Unlike Java or Python, flat (unboxed) data structure is commonly used in ATS, which leads to the benefit of a straightforward interaction between ATS and C code. In fact, ATS compiler simply generates C code from ATS files. Once you have a good understanding about how ATS compiler would generate the C code (observing the generated files is a good way to learn), mixing ATS with C code simply downgrades to the handling a pure C project.

In the following, we use “project” to refer to a bunch of files (ATS or C) which can be used to build an executable program. And we don’t tackle anything related to boxed types.

Scenario A.

There is only one ATS file in the project. Function “main” in ATS calls function “add2” in C. The later one calls function “add1” in ATS.

```
/*
* testmain01.dats
*/

%{$  // C code
// int add2(int n)
ats_int_type add2(ats_int_type n)
{
        int x = add1(n);
        x = add1(x);
        return x;
}
%}  // end of C code

extern fun add2 (n: int): int = "add2"

implement main () = let
  val n = 3
  val x = add2 (n)
  val () = printf ("add2(%d) = %d\n", @(n, x))
in end

extern fun add1 (n: int): int = "add1"
implement add1 (n) = n + 1
```

---

1. Enclosing C code

In ATS, the syntax for enclosing C code in dats file is
%{^ (some C code) %}
or
%{$ (some C code) %}
The former and the latter mean that the enclosed C code needs to be put at the top and the bottom of the generated C code, respectively. (Without specifying $ or ^, %{ represents %{$ by default.) In testmain01.dats, we use %{$ to ask the ATS compiler to put the C code to bottom of the generated Code. If we choose to use %{^ instead, then the file should be as follows

---

```
/*
* testmain02.dats
*/

%{^  // C code
extern ats_int_type add1(ats_int_type n);

// int add2(int n)
ats_int_type add2(ats_int_type n)
{
        int x = add1(n);
        x = add1(x);
        return x;
}
%}  // end of C code

extern fun add2 (n: int): int = "add2"

implement main () = let
  val n = 3
  val x = add2 (n)
  val () = printf ("add2(%d) = %d\n", @(n, x))
in end

extern fun add1 (n: int): int = "add1"
implement add1 (n) = n + 1
```

---

todo: ${# for sats (see testCommon14.sats listed below)

2. Function names

When declaring a function in ATS, we can specify the corresponding name in the generated C code. If this function is defined in ATS, then the generated C function would be of that name. For example, if we didn't specify extern fun add1 (n: int): int = "add1" in testmain01.dats, the generated C function would be of the name like "_2home_2grad2_library_2application_2SATS_2testmain_2esats__add", which is difficult to use in C code and also such naming convention is subject to the implementation of ATS compiler.On the other hand, for function defined in C (function "add2" for example), the correct name has to be provided when declaring the function in ATS code so that ATS compiler can generate the appropriate C code whenever the function is invoked in ATS.

3. Type

1) primitive types and boxed types

Unlike assigning names to functions, we cannot force ATS compiler to generate a special name in C for ATS' primitive types and types implemented in ATS (e.g. tuple or datatype). The mapping between primitive types in ATS and their name in C is fixed by ATS compiler. Part of the mapping is listed below.

ATS -> C
int -> ats_int_type
uint -> ats_uint_type
llint -> ats_llint_type
char -> ats_char_type
uchar-> ats_uchar_type
byte -> ats_byte_type
bool -> ats_byte_type
ptr -> ats_ptr_type
void -> ats_void_type.

The rule for forming such mapping is straightforward, it's easy for readers to form up other mappings not listed above.

Any boxed type defined in ATS is mapped to ats_ptr_type in C. And any dependent type is treated as non-dependent during the code generation. The following code demonstrates the mapping of such types. For functions declared in ATS, the implementation in C should of the appropriate type according to the mapping mentioned above.

```
/*
* testmain03.dats
*/

datatype tree (int) =
| Nil(0) | {n1,n2:two} Node(1) of (tree n1, int, tree n2)

viewtypedef Tree = [n:two] tree n

extern fun ref_tree (t: Tree): ref Tree = "ref_tree"


%{$

ats_ptr_type ref_tree (ats_ptr_type t) {
 ats_ptr_type *r;
 r = ats_malloc_gc (sizeof(ats_ptr_type)) ;
 *r =t ;
 return r ;
}

%}
```

Call by reference in function declaration in ATS is mapped to ats_ref_type. The following code shows such usage based on Scenario A.

```
/*
* testmain04.dats
*/

%{$  // C code
// int add2(int n)
ats_int_type add2(ats_ref_type pn)
{
        add1(pn);
        add1(pn);
        return *(ats_int_type *)pn;
}
%}  // end of C code

extern fun add2 (n: &int): int = "add2"
```

```
implement main () = let
  val n = 3
  var x = n
  val _ = add2 (x)
  val _ = printf ("add2(%d) = %d\n", @(n, x))
in end

extern fun add1 (n: &int): int = "add1"
implement add1 (n) = let
  val () = n := n + 1
in
  n
end
```

Those types (in the form ats_xxx_type) in C are defined in the file ccomp/runtime/ats_types.h. (We can change the definition of these types by overriding ats_type.h, which is useful when using ATS for Linux kernel development. We will talk about this topic later.) Part of the file is shown below.

```
typedef void ats_abs_type ;
typedef void *ats_ptr_type ;
typedef void *ats_ref_type ;
typedef void ats_var_type ;
typedef void ats_void_type ;

/* ****** ****** */

typedef int ats_bool_type ;
typedef unsigned char ats_byte_type ;

typedef char ats_char_type ;
typedef signed char ats_schar_type ;
typedef unsigned char ats_uchar_type ;

typedef double ats_double_type ;
typedef long double ats_ldouble_type ;
typedef float ats_float_type ;

…..............
```

Since all boxed types as well as call-by-reference are ultimately mapped to “void *” in C, the C implementation of those functions (e.g. “add2” in the previous example) should cast the pointer

to appropriate type in order to manipulate the data. The part of programming should be taken great care of due to the lack of protection from types.

2) unboxed types

…... todo …...

3) exporting ATS type to C

In the enclosing C code of testmain03.dats, some of the ats_ptr_type represents “ref Tree”, others represents “Tree”, which makes the code very confusing. We can enforce ATS compiler to give type in ATS an alias in C. The following code in ATS

---

extern typedef "exTree" = Tree

---

would be mapped to the following code in C.

---

typedef ats_ptr_type exTree ;

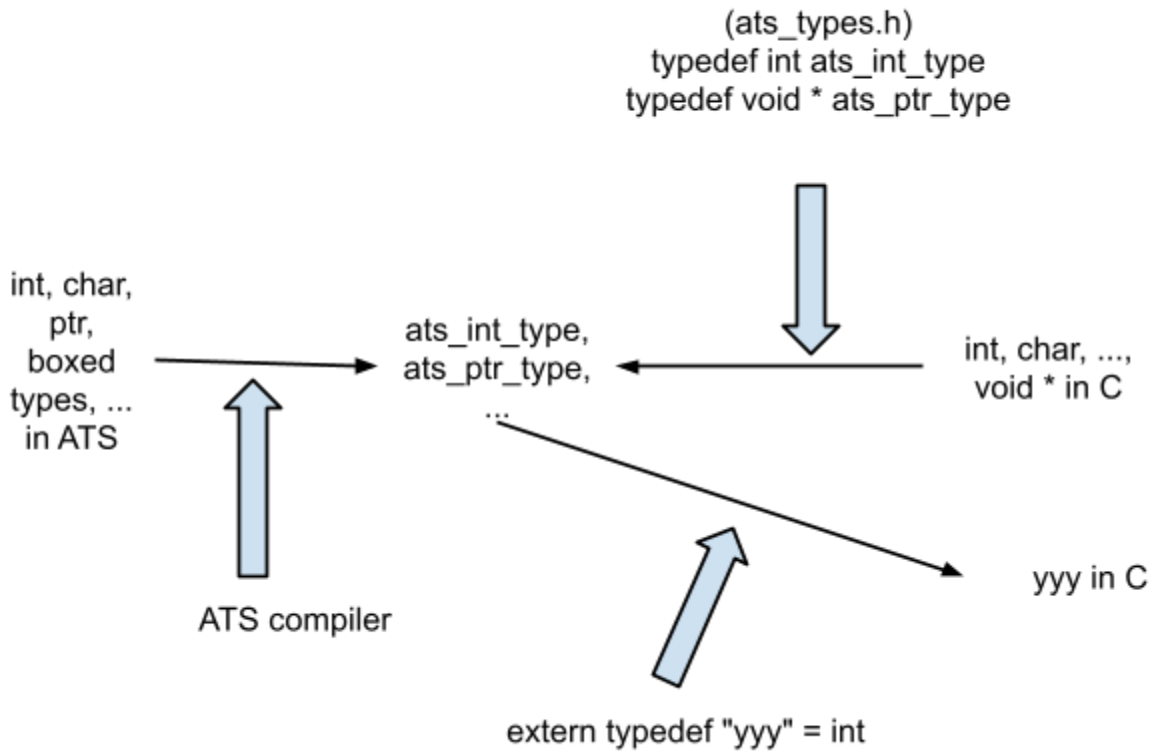
Using this trick, we can rewrite testmain03.dats as follows

```
/*
* testmain05.dats
*/

datatype tree (int) =
| Nil(0) | {n1,n2:two} Node(1) of (tree n1, int, tree n2)

viewtypedef Tree = [n:two] tree n

extern fun ref_tree (t: Tree): ref Tree = "ref_tree"

extern typedef "exTree" = Tree

%{$

ats_ptr_type ref_tree (exTree t) {
 exTree *r;
 r = ats_malloc_gc (sizeof(exTree)) ;
 *r =t ;
 return r ;
}
```

%}

This trick can also be applied to primitive types in ATS. But extern typedef "int" = int would be mapped to typedef ats_int_type int; which causes compiling error for gcc.

The mapping of types between ATS and C discussed so far is illustrated in the following chart.



4) exporting C types to ATS ($extype, typedef)

In ATS code, we can refer to types in C directly by the keyword “$extype”. ATS compiler shall map $extype “xxx” in ATS to xxx in C. The following code shows such usage.

```
/*
* testmain06.dats
*/
```

```
%{^

typedef struct XBox
{
        int x;
} XBox;

%}

// create alias in ATS for C types
typedef xbox_ptr = $extype "XBox *"
typedef xbox = $extype "XBox"

extern fun newXBox (x: int): xbox_ptr = "newXBox"
extern fun deleteXBox (x: xbox_ptr): void = "deleteXBox"

extern fun getX_ptr (x: xbox_ptr): int = "getX_ptr"
extern fun getX_ref (x: &xbox): int = "getX_ref"

extern fun setX (x: xbox_ptr, n: int): void = "setX"

extern fun showX (x: xbox_ptr): void = "showX"

extern fun printX (x: &xbox): void = "printX"
implement printX (x) = let
  val n = getX_ref (x)
  val () = printf ("X is %d\n", @(n))
in end

implement main () = let
  val x = newXBox (5)
  val n = getX_ptr (x)
  val () = setX (x, n + 1)
  val () = showX (x)
  val () = deleteXBox (x)
in end



%{$


XBox * newXBox(ats_int_type x)
```

```
{
	XBox *p = (XBox *)malloc(sizeof (XBox));
	p->x = x;
	return p;
}

ats_void_type deleteXBox(XBox *p)
{
	free(p);
	return;
}

ats_int_type getX_ptr(XBox *p)
{
	return p->x;
}

ats_int_type getX_ref(ats_ref_type p)
{
	return ((XBox *)p)->x;
}

ats_void_type setX(XBox *p, ats_int_type n)
{
	p->x = n;
}

ats_void_type showX(XBox *p)
{
	// The following two lines are unnecessaray since C compiler shall
	// do the type conversion (XBox * -> ats_ref_type) implicitly.
	// ats_ref_type ref = (ats_ref_type)p;
	// printX(ref);

	printX(p);  // call function in ATS
	return;
}

%}
```

ATS compiler cannot compare types declared by “$extype” even though two types may be literally the same. A good practice is to use “typedef” in ATS to create alias for C types and use the alias throughout ATS code as “testmain06.dats” does. As an example of bad practice, the

following code won’t compile in ATS.

```
/*
* testmain07.dats
*/


extern fun newXBox (x: int): $extype "XBox *" = "newXBox"
extern fun deleteXBox (x: $extype "XBox *"): void = "deleteXBox"

implement main () = let
  val x = newXBox (5)
  val () = deleteXBox (x)
in end
```

---

5) application of linear types ($extype, viewtypedef)

In testmain06.dats, if we erase the line “val () = deleteXBox (x)” in “main” fucntion, the code can still pass the type checker. By the linear type of ATS, it’s easy to force programmer to call “deleteXBox” after “createXBox”. The following example uses “viewtypedef” to achieve this goal. (abstract types of ATS can also be used for such purpose, which will be shown later.) Please compare it with testmain06.dats to see the modification necessary for using linear types. (All the enclosing C code is omitted since they are the same as testmain06.dats

---

```
/*
* testmain08.dats
*/

// create alias in ATS for C types
viewtypedef xbox_ptr = $extype "XBox *"
typedef xbox = $extype "XBox"

extern fun newXBox (x: int): xbox_ptr = "newXBox"
extern fun deleteXBox (x: xbox_ptr): void = "deleteXBox"

extern fun getX_ptr (x: !xbox_ptr): int = "getX_ptr"
extern fun getX_ref (x: &xbox): int = "getX_ref"

extern fun setX (x: !xbox_ptr, n: int): void = "setX"

extern fun showX (x: !xbox_ptr): void = "showX"
```

```
extern fun printX (x: &xbox): void = "printX"
implement printX (x) = let
  val n = getX_ref (x)
  val () = printf ("X is %d\n", @(n))
in end

implement main () = let
  val x = newXBox (5)
  val n = getX_ptr (x)
  val () = setX (x, n + 1)
  val () = showX (x)
  val () = deleteXBox (x)
in end
```

---

6) export C structure into ATS ($extype_struct, typedef)

In the previous example, to manipulate the member of a C structure, we have to provide the corresponding get/set functions in C and declare them in ATS. Using keyword $extype_struct can simplify the process. The following code create an alias for the type XBox. Furthermore, such code also exposes the member of XBox into ATS so that we can use “.” and “->” operators to access those members directly in ATS. Also it’s not mandatory to expose all the members of the C structure into ATS.

```
typedef xbox (n: int) = $extype_struct "XBox" of {
  x = int n
}
```

The following code shows the common usage of $extype_struct.

```
/*
* testmain09.dats
*/

%{^

typedef struct XBox
{
        int x;
        int y;
} XBox;

%}
```

```
// create alias in ATS for C types
typedef xbox (n: int) = $extype_struct "XBox" of {
  x = int n
}

extern fun newXBox {x:int} (x: int x): [l: agz] (xbox (x) @ l | ptr l) = "newXBox"
extern fun deleteXBox {l: agz}{x:int} (pf: xbox (x) @ l | p: ptr l): void = "deleteXBox"

extern fun showX {l:agz} {x:int} (pf: !xbox (x) @ l | p: ptr l): void = "showX"

typedef Xbox = [x:int] xbox (x)

extern fun printX (box: &Xbox): void = "printX"
implement printX (box) = let
  val n = box.x  // get the member of box
  val () = printf ("X is %d\n", @(n))
in end

fun incX {x:int} (box: &xbox (x) >> xbox (x+1)): void = let
  val x = box.x
  val () = box.x := (x + 1)  // set the member of box
in end

extern fun initX (x: & Xbox? >> xbox (0)): void = "initX"

extern fun createBox (): xbox (0) = "createBox"


implement main () = let
  val (pf | p) = newXBox (7)
  val n = p->x  // get the member
  val () = p->x := (n + 1)  // set the member
  val () = showX (pf | p)
  val () = printX (!p)
  val () = deleteXBox (pf | p)

  var box: Xbox
  val () = initX (box)  // must initialize
  val n = box.x
  val () = box.x := (n + 1)
  val () = incX (box)
  val () = printX (box)
```

```
  // printx is call-by-reference, so box2 has to be defined by var.
  var box2 = createBox ()
  val () = printX (box2)

in end


%{$

ats_ptr_type newXBox(ats_int_type x)
{
        XBox *p = (XBox *)malloc(sizeof (XBox));
        p->x = x;
        p->y = 0;
        return p;
}

ats_void_type deleteXBox(ats_ptr_type p)
{
        free((XBox *)p);
        return;
}

XBox createBox()
{
        XBox b = {.x = 0, .y = 0};
        return b;
}


ats_void_type initX(ats_ptr_type box)
{
        XBox *p = (XBox *)box;
        p->x = 0;
        p->y = 0;
        return;
}


ats_void_type showX(ats_ptr_type p)
{
        printX(p);  // call function in ATS
```

```
        return;
}

%}
```

---

7) abstype, abst@ype, absviewtype and absviewt@ype

In 4) and 5), we use $extype with typedef (or viewtypedef) to refer to types when writing ATS code. We can achieve the same goal by assigning C types to abstract types. For example, typedef xbox_ptr = $extype “XBox *” is equivalent (from the perspective of C code generation) to abstype xbox_ptr = $extype “XBox *”. And typedef xbox = $extype “XBox” is equivalent to abst@ype xbox_ptr = $extype “XBox *”. Similarly, absviewtype and absviewt@ype can be used when linear types are required.

ATS compiler won’t check the size of a type introduced by “$extype”. It’s the Programmers’ responsible to make sure that only C type of the same size as pointer be assigned to abstype and absviewtype.

8) mac#

In C, it’s a common practice to define an macro which is used like a function. (e.g. #define increase(x) (x) + 1) Simply declare a function in ATS with the name “increase” won’t work. This is because for every function declared in ATS, the ATS compiler would emit it’s corresponding declaration in the generated C code. For example the ATS code extern fun increase (x: int) = “increase” would cause ATS compiler to generate something like ATSextern_fun(ats_int_type, increase) (ats_int_type) ; in the C code, which is equivalent to extern ats_int_type increase(ats_int_type x). This would cause failure in compiling C code if increase is actually an macro defined in the C file. To tackle this, we can use prefix “mac” to the function name. The following code demonstrates its usage.

```
/*
* testmain11.dats
*/

extern fun increase (x: int): int = "mac#increase"

implement main () = let
  val x = 1
  val x = increase (x)
  val () = printf ("x is %d\n", @(x))
in end
```

```
%{^  // has to use ${^

#define increase(x) (x) + 1

%}
```

---

4. Global values ($extval)

The
By keyword $extval, we can refer to global variable in C. The syntax is $extval (sometype, “name”), in which sometype is the type of the value and name is the name of the variable in C. The following code demonstrates the usage.

---

testmain12.dats

On the other hand, we can assign a name to a global value in ATS by extern val name = exp where *name* is a string literal (representing a valid identifier in C) and *exp* ranges over dynamic expressions in ATS. When used in external C code, the string *name* refers to the value of the expression *exp*. The following code demonstrates the usage.

```
/*
* testmain13.dats
*/

fun getOne () = 1

extern val "globalX" = getOne ()

extern fun getGlobalX (): int = "getGlobalX"


implement main () = let
  val () = printf ("globalX is %d\n", @( $extval (int, "globalX") ))
  val gx = getGlobalX ()
  val () = printf ("globalX is %d\n", @(gx))
in end

%{$
int getGlobalX()
{
        return globalX;
```

```
}

%}
```

Scenario B.

The functionality and the control flow remains the same as testmain08.dats, but the project is split into multiple files. For small project like this it may not be necessary to introduce a bunch of header files (.h in C and .sats in ATS). But they are indispensable when dealing with large project of multiples modules. The following eight files (testmain14.dats, testCommon14.sats, testC14.h, testC14.sats, testC14.dats, testATS14.h, testATS14.sats, testATS14.dats) are the split version of testmain08.dats. Also it's not the only way to split testmain08.dats. To build the executable, the shell command is follows

```
>> atscc testmain14.dats testC14.c testATS14.dats
```

Scenario C.

There is one ATS file and one C file in the project. Function "main" in C calls function "add2" in ATS. The later one calls function "add1" in C.

Scenario D.

Merge the previous two files using a special feature of ATS. main_dymmy

Scenario E.

The functionality and the control flow remains the same, but the project is split into multiple files.

Good Example

For practical examples of mixing ATS with C program, please refer to $(ATSHOME)/libc/SATS/pthread.sats, $(ATSHOME)/libats/SATS/parworkshop.sats, and etc.

Makefile

The following address contains a Makefile template appropriate for developing large project consisting of both ATS and C source code.
https://github.com/alex-ren/ats_library/blob/master/application/Makefile

Search the keyword “todo” in the file, and do appropriate modification to meet your own requirement for inclusion and linking path.

---

todo list …

focus:

*.dats -> *.c

-D_ATS_GCATS has any influence on compilation?

atsopt -o xx.c -d xx.dats (Does the code contains gc? Or it depends on the code itself?)

#define ATS_STALOADFLAG 0
// no need to treat the sats file as a compiling module

#define ATS_DYNLOADFLAG 0

-D_ATS_HEADER_NONE -D_ATS_PRELUDE_NONE

%{^
%{$
%{# ordering

%{