

COMP 321 Homework 6: Product Types

Dan Licata

Fall 2026

Scheduling note: This homework is intended to be about the length of 1.5 weekly homeworks, and you have 2 weeks to complete it — so around 1.5 weeks if you take Fall break off.

1 Product Types with Split

This written part is to finish some of the lab problems from Wednesday, so if you already made progress on them during class you can hand that in.

In class, we discussed the following rules for product types:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{Pair}(e_1, e_2) : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{First}(e) : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{Second}(e) : \tau_2}$$

$$\frac{e_1 \mapsto e'_1}{\text{Pair}(e_1, e_2) \mapsto \text{Pair}(e'_1, e_2)} \text{ pair-search-1} \quad \frac{e_2 \mapsto e'_2 \quad e_1 \text{ done}}{\text{Pair}(e_1, e_2) \mapsto \text{Pair}(e_1, e'_2)} \text{ pair-search-2}$$

$$\frac{e_1 \text{ done} \quad e_2 \text{ done}}{\text{Pair}(e_1, e_2) \text{ done}} \text{ pair-done}$$

$$\frac{e \mapsto e'}{\text{First}(e) \mapsto \text{First}(e')} \text{ first-search} \quad \frac{\text{Pair}(e_1, e_2) \text{ done}}{\text{First}(\text{Pair}(e_1, e_2)) \mapsto e_1} \text{ first-instr}$$

$$\frac{e \mapsto e'}{\text{Second}(e) \mapsto \text{Second}(e')} \text{ second-search} \quad \frac{\text{Pair}(e_1, e_2) \text{ done}}{\text{Second}(\text{Pair}(e_1, e_2)) \mapsto e_2} \text{ second-instr}$$

These rules let you access the components of a tuple by projecting them individually. For example, here is what a function left-associates a triple looks like:¹

```
reassociate (t : int x (int x int)) : (int x int) x int =  
  ( ( first(t) , first(second(t)) ) , second(second(t)) )
```

A programmer might want a nicer notation in which one can pattern-match on a pair and give names to individual components. To start with, we can add a language feature for pattern-matching on a single pair so that the above example can be written

¹Assume we have functions whose bodies are expressions which return the value of that expression; we will cover this formally soon.

```

reassociate (t : int x (int x int)) : (int x int) x int =
  split (x,p) = t in
  split (y,z) = p in
  ((x,y),z)

```

We represent `split (x,y) = e1 in e2` by the abstract syntax $\text{Split}(e_1, x.y.e_2)$ with two variables x and y newly in scope in e_2 .

Task 1 (10 %). Give typing rules and operational semantics for `split`.

Task 2 (20 %). Prove the cases of progress and preservation for the typing and operational semantics rules for `split` that you defined in Task 1.

Task 3 (20 %). Show that the projections `First` and `Second` are *interdefinable* with `Split` (in a language with `Let` as in Section 1 of Wednesday’s lab). This means:

- In a language with `Split`, define $\text{First}(e)$ and $\text{Second}(e)$ and show that these definitions satisfy the above typing and operational semantics rules. That is, define

$\text{myFirst}(e) := \text{some expression using Split and } e$

and then show in the language with `Split` but without `First` that

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{myFirst}(e) : \tau_1} \quad \frac{e \mapsto e'}{\text{myFirst}(e) \mapsto \text{myFirst}(e')} \quad \frac{\text{Pair}(e_1, e_2) \text{ done}}{\text{myFirst}(\text{Pair}(e_1, e_2)) \mapsto e_1}$$

Then do the same thing for mySecond .

- In a language with `First` and `Second` (and `Let` as in the first problem in the lab), define `Split` and show that the definition satisfies the typing and operational semantics rules that you gave above in Task 1. A single operational semantics step in the language with `Split` may be implemented by several steps in the language with `First`, `Second`, and `Let`.

2 Implementing Structs

In this problem, you will implement product types as structs in IMP. The design you will implement is somewhat of a mix of C-like structs and record types in functional languages:

- At the expression level, we will have the ability to create a struct value by giving a value for each field, and to project a field of a struct. This part is analogous to the rules for record types from Monday’s class. For example, consider a struct definition

```
struct Point {x = int, y = int }
```

This represents a value with x and y coordinates, and defines a type write `struct Point`. We can make a value with the expression

```
struct Point {x = 1, y = 2} : struct Point
```

and if `e : struct Point` then we can write `(e.x) : int` and `(e.y) : int`.

- When represented in memory, a struct will be represented *flattened*, with no overhead. For example, the memory representation of `struct Point {x = 21, y = 22}` will be two adjacent memory addresses `[(a, 21), (a+1, 22)]`. As in C, the address of a struct will be the address of its first field, and the later fields can be set and accessed by their *offset*, which is how many memory cells past the beginning of the struct the field is stored. For example, in the above example, the offset of field `y` is 1. In the struct `struct Triple {x = int, y = int, z = int}`, the offset of `z` is 2, and in the struct `struct NestedTriple {p = struct Point, z = int}` the offset of `z` is also 2, because it will take two memory addresses to store the `struct Point`.

In general, an `int` and any pointer type `T *` will be represented by one memory cell, as we have done so far, while a struct will be represented by the sum of the number of memory cells needed to represent its fields.

2.1 Structs

The new syntax is defined as follows: We allow struct declarations written

```
struct S { f1 = T1 , ... , fn = Tn }
```

at the top level of the program. This defines a type `struct S` which is the record type with fields `fi` each containing data of type `Ti`. In abstract syntax, we write $\text{StructDecl}(S, [(f_1, \tau_1), \dots, (f_n, \tau_n)])$ for the declaration and $\text{StructTyp}(S)$ for the type.

To construct a value of this type, we have an expression `struct S {f1 = e1, ... , fn = en}`. In abstract syntax, we write $\text{StructVal}(S, [(f_1, e_1), \dots, (f_n, e_n)])$. The typing and operational semantics are:

$$\frac{\text{StructDecl}(S, [(f_1, \tau_1), \dots, (f_n, \tau_n)]) \in p \quad \Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \text{StructVal}(S, [(f_1, e_1), \dots, (f_n, e_n)]) : \text{StructTyp}(S)}$$

$$\frac{e_1 \text{ done} \quad \dots \quad e_n \text{ done}}{\text{StructVal}(S, [(f_1, e_1), \dots, (f_n, e_n)]) \text{ done}}$$

$$\frac{e_1 \text{ done} \quad \dots \quad e_{i-1} \text{ done} \quad \mu \mid e_i \mapsto e'_i}{\mu \mid \text{StructVal}(S, [(f_1, e_1), \dots, (f_i, e_i), \dots, (f_n, e_n)]) \mapsto \text{StructVal}(S, [(f_1, e_1), \dots, (f_i, e'_i), \dots, (f_n, e_n)])}$$

To access a field, in concrete syntax we write `(e.f)` (this can be iterated like `(e.f1.f2.f3)` if you have nested structs). In abstract syntax, we write $\text{ProjectField}(e, f)$:

$$\frac{\Gamma \vdash e : \text{StructTyp}(S) \quad \text{StructDecl}(S, [(f_1, \tau_1), \dots, (f_n, \tau_n)]) \in p}{\Gamma \vdash \text{ProjectField}(e, f) : \tau_i}$$

$$\frac{\mu \mid e \mapsto e'}{\mu \mid \text{ProjectField}(e, f) \mapsto \text{ProjectField}(e', f)}$$

$$\frac{\text{StructVal}(S, [(f_1, e_1), \dots, (f_i, e_i), \dots, (f_n, e_n)]) \text{ done}}{\mu \mid \text{ProjectField}(\text{StructVal}(S, [(f_1, e_1), \dots, (f_i, e_i), \dots, (f_n, e_n)]), f_i) \mapsto e_i}$$

Finally, we add a built-in function `offset(S, f)` to get the offset of a field in a struct:

$$\overline{\Gamma \vdash \text{Offset}(S, f) : \text{int}}$$

One of your tasks is to figure out the operational semantics for `Offset`.

2.2 Memory Representation

Combining the above operational semantics with the semantics of `LocalVar` and `Get` and `Set` that we have discussed previously will not work. Up until now, the implementation of these operations has worked with the contents of a single memory address — but now it will take many memory addresses to represent a single value. For example, if we implemented

```
{mut struct Point x;
  ...
}
```

by the implementation of `allocate` in Homework 3, which allocates one memory cell, we would not have enough space to store the two `ints` in a `Point`.

To fix this, we will annotate `Get` and `Set` with the type of the value stored in the memory location, e.g.

```
{mut int x;
  x :=[int] 1;
}

{mut struct Point p;
  p :=[struct Point] (struct Point {x = 1, y =1});;
}

{mut struct Point p;
  {mut int z;
    z :=[int] ( (get[struct Point] p).x);
  }
}
```

In abstract syntax, we write

$$\frac{\Gamma \vdash e : \tau^*}{\Gamma \vdash \text{Get}(\tau, e) : \tau} \quad \frac{\Gamma \vdash e_1 : \tau^* \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{Set}(\tau, e_1, e_2) \text{ok}}$$

One of your tasks in this homework is to figure out how the operational semantics for `Get` and `Set` and `LocalVar` should change to accommodate types of varying size. At a high level, the search rules will stay the same, while the instruction rules will be responsible for updating/looking up more than one memory location. For example, consider the instruction rule for `get`

$$\overline{\mu \mid \text{Get}(\tau, \text{Num}(a)) \mapsto e'}$$

Here, a is the memory address where the value of type τ starts (e.g. if τ is a struct type, then a is the address where the first field starts). If we always return $\text{Num}(\text{lookup}(\mu, a))$ as before, this will be fine when τ is

int or a pointer type, but incorrect when τ is a struct type. To satisfy type preservation, e' must do enough memory lookups to reconstruct a value of type τ , so when τ is `StructTyp( $S$ )` it must make a value for each field of S and package them up in a `StructVal`. Dually, `Set` should update the memory by writing each field of e_2 to the appropriate location. Finally, `LocalVar` must allocate enough memory space for the type of the variable.

Using this memory representation, we can assign to a field of a struct using `offset`. For example

```
struct Triple {first = int, second = int, third=int}

main() {
  {mut struct Triple x;
  {mut int y;
    x :=[struct Triple ] struct Triple { first = 1, second = 2 , third = 2 };
    ( cast[int*] (cast[int] x + offset(Triple,third)) ) :=[int] 3;
    y :=[int] (get[struct Triple](x).third);
  }
  }}
;
```

The first line declares a struct named `Triple` with three fields. The next line `mut struct Triple x;` therefore must allocate three consecutive memory addresses. The line

```
x :=[struct Triple ] struct Triple { first = 1, second = 2 , third = 2};
```

should write the numbers 1,2,and 2 to each of those three addresses, because it creates a `struct` value and then does a `:=` at type `struct Triple`. In the next line,

```
( cast[int*] (cast[int] x + offset(Triple,third)) )
```

the offset of `third` should be 2 (because there are two ints to the left of it), so location `x+2` will be the third memory address. Thus, in the final line `y` is set to 3, because the `get[struct Triple](x)` reads in the values of the all three memory locations as a value

```
struct Triple { first = 1, second = 2 , third = 3 }
```

and then projects the `third` field.

As an aside, this syntax, where each `get/set` is annotated with a type, is needlessly verbose, because the type checker could insert these annotations based on the type of the arguments to `get/set`. However, to avoid making this homework too long, we will not implement that this week. So you will need to insert the type annotations yourself when writing examples, so that the operational semantics can do the correct number of memory lookups/updates for the type in question.

2.3 Tasks

Task 1 (25 %). In `check.sml`, extend your type checker from Homework 5 to type check structs. Start from your code for Homework 5. To prepare, add the program `p` as an input to `check_exp` (in the function parameters and in all the recursive calls and the calls from `check_statement`), and make the cases for `Get` and `Set` check the new type annotation. Then, add cases for type checking the new declaration `StructDecl( $S, [(f_1, \tau_1), \dots, (f_n, \tau_n)]$ )`, and the new expressions `StructVal( $S, [(f_1, e_1), \dots, (f_n, e_n)]$ )` and `ProjectField( $e, f$ )`.

Task 2 (25 %). Start from your `step.sml` from Homework 3. To prepare, modify the cases for `Get` and `Set` and `LocalVar` to take the type information. Modify the case for function calls to propagate the type information. Modify `step_exp` to take the program as input. Then, in `step_exp`, implement the above rules for `ProjectField` and `StructVal`.

Task 3 (40 %). Next, in `step_exp` and `step_statement`, modify the `Get`, `Set`, `LocalVar` and `Free` cases, and add the `Offset` case to store structs in memory, as discussed above. Much of the work can go in the helper functions `read_value` and `write_value`. The `allocate` and `deallocate` functions may also need to be changed from HW3 — note that `allocate` now takes the type of the variable being allocated.

Task 4 (10 %). There are a few tests in `test-struct1.imp` through `test-struct4.imp` and `test-offset.imp` through `test-offset3.imp` that you can start with.

To more thoroughly test your code, in `reverse-dll.imp`, write code to in-place reverse a doubly-linked list. Each node in a doubly linked list is represented by a struct

```
struct Node {value = int, prev = struct Node *, next = struct Node *}
```

with a value and pointers to the previous and next nodes. A NULL pointer is represented by 0. The first node in a list has NULL previous, while the last node has NULL next. `reverse` takes a pointer to a node, and reverses the segment of the list from that node until the last node. Reversing a doubly-linked list means changing the `next/previous` pointers in each node so that looping along the new `next` pointers from the original *last* element encounters the original values in the opposite order.