Compiler Construction

Tuples & Garbage Collection

Hannes Saffrich

University of Freiburg
Department of Computer Science
Programming Languages

4. Juni 2024

Tuples

- Tuples are immutable, heterogeneous lists, e.g. `(10, True, 30)`
- Immutable: no entries can be removed, added, or changed
- Heterogeneous: the entries can have different types
- The type of a tuple is a tuple of its component types, e.g.
  `(10, True, 20)  :  tuple[int, bool, int]`
- We model the following operations on tuples:

```
t = (10, True, 20)   # Tuple literals
x = t[1]             # Element Access; x is True
y = len(t)           # Length of a tuple; y is 3
```

- We restrict element access to constant indices due to static typing
  - `t[1]` has type `int`
  - `t[f()]` is forbidden, because type checking would be undecidable for arbitrary turing complete functions `f`

Tuples: Memory Layout

- With dynamic typing, tuples need to be allocated on the heap
- With static typing, tuples can also be allocated on the stack
- The book allocates tuples on the heap, as they later also introduce optional dynamic typing
- We follow the book to showcase how heap allocations work

Tuples: Static Memory Layout

- As tuples or structs in C/++ or Rust

```
fn f() {
    // 10 and 20 are placed next to each other on the stack
    let x = (10, 20);

    // 10 and 20 are both passed as part of the argument
    g(x);

    // A pointer to the begin of the tuple is passed as argument
    // which allows h to read the tuple out of f's stack frame
    h(&x);
}
```

Tuples: Dynamic Memory Layout

- As tuples in Python or $DYNAMICALLY_TYPED_LANGUAGE

```
def f():
    # A new memory region is allocated on the heap.
    # 10 and 20 are placed next to each other in that region.
    # x contains a pointer to the beginning of that region.
    x = (10, 20)

    # The pointer to the tuple is passed as an argument
    g(x)

    # If f returns, the stack contains no pointer to the
    # memory region, which a garbage collector will
    # eventually detect and free the memory again
    # (assuming g doesn't store the pointer in a global variable)
```

Managing Heap Memory

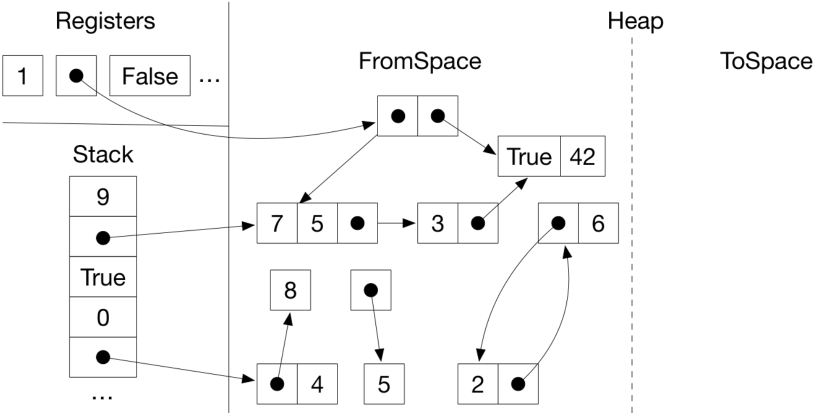
- Manual Memory Management
  - the user is responsible for allocating and freeing heap memory
  - can be achieved with functions from the C standard library
  - `malloc` takes a number of bytes as an argument, finds a free region on the heap of that size, marks it as used, and returns a pointer to that region.
  - `free` takes a pointer, which was previously returned by `malloc`, and marks that memory region as free again, such that it can be reused by subsequent calls to `malloc`.
- Automatic Memory Management
  - Garbage collectors only provide a function to allocate memory, and periodically scan for memory regions, which cannot be reached anymore, and free them automatically.
  - Garbage collectors trade runtime performance for memory safety and simplicity.

Garbage Collection: Two-Space Copying Collector

- This is the garbage collector we use
- Divides the heap into two regions: the *from-space* and *to-space*
- Allocated memory is taken sequentially from the from space
- If the from-space runs out of memory $\rightarrow$ garbage collection:
  - Find objects that are reachable transitively from the stack
  - Reachable objects are copied from *from-space* to *to-space*
  - Pointers in stack and objects are adjusted accordingly
  - The *from-space* becomes the new *to-space*, and vice versa

Garbage Collection: Example

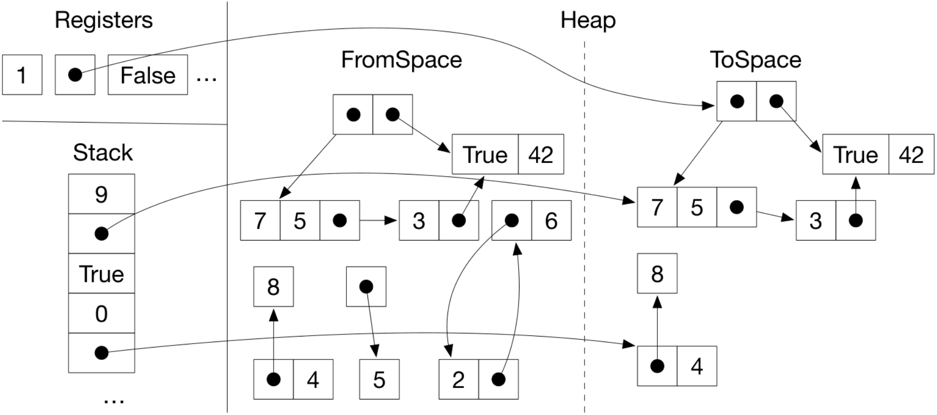
Before Collection



[Essentials of Compilations, Jeremy Siek, Figure 7.5]

Garbage Collection: Example

After Collection



[Essentials of Compilations, Jeremy Siek, Figure 7.5]

Garbage Collection: Pointer or Integer?

- Garbage collector (GC) needs to identify pointers to heap space
- But integers and pointers are indistinguishable in memory:
  - 32 could be a number $\rightarrow$ GC should ignore it
  - 32 could be the address to heap space $\rightarrow$ GC should follow it
- This affects both values stored on the stack and values stored inside of heap objects, e.g. tuple entries
- The book addresses this issue by having a second stack for heap pointers ("shadow stack"), and adding metadata to the tuple objects describing, which tuple entries are heap pointers
- We take a different approach that we call *value tagging*

Garbage Collection: Value Tagging

- All our values take exactly 64bit of space:
  - Integers
  - Booleans
  - Pointers (to tuples on the heap)
- Crazy idea: can we steal one of the 64 bits and use it as a tag?
- 1 would mean "heap pointer"; 0 would mean "no heap pointer"
- The answer is "yes" by using a sufficient amount of black magic!
  - Instead of 64 bit integers, we have 63 bit integers
  - We still have the full range of pointers (!?)
  - Booleans have lots of unused bits anyways
  - Only minimal adjustments in code generation!
  - We choose the *least* significant bit to encode the tag

Garbage Collection: Value Tagging for Integers

- Integers are not heap pointers, so the tag bit should be `0`
- Example: `5` was previously encoded as `101`, but now it is `1010`
- Additon, subtraction, negation, and multiplication still work!
- Key Insight: adding a `0` at the right means multiplying by two
- Example for Addition:
  - Let's say we want to add two numbers x and y
  - Then they are encoded as $2x$ and $2y$
  - Adding them is $2x + 2y = 2(x + y)$ which is the encoding of $x + y$
  - The overflow behaviour is also as expected for 63 bit integers
- Only caveat: When calling a C-function with integer arguments (like `print_int`), the tags need to be removed by right-shifting

Garbage Collection: Value Tagging for Booleans

- Booleans are not heap pointers, so the tag bit should be `0`
- Example: `True` was previously encoded as `1`, but now it is `10`
- Key Insight: Logic instructions are bitwise
- Conjunction and Disjunction still work the same!
  - it doesn't matter which bit we use for the actual boolean data
  - tag bit remains `0` as all operands have a `0` tag bit
- Negation does *not* work the same:
  - bitwise `not` also flips the tag bit
  - but we can compute `not x` as `2 - x`!

Garbage Collection: Value Tagging for Pointers

- Pointers can be heap pointers (tuples) or not (spilled framepointer)
- Key Insight: All addresses are multiples of 8 due to alignment
- This means the three least significant bits are always 0
- Non-Heap-Pointers behave as before:
  - Let's say register `a0` contains a non-heap-pointer
  - Then `ld a1, 0(a0)` loads the corresponding word into `a1`
- Heap-Pointers can be adjusted via offset:
  - Let's say register `a0` contains a heap-pointer
  - Then `ld a1, -1(a0)` loads the corresponding word into `a1`

Garbage Collection: Value Tagging and FFI

- What about calling C-functions like `print_int64` or `input_int64`?
- Tag needs to be removed from arguments and added to return values
- Integers and booleans
  - need to be divided by two, before used as arguments
  - need to be multiplied by two, after retrieved as a return values
- Heap-Pointers
  - need to have their tag bit set to 0, before used as arguments
  - need to have their tag bit set to 1, after retrieved as a return values

Garbage Collection: Cycles

- How should the garbage collector deal with cycles in the heap graph?
- Does not happen with tuples, but could happen for more complicated data structures, like graphs, where each node contains a pointers to all neighbors
- During collection the GC needs to track which object it already visited
- Idea: Each heap object starts with an extra word with meta data
- The least significant bit of that word is used during garbage collection to store whether this object has already been copied or not
  - If the bit is 0, then it hasn't been copied, and the other 63 bits store the word length of the object
  - If the bit is 1, then it has already been copied, and the other 63 bits store the address to where it has been copied
- If the GC finds a pointer to an object that was not yet copied, it uses the word length to know how much it needs to copy
- If the GC finds a pointer to an object that was already copied, it changes the pointer to the new address.

Garbage Collection: Detailed Example

- Let's consider the following program:

```
i = 42      # no allocation
x = (0,)    # 2 words = 16 bytes (length + 1 entry)
x = (1,)    # 2 words = 16 bytes (length + 1 entry)
y = (x,2)   # 3 words = 24 bytes (length + 2 entries)
z = (3,)    # 2 words = 16 bytes (length + 1 entry)
```

- Let's assume from- and to-space are both 8 words = 64 bytes long
- Let's assume all variables are spilled to the stack
- When `z = (3,)` is executed, we run out of space
- Garbage collector kicks in and copies `(1,)` and `(x,2)`, but not `(0,)`
- After collection, 3 words are free, so the object for `(5, x)` is allocated

Garbage Collection: Detailed Example

- Current Statement: `i = 42`

Stack

Loc	63bit	Tag
i	42	0

FromSpace

Loc	63bit	Tag

ToSpace

Loc	63bit	Tag

Garbage Collection: Detailed Example

- Current Statement: `x = (0,)`

Stack

Loc	63bit	Tag
i	42	0
x	•	1

FromSpace

Loc	63bit	Tag
meta	1	0
entry_1	0	0

ToSpace

Loc	63bit	Tag

Garbage Collection: Detailed Example

- Current Statement: `x = (1,)`

Stack

Loc	63bit	Tag
i	42	0
x	•	1

FromSpace

Loc	63bit	Tag
meta	1	0
$entry_1$	0	0
meta	1	0
$entry_1$	1	0

ToSpace

Loc	63bit	Tag

Garbage Collection: Detailed Example

- Current Statement: `y = (x,2)`

Stack

Loc	63bit	Tag
i	42	0
x	•	1
y	•	1

FromSpace

Loc	63bit	Tag
meta	1	0
$entry_1$	0	0
meta	1	0
$entry_1$	1	0
meta	2	0
$entry_1$	•	1
$entry_2$	2	0

ToSpace

Loc	63bit	Tag

Garbage Collection: Detailed Example

- Current Statement: `z = (3,)`

Stack

Loc	63bit	Tag
i	42	0
x	•	1
y	•	1

FromSpace

Loc	63bit	Tag
meta	1	0
entry_1	0	0
meta	1	0
entry_1	1	0
meta	2	0
entry_1	•	1
entry_2	2	0

ToSpace

Loc	63bit	Tag

- Not enough space → Garbage Collection
- Stack is scanned for heap pointers
- By looking at the tags, it finds that x and y contain heap pointers

Garbage Collection: Detailed Example

- Current Statement: `z = (3,)`

Stack

Loc	63bit	Tag
i	42	0
x	•	1
y	•	1

FromSpace

Loc	63bit	Tag
meta	1	0
$entry_1$	0	0
meta	•	1
$entry_1$	1	0
meta	2	0
$entry_1$	•	1
$entry_2$	2	0

ToSpace

Loc	63bit	Tag
meta	1	0
$entry_1$	1	0

- The object behind x got copied to ToSpace
- The pointer in x was updated
- The old object was marked as "already copied" and a *forwarding pointer* was stored in the meta information

Garbage Collection: Detailed Example

- Current Statement: `z = (3,)`

Stack

Loc	63bit	Tag
i	42	0
x	●	1
y	●	1

FromSpace

Loc	63bit	Tag
meta	1	0
$entry_1$	0	0
meta	●	1
$entry_1$	1	0
meta	●	1
$entry_1$	●	1
$entry_2$	2	0

ToSpace

Loc	63bit	Tag
meta	1	0
$entry_1$	1	0
meta	2	0
$entry_1$	●	1
$entry_2$	2	0

- The object behind y got copied to ToSpace
- The pointer in y was updated
- The old object was marked as “already copied” and a *forwarding pointer* was stored in the meta information

Garbage Collection: Detailed Example

- Current Statement: `z = (3,)`

Stack

Loc	63bit	Tag
i	42	0
x	•	1
y	•	1

FromSpace

Loc	63bit	Tag
meta	1	0
$entry_1$	0	0
meta	•	1
$entry_1$	1	0
meta	•	1
$entry_1$	•	1
$entry_2$	2	0

ToSpace

Loc	63bit	Tag
meta	1	0
$entry_1$	1	0
meta	2	0
$entry_1$	•	1
$entry_2$	2	0

- All objects pointed to from the stack have been copied
- Now the objects in the to-space are scanned
- The second object contains a heap pointer
- That heap pointer points to an already copied object
- The heap pointer is replaced with the forwarding pointer

Garbage Collection: Detailed Example

- Current Statement: `z = (3,)`

Stack

Loc	63bit	Tag
i	42	0
x	•	1
y	•	1

ToSpace

Loc	63bit	Tag
meta	1	0
$entry_1$	0	0
meta	•	1
$entry_1$	1	0
meta	•	1
$entry_1$	•	1
$entry_2$	2	0

FromSpace

Loc	63bit	Tag
meta	1	0
$entry_1$	1	0
meta	2	0
$entry_1$	•	1
$entry_2$	2	0

- Garbage collection finishes by treating the ToSpace as FromSpace and vice versa

Garbage Collection: Detailed Example

- Current Statement: `z = (3,)`

Stack

Loc	63bit	Tag
i	42	0
x	•	1
y	•	1
z	•	1

ToSpace

Loc	63bit	Tag
meta	1	0
$entry_1$	0	0
meta	•	1
$entry_1$	1	0
meta	•	1
$entry_1$	•	1
$entry_2$	2	0

FromSpace

Loc	63bit	Tag
meta	1	0
$entry_1$	1	0
meta	2	0
$entry_1$	•	1
$entry_2$	2	0
meta	1	0
$entry_1$	3	0

- Finally, the current statement finishes by allocating the tuple

Garbage Collection: Registers

- In the example, we assumed all registers are spilled
- This does not need to be the case in practice
- To ensure that the GC updates all pointers, all registers containing heap pointers need to be spilled before running the GC
- Safe and simple: make all registers interfere with calls to the GC

Garbage Collection: Size of From- and To-Space

- GCs need to reserve the memory for from- & to-space upfront
- Reserving all available memory is a bad idea:
  - only a small amount of memory might be needed
  - reserved memory is not available to other processes
- Solution: Start with a small size for from- and to-space and reallocate on demand
- Can be done in amortized constant time using the same technique as for `std::vector` in C++, `Vec` in Rust, or `list` in python
- If after garbage collection with a from- and to-space of `m` bytes, we still need `n` more bytes, then the new from- and to-space are `p` bytes large, where `p` is the next power of two larger than `m + n`
- Copying from the old from-space to the new from-space is the same as copying from from-space to to-space when collecting garbage

Garbage Collection: API

- The garbage collector is part of the runtime and implemented in C
- It provides the following API:

```
int64_t* gc_free_ptr;        // first free byte on from-space
int64_t* gc_fromspace_end;   // first byte after from-space
void gc_init(
    int64_t* stack_begin,    // stack pointer at begin of main
    uint64_t heap_size       // initial size of from-/to-space
);
void gc_collect(
    int64_t* stack_end,          // current stack pointer
    uint64_t requested_bytes     // to reallocate from-/to-space
);                               // if collection didn't free
                                 // enough space
```

- The pointers are regular C-pointers, i.e. no value tagging

Global Variables

- The following C-program

```
int x = 42;
int main() { return x; }
```

can be compiled to

```
    .data
x:
    .word 42
    .text
    .globl main
main:
    la a0, x
    ld a0, 0(a0)
    ret
```

- Addresses of global variables are accessed via labels
- Their data is put into the `.data`-Segment of a program
- The load address instruction `la` retrieves the address of a label

Tuple Compilation

- Tuple expressions need to be translated to allocation, potential garbage collection, and entry initialization
  - New pass early in the compilation pipeline, such that we can use high-level features like `if`-statements
  - Initializing tuple entries requires the intermediate languages to allow subscripts in the left-hand-sides of assignments, e.g. `x[1] = y`
  - Instruction selection pass compiles the allocation and garbage collection expressions into call instructions, load and store instructions
  - Expressions for global variables need to be added
- Tuple subscripts, e.g. `x = y[i]`, simply retrieve the corresponding entry value of the tuple object using register offsets, e.g. `-9(a0)`
- Tuple length, e.g. `x = len(y)`, retrieves the meta data of the tuple object using register offsets and then shifts right to remove the GC tag

Tuple Compilation: New Pass

The python code

```
x = (1, 2)
```

is translated to

```
e1 = 1
e2 = 2
if free_ptr + num_bytes < fromspace_end:
else:
    collect (num_bytes)
t = allocate(num_bytes)
t [0] = e1
t [1] = e2
x = t
```

where `num_bytes = 8 + 8 * 2`

Tuple Compilation: Instruction Selection

- `x = allocate(n)` is translated to

  addi x 0(free_ptr) 1
  add free_ptr 0(free_ptr) (8 + 8 * n)
  mv −1(x) METADATA

  - `free_ptr` represents the *address* of the global variable
  - label offsets, e.g. `0(free_ptr)`, are not allowed in RISC-V and need to be translated to load address `la` and load instructions with offsets in `patch_instructions`

- `collect(n)` is translated to

  mv a0 sp
  mv a1 n
  call gc_collect

Tuple Compilation: Main Prelude

- In the prelude of the `main` function, we need to initialize the garbage collector by calling `gc_init`

Tuple Compilation: Interaction with `while`

- Tuple expressions can appear as `while` conditions, e.g.

```
x = 0
while (x, 1)[0] < 10:
    x = x + 1
```

- While non-sensical, we need to take care of such expressions
- Allocation needs to happen for each iteration, and not just the first
- Same trick as before, but already in earlier compiler passes:

```
@dataclass
class SWhile:
    test_body: IList[Stmt]
    test_expr: ECompare
    loop_body: IList[Stmt]
```