Utrecht University

Concepts of Programming Language Design

Reference Types

Gabriele Keller
Tom Smeding

Reference types

- Variables in TinyC represent values stored in a fixed memory location
  - assigning a new value to a variable updated the value in that location
- Reference types refer to a location a value is stored
- Reference types are usually implemented as pointers, that is as address into the memory of a process (often with some associated meta data, such as the size of the data pointed to)
- Most high-level languages support or use reference types in one way or another
  - explicitly, in an abstract way: only expose the interface (creation, read and write a value)
  - implicitly, using them behind the scenes to implement data structures
  - explicitly, exposing the implementation as pointer: C

Reference types vs pointers

- A pointer:
  - machine address
  - some meta data (size & stats of the data it points to)
- A reference is an abstract data type:
  - we have a set of operations defined on it
    - create, read, write, update
  - often represented internally as pointer, but no guarantee that the address stays constant, runtime system may relocate the data

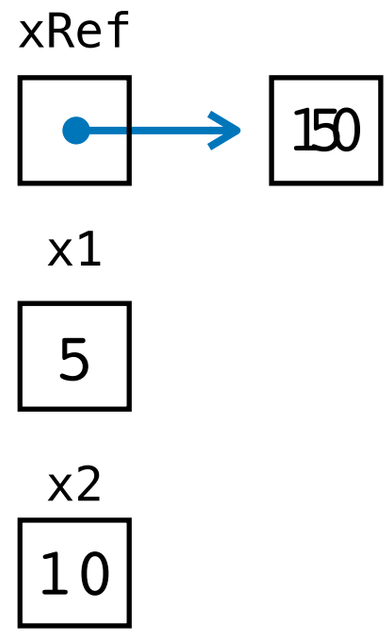
Reference types

- Haskell has not explicit built-in reference types, but `Data.IORef` provides it as abstract data type:

```
newIORef    :: a                -> IO (IORef a)
writeIORef :: a -> IORef a -> IO ()
readIORef   :: IORef            -> IO a
```

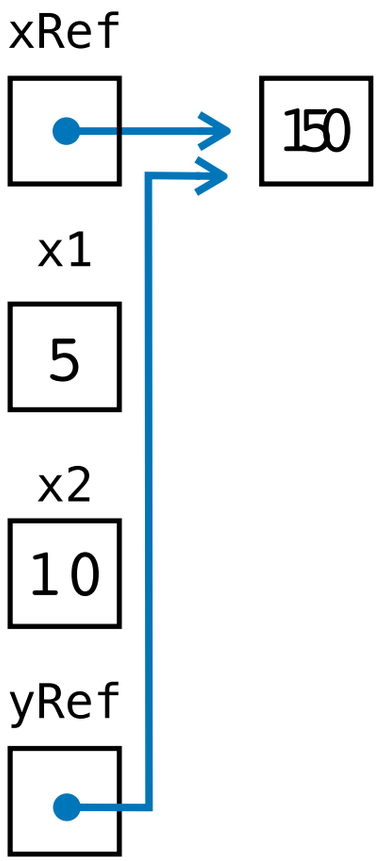
- these are functions which have an effect on the world (or depend on the current state of the world)

```
main = do
  xRef <- newIORef 5
  x1 <- readIORef xRef
  writeIORef xRef 10
  x2 <- readIORef xRef
  putStrLn ("x1: " ++ (show x1) ++ " x2: " ++ (show x2))
```



Reference types

```
main = do
  xRef <- newIORef 5
  x1 <- readIORef xRef
  let yRef = xRef
  writeIORef yRef 10
  x2 <- readIORef xRef
  putStrLn ("x1: " ++ (show x1) ++ " x2: " ++ (show x2))
```




```
x1: 5  x2: 10
```

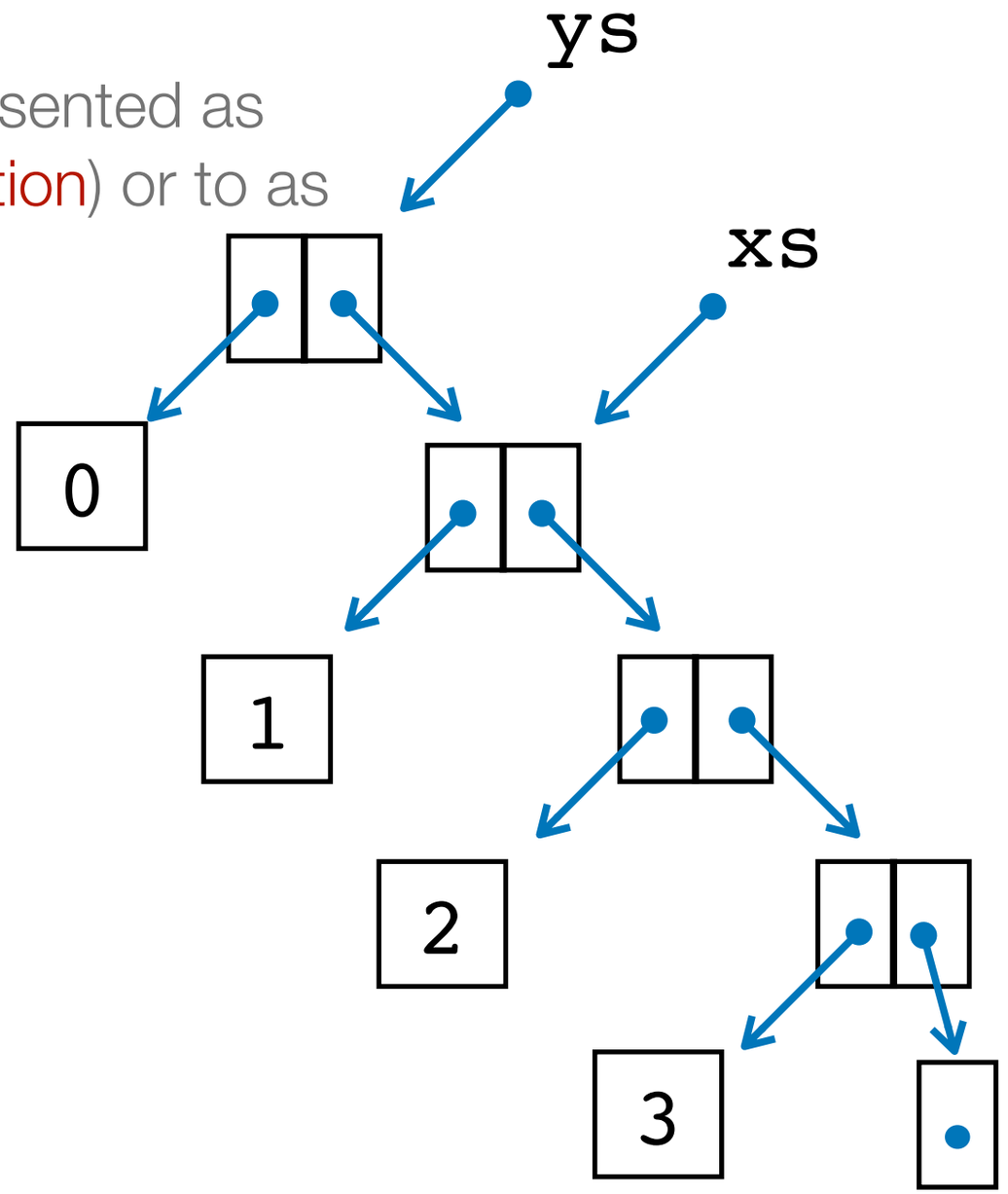
Reference types

- Haskell also uses references behind the scenes

  - even basic values (Int etc) are internally represented as references to these values (boxed representation) or to as to yet unevaluated computations

  - enables sharing

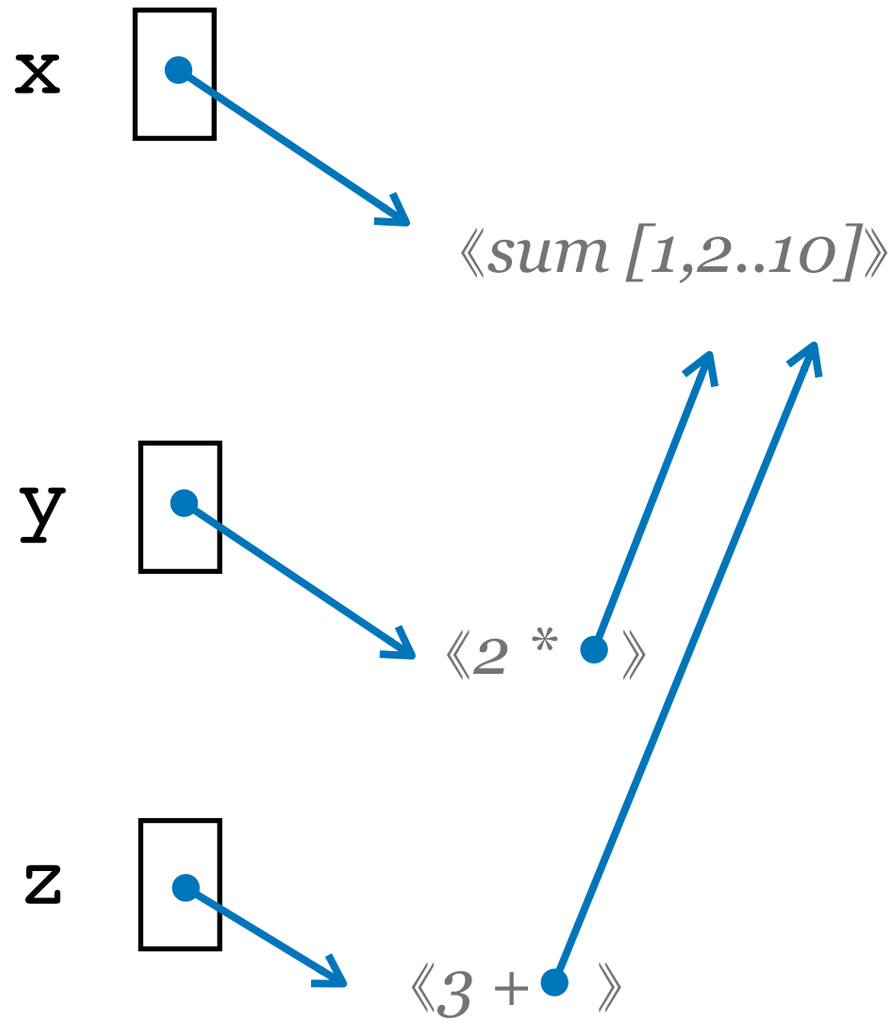
```
let
  xs = [1,2,3]
  ys = 0 : xs
```



Reference types

- The boxed representation is an efficient representation for lazy evaluation

```
let
  x = sum [1,2..10]
  y = 2 * x
  z = 3 + x
```

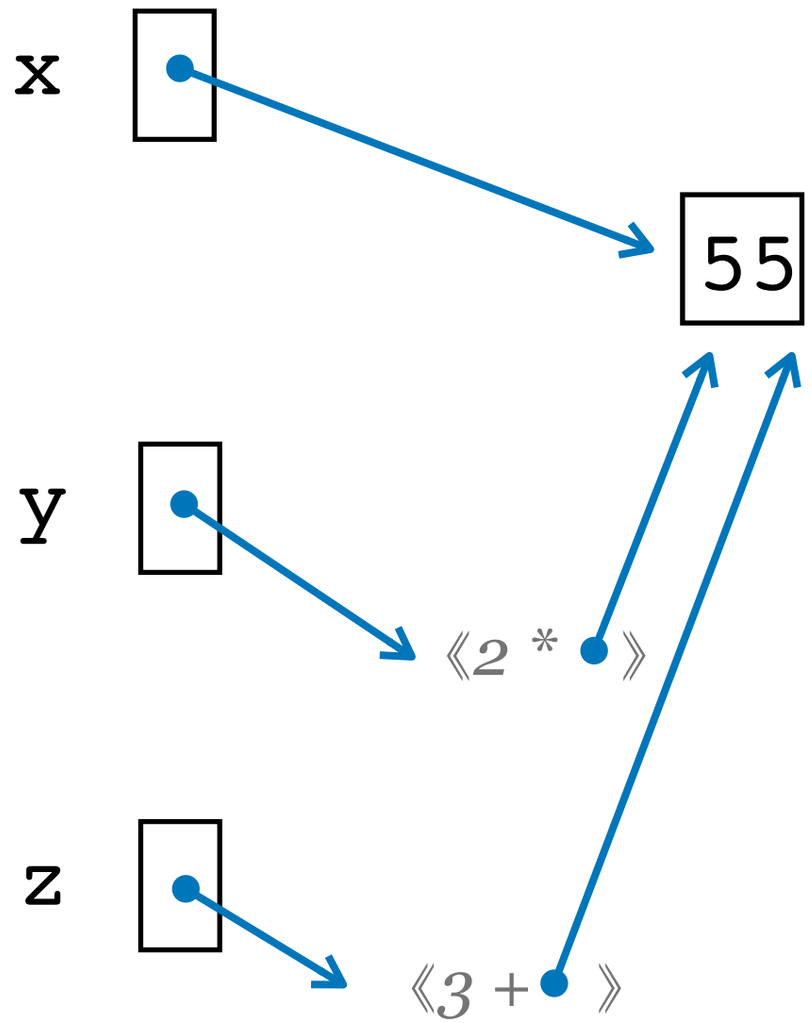


- This means evaluation has a side effect (this can be problematic for parallel execution)

Reference types

- The boxed representation is an effective representation for sharing (lazy evaluation!)

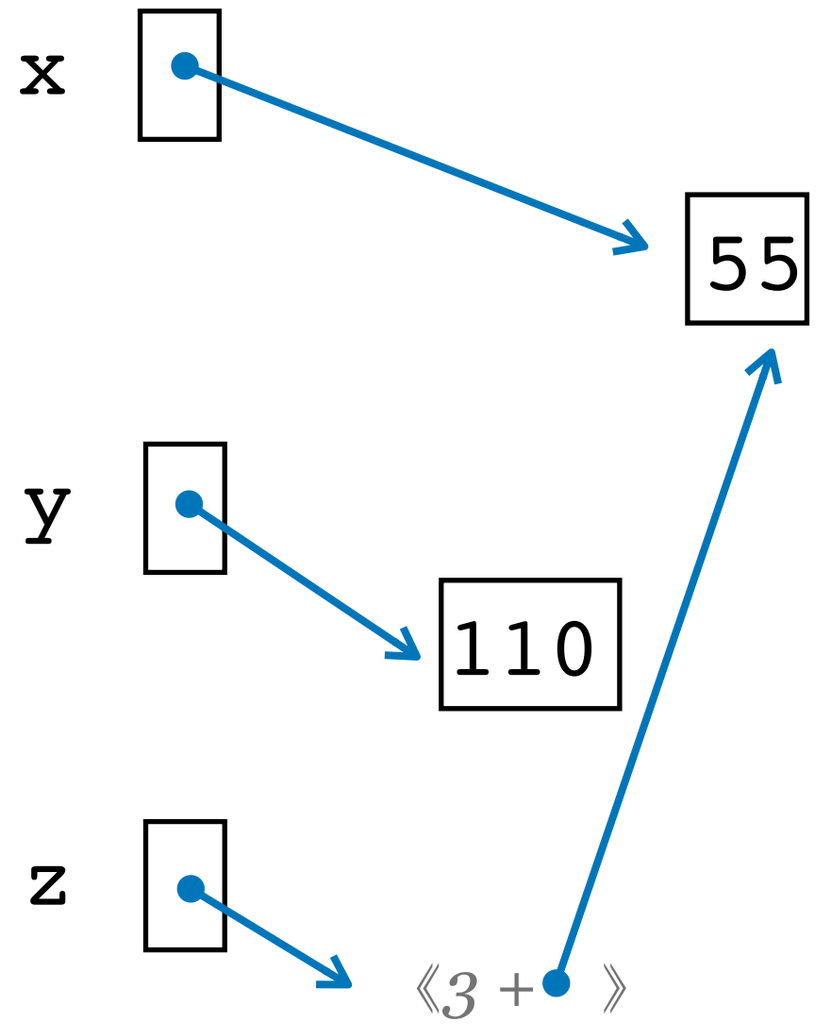
```
let
  x = sum [1,2..10]
  y = 2 * x
  z = 3 + x
```



Reference types

- The boxed representation is an effective representation for sharing (lazy evaluation!)

```
let
  x = sum [1,2..10]
  y = 2 * x
  z = 3 + x
```



- This means evaluation has a side effect (this can be problematic for parallel execution)

Reference types

- In functional languages, it doesn't matter for the semantics of a program whether a value has a boxed or unboxed representation

  - it does affect performance, as dereferencing is expensive

  - in Haskell, it's possible to explicitly use unboxed types (denoted by # - `Int#`…)

References in stateful languages

- In languages with side effect, it is important to know whether we deal with reference or value types to understand the behaviour of
  - assignments
  - function calls
- Unfortunately, this is not uniform, even across closely related languages

Reference types vs value types

- In C#, Java some values are represented via references, some directly a the value
  - value types:
    - value types: boolean, integers, floating point numbers
  - reference types:
    - classes, interfaces, arrays
- In Swift
  - everything is a value type, with the exception of functions, closures and classes
- C++
  - all value types, references and pointers are explicit

C#:

```
public class MyClass
{
  public int value;
}

public class Program
{
  public static void Main()
  {
    MyClass ob1 = new MyClass();
    ob1.value   = 20;
    MyClass ob2 = ob1;
    ob1.value   = 10;
    Console.WriteLine("ob2.value = {0}", ob2.value);
}
```

ob2.value = 10

C++:

```
class MyClass
{
  public: int value;
};

int main() {
    MyClass ob1;
    ob1.value   = 20;
    MyClass ob2 = ob1;
    ob1.value   = 10;
    std::cout << "obj2.value = "<< ob2.value;
    return 0;
}
```

ob2.value = 20

Check out differences in value & reference type classification when switching to a new language!

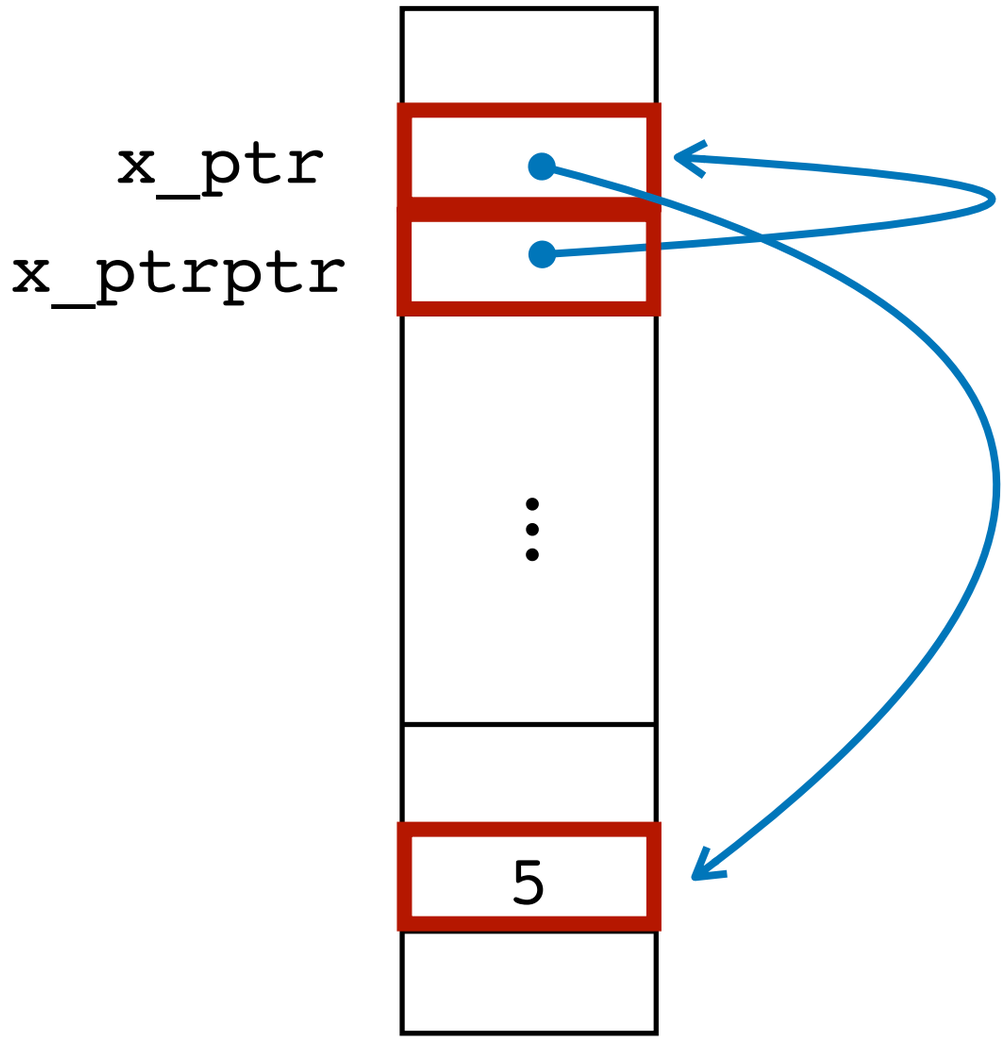
Language	Value type	Reference type
C++[3]	booleans, characters, integer numbers, floating-point numbers, classes (including strings, lists, maps, sets, stacks, queues), enumerations	references, pointers
Java[4]	booleans, characters, integer numbers, floating-point numbers	arrays, classes (including immutable strings, lists, dictionaries, sets, stacks, queues, enumerations), interfaces, null pointer
C#[5]	structures (including booleans, characters, integer numbers, floating-point numbers, point in time i.e. DateTime, optionals i.e. Nullable<T>), enumerations	classes (including immutable strings, arrays, tuples, lists, dictionaries, sets, stacks, queues), interfaces, pointers
Swift[6][7]	structures (including booleans, characters, integer numbers, floating-point numbers, fixed-point numbers, mutable strings, tuples, mutable arrays, mutable dictionaries, mutable sets), enumerations (including optionals), and user-defined structures and enumerations composing other value types.	functions, closures, classes
Python[8]		classes (including immutable booleans, immutable integer numbers, immutable floating-point numbers, immutable complex numbers, immutable strings, byte strings, immutable byte strings, immutable tuples, immutable ranges, immutable memory views, lists, dictionaries, sets, immutable sets, null pointer)
JavaScript[9]	immutable booleans, immutable floating-point numbers, immutable integer numbers (bigint), immutable strings, immutable symbols, undefined, null	objects (including functions, arrays, typed arrays, sets, maps, weak sets and weak maps)
OCaml[10] [11]	immutable characters, immutable integer numbers, immutable floating-point numbers, immutable tuples, immutable enumerations (including immutable units, immutable booleans, immutable lists, immutable optionals), immutable exceptions, immutable formatting strings	arrays, immutable strings, byte strings, dictionaries (including pointers)

https://en.wikipedia.org/wiki/Value_type_and_reference_type

Pointers in C

- Pointer types in C are denoted by an asterisk * after the type name
- Dereferencing a pointer is done by the * operator
- The operator & returns the address of a variable
- Memory has to allocated and freed explicitly by the programmer

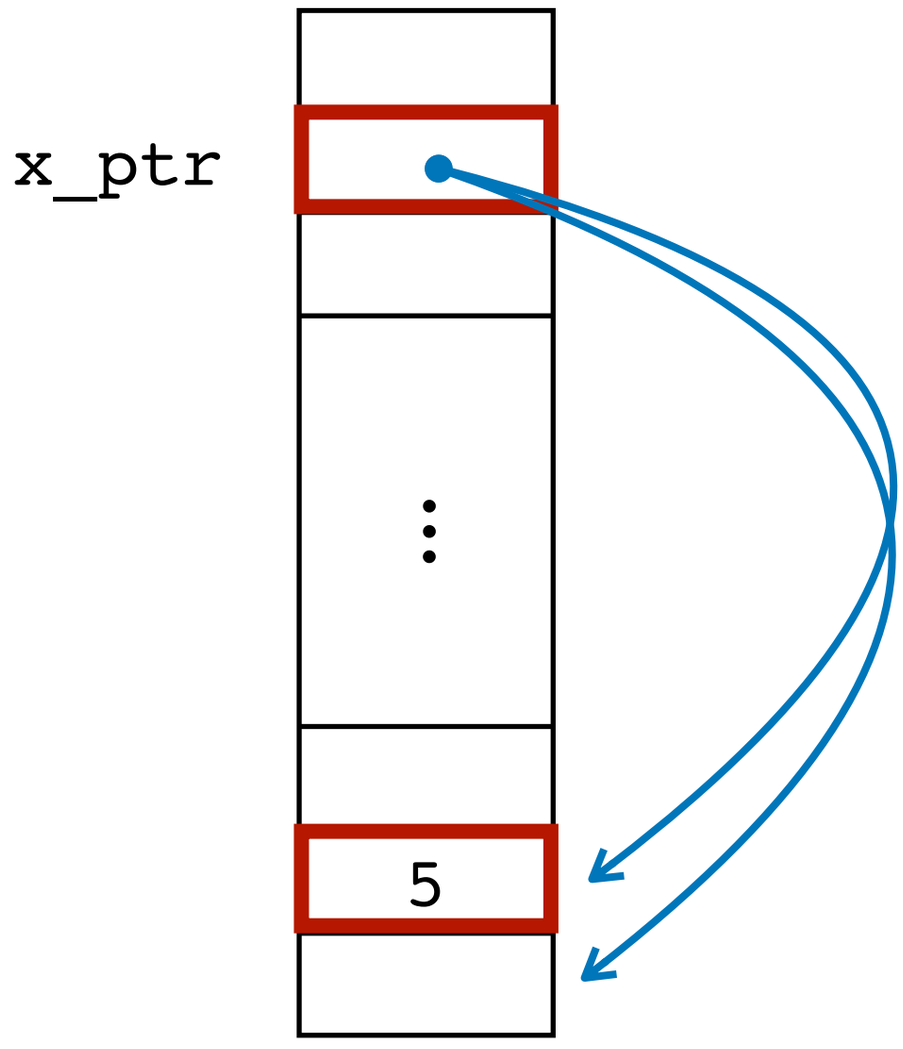
```
int * x_ptr;

x_ptr = (int *) malloc (sizeof (int));

*x_ptr = 5;

int * * x_ptrptr;
x_ptrptr = & x_ptr;


free (x_ptr);
```



Pointers in C

- Pointers in C are very powerful, but very unsafe!
  - forget to free memory
  - free an address which has not been allocated
  - access a memory location after freeing
  - return an address of a local variable
- Pointer arithmetic

```
x_ptr++;
```



Call-by-value vs call-by reference

- Also called pass-by-reference/pass-by-value
- What is the calling convention for procedures/functions/methods?
- Call by value
  - like in TinyC (and C): the value of the argument expression gets bound to the formal parameter.
  - function calls don't affect the values of the variables in the caller
- Java, C#, C++ are all call by value, but since classes are reference types in C# & Java, the behaviour is different
  - Java/C#: the reference gets copied
  - C++, the object gets copied)
- Fortran is always call by reference - even on constant values!
  - this can result of constant values being changed!

Call-by-value vs call-by reference

```
void swap1 (int x, int y) {
  int tmp;
  tmp = x;
  x = y;
  y = tmp;
}


void swap2 (int * x, int * y) {
  int tmp;
  tmp = *x;
  *x = *y;
  *y = tmp;
}

int a = 5;
int b = 7;

swap1 (a, b);
swap2 (&a, &b);
```

TinyC with references

- We add references as abstract data type to the language
  - declaration & instantiation of a reference and the value it points to
  - reading the value a reference points to
  - updating the value a reference points to

Adding references

$$
\begin{array}{lll}
\mathit{prgm} & ::= & \mathit{gdecs}\ \mathit{rdecs}\ \mathit{stmt} \\
\mathit{gdecs} & ::= & \epsilon \mid \mathit{gdec}\ \mathit{gdecs} \\
\mathit{gdec} & ::= & \mathit{fdec} \mid \mathit{vdec} \\
\mathit{vdecs} & ::= & \epsilon \mid \mathit{vdec}\ \mathit{vdecs} \\
\mathit{type} & ::= & \texttt{int} \mid \texttt{int}* \\
\mathit{vdec} & ::= & \texttt{int}\ \mathit{Ident} = \mathit{Int}; \\
\mathit{rdecs} & ::= & \epsilon \mid \mathit{rdec}\ \mathit{rdecs} \\
\mathit{rdec} & ::= & \texttt{int}*\ \mathit{Ident} = \texttt{alloc}(\mathit{Int}); \\
\mathit{fdec} & ::= & \texttt{int}\ \mathit{Ident}\ (\mathit{arguments})\ \mathit{stmt} \\
\mathit{stmt} & ::= & \mathit{expr}; \mid \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{stmt}\ \texttt{else}\ \mathit{stmt}; \mid \texttt{return}\ \mathit{expr}; \mid \\
 & & \{\ \mathit{vdecs}\ \mathit{rdecs}\ \mathit{stmts}\ \} \mid \texttt{while}\ (\ \mathit{expr}\ )\ \mathit{stmt} \\
\mathit{stmts} & ::= & \epsilon \mid \mathit{stmt}\ \mathit{stmts} \\
\mathit{expr} & ::= & \mathit{Int} \mid \mathit{Ident} \mid *\mathit{Ident} \mid \mathit{expr} + \mathit{expr} \mid \mathit{expr} - \mathit{expr} \mid \\
 & & \mathit{Ident} = \mathit{expr} \mid *\mathit{Ident} = \mathit{expr} \mid \mathit{Ident}\ (\mathit{exprs}) \\
\mathit{arguments} & ::= & \epsilon \mid \texttt{int}\ \mathit{Ident}\ ,\ \mathit{arguments} \\
\mathit{exprs} & ::= & \epsilon \mid \mathit{expr}\ ,\ \mathit{exprs}
\end{array}
$$

Utrecht University

Static semantics

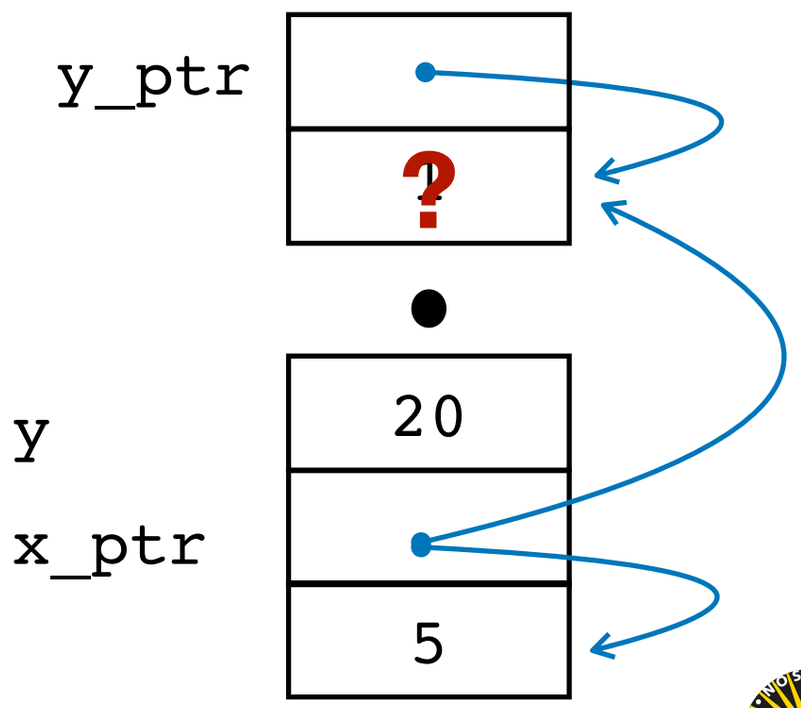
- We have to do ‘proper’ type checking now:
  - Environment of variables with type:
    - $V = \{x_1 : type_1,\ x_2 : type_2, ...\}$
  - Environment of functions with their type:
    - $F = \{f_1 : (type_1, type_2\ ...) \rightarrow type\ , ...\}$
  - make sure arithmetic operations only applied to `int`
  - dereferencing only on `int *` types, etc
  - type of the return value of a function is what it is supposed to be
- same technique as for MinHs, so we skip this step and focus on the dynamic semantics

Dynamic semantics

- Consider how we modelled the memory in TinyC:

$$\frac{g@x = v}{(g, x) \Downarrow (g, v)} \qquad \frac{(g, e) \Downarrow (g', v)}{(g, x{=}e) \Downarrow (g'@x \leftarrow v, v)} \qquad \frac{(g.l, ss) \Downarrow (g'.l', rv)}{(g, \{l\ ss\}) \Downarrow (g', rv)}$$

```
int y = 20;
int * x_ptr = alloc (5);

{
  int * y_ptr = alloc (1);
  x_ptr = y_ptr;
}
y = *x_ptr;
```

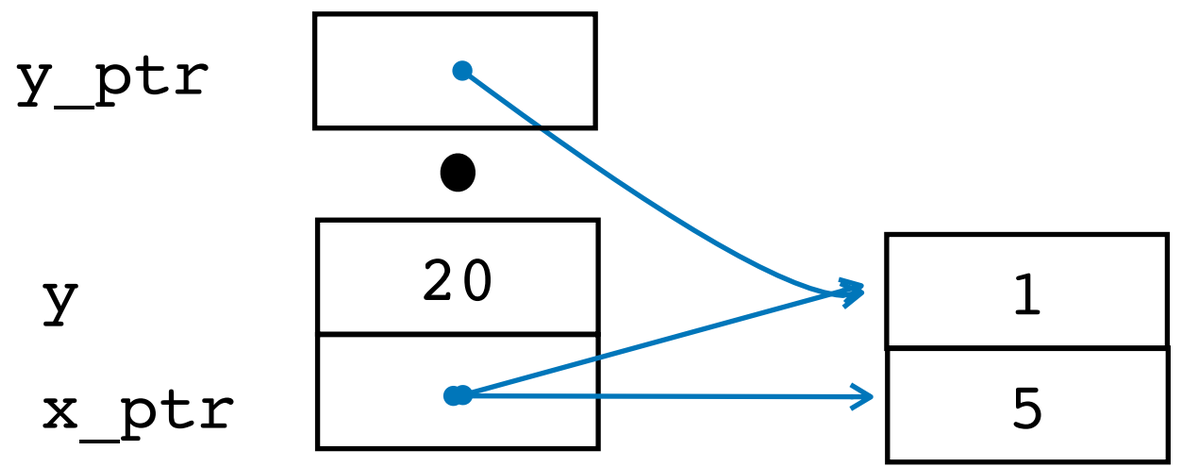


Dynamic semantics

- Problem:
  - we can't get away with only storing everything in a stack frame which we remove from the stack when exiting a block
  - we need persistent memory (in addition to the stack)!
  - we need to keep track of which addresses in the persistent memory are available

```
int y = 20;
int * x_ptr = alloc (5);

{
  int * y_ptr = alloc (1);
  x_ptr = y_ptr;
}
y = *x_ptr;
```

g h



Dynamic semantics

- Modelling the persistent memory (heap)

  - state as a triple: stack g, heap h with next free address k, current statement s

$$(g \blacklozenge h_k , s)$$

  - judgements:

$$prgm \Downarrow (g \blacklozenge h_k , rv)$$

$$(g \blacklozenge h_k , expr) \Downarrow (g' \blacklozenge h'_{k'} , v)$$

$$(g \blacklozenge h_k , stmt) \Downarrow (g' \blacklozenge h'_{k'} , rv)$$

Dynamic semantics

- Declaration and initialisation of references:

$$\frac{}{(g \blacklozenge h_k,\ \texttt{int * } x \texttt{ = alloc(}v\texttt{);}) \Downarrow ((g,\ \texttt{int * } x \texttt{ = } k) \blacklozenge (h.\ k = v)_{k+1},\ v)}$$

- Dereferencing and assignment

$$\frac{h_k @ (g\ @\ x) = v}{(g \blacklozenge h_k\ ,\ *x) \Downarrow (g \blacklozenge h_k,\ v)}$$

$$\frac{(g \blacklozenge h_k\ ,\ e) \Downarrow (g' \blacklozenge h'_{k'}\ ,\ v) \qquad (g\ @\ x) = j}{(g \blacklozenge h_k\ ,*x = e) \Downarrow (g' \blacklozenge (h'\ @\ j \leftarrow v)_{k'}\ ,\ v)}$$

Utrecht University

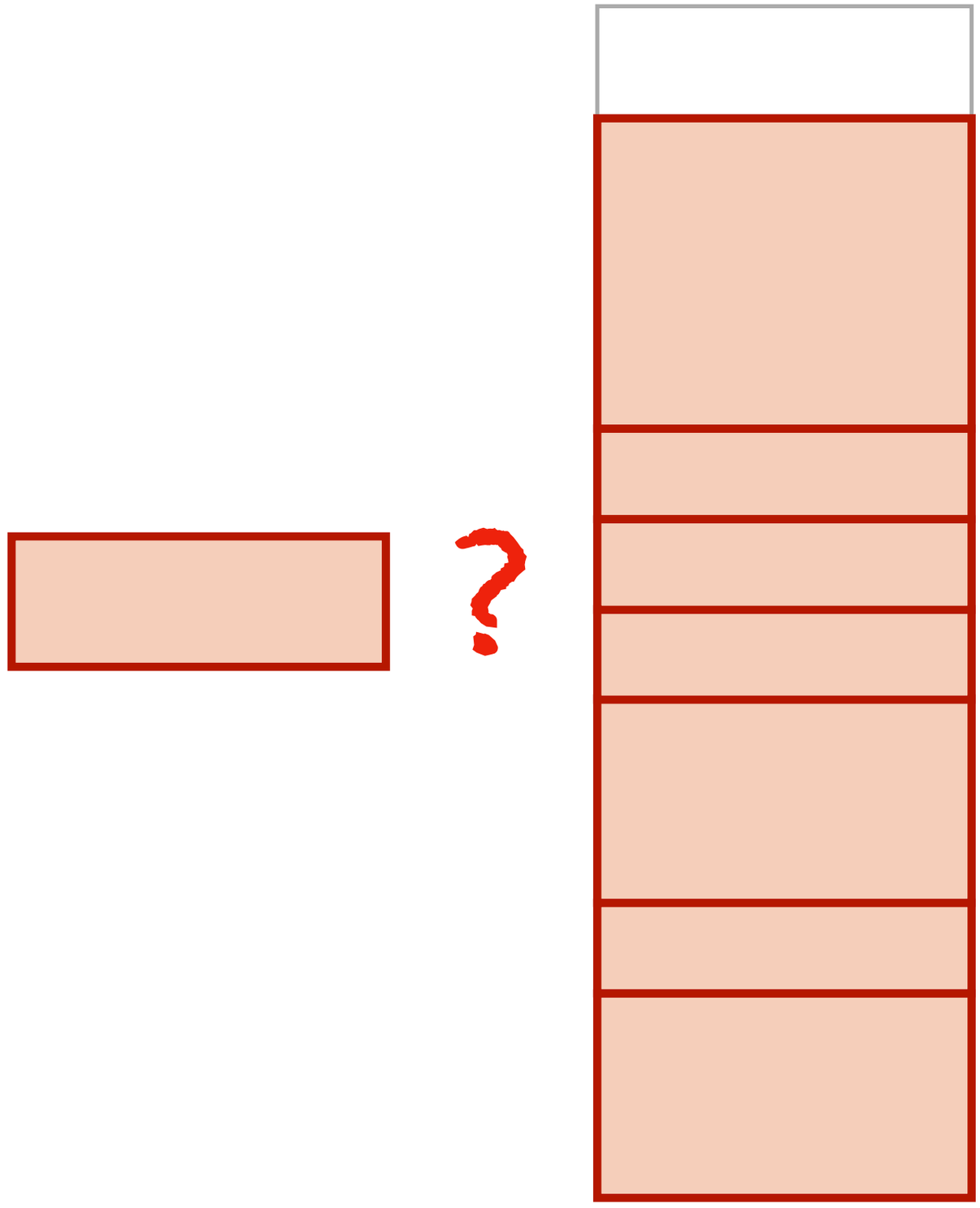
Observations

- Is TinyC with references still type safe (assuming we did the type checking properly)?
  - dereferencing will always return a result, since all variables of type `int *` are either
    - introduced and instantiated by a declaration and therefore have a value, or
    - assigned to a variable of type `int *` (has a value if that variable has a value)
    - or bound via a function call to another value of type `int *`
  - we don't have an address operator (&) like C, so we don't have to worry about addresses not pointing to h
  - once a legal reference, always a legal reference (never freed, never out of scope)
  - the actual address k is not observable and can't be manipulated (no pointer arithmetic)

Observations

- The problem with our heap h:
  - the heaps keeps growing, as memory on the heap is never freed
- Should we introduce a `free` operation?
  - we could introduce a runtime error whenever we dereference a freed reference
  - but how can we re-use an address then?
  - still no guarantee that the programmer frees memory once its not required anymore
  - in real languages, this leads to memory fragmentation

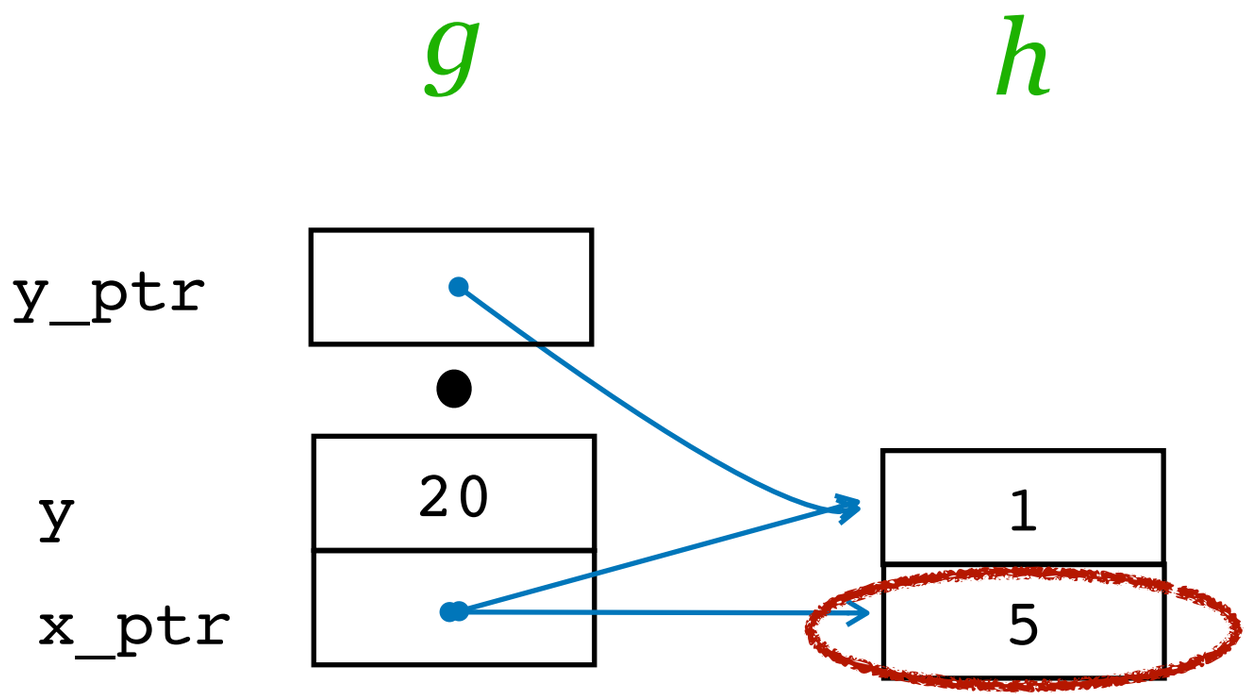
Memory Fragmentation



Observations

- We know that only addresses which are still referred to from the stack are reachable

```
int y = 20;
int * x_ptr = alloc (5);

{
  int * y_ptr = alloc (1);
  x_ptr = y_ptr;
}
y = *x_ptr;
```



can be reused!

- How can we identify those addresses?

Memory management

- Many ways to do this, for example:
  - keep track of number of references to each heap location during execution
    - disadvantage: runtime overhead, possible fragmentation of memory if allocated blocks have different size (not a problem in Tiny, since only space for ints is allocated), cyclic structures?
  - tracing garbage collection:
    - stop execution of program and trace all stack references, copy only live data to new heap space, free old heap space
    - it is necessary to be able to identify references on the stack
- keep track of references via static semantics/type system
  - we'll discuss this later in the course

Utrecht University

Garbage Collection

- Garbage collection in our simple model can be viewed as a mapping from
  - a memory state $(g \blacklozenge h_k)$ to a state $(g' \blacklozenge h'_{k'})$
  - which yields the same result for all data look ups.
- More formally, $(g \blacklozenge h_k) \approx (g' \blacklozenge h'_{k'})$, with

$$\frac{}{(\circ \blacklozenge h_k) \approx (\circ \blacklozenge h'_{k'})}$$

$$\frac{(g \blacklozenge h_k) \approx (g' \blacklozenge h'_{k'})}{(g\ ,\ \texttt{int}\ x = v \blacklozenge h_k) \approx (g'\ ,\ \texttt{int}\ x = v \blacklozenge h'_{k'})}$$

$$\frac{(g \blacklozenge h_k) \approx (g' \blacklozenge h'_{k'}) \qquad h_k@| = h'_{k'}@|}{(g\ ,\ \texttt{int*}\ \ x = l \blacklozenge h_k) \approx (g'\ ,\ \texttt{int}\ x = l' \blacklozenge h'_{k'})}$$

Utrecht University

Garbage collection

- Stop-and-copy:
  - we maintain two heaps, the current and the new heap
  - when the current heap gets too full, traverse the stack
    - copy each object referenced from the stack into the new heap
    - for realistic languages, where we also have references on the heap, we follow the references recursively, until all objects referenced from the stack directly or indirectly have been moved to the new heap
  - swap current and new heap, repeat
  - disadvantage:
    - (large data structures which live for a long time get copied back and forth repeatedly

Garbage collection

- Generational garbage collection:
  - based on the observation that most objects can be freed soon after creation, the older objects are, the more like they are still in use in the next interval
  - multiple heaps, $gen\ 0$ to $gen\ n$
  - objects get copied from $gen\ i$ to $gen\ (i+1)$
  - garbage collect more frequently in younger generations
  - only $gen\ n$ has its new heap
  - objects cannot refer back to objects of a younger generation

Memory leaks

- In languages with manual memory management, we can have memory leaks if the programmer forgets to free memory

- How about languages with automatic memory management?

  - external resources the runtime system doesn't have control over

  - references to unused resources in an object, or a closure

  - interaction with sharing and laziness, e.g., Haskell

```
let
  xs = [1,2..1000000]
  ys = sum xs
  z  = maximum xs
in ....
```

- and laziness, e.g., Haskell

Summary

- It is important to know which types are represented as value types, which as reference types
  - for pure languages, it influences performance
  - for stateful languages, performance and semantics!
- Memory management
  - manual memory management
    - allows fine grained control, potentially much more economical regarding memory consumption
    - also, huge source of bugs, security vulnerabilities
  - automatic memory management
    - convenient
    - safer
    - hard to predict runtime performance, still necessary to be aware of what's going on behind the scenes to some extend
  - we will hear about a third approach later in the course!

Utrecht University