst composes of three t2boxs: t2x1, one for the generator, another for the argument. Finally, the translator returns "strm_cons" along with the new t2boxlst.

Running Instruction In *project.dats*, after obtaining LAMBDA syntax tree **t1ds**, we pass it to the function *t1dclist_atrans0()* to get A-normal expression **t2trans**. Then, we use *println!()* to print **t2trans**.

3.6 Emitter

Overall Design All the data structure and operation functions have been implemented in *runtime.h*, so we don't need to emit any basic code block in the target C code. The whole procedure is described in Figure4.

Basic Data Types For convenience, all the basic data types have been implemented in *runtime.h*. This section will introduce some important types.

- **lamval1_lst** refers to list2. For list(x,list(x)) in the functional language, *ele* corresponds to x, and *lst* corresponds to list(x). In C, *lst* refers to another lamval1_lst.

Listing 2: lamval1_lst data structure

```
typedef
struct{
  int tag;
  lamval1 ele;
  lamval1 lst;
} lamval0_lst;

typedef lamval0_lst *lamval1_lst;
```

Listing 3: lamval1_strm data structure

```
typedef
struct{
  int tag;
  lamval1 ele;
  lamval1 (*func)(lamval1 arg);
  lamval1 arg;
} lamval0_strm;

typedef lamval0_strm *lamval1_strm;
```

- **lamval1_strm** refers to list3. As we discussed in **T1Mlazy(tm1) Translation**, lamval1_strm is composed of three data entries. *ele* corresponds to the next generated data, *func* corresponds to a function pointer referring to the generator, and *arg* represents arguments of the generator.

temit3_function() This function handles all the functions' declaration and implementation. The procedure can be described as follows in order:

- **find_funs_in_dcls()** Given A-normal syntax tree *t2atrans*, this function searches the entire tree to find all the functions including nested functions, and returns a function list **funs**. The type of **funs** is *t3funlst of mylist(@(t1var, t2cmp))*.

  This function may call two additional functions *find_funs_in_cmp()* and *find_funs_in_bndlst()*. The former searches functions in t2cmp, and the latter in t2bndlst.
- **print_funs_dcls()** Given the function list **funs**, this function prints the declaration of each function, such as

$$\begin{gathered} extern \\ lamval1 \\ func1(lamval1 \quad x); \end{gathered}$$

- **print_funs()** Given the function list **funs**, this function prints the implementation of each function, such as

$$\begin{array}{l} lamval1 \\ func1(lamval1 \quad x)\{ \\ < body > \\ \} \end{array}$$

  The $<$ *body* $>$ section has four reference and printing functions. We will talk about them in the next four sections.

t2box	printing result
T2Vint(i)	LAMVAL_int(i)
T2Vbtf(b)	LAMVAL_int(1) or LAMVAL_int(0)
T2Vstr(s)	LAMVAL_str(s)
T2Vvar(v)	v
T2Vfix(f)	f
T2Varg(a)	x
T2Vreg(r)	tmp + r
T2Vlam(_)	()
T2Vnil	()

Table 4: Printing Reference Table

- **collect_regs()** The first thing to print the body is to declare all the tmp (treg) variables. This function collects all the tregs, and returns them in a list.

  Inside *collect_regs()*, we call *collect_regs_from_tins()* to collect tregs from t2ins.

  Because **T2Iif0** and **T2Ilet** have types other than t2box, the collections of these two t2inses are a little more complicated than others. We implement *collect_regs_from_if()* and *collect_regs_from_let()* for this purpose.
- **print_regs()** After collecting regs, we print all the regs like

$$lamval1 \quad tmp1, \quad tmp2, \quad ... \quad ;$$

- **print_bndl()** This function handles the body printing after variables declaration. It prints every t2bnd in t2bndlst. Moreover, the printing methods vary from different t2ins. Every type of t2ins has its own print function. We will talk about them after introducing the overall design.
- **print_return()** This function prints the return sentence.

temit3_main() This function handles the main function of body printing. The procedure can be described as follows in order:

- **main_collect_regs()** This function collects all the tregs in the main body.
- **print_regs_main()** This function declares all the tregs collected from the previous step.
- **print_dcls()** This function prints all the declarations from the main body. The printing format looks like:

$$lamval1 \quad t1var \quad = \quad ... \quad ;$$

print_t2box() This function prints t2box types. The printing reference table is shown in Table4

print_mov() This function prints T2Imov type like:

$$tmp + reg \quad = \quad t2box \quad ;$$

print_cal() This function prints T2Ical type like:

$$tmp + reg \quad = \quad t2box1(\quad t2box2 \quad);$$

print_opr() This function calls *print_opr_detail()* to print T2Iopr type.

print_fst() This function prints T2Ifst type like:

$$tmp + reg \quad = \quad \text{LAMVAL\_fst(} \quad t2box \quad);$$

print_snd() This function prints T2Isnd type like:

$$tmp + reg \quad = \quad \text{LAMVAL\_snd(} \quad t2box \quad);$$

print_tup() This function prints T2Itup type like:

$$tmp + reg \quad = \quad \text{LAMVAL\_tup(} \quad t2box1, \quad t2box2);$$

print_if() This function prints T2Iif0 type like:

$$\begin{array}{l} if(((lamval1\_int)t2box)- > data)\{ \\ \} \\ else\{ \\ \} \end{array}$$

To print the body of the ”then” and ”else” blocks, we call *print_bndl()*.

print_let() This function calls *print_dcls()* to print T2Ilet type.

3.7 runtime.h

The basic data structures such as lamval1_lst and lamval1_strm have been introduced before. This section will discuss the print operation, and some list and lazy evaluation operations in C.

LAMOPR_print() This operation prints integer, string, and list. For the list printing, this function will call *print_list()* to print the list like $[l1, l2, l3]$.

LAMOPR_lst_nil() This operation returns LAMVAL_lst(NULL, NULL).

LAMOPR_lst_cons(lamval1 x, lamval1 y) This operation returns LAMVAL_lst(x, y).

LAMOPR_lst_uncons1(lamval1 x) This operation returns ((lamval1_lst)x)->ele.

LAMOPR_lst_uncons2(lamval1 x) This operation returns ((lamval1_lst)x)->lst.

strm_nil() This operation returns LAMVAL_strm(NULL, NULL, NULL).

strm_cons(lamval1 x, lamval1 (*y)(lamval1 arg), lamval1 z) x represents the data element, y is the function pointer of the generator function, and z is the generator's arguments. This operation returns LAMVAL_strm(x, y, z).

eval(lamval1 x) This operation evaluates the stream. The input x is a LAMVAL_strm(). By fetching the generator pointer *func*, generated element *ele*, and generator's argument *arg* from the input x, this function applies *arg* to *func* and obtains its result *strm* as the next stream. Finally, it outputs a tuple *LAMVAL_tup(ele, strm)* as the evaluated stream.

Listing 4: lamval1_lst data structure

```
typedef
struct{
  int tag;
  lamval1 (*cfp_func)(lamval1 arg);
  lamval1 env[];
} lamval0_cfp;

typedef lamval0_cfp *lamval1_cfp;

lamval1
LAMVAL_cfp(lamval1 cfp_func, lamval1 env
    )
{
  lamval1_cfp tmp0;
  tmp0 = (lamval1_cfp)malloc(sizeof(
      lamval0_cfp));
  tmp0->tag = TAGcfp;
  tmp0->cfp_func = (lamval1 (*)(lamval1)
      )cfp_func;
  memcpy(tmp0->env, &env, sizeof(env));
  return (lamval1)tmp0;
}
```

strm_uncons1(lamval1 x) This operation returns the first element of the evaluated stream, that is, *LAMVAL_fst(x)*.

Closure Conversion Assumption Although we didn't accomplish the closure conversion's implementation in C, we still want to share the data structure we assumed in list4.

cfp_func refers to the function pointer and *env* represents the environment of the correspoing function.

4 Future Works

Although we implement some basic functions of this compiler, there are still multiple features that we can improve.

4.1 Closure Conversion

We only finished the work in A-normal translator. However, the idea seems not to be correct. We plan to dive deeper into the relationship between closure conversion and C language.

4.2 tregs Printing Improvement

Due to the special collection procedure of tregs in our emitter, the declaration of tregs may not be in ascending order. The next work is to correctly order them.

4.3 Dcls Printing Improvement

For now, when printing the variables declaration, our emitter will also output duplicate tregs. For instance, "lamval1 tvar = LAMVAL_int(1);" is reasonable and straightforward. However, our emitter will output "lamval1 tmp0 = LAMVAL_int(1); lamval1 tvar = tmp0;".

4.4 Function Optimization

Our compiler doesn't support directly nested functions such as *func => lam(x)=>lam(y)*. The main reason is that in our emitter, the interpretation of T2Vlam has a restriction for continually T2Vlam nesting.

4.5 Type Checker Optimization

Our compiler doesn't insert error-type messages, so it's much more difficult for users to locate their bugs. The unmatched error also refers to *project_tcheck2.dats* instead of the test file.

5 Conclusion

From this project, we systematically established the procedure of a compiler.

The implementation of the interpreter helps us preliminarily taste how to interpret a LAMBDA syntax tree. The typechecker lets us know the importance of human-friendly programming, and methods to convey type along T1Mlam and T1Mfix.

A-normal form teaches us how to remove complex language features(Might 2022) from the syntax tree, which is essential to consider programming sentences in sequence. Implementing the emitter also teaches us many C features and tricks to lift an inner function to a global one. This class is also a good choice for training programming with good habits.

References

Might, M. 2022. A-Normalization: Why and How (with code). https://matt.might.net/articles/a-normalization/. (Accessed on 12/17/2022).

Wikipedia. 2022. Lambda calculus — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Lambda%20calculus&oldid=1127027221. [Online; accessed 17-December-2022].

Xi, H. 2022. hwxi/CS525-2022-Fall: For teaching BU CAS CS 525: Compiler Design and Implementation. https://github.com/hwxi/CS525-2022-Fall. (Accessed on 12/17/2022).

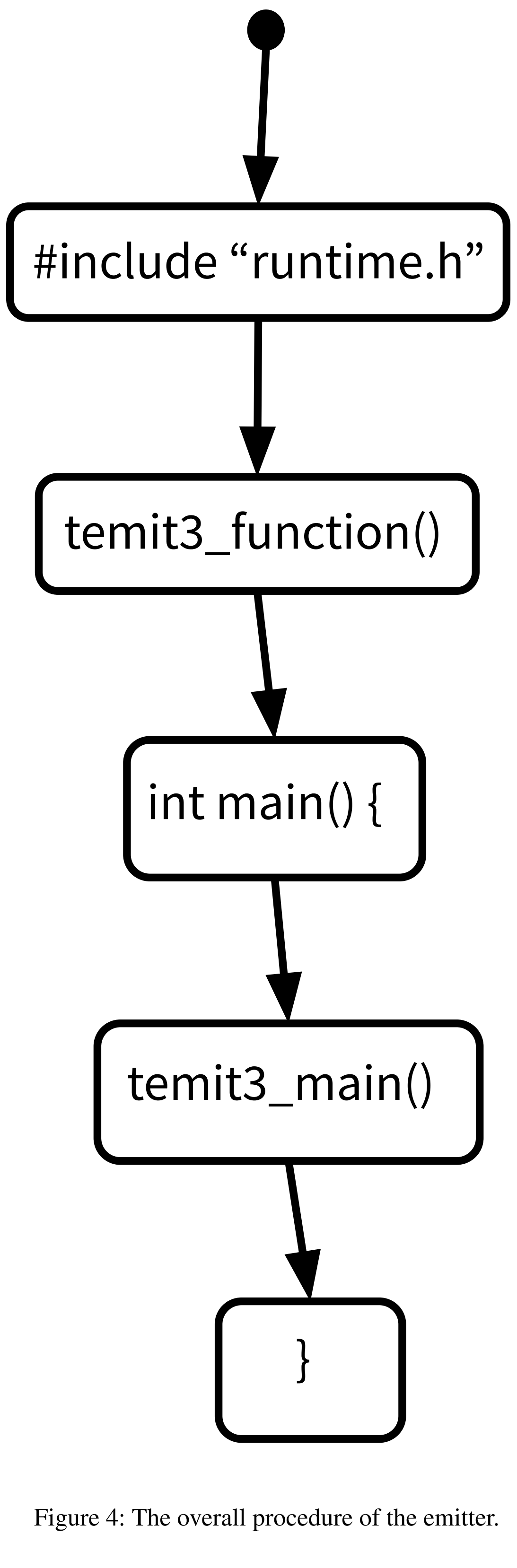

Figure 4: The overall procedure of the emitter.