7 Tuples and Garbage Collection

Specialized version of this chapter for use at PSU, Winter 2024.

In this chapter we study the implementation of tuples. A tuple is a fixed-length sequence of elements in which each element may have a different type. This language feature is the first to use the computer's *heap*, because the lifetime of a tuple is indefinite; that is, a tuple lives forever from the programmer's viewpoint. Of course, from an implementer's viewpoint, it is important to reclaim the space associated with a tuple when it is no longer needed, which is why we also study *garbage collection* techniques in this chapter.

Section 7.1 introduces the $\mathcal{L}_{\mathsf{Tup}}$ language, including its interpreter and type checker. The $\mathcal{L}_{\mathsf{Tup}}$ language extends the $\mathcal{L}_{\mathsf{While}}$ language (chapter 6) with tuples. Section 7.2 describes a garbage collection algorithm based on copying live tuples back and forth between two halves of the heap. The garbage collector requires coordination with the compiler so that it can find all the live tuples. Sections 7.4 through 7.8 discuss the necessary changes and additions to the compiler passes, including a new compiler pass named `expose_allocation`.

7.1 The $\mathcal{L}_{\mathsf{Tup}}$ Language

Figure 7.1 shows the definition of the concrete syntax for $\mathcal{L}_{\mathsf{Tup}}$, and figure 7.2 shows the definition of the abstract syntax. The $\mathcal{L}_{\mathsf{Tup}}$ language adds (1) tuple creation via a comma-separated list of expressions; (2) accessing or updating an element of a tuple with the square bracket notation (i.e., `t[n]` returns the element at index `n` of tuple `t`); and (3) the `is` comparison operator~~; and (4) obtaining the number of elements (the length) of a tuple~~. In this chapter, we restrict access indices to constant integers.

Our tuples follow the standard Python syntax for tuples. In particular, a one-element tuple containing e is written `(`e`,)` and a zero-element tuple is written `()`. Our tuples also have similar semantics to those of Python, except that (1) we restrict indices to be constant integers rather than arbitrary expressions, and (2) we allow tuple fields to be updated (using an assignment statement) as long as the type of the field does not change, whereas Python treats tuples as immutable data structures.

The following program shows an example of the use of tuples. It creates a tuple `t` containing the elements `40`, `True`, and another tuple that contains just `3`. It then

exp	::=	*int* | `input_int()` | `-` *exp* | *exp* `+` *exp* | *exp* `-` *exp* | `(`*exp*`)`
stmt	::=	`print(`*exp*`)` | *exp*
exp	::=	*var*
stmt	::=	*var* `=` *exp*
cmp	::=	`==` | `!=` | `<` | `<=` | `>` | `>=`
exp	::=	`True` | `False` | *exp* `and` *exp* | *exp* `or` *exp* | `not` *exp*
	|	*exp cmp exp* | *exp* `if` *exp* `else` *exp*
stmt	::=	`if` *exp*`:` $stmt^+$ `else:` $stmt^+$
stmt	::=	`while` *exp*`:` $stmt^+$
cmp	::=	`is`
exp	::=	*exp*`,` ... `,`*exp* | `()` | *exp*`[`*int*`]`
stmt	::=	*exp*`[`*int*`]` `=` *exp*
$\mathcal{L}_{\text{Tup}}$	::=	$stmt^*$

Figure 7.1
The concrete syntax of $\mathcal{L}_{\text{Tup}}$, extending $\mathcal{L}_{\text{While}}$ (figure 6.1).

updates the initial element of the tuple to be 39. The element at index 1 of `t` is `True`, so the *then* branch of the `if` is taken. The element at index 0 of `t` is now 39, to which we add 3, the element at index 0 of the tuple. The result of the program is 42.

```
t = 40, True, (3,)
t[0] = 39
print(t[0] + t[2][0] if t[1] else 44)
```

Tuples raise several interesting new issues. First, variable binding performs a shallow copy in dealing with tuples, which means that different variables can refer to the same tuple; that is, two variables can be *aliases* for the same entity. Consider the following example, in which `t1` and `t2` refer to the same tuple value and `t3` refers to a different tuple value with equal elements. The result of the program is 42.

```
t1 = 3, 7
t2 = t1
t3 = 3, 7
print(42 if (t1 is t2) and not (t1 is t3) else 0)
```

Whether two variables are aliased or not affects what happens when the underlying tuple is mutated. Consider the following example in which `t1` and `t2` again refer to the same tuple value.

```
t1 = 3, 7
t2 = t1
t2[0] = 42
print(t1[0])
```

```
exp    ::= Constant(int) | Call(Name('input_int'),[])
         | UnaryOp(USub(),exp) | BinOp(exp,Add(),exp)
         | BinOp(exp,Sub(),exp)
stmt   ::= Expr(Call(Name('print'),[exp])) | Expr(exp)
-----------------------------------------------------
exp    ::= Name(var)
stmt   ::= Assign([Name(var)], exp)
-----------------------------------------------------
boolop ::= And() | Or()
cmp    ::= Eq() | NotEq() | Lt() | LtE() | Gt() | GtE()
bool   ::= True | False
exp    ::= Constant(bool) | BoolOp(boolop,[exp,exp])
         | UnaryOp(Not(),exp) | Compare(exp,[cmp],[exp])
         | IfExp(exp,exp,exp)
stmt   ::= If(exp, stmt+, stmt+)
-----------------------------------------------------
stmt   ::= While(exp, stmt+, [])
-----------------------------------------------------
cmp    ::= Is()
exp    ::= Tuple(exp*,Load()) | Subscript(exp,Constant(int),Load())
stmt   ::= Assign([Subscript(exp,Constant(int),Store())], exp)
L_Tup  ::= Module(stmt*)
```

Figure 7.2
The abstract syntax of $\mathcal{L}_{\text{Tup}}$.

The mutation through `t2` is visible in referencing the tuple from `t1`, so the result of this program is again 42.

The next issue concerns the lifetime of tuples. When does a tuple's lifetime end? Notice that $\mathcal{L}_{\text{Tup}}$ does not include an operation for deleting tuples. Furthermore, the lifetime of a tuple is not tied to any notion of static scoping. For example, the following program returns `42` even though the variable `x` goes out of scope when the function returns, prior to reading the tuple element at index 0. (We study the compilation of functions in chapter 8.)

```
def f():
    x = 42, 43
    return x
t = f()
print(t[0])
```

From the perspective of programmer-observable behavior, tuples live forever. However, if they really lived forever then many long-running programs would run out of memory. To solve this problem, the language's runtime system performs automatic garbage collection.

Figure 7.3 shows the definitional interpreter for the $\mathcal{L}_{\text{Tup}}$ language. We represent tuples with Python lists in the interpreter because we need to write to them (section 7.4). (Python tuples are immutable.) We define element access, update, and the `is` operator~~, and the len operator~~ for $\mathcal{L}_{\text{Tup}}$ in terms of the corresponding operations in Python.

```
class InterpLtup(InterpLwhile):
  def interp_cmp(self, cmp):
    match cmp:
      case Is():
        return lambda x, y: x is y
      case _:
        return super().interp_cmp(cmp)

  def interp_exp(self, e:expr, env:Env) -> Any:
    match e:
      case Tuple(es, Load()):
        return [self.interp_exp(e, env) for e in es]
      case Subscript(tup, Constant(index), Load()):
        t = self.interp_exp(tup, env)
        return t[index]
      case _:
        return super().interp_exp(e, env)

  def interp_stmt(self, s:stmt, env:Env, cont:list[stmt]):
    match s:
      case Assign([Subscript(tup, Constant(index))], value):
        tup = self.interp_exp(tup, env)
        tup[index] = self.interp_exp(value, env)
        return self.interp_stmts(cont, env)
```

Figure 7.3
Interpreter for the $\mathcal{L}_{\mathsf{Tup}}$ language.

Figure 7.4 shows the type checker for $\mathcal{L}_{\mathsf{Tup}}$. The type of a tuple is a `TupleType` type that contains a type for each of its elements. The type of accessing the ith element of a tuple is the ith element type of the tuple's type, if there is one. If not, an error is signaled. Note that the index `i` is required to be a constant integer (and not, for example, a call to `input_int`) so that the type checker can determine the element's type given the tuple type.

7.2 Garbage Collection

Garbage collection is a runtime technique for reclaiming space on the heap that will not be used in the future of the running program. We use the term *object* to refer to any value that is stored in the heap, which for now includes only tuples.[1] Unfortunately, it is impossible to know precisely which objects will be accessed in the future and which will not. Instead, garbage collectors overapproximate the set of objects that will be accessed by identifying which objects can possibly be accessed.

1. The term *object* as it is used in the context of object-oriented programming has a more specific meaning than the way in which we use the term here.

```
class TypeCheckLtup(TypeCheckLwhile):
  def type_check_exp(self, e, env):
    match e:
      case Compare(left, [cmp], [right]) if isinstance(cmp, Is):
        l = self.type_check_exp(left, env)
        r = self.type_check_exp(right, env)
        check_type_equal(l, r, e)
        return bool
      case Tuple(es, Load()):
        ts = [self.type_check_exp(e, env) for e in es]
        e.has_type = TupleType(ts)
        return e.has_type
      case Subscript(tup, Constant(i), Load()):
        tup_ty = self.type_check_exp(tup, env)
        i_ty = self.type_check_exp(Constant(i), env)
        check_type_equal(i_ty, int, i)
        match tup_ty:
          case TupleType(ts):
            return ts[i]
          case _:
            raise Exception('expected a tuple, not ' + repr(tup_ty))
      case _:
        return super().type_check_exp(e, env)

  def type_check_stmts(self, ss, env):
    if len(ss) == 0:
      return VoidType()
    match ss[0]:
      case Assign([Subscript(tup, Constant(index), Store())], value):
        tup_t = self.type_check_exp(tup, env)
        index_ty = self.type_check_exp(Constant(index), env)
        self.check_type_equal(index_ty, IntType(), index)
        value_t = self.type_check_exp(value, env)
         match tup_t:
          case TupleType(ts):
            self.check_type_equal(ts[index], value_t, ss[0])
          case _:
            raise Exception('expected a tuple, not ' + repr(tup_t))
        return self.type_check_stmts(ss[1:], env)
      case _:
        return super().type_check_stmts(ss, env)
```

Figure 7.4
Type checker for the $\mathcal{L}_{\mathsf{Tup}}$ language.

The running program can directly access objects that are in registers and on the procedure call stack. It can also transitively access the elements of tuples, starting with a tuple whose address is in a register or on the procedure call stack. We define the *root set* to be all the tuple addresses that are in registers or on the procedure call stack. We define the *live objects* to be the objects that are reachable from the root set. Garbage collectors reclaim the space that is allocated to objects that are no longer live. That means that some objects may not get reclaimed as soon as they could be, but at least garbage collectors do not reclaim the space dedicated to objects that will be accessed in the future! The programmer can influence which objects get reclaimed by causing them to become unreachable.

So the goal of the garbage collector is twofold:

1. to preserve all the live objects, and
2. to reclaim the memory of everything else, that is, the *garbage*.

7.2.1 Two-Space Copying Collector

Here we study a relatively simple algorithm for garbage collection that is the basis of many state-of-the-art garbage collectors (Lieberman and Hewitt 1983; Ungar 1984; Jones and Lins 1996; Detlefs et al. 2004; Dybvig 2006; Tene, Iyengar, and Wolf 2011). In particular, we describe a two-space copying collector (Wilson 1992) that uses Cheney's algorithm to perform the copy (Cheney 1970). Figure 7.5 gives a coarse-grained depiction of what happens in a two-space collector, showing two time steps, prior to garbage collection (on the top) and after garbage collection (on the bottom). In a two-space collector, the heap is divided into two parts named the FromSpace and the ToSpace. Initially, all allocations go to the FromSpace until there is not enough room for the next allocation request. At that point, the garbage collector goes to work to make room for the next allocation.

A copying collector makes more room by copying all the live objects from the FromSpace into the ToSpace and then performs a sleight of hand, treating the ToSpace as the new FromSpace and the old FromSpace as the new ToSpace. In the example shown in figure 7.5, the root set consists of three pointers, one in a register and two on the stack. All the live objects have been copied to the ToSpace (the right-hand side of figure 7.5) in a way that preserves the pointer relationships. For example, the pointer in the register still points to a tuple that in turn points to two other tuples. There are four tuples that are not reachable from the root set and therefore do not get copied into the ToSpace.

The exact situation shown in figure 7.5 cannot be created by a well-typed program in $\mathcal{L}_{\mathsf{Tup}}$ because it contains a cycle. However, creating cycles will be possible once we get to $\mathcal{L}_{\mathsf{Dyn}}$ (chapter 10). We design the garbage collector to deal with cycles to begin with, so we will not need to revisit this issue.

7.2.2 Graph Copying via Cheney's Algorithm

Let us take a closer look at the copying of the live objects. The allocated objects and pointers can be viewed as a graph, and we need to copy the part of the graph that is reachable from the root set. To make sure that we copy all the reachable

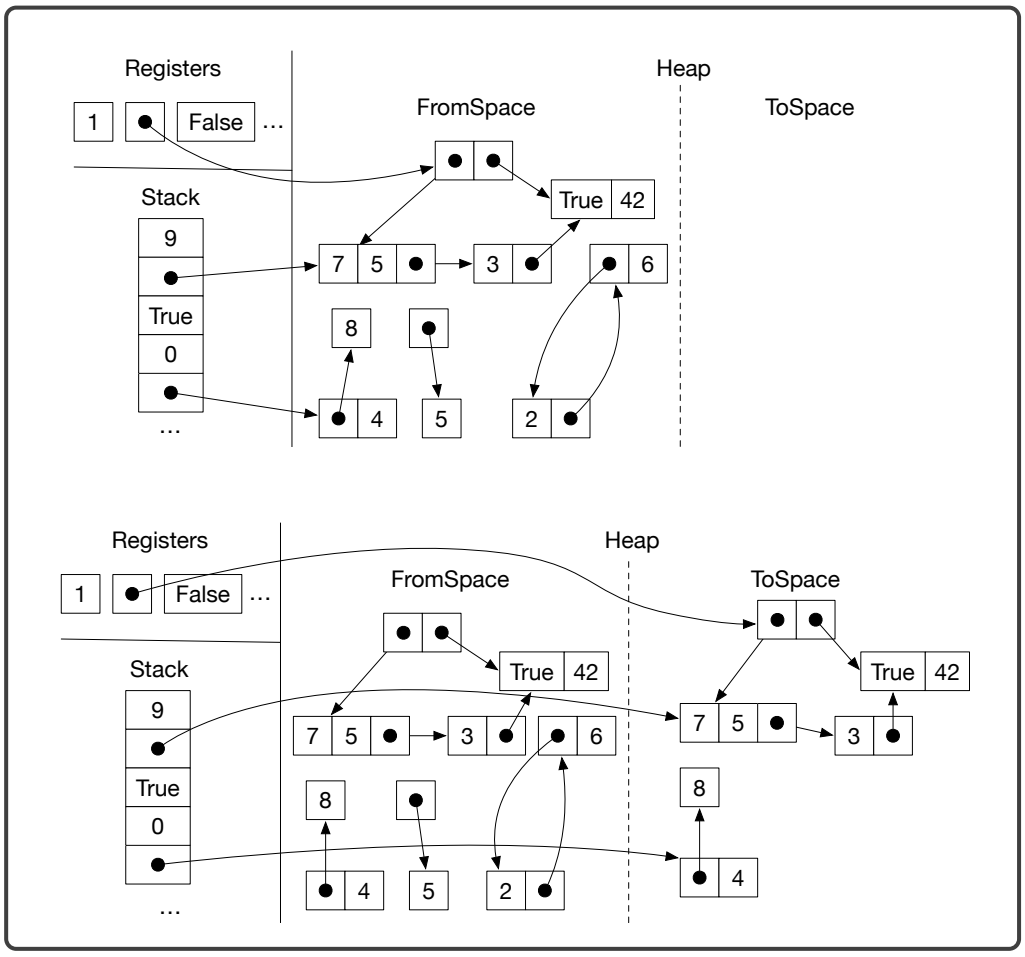

Figure 7.5
A copying collector in action.

vertices in the graph, we need an exhaustive graph traversal algorithm, such as depth-first search or breadth-first search (Moore 1959; Cormen et al. 2001). Recall that such algorithms take into account the possibility of cycles by marking which vertices have already been visited, so to ensure termination of the algorithm. These search algorithms also use a data structure such as a stack or queue as a to-do list to keep track of the vertices that need to be visited. We use breadth-first search and a trick due to Cheney (1970) for simultaneously representing the queue and copying tuples into the ToSpace.

Figure 7.6 shows several snapshots of the ToSpace as the copy progresses. The queue is represented by a chunk of contiguous memory at the beginning of the ToSpace, using two pointers to track the front and the back of the queue, called the *free pointer* and the *scan pointer*, respectively. The algorithm starts by copying all tuples that are immediately reachable from the root set into the ToSpace to form the initial queue. When we copy a tuple, we mark the old tuple to indicate that it has been visited. We discuss how this marking is accomplished in section 7.2.3.

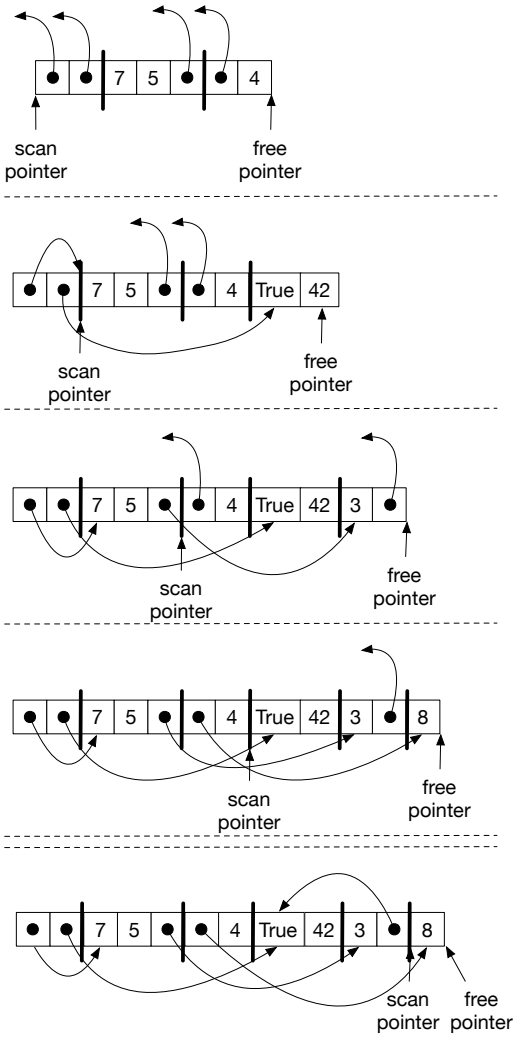

Figure 7.6
Depiction of the Cheney algorithm copying the live tuples.

Note that any pointers inside the copied tuples in the queue still point back to the FromSpace. Once the initial queue has been created, the algorithm enters a loop in which it repeatedly processes the tuple at the front of the queue and pops it off the queue. To process a tuple, the algorithm copies all the objects that are directly reachable from it to the ToSpace, placing them at the back of the queue. The algorithm then updates the pointers in the popped tuple so that they point to the newly copied objects.

As shown in figure 7.6, in the first step we copy the tuple whose second element is 42 to the back of the queue. The other pointer goes to a tuple that has already been copied, so we do not need to copy it again, but we do need to update the pointer to the new location. This can be accomplished by storing a *forwarding pointer* to the new location in the old tuple, when we initially copied the tuple into the ToSpace. This completes one step of the algorithm. The algorithm continues in this way until the queue is empty; that is, when the scan pointer catches up with the free pointer.

7.2.3 Data Representation

The garbage collector places some requirements on the data representations used by our compiler. First, the garbage collector needs to distinguish between pointers and other kinds of data such as integers. The following are three ways to accomplish this:

1. Attach a tag to each object that identifies what type of object it is (McCarthy 1960).
2. Store different types of objects in different regions (Steele 1977).
3. Use type information from the program to either (a) generate type-specific code for collecting, or (b) generate tables that guide the collector (Appel 1989; Goldberg 1991; Diwan, Moss, and Hudson 1992).

Dynamically typed languages, such as Python, need to tag objects in any case, so option 1 is a natural choice for those languages. However, $\mathcal{L}_{\mathsf{Tup}}$ is a statically typed language, so it would be unfortunate to require tags on every object, especially small and pervasive objects like integers and Booleans. Option 3 is the best-performing choice for statically typed languages, but it comes with a relatively high implementation complexity. To keep this chapter within a reasonable scope of complexity, we recommend a combination of options 1 and 2, using separate strategies for the stack and the heap.

Regarding the stack, we recommend using a separate stack for pointers, which we call the *root stack* (aka *shadow stack*) (Siebert 2001; Henderson 2002; Baker et al. 2009). That is, when a local variable needs to be spilled and is of type `TupleType`, we put it on the root stack instead of putting it on the procedure call stack. Furthermore, we always spill tuple-typed variables if they are live during a call to the collector, thereby ensuring that no pointers are in registers during a collection. Figure 7.7 reproduces the example shown in figure 7.5 and contrasts it with the data layout using a root stack. The root stack contains the two pointers from the regular stack and also the pointer in the second register.

The problem of distinguishing between pointers and other kinds of data also arises inside each tuple on the heap. We solve this problem by attaching a tag, an extra 64 bits, to each tuple. Figure 7.8 shows a zoomed-in view of the tags for two of the tuples in the example given in figure 7.5. Note that we have drawn the bits in a big-endian way, from right to left, with bit location 0 (the least significant bit) on the far right, which corresponds to the direction of the x86 shifting instructions `salq` (shift left) and `sarq` (shift right). Part of each tag is dedicated to specifying which elements of the tuple are pointers, the part labeled *pointer mask*. Within the

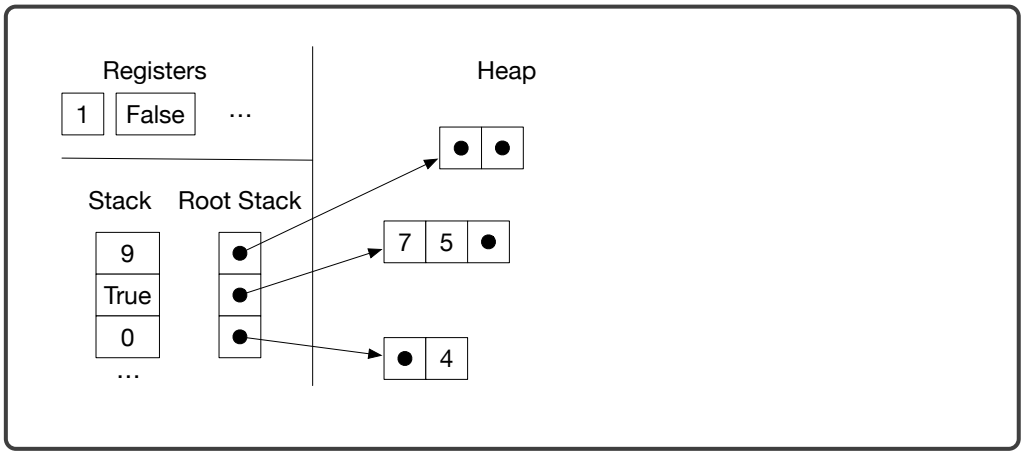

Figure 7.7
Maintaining a root stack to facilitate garbage collection.

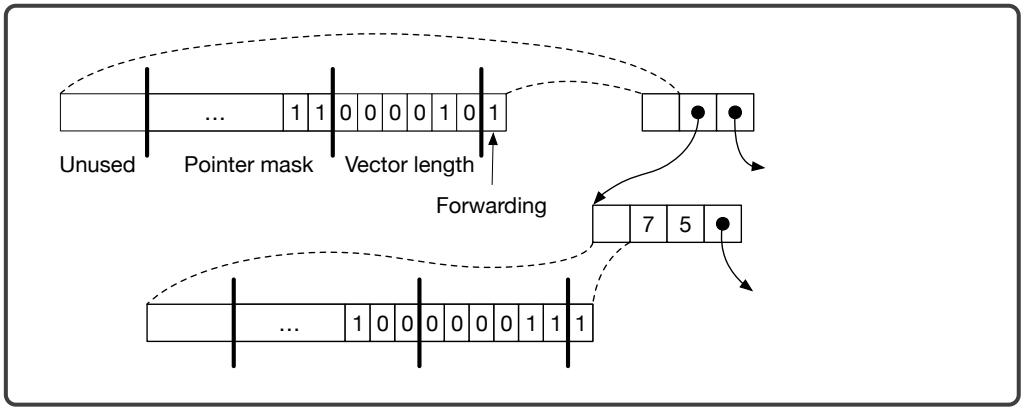

Figure 7.8
Representation of tuples in the heap.

pointer mask, a 1 bit indicates that there is a pointer, and a 0 bit indicates some other kind of data. The pointer mask starts at bit location 7. We limit tuples to a maximum size of fifty elements, so we need 50 bits for the pointer mask.[2] The tag also contains two other pieces of information. The length of the tuple (number of elements) is stored in bits at locations 1 through 6. Finally, the bit at location 0 indicates whether the tuple has yet to be copied to the ToSpace. If the bit has value 1, then this tuple has not yet been copied. If the bit has value 0, then the entire tag is a forwarding pointer. (The lower 3 bits of a pointer are always zero in any case, because our tuples are 8-byte aligned.)

2. A production-quality compiler would handle arbitrarily sized tuples and use a more complex approach.

```
void initialize(uint64_t rootstack_size, uint64_t heap_size);
void collect(int64_t** rootstack_ptr, uint64_t bytes_requested);
int64_t* free_ptr;
int64_t* fromspace_begin;
int64_t* fromspace_end;
int64_t** rootstack_begin;
```

Figure 7.9
The compiler's interface to the garbage collector.

7.2.4 Implementation of the Garbage Collector

An implementation of the copying collector is provided in the `runtime.c` file. Figure 7.9 defines the interface to the garbage collector that is used by the compiler. The `initialize` function creates the FromSpace, ToSpace, and root stack and should be called in the prelude of the `main` function. The arguments of `initialize` are the root stack size and the heap size. Both need to be multiples of 16, but otherwise these choices are pretty arbitrary: we use 65536 for the root stack size and 16 for the initial heap size. The root stack size should be large enough to make sure that this stack does not overflow (because we will live dangerously and not check for this). It would require a deeply recursive program to use up this much root stack. Our collector implementation automatically resizes the heap as needed, so the initial heap size doesn't matter much, but by setting it to 16, we guarantee to exercise the collector as vigorously as possible, which is good for finding bugs!

The `initialize` function puts the address of the beginning of the FromSpace into the global variable `free_ptr`. The global variable `fromspace_end` points to the address that is one past the last element of the FromSpace. We use half-open intervals to represent chunks of memory (Dijkstra 1982). The `rootstack_begin` variable points to the first element of the root stack.

As long as there is room left in the FromSpace, your generated code can allocate tuples simply by moving the `free_ptr` forward. The amount of room left in the FromSpace is the difference between the `fromspace_end` and the `free_ptr`. The `collect` function should be called when there is not enough room left in the FromSpace for the next allocation. The `collect` function takes a pointer to the current top of the root stack (one past the last item that was pushed) and the number of bytes that need to be allocated. The `collect` function performs the copying collection and leaves the heap in a state such that there is enough room for the next allocation.

The introduction of garbage collection has a nontrivial impact on our compiler passes. We introduce a new compiler pass named `expose_allocation` that elaborates the code for allocating tuples. We also make significant changes to `select_instructions`, `build_interference`, `allocate_registers`, and `prelude_and_conclusion` and make minor changes in several more passes.

The following program serves as our running example. It creates two tuples, one nested inside the other. Both tuples have length one. The program accesses the element in the inner tuple.

```
v1 = (42,)
v2 = (v1,)
print(v2[0][0])
```

7.3 Shrink and Remove Complex Operands

The `shrink` pass needs minor additions to cover the new forms in the language.

In the `remove_complex_operands` pass, the tuple creation and tuple subscripting expressions should be treated as complex operands. The field subexpressions of the tuple creation expression and the first subexpression of the subscripting expression and of the tuple assignment statement must be atomic. The right-hand side expression of the tuple assignment statement should also be made atomic (this could be avoided, but would require substantial reworking of the existing code in later passes). The output of this pass is called $\mathcal{L}^{mon}_{\mathsf{Tup}}$.

7.4 Expose Allocation

The pass `expose_allocation` lowers tuple creation into making a conditional call to the collector followed by allocating the appropriate amount of memory and initializing it.

Since we choose to place the `expose_allocation` pass after `remove_complex_operands` pass, we must make sure that it does not create any code that contains complex operands. The output of `expose_allocation` is a language $\mathcal{L}^{mon}_{\mathsf{Alloc}}$ that replaces tuple creation with new lower-level forms that we use in the translation of tuple creation. Figure 7.11 shows the grammar for $\mathcal{L}^{mon}_{\mathsf{Alloc}}$.

The `collect(`n`)` form runs the garbage collector, requesting that there be n bytes ready to be allocated. During instruction selection, the `collect(`n`)` form will become a call to the `collect` function in `runtime.c`. The `allocate(`n`,`*type*`)` form obtains memory for n elements (and space at the front for the 64-bit tag), but the elements are not initialized. The *type* parameter is the type of the tuple: `TupleType([`$type_1$`, ...,`$type_n$`])` where $type_i$ is the type of the ith element. The `global_value(`*name*`)` form reads the value of a global variable, such as `free_ptr`.

The following shows the transformation of tuple creation into (1) a sequence of temporary variable bindings for the initializing expressions, (2) a conditional call to `collect`, (3) a call to `allocate`, and (4) the initialization of the tuple. The *len* placeholder refers to the length of the tuple, and *bytes* is the total number of bytes that need to be allocated for the tuple, which is 8 for the tag plus *len* times 8. The *type* needed for the second argument of the `allocate` form can be obtained from the `has_type` field of the tuple AST node, which is stored there by running the type checker for $\mathcal{L}_{\mathsf{Tup}}$ immediately before this pass.

```
v1 = {
  newp.2 = (free_ptr + 16)
  if newp.2 < fromspace_end:
  else:
    collect(16)
  tuple.3 = allocate(1,tuple[int])
  tuple.3[0] = 42
  produce tuple.3}
v2 = {
  newp.4 = (free_ptr + 16)
  if newp.4 < fromspace_end:
  else:
    collect(16)
  tuple.5 = allocate(1,tuple[tuple[int]])
  tuple.5[0] = v1
  produce tuple.5}
tmp.0 = v2[0]
tmp.1 = tmp.0[0]
print(tmp.1)
```

Figure 7.10
Output of the `expose_allocation` pass.

```
(a0, ..., an-1)
⟹
{
    p = global_value(free_ptr) + bytes
    if p < global_value(fromspace_end):
        0
    else:
        collect(bytes)
    v = allocate(len, type)
    v[0] = a0
         ⋮
    v[n-1] = an-1
    produce v
}
```

It is important that we sequence the expose allocation pass *after* the removing complex subexpressions pass, because this guarantees that the fields of the tuple have already been evaluated to atoms prior to the call to `allocate`. If we were instead to perform the `allocate` first and then compute the values of the fields, those computations might themselves trigger a garbage collection, but we must not have an allocated but uninitialized tuple on the heap during a collection!

Figure 7.10 shows the output of the `expose_allocation` pass on our running example.

$$
\begin{array}{rcl}
\mathit{atm} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Name(}\mathit{var}\texttt{)} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{Call(Name('input\_int'),[])} \\
 & \mid & \texttt{UnaryOp(USub(),}\mathit{atm}\texttt{)} \mid \texttt{BinOp(}\mathit{atm}\texttt{,Add(),}\mathit{atm}\texttt{)} \\
 & \mid & \texttt{BinOp(}\mathit{atm}\texttt{,Sub(),}\mathit{atm}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\mathit{atm}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\
 & \mid & \texttt{Assign([Name(}\mathit{var}\texttt{)], }\mathit{exp}\texttt{)} \\
\hline
\mathit{atm} & ::= & \texttt{Constant(}\mathit{bool}\texttt{)} \\
\mathit{exp} & ::= & \texttt{UnaryOp(Not(),}\mathit{exp}\texttt{)} \mid \texttt{Compare(}\mathit{atm}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{atm}\texttt{])} \\
 & \mid & \texttt{IfExp(}\mathit{exp}\texttt{,}\mathit{exp}\texttt{,}\mathit{exp}\texttt{)} \mid \texttt{Begin(}\mathit{stmt}^*\texttt{, }\mathit{exp}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{If(}\mathit{exp}\texttt{, }\mathit{stmt}^*\texttt{, }\mathit{stmt}^*\texttt{)} \\
\hline
\mathit{stmt} & ::= & \texttt{While(}\mathit{exp}\texttt{, }\mathit{stmt}^+\texttt{, []})
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{atm} & ::= & \texttt{GlobalValue(}\mathit{var}\texttt{)} \\
\mathit{exp} & ::= & \texttt{Subscript(}\mathit{atm}\texttt{,}\mathit{atm}\texttt{,Load())} \mid \texttt{Allocate(}\mathit{int}\texttt{,}\mathit{type}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Assign([Subscript(}\mathit{atm}\texttt{,}\mathit{atm}\texttt{,Store())], }\mathit{atm}\texttt{)} \\
 & \mid & \texttt{Collect(}\mathit{int}\texttt{)} \\
\mathcal{L}^{mon}_{\mathsf{Alloc}} & ::= & \texttt{Module(}\mathit{stmt}^*\texttt{)}
\end{array}
$$

Figure 7.11
$\mathcal{L}^{mon}_{\mathsf{Alloc}}$ is $\mathcal{L}_{\mathsf{Alloc}}$ in monadic normal form.

$$
\begin{array}{rcl}
\mathit{atm} & ::= & \texttt{Constant(}\mathit{int}\texttt{)} \mid \texttt{Name(}\mathit{var}\texttt{)} \mid \texttt{Constant(}\mathit{bool}\texttt{)} \\
\mathit{exp} & ::= & \mathit{atm} \mid \texttt{Call(Name('input\_int'),[])} \mid \texttt{UnaryOp(USub(),}\mathit{atm}\texttt{)} \\
 & \mid & \texttt{BinOp(}\mathit{atm}\texttt{,Sub(),}\mathit{atm}\texttt{)} \mid \texttt{BinOp(}\mathit{atm}\texttt{,Add(),}\mathit{atm}\texttt{)} \\
 & \mid & \texttt{Compare(}\mathit{atm}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{atm}\texttt{])} \\
\mathit{stmt} & ::= & \texttt{Expr(Call(Name('print'),[}\mathit{atm}\texttt{]))} \mid \texttt{Expr(}\mathit{exp}\texttt{)} \\
 & \mid & \texttt{Assign([Name(}\mathit{var}\texttt{)], }\mathit{exp}\texttt{)} \\
\mathit{tail} & ::= & \texttt{Return(}\mathit{exp}\texttt{)} \mid \texttt{Goto(}\mathit{label}\texttt{)} \\
 & \mid & \texttt{If(Compare(}\mathit{atm}\texttt{,[}\mathit{cmp}\texttt{],[}\mathit{atm}\texttt{]), [Goto(}\mathit{label}\texttt{)], [Goto(}\mathit{label}\texttt{)])} \\
\hline
\mathit{atm} & ::= & \texttt{GlobalValue(}\mathit{var}\texttt{)} \\
\mathit{exp} & ::= & \texttt{Subscript(}\mathit{atm}\texttt{,}\mathit{atm}\texttt{,Load())} \mid \texttt{Allocate(}\mathit{int}\texttt{,}\mathit{type}\texttt{)} \\
\mathit{stmt} & ::= & \texttt{Collect(}\mathit{int}\texttt{)} \mid \texttt{Assign([Subscript(}\mathit{atm}\texttt{,}\mathit{atm}\texttt{,Store())], }\mathit{atm}\texttt{)} \\
\mathcal{C}_{\mathsf{Tup}} & ::= & \texttt{CProgram(\{}\mathit{label}\texttt{: }\mathit{stmt}^*\ \mathit{tail}\texttt{, ... \})}
\end{array}
$$

Figure 7.12
The abstract syntax of $\mathcal{C}_{\mathsf{Tup}}$, extending $\mathcal{C}_{\mathsf{If}}$ (figure 5.8).

7.5 Explicate Control and the $\mathcal{C}_{\mathsf{Tup}}$ Language

The output of `explicate_control` is a program in the intermediate language $\mathcal{C}_{\mathsf{Tup}}$, for which figure 7.12 shows the definition of the abstract syntax. The new expressions of $\mathcal{C}_{\mathsf{Tup}}$ include `allocate`, accessing tuple elements, and `global_value`. $\mathcal{C}_{\mathsf{Tup}}$ also includes the `collect` statement and assignment to a tuple element. The `explicate_control` pass can treat these new forms much like the other forms that we've already encountered. The output of the `explicate_control` pass on the running example is shown on the left side of figure 7.15 in the next section.

7.6 Select Instructions and the $\text{x86}_{\text{Global}}$ Language

In this pass we generate x86 code for most of the new operations that are needed to compile tuples, including `Allocate`, `Collect`, accessing tuple elements, and the `Is` comparison. We compile `GlobalValue` to `Global` because the latter has a different concrete syntax (see figures 7.13 and 7.14).

The tuple read and write forms translate into `movq` instructions. (The `+1` in the offset serves to move past the tag at the beginning of the tuple representation.)

```
lhs = tup[n]
==>
movq tup', %r11
movq 8(n+1)(%r11), lhs'

tup[n] = rhs
==>
movq tup', %r11
movq rhs', 8(n+1)(%r11)
```

The *tup′* and *rhs′* are obtained by translating from $\mathcal{C}_{\mathsf{Tup}}$ to x86. The move of *tup′* to register `r11` ensures that the offset expression `8(n+1)(%r11)` contains a register operand. This requires removing `r11` from consideration by the register allocator.

Why not use `rax` instead of `r11`? Suppose that we instead used `rax`. Then the generated code for tuple assignment would be

```
movq tup', %rax
movq rhs', 8(n+1)(%rax)
```

Next, suppose that *rhs′* ends up as a stack location, so `patch_instructions` would insert a move through `rax` as follows:

```
movq tup', %rax
movq rhs', %rax
movq %rax, 8(n+1)(%rax)
```

However, this sequence of instructions does not work because we're trying to use `rax` for two different values (*tup′* and *rhs′*) at the same time!

~~The len operation should be translated into a sequence of instructions that read the tag of the tu~~

We compile the `allocate` form to operations on the `free_ptr`, as shown next. This approach is called *inline allocation* because it implements allocation without a function call by simply incrementing the allocation pointer. It is much more efficient than calling a function for each allocation. The address in the `free_ptr` is the next free address in the FromSpace, so we copy it into `r11` and then move it forward by enough space for the tuple being allocated, which is $8(len+1)$ bytes because each element is 8 bytes (64 bits) and we use 8 bytes for the tag. We then initialize the *tag* and finally copy the address in `r11` to the left-hand side. Refer to figure 7.8 to see how the tag is organized. We recommend using the bitwise-or operator `|` and the shift-left operator `«` to compute the tag during compilation. The type annotation in the `allocate` form is used to determine the pointer mask region of

the tag. The addressing mode `free_ptr(%rip)` essentially stands for the address of the `free_ptr` global variable using a special instruction-pointer-relative addressing mode of the x86-64 processor. In particular, the assembler computes the distance d between the address of `free_ptr` and where the `rip` would be at that moment and then changes the `free_ptr(%rip)` argument to d`(%rip)`, which at runtime will compute the address of `free_ptr`.

```
lhs = allocate(len, TupleType([type, ...]));
⟹
movq free_ptr(%rip), %r11
addq 8(len+1), free_ptr(%rip)
movq $tag, 0(%r11)
movq %r11, lhs'
```

The `collect` form is compiled to a call to the `collect` function in the runtime. The arguments to `collect` are (1) the top of the root stack, and (2) the number of bytes that need to be allocated. We use another dedicated register, `r15`, to store the pointer to the top of the root stack. Therefore `r15` is not available for use by the register allocator.

```
collect(bytes)
⟹
movq %r15, %rdi
movq $bytes, %rsi
callq collect
```

The `is` comparison is compiled similarly to the other comparison operators, using the `cmpq` instruction. Because the value of a tuple is its address, we can translate `is` into a simple check for equality using the `e` condition code.

var `=` `(`atm_1 `is` atm_2`)` $\Rightarrow$

```
cmpq arg2, arg1
sete %al
movzbq %al, var
```

The definitions of the concrete and abstract syntax of the $\text{x86}_{\text{Global}}$ language are shown in figures 7.13 and 7.14. It differs from x86_{If} only in the addition of global variables. Figure 7.15 shows the output of the `select_instructions` pass on the running example.

```
reg    ::= rsp | rbp | rax | rbx | rcx | rdx | rsi | rdi |
           r8 | r9 | r10 | r11 | r12 | r13 | r14 | r15
arg    ::= $int | %reg | int(%reg)
instr  ::= addq arg,arg | subq arg,arg | negq arg | movq arg,arg |
           pushq arg | popq arg | callq label | retq | jmp label |
           label: instr
-------------------------------------------------------------------
bytereg ::= ah | al | bh | bl | ch | cl | dh | dl
arg     ::= %bytereg
cc      ::= e | ne | l | le | g | ge
instr   ::= xorq arg, arg | cmpq arg, arg | setcc arg | movzbq arg, arg
          | jcc label
-------------------------------------------------------------------
arg    ::= label(%rip)
x86Global ::= .globl main
              main: instr*
```

Figure 7.13
The concrete syntax of $x86_{\text{Global}}$ (extends $x86_{\text{If}}$ shown in figure 5.9).

```
reg     ::= rsp | rbp | rax | rbx | rcx | rdx | rsi | rdi |
            r8 | r9 | r10 | r11 | r12 | r13 | r14 | r15
arg     ::= Immediate(int) | Reg(reg) | Deref(reg,int)
instr   ::= Instr('addq',[arg,arg]) | Instr('subq',[arg,arg])
          | Instr('negq',[arg]) | Instr('movq',[arg,arg])
          | Instr('pushq',[arg]) | Instr('popq',[arg])
          | Callq(label,int) | Instr('retq',[]) | Jump(label)
block   ::= instr+
-------------------------------------------------------------------
bytereg ::= 'ah' | 'al' | 'bh' | 'bl' | 'ch' | 'cl' | 'dh' | 'dl'
arg     ::= Immediate(int) | Reg(reg) | Deref(reg,int) | ByteReg(bytereg)
cc      ::= 'e' | 'ne' | 'l' | 'le' | 'g' | 'ge'
instr   ::= Jump(label)
          | Instr('xorq',[arg,arg]) | Instr('cmpq',[arg,arg])
          | Instr('set'+cc,[arg]) | Instr('movzbq',[arg,arg])
          | JumpIf(cc,label)
-------------------------------------------------------------------
arg    ::= Global(label)
x86Global ::= X86Program({label : block, ... })
```

Figure 7.14
The abstract syntax of $x86_{\text{Global}}$ (extends $x86_{\text{If}}$ shown in figure 5.10).

```
block.6:
    tuple.5 = allocate(1,tuple[tuple[int]])
    tuple.5[0] = v1
    v2 = tuple.5
    tmp.0 = v2[0]
    tmp.1 = tmp.0[0]
    print(tmp.1)
    return 0

block.7:
    collect(16)
    goto block.6

block.8:
    tuple.3 = allocate(1,tuple[int])
    tuple.3[0] = 42
    v1 = tuple.3
    newp.4 = (free_ptr + 16)
    if newp.4 < fromspace_end:
      goto block.6
    else:
      goto block.7

block.9:
    collect(16)
    goto block.8

start:
    newp.2 = (free_ptr + 16)
    if newp.2 < fromspace_end:
      goto block.8
    else:
      goto block.9
```

⇒

```
block.6:
    movq free_ptr(%rip), %r11
    addq $16, free_ptr(%rip)
    movq $131, 0(%r11)
    movq %r11, tuple.5
    movq tuple.5, %r11
    movq v1, 8(%r11)
    movq tuple.5, v2
    movq v2, %r11
    movq 8(%r11), tmp.0
    movq tmp.0, %r11
    movq 8(%r11), tmp.1
    movq tmp.1, %rdi
    callq print_int
    movq $0, %rax
    jmp conclusion

block.7:
    movq %r15, %rdi
    movq $16, %rsi
    callq _collect
    jmp block.6

b0lock.8:
    movq free_ptr(%rip), %r11
    addq $16, free_ptr(%rip)
    movq $3, 0(%r11)
    movq %r11, tuple.3
    movq tuple.3, %r11
    movq $42, 8(%r11)
    movq tuple.3, v1
    movq free_ptr(%rip), newp.4
    addq $16, newp.4
    cmpq fromspace_end(%rip), newp.4
    jl block.6
    jmp block.7

block.9:
    movq %r15, %rdi
    movq $16, %rsi
    callq collect
    jmp block.8

start:
    movq free_ptr(%rip), newp.2
    addq $16, newp.2
    cmpq fromspace_end(%rip), newp.2
    jl block.8
    jmp block.9
```

Figure 7.15
Output of `explicate_control` (*left*) and `select_instructions` (*right*) on the running example.

7.7 Register Allocation

As discussed previously in this chapter, the garbage collector needs to access all the pointers in the root set, that is, all variables that are tuples. It will be the responsibility of the register allocator to make sure that

1. the root stack is used for spilling tuple-typed variables, and
2. if a tuple-typed variable is live during a call to the collector, it must be spilled to ensure that it is visible to the collector.

The latter responsibility can be handled during construction of the interference graph, by adding interference edges between the call-live tuple-typed variables and all the callee-saved registers. (They already interfere with the caller-saved registers.) The type information for variables is generated by the type checker for $\mathcal{C}_{\mathsf{Tup}}$, stored in a field named `var_types` in the `CProgram` AST mode. You'll need to propagate that information so that it is available in this pass.

The spilling of tuple-typed variables to the root stack can be handled after graph coloring, in choosing how to assign the colors (integers) to registers and stack locations. The `CProgram` output of this pass changes to also record the number of spills to the root stack.

7.8 Generate Prelude and Conclusion

Figure 7.16 shows the output of the `prelude_and_conclusion` pass on the running example. In the prelude of the `main` function, we allocate space on the root stack to make room for the spills of tuple-typed variables. We do so by incrementing the root stack pointer (`r15`), taking care that the root stack grows up instead of down. For the running example, there was just one spill, so we increment `r15` by 8 bytes. In the conclusion we subtract 8 bytes from `r15`.

One issue that deserves special care is that there may be a call to `collect` prior to the initializing assignments for all the variables in the root stack. We do not want the garbage collector to mistakenly determine that some uninitialized variable is a pointer that needs to be followed. Thus, we zero out all locations on the root stack in the prelude of `main`. In figure 7.16, the instruction `movq $0, 0(%r15)` is sufficient to accomplish this task because there is only one spill. In general, we have to clear as many words as there are spills of tuple-typed variables. The garbage collector tests each root to see if it is null prior to dereferencing it.

Figure 7.17 gives an overview of all the passes needed for the compilation of $\mathcal{L}_{\mathsf{Tup}}$.

```
    .globl main
main:
    pushq %rbp
    movq %rsp, %rbp
    pushq %rbx
    subq $8, %rsp
    movq $65536, %rdi
    movq $16, %rsi
    callq initialize
    movq rootstack_begin(%rip), %r15
    movq $0, 0(%r15)
    addq $8, %r15
    jmp start

conclusion:
    subq $8, %r15
    addq $8, %rsp
    popq %rbx
    popq %rbp
    retq
```

Figure 7.16
The prelude and conclusion for the running example.

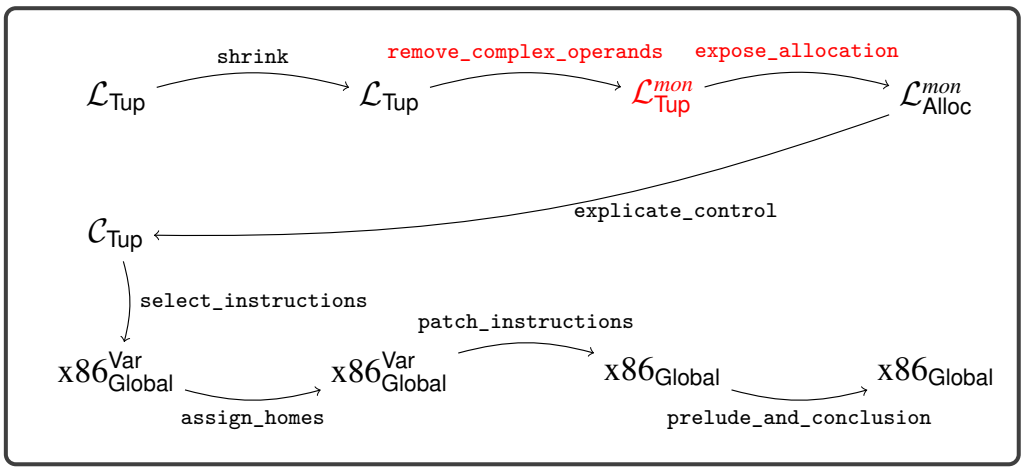

Figure 7.17
Diagram of the passes for $\mathcal{L}_{\text{Tup}}$, a language with tuples.

7.9 ~~Challenge: Arrays~~

7.10 Further Reading

Appel (1990) describes many data representation approaches including the ones used in the compilation of Standard ML.

There are many alternatives to copying collectors (and their bigger siblings, the generational collectors) with regard to garbage collection, such as mark-and-sweep (McCarthy 1960) and reference counting (Collins 1960). The strengths of copying collectors are that allocation is fast (just a comparison and pointer increment), there is no fragmentation, cyclic garbage is collected, and the time complexity of collection depends only on the amount of live data and not on the amount of garbage (Wilson 1992). The main disadvantages of a two-space copying collector is that it uses a lot of extra space and takes a long time to perform the copy, though these problems are ameliorated in generational collectors. Object-oriented programs tend to allocate many small objects and generate a lot of garbage, so copying and generational collectors are a good fit (Dieckmann and Hölzle 1999). Garbage collection is an active research topic, especially concurrent garbage collection (Tene, Iyengar, and Wolf 2011). Researchers are continuously developing new techniques and revisiting old trade-offs (Blackburn, Cheng, and McKinley 2004; Jones, Hosking, and Moss 2011; Shahriyar et al. 2013; Cutler and Morris 2015; Shidal et al. 2015; Österlund and Löwe 2016; Jacek and Moss 2019; Gamari and Dietz 2020). Researchers meet every year at the International Symposium on Memory Management to present these findings.