

# Values (a.k.a. data) representation

Advanced Compiler Construction  
Michel Schinz – 2016-03-10

# The problem

# Values representation

A compiler must have a way of representing the values of the source language using values of the target language. This representation must be as efficient as possible, in terms of both memory consumption and execution speed.

For simple languages like C, the representation is trivial as most source values (e.g. C's `int`, `float`, `double`, etc.) map directly to the corresponding values of the target machine.

For more complex languages, things are not so easy...

# The problem

Most high-level languages have the ability to manipulate values whose exact type is unknown at compilation time. This is trivially true of all “dynamically-typed” languages, but also of statically-typed languages that offer parametric polymorphism – e.g. Scala, Java 5 and later, Haskell, etc. Generating code to manipulate such values is problematic: how should values whose size is unknown be stored in variables, passed as arguments, etc. ?

# Example

To illustrate the problem of values representation, consider the following L<sub>3</sub> function:

```
(def pair-make
  (fun (f s)
    (let ((p (@block-alloc-0 2)))
      (@block-set! p 0 f)
      (@block-set! p 1 s) p)))
```

Obviously, nothing is known about the type of `f` and `s` at compilation time. How can the compiler generate code to pass `f` and `s` around, copy them in the allocated block, etc. given that their size is unknown?

# Example

Notice that the very same problem appears in statically-typed languages offering parametric polymorphism, e.g.

Scala:

```
def pairMake[T,U](f: T, s: U): Pair[T,U] =  
  new Pair[T,U](f, s)
```

**The solutions**

# Boxed representation

The simplest solution to the values representation problem is to use a **(fully) boxed values representation**.

The idea is that all source values are represented by a pointer to a tagged block (a.k.a. a box), allocated in the heap. The block contains the representation of the source value, and the tag identifies the type of that value.

While simple, fully boxed values representation is very costly, since even small values like integers or booleans are stored in the heap. A simple operation like addition requires fetching the integers to add from their box, adding them and storing the result in a newly-allocated box...



# Tagging

To avoid paying the hefty price of fully boxed representation for small values like integers, booleans, etc., **tagging** can be used for them.

Tagging takes advantage of the fact that, on most target architectures, heap blocks are aligned on multiple of 2, 4 or more addressable units. Therefore, some of their least significant bits (LSBs) are always zero.

As a consequence, it is possible to use values with non-zero LSBs to represent small values.

# Integer tagging example

A popular technique to represent a source integer  $n$  as a tagged target value is to encode it as  $2n + 1$ . This ensures that the LSB of the encoded integer is always 1, which makes it distinguishable from pointers at run time.

The drawback of this encoding is that the source integers have one bit less than the integers of the target architecture. While rarely a problem, some applications become very cumbersome to write in such conditions – e.g. the computation of a 32 bits checksum.

For that reason, some languages offer several kinds of integer-like types: fast, tagged integers with reduced range, and slower, boxed integers with the full range of the target architecture.

# NaN tagging

With the current move towards 64 bits architectures, it can make sense to use 64 bits words to represent values.

Although standard tagging techniques can be used in that case, it is also possible to take advantage of a characteristic of 64 bits IEEE 754 floating-point values (i.e. `double` in Java): so-called not-a-number values (NaN), which are used to indicate errors, are identified only by the 12 most-significant bits, which must be 1. The 52 least-significant bits can be arbitrary.

Therefore, floating-point values can be represented naturally, and other values (pointers, integers, booleans, etc.) as specific NaN values.

# On-demand boxing

Statically-typed languages have the option of using unboxed/untagged values for monomorphic code, and switching to (and from) boxed/tagged representation only for polymorphic code.

Dynamically-typed language implementations can try to infer types to obtain the same result.

# (Full) specialization

For statically-typed, polymorphic languages, **specialization** (or **monomorphization**) is another way to represent values. Specialization consists in translating polymorphism away by producing several specialized versions of polymorphic code.

For example, when the type `List[Int]` appears in a program, the compiler produces a special class that represents lists of integers – and of nothing else.

# (Full) specialization issues

Full specialization removes the need for a uniform representation of values in polymorphic code. However, this is achieved at a considerable cost in terms of code size.

Moreover, the specialization process can loop for ever in pathological cases like:

```
class C[T]  
class D[T] extends C[D[D[T]]]
```

# Partial specialization

To reap some of the benefits of specialization without paying its full cost, it is possible to perform **partial specialization**.

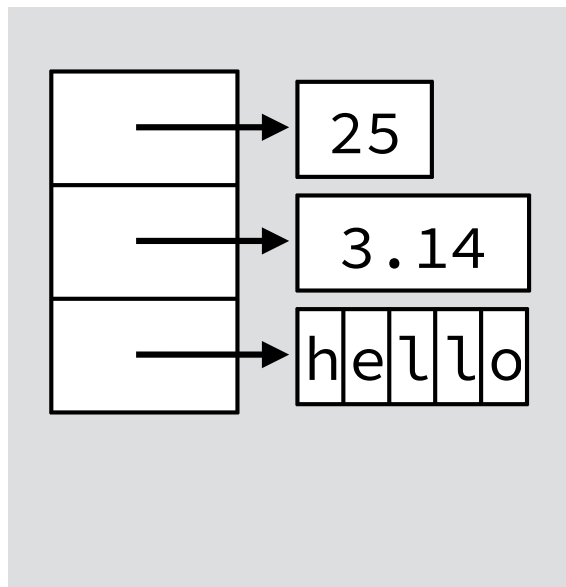
For example, since in Java all objects are manipulated by reference, there is no point in specializing for more than one kind of reference (e.g. `String` and `Object`).

It is also possible to generate specialized code only for types that are deemed important – e.g. `double` in a numerical application – and fall back to non-specialized code (with a uniform representation) for the other types. See for example Scala's `@specialized` annotation.

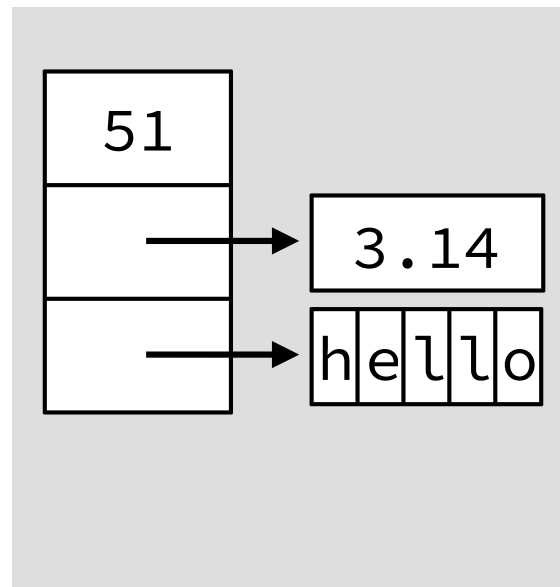
# Comparing solutions

The figures below show how an object containing the integer 25, the real 3.14 and the string *hello* could be represented using the three techniques previously described.

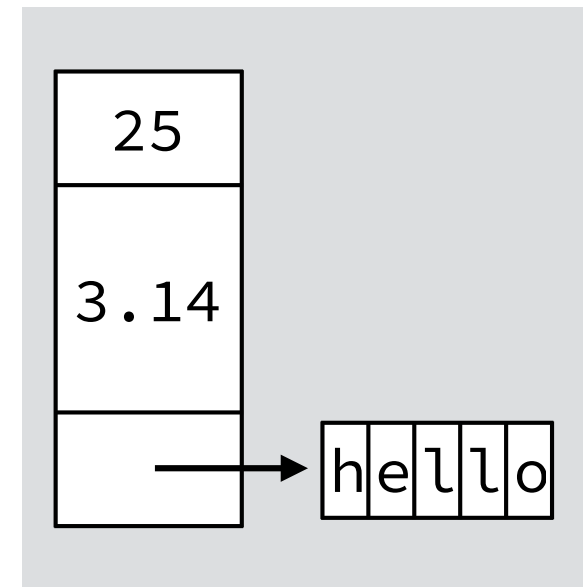
fully boxed



boxed with  
integer tagging



(fully) specialized





# Translation of operations

When a source value is encoded to a target value, the operations on the source values have to be compiled according to the encoding.

For example, if integers are boxed, addition of two source integers has to be compiled as the following sequence: the two boxed integers are unboxed, the sum of these unboxed values is computed, and finally a new box is allocated and filled with that result. This new box is the result of the addition.

Similarly, if integers are tagged, the tags must be removed before the addition is performed, and added back afterwards – at least in principle. In the case of tagging, it is however possible to do better for several operations...

# Tagged integer arithmetic

The table below illustrates how the encoded version of three basic arithmetic primitives can be derived for tagged integers. Similar derivations can be made for other operations (division, remainder, bitwise operations, etc.).

$$\begin{aligned} \llbracket n + m \rrbracket &= 2[(\llbracket n \rrbracket - 1) / 2 + (\llbracket m \rrbracket - 1) / 2] + 1 \\ &= (\llbracket n \rrbracket - 1) + (\llbracket m \rrbracket - 1) + 1 \\ &= \llbracket n \rrbracket + \llbracket m \rrbracket - 1 \end{aligned}$$

$$\begin{aligned} \llbracket n - m \rrbracket &= 2[(\llbracket n \rrbracket - 1) / 2 - (\llbracket m \rrbracket - 1) / 2] + 1 \\ &= (\llbracket n \rrbracket - 1) - (\llbracket m \rrbracket - 1) + 1 \\ &= \llbracket n \rrbracket - \llbracket m \rrbracket + 1 \end{aligned}$$

$$\begin{aligned} \llbracket n \times m \rrbracket &= 2[((\llbracket n \rrbracket - 1) / 2) \times ((\llbracket m \rrbracket - 1) / 2)] + 1 \\ &= (\llbracket n \rrbracket - 1) \times ((\llbracket m \rrbracket - 1) / 2) + 1 \\ &= (\llbracket n \rrbracket - 1) \times (\llbracket m \rrbracket \gg 1) + 1 \end{aligned}$$

# **L<sub>3</sub> values representation**

# L<sub>3</sub> values

L<sub>3</sub> has the following kinds of values: tagged blocks, functions, integers, characters, booleans, unit.

Tagged blocks are represented as pointers to themselves.

Functions are currently represented as code pointers, although we will see later that this is incorrect!

Integers, characters, booleans and the unit value are represented as tagged values.

# L<sub>3</sub> tagging scheme

For L<sub>3</sub>, we require that the two least-significant bits (LSBs) of pointers are always 0. This enables us to represent integers, booleans and unit as tagged values.

The tagging scheme for L<sub>3</sub> is given by the table below.

Kind of value	LSBs
Integer	...1 <sub>2</sub>
Block (pointer)	...00 <sub>2</sub>
Character	...110 <sub>2</sub>
Boolean	...1010 <sub>2</sub>
Unit	...0010 <sub>2</sub>

# Values representation phase

The values representation phase of the  $L_3$  compiler takes as input a CPS program where all values and primitives are “high-level”, in that they have the semantics of the  $L_3$  language gives them. It produces a “low-level” version of that program as output, in which all values are either integers or pointers and primitives correspond directly to instructions of a typical microprocessor.

As usual, we will specify this phase as a transformation function called  $\llbracket \cdot \rrbracket$ , which maps high-level CPS terms to their low-level equivalent.

# Representing L<sub>3</sub> functions

L<sub>3</sub> functions also need to be represented specially so that the operations permitted on them – e.g. function? – can be implemented in the target language.

The phase that takes care of representing functions is commonly known as **closure conversion**. While it logically belongs to the values conversion phase, it will be presented separately in the next lecture.

For now, we assume – incorrectly – that functions are not translated specially:

$$\llbracket (\text{let}_f ((f_1 (\text{fun } (c_1 \ n_{1,1} \dots) \ e_1)) \dots) \ e) \rrbracket = \\ (\text{let}_f ((f_1 (\text{fun } (c_1 \ n_{1,1} \dots) \ \llbracket e_1 \rrbracket)) \dots) \ \llbracket e \rrbracket)$$
$$\llbracket (\text{app}_f \ n \ nc \ n_1 \dots) \rrbracket = \\ (\text{app}_f \ n \ nc \ n_1 \dots)$$

# Representing L<sub>3</sub> continuations

Continuations, unlike functions, are limited enough that they do not need to be transformed!

Their body must still be transformed recursively:

$$\llbracket (\text{let}_c ((c_1 (\text{cnt } (n_{1,1} \dots) e_1)) \dots) e) \rrbracket =$$
$$(\text{let}_c ((c_1 (\text{cnt } (n_{1,1} \dots) \llbracket e_1 \rrbracket)) \dots) \llbracket e \rrbracket)$$
$$\llbracket (\text{app}_c n n_1 \dots) \rrbracket =$$
$$(\text{app}_c n n_1 \dots)$$



# Representing $L_3$ integers (1)

$\llbracket (\text{let}_1 ((n\ i))\ e) \rrbracket$  where  $i$  is an integer literal =

$(\text{let}_1 ((n\ 2i+1))\ \llbracket e \rrbracket)$

$\llbracket (\text{if (int? n) ct cf}) \rrbracket =$

$(\text{let}^* ((\underline{c1}\ 1)$   
           $(\underline{t1}\ (&\ n\ c1)))$   
           $(\text{if (= t1 c1) ct cf)))$

& is bit-wise and

# Representing L<sub>3</sub> integers (2)

```
[[letp ((n (+ n1 n2))) e]] =  
  (let* ((c1 1)  
         (t1 (+ n1 n2))  
         (n (- t1 c1)))  
    [[e]])
```

... other arithmetic primitives are similar.

```
[[if (< n1 n2) ct cf]] =  
  (if (< n1 n2) ct cf)
```

... other integer comparison primitives are similar.

# Representing L<sub>3</sub> integers (3)

```
[[letp ((n (block-alloc-k n1))) e]] =  
  (let* ((c1 1)  
        (t1 (>> n1 c1))  
        (n (block-alloc-k t1)))  
    [[e]])
```

>> is right shift

```
[[letp ((n (block-tag n1))) e]] =  
  (let* ((c1 1)  
        (t1 (block-tag n1))  
        (t2 (<< t1 c1))  
        (n (+ t2 c1)))  
    [[e]])
```

<< is left shift

... other block primitives are similar.

# Representing L<sub>3</sub> integers (4)

```
[[letp ((n (byte-read))) e]] =  
  (let* ((t1 (byte-read))  
         (c1 1)  
         (t2 (<< t1 c1))  
         (n (+ t2 c1)))  
    [[e]])
```

```
[[letp ((n (byte-write n1))) e]] =  
  left as an exercise
```

# Representing L<sub>3</sub> characters

$\llbracket (\text{let}_l ((n\ c))\ e) \rrbracket$  where  $c$  is a character literal =  
 $(\text{let}_l ((n\ [(\text{code-point}(c) \ll 3) | 110_2]))\ \llbracket e \rrbracket)$

$\llbracket (\text{let}_p ((n\ (\text{char} \rightarrow \text{int}\ n_1)))\ e) \rrbracket =$   
 $(\text{let}^* ((\underline{c2}\ 2)$   
 $\quad (n\ (>> n_1\ c2)))$   
 $\llbracket e \rrbracket)$

$\llbracket (\text{let}_p ((n\ (\text{int} \rightarrow \text{char}\ n_1)))\ e) \rrbracket =$   
 $(\text{let}^* ((\underline{c2}\ 2)$   
 $\quad (\underline{t1}\ (<< n_1\ c2))$   
 $\quad (n\ (+\ t1\ c2)))$   
 $\llbracket e \rrbracket)$

# Representing L<sub>3</sub> booleans

$\llbracket (\text{let}_1 ((n \ \#t)) e) \rrbracket =$   
     $(\text{let}_1 ((n \ 11010_2)) \llbracket e \rrbracket)$

$\llbracket (\text{let}_1 ((n \ \#f)) e) \rrbracket =$   
     $(\text{let}_1 ((n \ 01010_2)) \llbracket e \rrbracket)$

$\llbracket (\text{if} (\text{bool? } n) \text{ ct } \text{cf}) \rrbracket =$   
     $(\text{let}^* ((\underline{m} \ 1111_2)$   
             $(\underline{t} \ 1010_2)$   
             $(\underline{r} \ (& \ n \ m)))$   
           $(\text{if} (= \ r \ t) \ \text{ct} \ \text{cf}))$

# Representing $L_3$ unit, etc.

$\llbracket (\text{let}_1 ((n \ \#u)) \ e) \rrbracket =$   
 $(\text{let}_1 ((n \ 0010_2)) \llbracket e \rrbracket)$

$\llbracket (\text{if} (\text{unit?} \ n) \ ct \ cf) \rrbracket =$   
*left as an exercise*

$\llbracket (\text{halt} \ n) \rrbracket =$   
*left as an exercise*

Names are left untouched by the values representation transformation:

$\llbracket n \rrbracket =$   
 $n$

# Exercise

How does the values representation phase translate the following CPS/L<sub>3</sub> version of the successor function?

```
(letf ((succ (fun (c x)
              (let* ((c1 1)
                    (t1 (+ x c1)))
                    (appc c t1))))))
succ)
```