Runtime systems

Programmation Fonctionnelle Avancée

http://www-lipn.univ-paris13.fr/~saiu/teaching/PFA-2010

Luca Saiu
saiu@lipn.univ-paris13.fr

Master Informatique 2ème année, spécialité *Programmation et Logiciels Sûrs*
Laboratoire d'Informatique de l'Université Paris Nord — Institut Galilée

2011-12-05

Runtime systems

- Functional programs are very high-level: it's not obvious how to implement them on the machine.
- Complex library support at run time
  - How do we represent objects in memory?
    - Think about memory words, bits and pointers
  - How do we release memory?
  - The runtime must be written **in a low-level language** (C, assembly)

Runtime systems

- Functional programs are very high-level: it's not obvious how to implement them on the machine.
- Complex library support at run time
  - How do we represent objects in memory?
    - Think about memory words, bits and pointers
  - How do we release memory?
  - The runtime must be written **in a low-level language** (C, assembly)

Runtime systems

- Functional programs are very high-level: it's not obvious how to implement them on the machine.
- Complex library support at run time
  - How do we represent objects in memory?
    - Think about memory words, bits and pointers
  - How do we release memory?
  - The runtime must be written **in a low-level language** (C, assembly)

Runtime systems

- Functional programs are very high-level: it's not obvious how to implement them on the machine.
- Complex library support at run time
  - How do we represent objects in memory?
    - Think about memory words, bits and pointers
  - How do we release memory?
  - The runtime must be written **in a low-level language** (C, assembly)

Runtime systems

- Functional programs are very high-level: it's not obvious how to implement them on the machine.
- Complex library support at run time
  - How do we represent objects in memory?
    - Think about memory words, bits and pointers
  - How do we release memory?
  - The runtime must be written **in a low-level language** (C, assembly)

Runtime systems

- Functional programs are very high-level: it's not obvious how to implement them on the machine.
- Complex library support at run time
  - How do we represent objects in memory?
    - Think about memory words, bits and pointers
  - How do we release memory?
  - The runtime must be written **in a low-level language** (C, assembly)

Think of an efficient implementation

Return the given list with 42 prepended, without modifying anything: `let f xs = 42 :: xs;;`

- What if we call `f` with a ten-million-element list?
- *We don't need to copy anything!* We're just building a very small structure (one cons) which refers the given one
  - We should always pass and return *pointers* to data structures in memory
  - Fast and simple!
    - but **when do we destroy** lists...?
  - Anyway we *don't* want to allocate in memory small data which fit in registers: avoid allocation whenever possible

Think of an efficient implementation

Return the given list with 42 prepended, without modifying anything: `let f xs = 42 :: xs;;`

- What if we call `f` with a ten-million-element list?
- *We don't need to copy anything!* We're just building a very small structure (one cons) which refers the given one
  - We should always pass and return *pointers* to data structures in memory
  - Fast and simple!
    - but **when do we destroy** lists...?
  - Anyway we *don't* want to allocate in memory small data which fit in registers: avoid allocation whenever possible

Think of an efficient implementation

Return the given list with 42 prepended, without modifying anything: `let f xs = 42 :: xs;;`

- What if we call `f` with a ten-million-element list?
- *We don't need to copy anything!* We're just building a very small structure (one cons) which refers the given one
  - We should always pass and return *pointers* to data structures in memory
  - Fast and simple!
    - but **when do we destroy** lists...?
  - Anyway we *don't* want to allocate in memory small data which fit in registers: avoid allocation whenever possible

Think of an efficient implementation

Return the given list with 42 prepended, without modifying anything: `let f xs = 42 :: xs;;`

- What if we call `f` with a ten-million-element list?
- *We don't need to copy anything!* We're just building a very small structure (one cons) which refers the given one
  - We should always pass and return *pointers* to data structures in memory
  - Fast and simple!
    - but **when do we destroy** lists...?
  - Anyway we *don't* want to allocate in memory small data which fit in registers: avoid allocation whenever possible

Think of an efficient implementation

Return the given list with 42 prepended, without modifying anything: `let f xs = 42 :: xs;;`

- What if we call `f` with a ten-million-element list?
- *We don't need to copy anything!* We're just building a very small structure (one cons) which refers the given one
  - We should always pass and return *pointers* to data structures in memory
  - Fast and simple!
    - but **when do we destroy** lists...?
  - Anyway we *don't* want to allocate in memory small data which fit in registers: avoid allocation whenever possible

Think of an efficient implementation

Return the given list with 42 prepended, without modifying anything: `let f xs = 42 :: xs;;`

- What if we call `f` with a ten-million-element list?
- *We don't need to copy anything!* We're just building a very small structure (one cons) which refers the given one
  - We should always pass and return *pointers* to data structures in memory
  - Fast and simple!
    - but **when do we destroy** lists...?
  - Anyway we *don't* want to allocate in memory small data which fit in registers: avoid allocation whenever possible

Think of an efficient implementation

Return the given list with 42 prepended, without modifying anything: `let f xs = 42 :: xs;;`

- What if we call `f` with a ten-million-element list?
- *We don't need to copy anything!* We're just building a very small structure (one cons) which refers the given one
  - We should always pass and return *pointers* to data structures in memory
  - Fast and simple!
    - but **when do we destroy** lists...?
  - Anyway we *don't* want to allocate in memory small data which fit in registers: avoid allocation whenever possible

Binary words

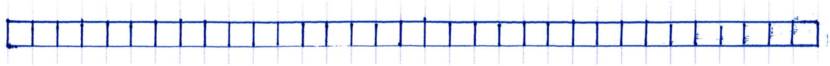


Figure: A 32-bit word

- Modern machines have 32- or 64-bit words. There are assembly instructions working efficiently on word-sized data (arithmetics, load, store)
- At the hardware level, *memory is untyped*: a binary word can represent an integer, a boolean, a float, some characters, a pointer[1], a sum type element with no parameters...

[1] Today we ignore internal pointers for simplicity

Binary words

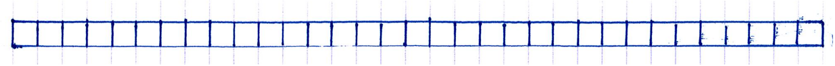


Figure: A 32-bit word

- Modern machines have 32- or 64-bit words. There are assembly instructions working efficiently on word-sized data (arithmetics, load, store)
- At the hardware level, *memory is untyped*: a binary word can represent an integer, a boolean, a float, some characters, a pointer[1], a sum type element with no parameters...

[1] Today we ignore internal pointers for simplicity

Boxed vs. unboxed

Two ways of representing objects:

- *boxed*: allocate the object in memory, and pass around a pointer to it
- *unboxed*: pass around the object itself, which is small (word-sized)

Stack allocation is not enough

```
int f(int x){
  struct big_struct s;
  s.q = x;
  s.w = g(x, &s);
  ...
  return 42;
}
```

- `s` is visible to `g`. `s` remains *alive* in memory until `f` returns, so also the functions called by `g` might access it.
- When `f` returns `s` is **automatically** destroyed: its memory will be reused
- LIFO policy implemented with a stack: push at function entry, pop at function exit
  - Very, *very* efficient. But not general enough.

Stack allocation is not enough

```
int f(int x){
  struct big_struct s;
  s.q = x;
  s.w = g(x, &s);
  ...
  return 42;
}
```

- s is visible to g. s remains *alive* in memory until f returns, so also the functions called by g might access it.
- When f returns s is **automatically** destroyed: its memory will be reused
- LIFO policy implemented with a stack: push at function entry, pop at function exit
  - Very, *very* efficient. But not general enough.

Stack allocation is not enough

```
int f(int x){
  struct big_struct s;
  s.q = x;
  s.w = g(x, &s);
  ...
  return 42;
}
```

- `s` is visible to `g`. `s` remains *alive* in memory until `f` returns, so also the functions called by `g` might access it.
- When `f` returns `s` is **automatically** destroyed: its memory will be reused
- LIFO policy implemented with a stack: push at function entry, pop at function exit
  - Very, *very* efficient. But not general enough.

Stack allocation is not enough

```
int f(int x){
  struct big_struct s;
  s.q = x;
  s.w = g(x, &s);
  ...
  return 42;
}
```

- s is visible to g. s remains *alive* in memory until f returns, so also the functions called by g might access it.
- When f returns s is **automatically** destroyed: its memory will be reused
- LIFO policy implemented with a stack: push at function entry, pop at function exit
  - Very, *very* efficient. But not general enough.

Stack allocation is not enough

```
int f(int x){
  struct big_struct s;
  s.q = x;
  s.w = g(x, &s);
  ...
  return 42;
}
```

- s is visible to g. s remains *alive* in memory until f returns, so also the functions called by g might access it.
- When f returns s is **automatically** destroyed: its memory will be reused
- LIFO policy implemented with a stack: push at function entry, pop at function exit
  - Very, *very* efficient. But not general enough.

Stack allocation is not enough

```
int f(int x){
  struct big_struct s;
  s.q = x;
  s.w = g(x, &s);
  ...
  return 42;
}
```

- `s` is visible to `g`. `s` remains *alive* in memory until `f` returns, so also the functions called by `g` might access it.
- When `f` returns `s` is **automatically** destroyed: its memory will be reused
- LIFO policy implemented with a stack: push at function entry, pop at function exit
  - Very, *very* efficient. But not general enough.

Heap allocation and de-allocation

Any reasonable programming language also lets you explicitly create new objects in memory without following a LIFO policy:

```
/* C */
p = malloc(sizeof(int) * 2);
p[0] = 42;
...
free(p);
```

```
(* OCaml *)
x :: xs
...
```

(the memory is freed "automagically")

A heap with free list

A *heap* is the data structure on which `malloc` and `free` are implemented:

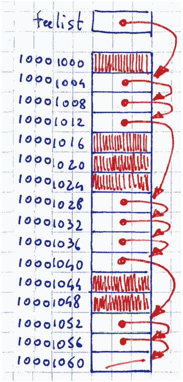


Figure: A 16-word heap with a *free list*: each *unused* word in the *heap* points to the next one. "Filled" words belong to alive objects.

How to interpret a word-sized datum

Figure: Is this a number, a boolean, a pointer, or maybe an object of type `t = A | B | C`, or ...?

- $100110001001101010001000_2 = 10001032_{10}$
- Number, memory address, or what else?
- In general we can't tell
- We could establish a non-standard convention in our runtime so that all objects are *tagged* with an encoding of their type

How to interpret a word-sized datum

Figure: Is this a number, a boolean, a pointer, or maybe an object of type `t = A | B | C`, or ...?

- $100110001001101010001000_2 = 10001032_{10}$
- Number, memory address, or what else?
- In general we can't tell
- We could establish a non-standard convention in our runtime so that all objects are *tagged* with an encoding of their type

How to interpret a word-sized datum

Figure: Is this a number, a boolean, a pointer, or maybe an object of type `t = A | B | C`, or ...?

- $100110001001101010001000_2 = 10001032_{10}$
- Number, memory address, or what else?
- In general we can't tell
- We could establish a non-standard convention in our runtime so that all objects are *tagged* with an encoding of their type

How to interpret a word-sized datum

Figure: Is this a number, a boolean, a pointer, or maybe an object of type `t = A | B | C`, or ...?

- $100110001001101010001000_2 = 10001032_{10}$
- Number, memory address, or what else?
- In general we can't tell
- We could establish a non-standard convention in our runtime so that all objects are *tagged* with an encoding of their type

How to interpret a word-sized datum

Figure: Is this a number, a boolean, a pointer, or maybe an object of type `t = A | B | C`, or ...?

- $100110001001101010001000_2 = 10001032_{10}$
- Number, memory address, or what else?
- In general we can't tell
- We could establish a non-standard convention in our runtime so that all objects are *tagged* with an encoding of their type

Object headers - I

We can reserve a word for runtime type information at the beginning of each object:

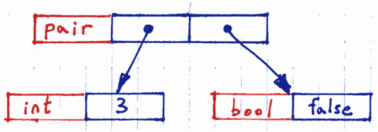


Figure: The pair `(3, false)` represented with object headers. The words shown in red contain some binary encoding of the type.

Object headers - II

Another example with object headers:

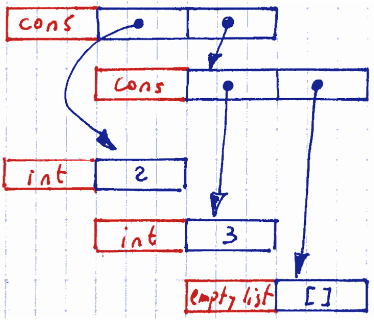


Figure: The list `2 :: 3 :: []`, also written as `[2; 3]`, with object headers

Object headers - III

Pros

- Easy to understand and implement
- One word per object suffices to encode any type
  - The header can also be a pointer, if needed

Cons

- Inefficient: we have to unbox everything

Object headers - III

Pros

- Easy to understand and implement
- One word per object suffices to encode any type
    - The header can also be a pointer, if needed

Cons

- Inefficient: we have to unbox everything

Object headers - III

Pros

- Easy to understand and implement
- One word per object suffices to encode any type
    - The header can also be a pointer, if needed

Cons

- Inefficient: we have to unbox everything

Object headers - III

Pros

- Easy to understand and implement
- One word per object suffices to encode any type
    - The header can also be a pointer, if needed

Cons

- Inefficient: we have to unbox everything

Object headers - III

Pros

- Easy to understand and implement
- One word per object suffices to encode any type
    - The header can also be a pointer, if needed

Cons

- Inefficient: we have to unbox everything

Tagging within a datum word - I

Instead of using a prefix word, we can *reserve some bits in a fixed position within a datum to encode its type*. Example (3-bit tag):

- 000: unique values (booleans, empty list, unit, ...)
- 001: integer
- 010: cons
- 011: character
- 100: float
- 101: ref
- 110: string
- 111: (not used)

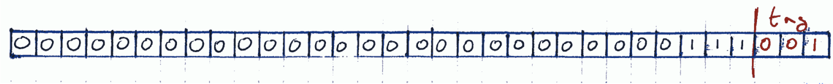


Figure: A 32-bit word with a 3-bit tag. What's this?

Tagging within a datum word - I

Instead of using a prefix word, we can *reserve some bits in a fixed position within a datum to encode its type*. Example (3-bit tag):

- 000: unique values (booleans, empty list, unit, ...)
- 001: integer
- 010: cons
- 011: character
- 100: float
- 101: ref
- 110: string
- 111: (not used)

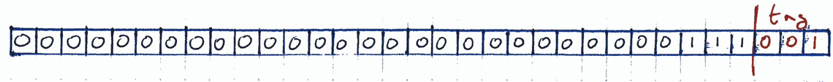


Figure: A 32-bit word with a 3-bit tag. What's this?

Tagging within a datum word - I

Instead of using a prefix word, we can *reserve some bits in a fixed position within a datum to encode its type*. Example (3-bit tag):

- 000: unique values (booleans, empty list, unit, ...)
- 001: integer
- 010: cons
- 011: character
- 100: float
- 101: ref
- 110: string
- 111: (not used)

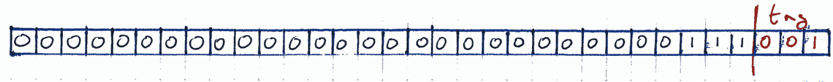


Figure: A 32-bit word with a 3-bit tag. What's this?

Tagging within a datum word - I

Instead of using a prefix word, we can *reserve some bits in a fixed position within a datum to encode its type*. Example (3-bit tag):

- 000: unique values (booleans, empty list, unit, ...)
- 001: integer
- 010: cons
- 011: character
- 100: float
- 101: ref
- 110: string
- 111: (not used)

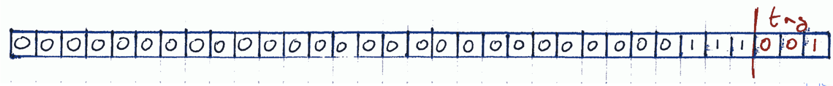


Figure: A 32-bit word with a 3-bit tag. What's this? *The integer* 7_{10}

Tagging within a datum word - II

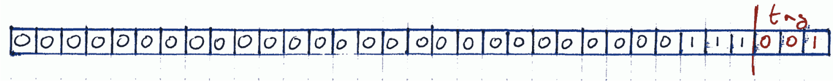


Figure: A 32-bit word: 29-bit payload plus 3-bit tag

- Pros:
  - compact: no additional space is used
- Cons:
  - operating on data is harder and possibly slower (think of adding two tagged integers)
    - ...but we can choose tags in a smart way (any ideas?)
  - less space available for the datum payload
  - very few tags available: at most 2^n with n tag bits

Tagging within a datum word - II

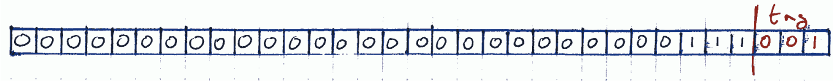


Figure: A 32-bit word: 29-bit payload plus 3-bit tag

- Pros:
  - compact: no additional space is used
- Cons:
  - operating on data is harder and possibly slower (think of adding two tagged integers)
    - ...but we can choose tags in a smart way (any ideas?)
  - less space available for the datum payload
  - very few tags available: at most 2^n with n tag bits

Tagging within a datum word - II

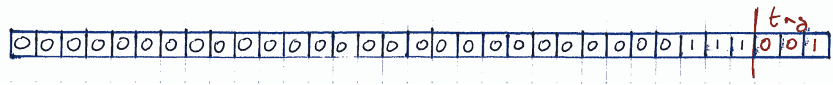


Figure: A 32-bit word: 29-bit payload plus 3-bit tag

- Pros:
  - compact: no additional space is used
- Cons:
  - operating on data is harder and possibly slower (think of adding two tagged integers)
    - ...but we can choose tags in a smart way (any ideas?)
  - less space available for the datum payload
  - very few tags available: at most 2^n with n tag bits

Tagging within a datum word - II

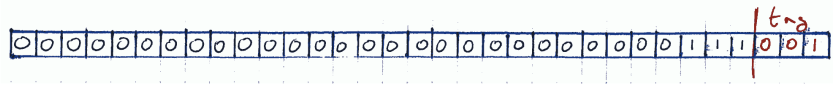


Figure: A 32-bit word: 29-bit payload plus 3-bit tag

- Pros:
  - compact: no additional space is used
- Cons:
  - operating on data is harder and possibly slower (think of adding two tagged integers)
    - ...but we can choose tags in a smart way (any ideas?)
  - less space available for the datum payload
  - very few tags available: at most 2^n with n tag bits

Tagging within a datum word - II

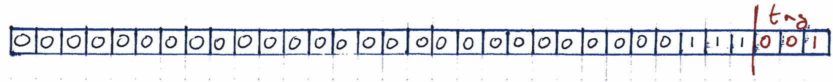


Figure: A 32-bit word: 29-bit payload plus 3-bit tag

- Pros:
  - compact: no additional space is used
- Cons:
  - operating on data is harder and possibly slower (think of adding two tagged integers)
    - ...but we can choose tags in a smart way (any ideas?)
  - less space available for the datum payload
  - very few tags available: at most 2^n with n tag bits

Tagging within a datum word - II

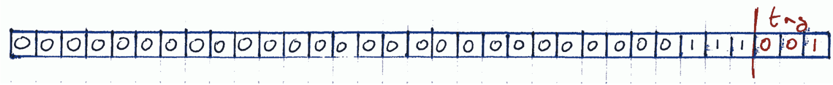


Figure: A 32-bit word: 29-bit payload plus 3-bit tag

- Pros:
  - compact: no additional space is used
- Cons:
  - operating on data is harder and possibly slower (think of adding two tagged integers)
    - ...but we can choose tags in a smart way (any ideas?)
  - less space available for the datum payload
  - very few tags available: at most 2^n with n tag bits

Tagging within a datum word - Example

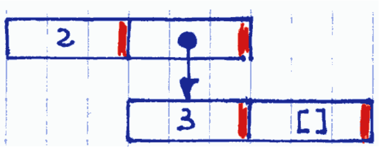


Figure: A list with in-word tags

Notice that we have tagged a *pointer* word. Why can we do it?

- If heap objects are aligned on word boundary, the rightmost two (for 32-bit architectures) or three (for 64-bit architectures) bits are always zero in native pointers.
- Aligning on a wider boundary gives us more bits to use for tagging, but may waste heap space

Tagging within a datum word - Example

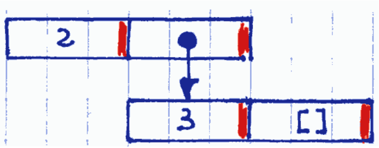


Figure: A list with in-word tags

Notice that we have tagged a *pointer* word. Why can we do it?

- If heap objects are aligned on word boundary, the rightmost two (for 32-bit architectures) or three (for 64-bit architectures) bits are always zero in native pointers.
- Aligning on a wider boundary gives us more bits to use for tagging, but may waste heap space

Tagging within a datum word - Example

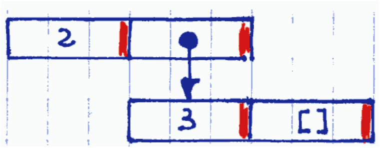


Figure: A list with in-word tags

Notice that we have tagged a *pointer* word. Why can we do it?

- If heap objects are aligned on word boundary, the rightmost two (for 32-bit architectures) or three (for 64-bit architectures) bits are always zero in native pointers.
- Aligning on a wider boundary gives us more bits to use for tagging, but may waste heap space

Alignment: look at the heap again

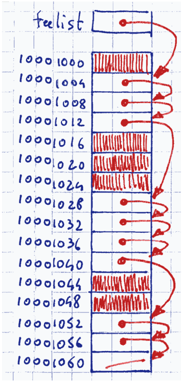


Figure: Think of the binary representation of pointers to heap objects: here any pointer will end with 00 (because addresses in radix 10 are divisible by 4).

Hybrid tagging

In-word tags are more efficient than headers, but we would need much more than two or three bits... We can find a compromise

Use a short in-word tag of two or three bits for the most common types which we want to keep unboxed or boxed without header, reserving one value for boxed objects with headers.

Example (with two-bit in-word tag):

- 00 `int` (unboxed)
- 01 pointer to `cons` (boxed, but no header)
- 10 unique (unboxed)
- 11 *pointer to a boxed object with header*

Very efficient **if integers and lists are used a lot.**

Hybrid tagging

In-word tags are more efficient than headers, but we would need much more than two or three bits... We can find a compromise

Use a short in-word tag of two or three bits for the most common types which we want to keep unboxed or boxed without header, reserving one value for boxed objects with headers.

Example (with two-bit in-word tag):

- 00 `int` (unboxed)
- 01 pointer to `cons` (boxed, but no header)
- 10 unique (unboxed)
- 11 *pointer to a boxed object with header*

Very efficient **if integers and lists are used a lot.**

Hybrid tagging

In-word tags are more efficient than headers, but we would need much more than two or three bits... We can find a compromise

Use a short in-word tag of two or three bits for the most common types which we want to keep unboxed or boxed without header, reserving one value for boxed objects with headers.

Example (with two-bit in-word tag):

- 00 `int` (unboxed)
- 01 pointer to `cons` (boxed, but no header)
- 10 unique (unboxed)
- 11 *pointer to a boxed object with header*

Very efficient if integers and lists are used a lot.

Hybrid tagging

In-word tags are more efficient than headers, but we would need much more than two or three bits... We can find a compromise

Use a short in-word tag of two or three bits for the most common types which we want to keep unboxed or boxed without header, reserving one value for boxed objects with headers.

Example (with two-bit in-word tag):

- 00 `int` (unboxed)
- 01 pointer to `cons` (boxed, but no header)
- 10 unique (unboxed)
- 11 *pointer to a boxed object with header*

Very efficient **if integers and lists are used a lot.**

Hybrid tagging – example

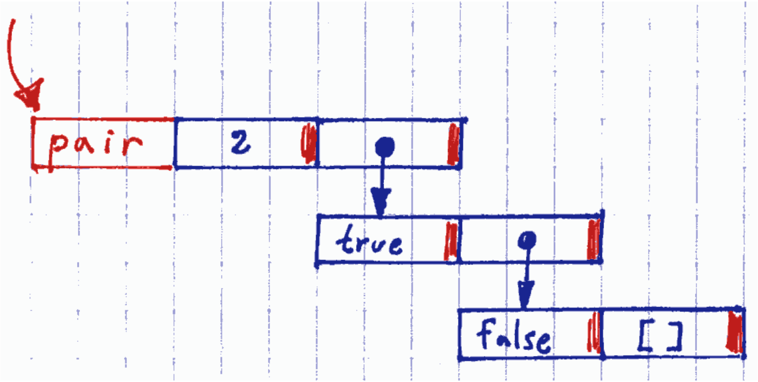


Figure: The pair `(2, [true; false])`, following the convention of the previous slide. The pair has a header because we didn't consider pairs to be "common" enough: but integers, conses and unique values (booleans and the empty list) need no header.

...and what about statically-typed languages?

We ignored *static typing* until now. Does OCaml need runtime tags?

[Tell me!]

Automatic memory management

We want to work under the illusion that memory is infinite.

- **The program just allocates objects**, ignoring the problem
- Unneeded objects are automatically destroyed by the runtime, which "wakes up" when needed

Pros:

- No dangling pointers
- No double free
- No (trivial) memory leaks

Cons:

- Inefficient.
  - Mmm. Is it *really* inefficient?

Automatic memory management

We want to work under the illusion that memory is infinite.

- **The program just allocates objects**, ignoring the problem
- Unneeded objects are automatically destroyed by the runtime, which "wakes up" when needed

Pros:

- No dangling pointers
- No double free
- No (trivial) memory leaks

Cons:

- Inefficient.
  - Mmm. Is it *really* inefficient?

Automatic memory management

We want to work under the illusion that memory is infinite.

- **The program just allocates objects**, ignoring the problem
- Unneeded objects are automatically destroyed by the runtime, which "wakes up" when needed

Pros:

- No dangling pointers
- No double free
- No (trivial) memory leaks

Cons:

- Inefficient.
  - Mmm. Is it *really* inefficient?

Automatic memory management

We want to work under the illusion that memory is infinite.

- **The program just allocates objects**, ignoring the problem
- Unneeded objects are automatically destroyed by the runtime, which "wakes up" when needed

Pros:

- No dangling pointers
- No double free
- No (trivial) memory leaks

Cons:

- Inefficient.
    - Mmm. Is it *really* inefficient?

Automatic memory management

We want to work under the illusion that memory is infinite.

- **The program just allocates objects**, ignoring the problem
- Unneeded objects are automatically destroyed by the runtime, which "wakes up" when needed

Pros:

- No dangling pointers
- No double free
- No (trivial) memory leaks

Cons:

- Inefficient.
    - Mmm. Is it *really* inefficient?

Automatic memory management

We want to work under the illusion that memory is infinite.

- **The program just allocates objects**, ignoring the problem
- Unneeded objects are automatically destroyed by the runtime, which "wakes up" when needed

Pros:

- No dangling pointers
- No double free
- No (trivial) memory leaks

Cons:

- Inefficient.
    - Mmm. Is it *really* inefficient?

Automatic memory management

We want to work under the illusion that memory is infinite.

- **The program just allocates objects**, ignoring the problem
- Unneeded objects are automatically destroyed by the runtime, which “wakes up” when needed

Pros:

- No dangling pointers
- No double free
- No (trivial) memory leaks

Cons:

- Inefficient.
  - Mmm. Is it *really* inefficient?

Automatic memory management — Definitions

At a given time, we call heap objects which will never be used again by the program *semantic garbage*.

- The runtime system works with *roots* (processor registers and stack, global variables): heap objects can only be reached via pointers from roots, or...
- ...from other heap objects. For example, many conses *refer* other conses
- A piece of *syntactic* garbage is a heap object which can't be reached by recursively following pointers starting from roots
  - Automatic memory management recycles *syntactic* garbage
  - Because of deep theoretical reasons **it's impossible to find all semantic garbage**; but recycling syntactic garbage is a conservative approximation

Automatic memory management — Main approaches

Two main approaches:

- *Tracing garbage collection* (or just *garbage collection*):
  - when the memory is full visit the graph of alive objects, starting from roots;
  - what we *didn't* visit is garbage: destroy it
- *Reference counting*
  - count the pointers *to* each object
  - when an object has zero pointers destroy it
- Only two main approaches. But there are many, many, many variants

Automatic memory management — Main approaches

Two main approaches:

- *Tracing garbage collection* (or just *garbage collection*):
  - when the memory is full visit the graph of alive objects, starting from roots;
  - what we *didn't* visit is garbage: destroy it
- *Reference counting*
  - count the pointers *to* each object
  - when an object has zero pointers destroy it
- Only two main approaches. But there are many, many, many variants

Automatic memory management — Main approaches

Two main approaches:

- *Tracing garbage collection* (or just *garbage collection*):
  - when the memory is full visit the graph of alive objects, starting from roots;
  - what we *didn't* visit is garbage: destroy it
- *Reference counting*
  - count the pointers *to* each object
  - when an object has zero pointers destroy it
- Only two main approaches. But there are many, many, many variants

Automatic memory management — Main approaches

Two main approaches:

- *Tracing garbage collection* (or just *garbage collection*):
  - when the memory is full visit the graph of alive objects, starting from roots;
  - what we *didn't* visit is garbage: destroy it
- *Reference counting*
  - count the pointers *to* each object
  - when an object has zero pointers destroy it
- Only two main approaches. But there are many, many, many variants

Automatic memory management — Main approaches

Two main approaches:

- *Tracing garbage collection* (or just *garbage collection*):
    - when the memory is full visit the graph of alive objects, starting from roots;
    - what we *didn't* visit is garbage: destroy it
- *Reference counting*
    - count the pointers *to* each object
    - when an object has zero pointers destroy it
- Only two main approaches. But there are many, many, many variants

Automatic memory management — Main approaches

Two main approaches:

- *Tracing garbage collection* (or just *garbage collection*):
  - when the memory is full visit the graph of alive objects, starting from roots;
  - what we *didn't* visit is garbage: destroy it
- *Reference counting*
  - count the pointers *to* each object
  - when an object has zero pointers destroy it
- Only two main approaches. But there are many, many, many variants

History

- John McCarthy (1927-2011) proposed mark-sweep garbage collection in his famous 1959 (!) paper introducing Lisp

Figure: John McCarthy in 2006. Photo by null0, released under the "Creative Commons Attribution 2.0 Generic" license: `http://www.flickr.com/photos/null0/272015955/`

- George E. Collins responded in 1960 by proposing reference counting as a "more efficient" alternative
- Popularly considered inefficient. Many languages depend on it, but accepted into the mainstream only in the 1990s

History

- John McCarthy (1927-2011) proposed mark-sweep garbage collection in his famous 1959 (!) paper introducing Lisp

Figure: John McCarthy in 2006. Photo by null0, released under the "Creative Commons Attribution 2.0 Generic" license: http://www.flickr.com/photos/null0/272015955/

- George E. Collins responded in 1960 by proposing reference counting as a "more efficient" alternative
- Popularly considered inefficient. Many languages depend on it, but accepted into the mainstream only in the 1990s

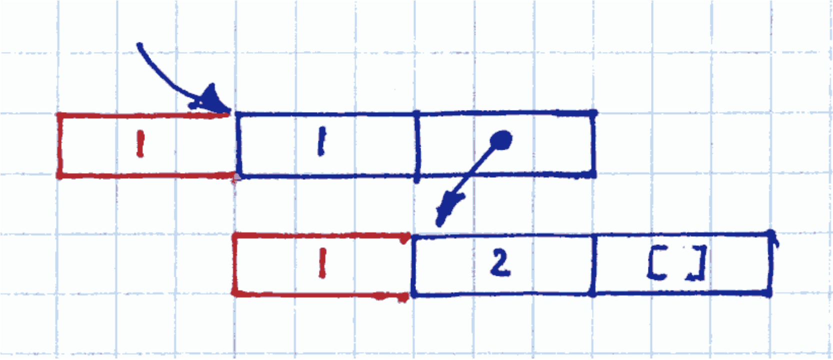
History

- John McCarthy (1927-2011) proposed mark-sweep garbage collection in his famous 1959 (!) paper introducing Lisp

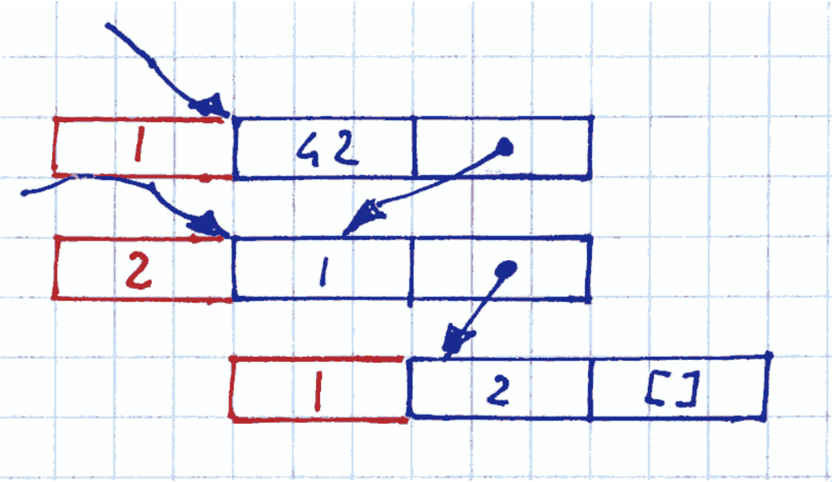
Figure: John McCarthy in 2006. Photo by null0, released under the "Creative Commons Attribution 2.0 Generic" license: http://www.flickr.com/photos/null0/272015955/

- George E. Collins responded in 1960 by proposing reference counting as a "more efficient" alternative
- Popularly considered inefficient. Many languages depend on it, but accepted into the mainstream only in the 1990s

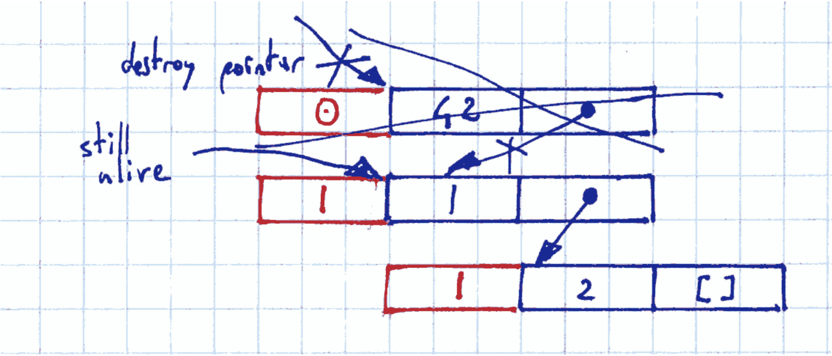
Reference-counting - I



Reference-counting - II



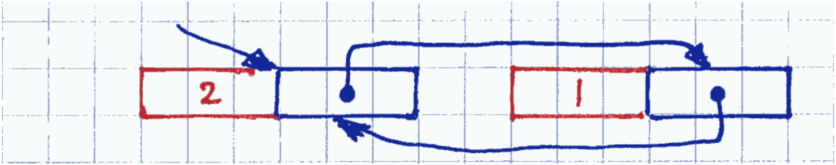
Reference-counting - III



Reference-counting problems - I

- Very inefficient (one word overhead per object, keeping counters up-to-date costs more than payload operations)...
- ...but this is not the main problem

Reference-counting problems - II



Reference-counting problems - III

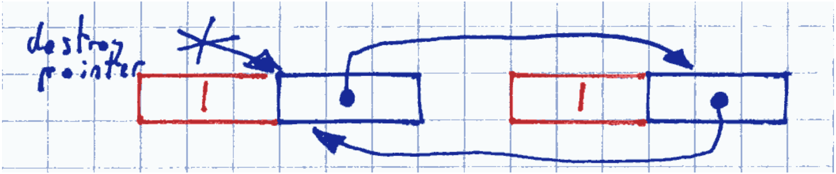


Figure: Circular garbage is never destroyed!

Reference-counting problems - IV

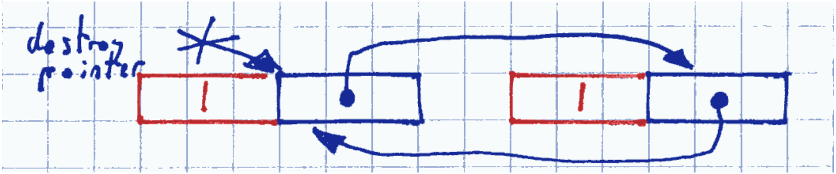


Figure: This is definitely syntactic garbage; but *cyclic* objects can't be destroyed by the reference counter.

Tracing garbage collection