Values (or data) representation

Advanced Compiler Construction
Michel Schinz – 2021-03-11

The problem

Values representation

The **values representation** problem: how to represent the values of the source language in the target language?

Trivial in C and similar languages that have:

- no parametric polymorphism, and
- types corresponding directly to those of the target language (e.g. `int`, `long`, `double`),

More difficult in languages that have either:

- parametric polymorphism, as exact types are not at compilation time, or
- dynamic types, for the same reason, or
- types not corresponding directly to those of the target.

Example

Consider the following L_3 function:

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

The L_3 compiler knows nothing about the type of `f` and `s`, so some uniform representation must be used.

Example

The same problem exists in Scala when using parametric polymorphism:

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


The solutions

Boxing

Boxing: all values are represented uniformly by a pointer to a heap-allocated block called a **box** and containing:

- the value,
- some information about its type.

Pros and cons:

- simple,
- very costly for small values (e.g. integers).

Tagging

Tagging: all values are represented uniformly by a pointer-sized word containing either:

- a pointer to a boxed value, as before, or
- a small value (e.g. integer) with a tag identifying its type.

Pros and cons:

- simple,
- less costly than boxing,
- reduced range for some small values (e.g. integers).

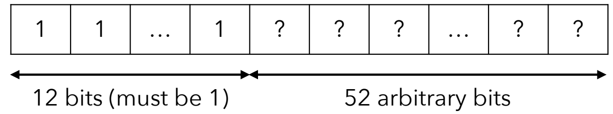
Example: integer tagging

Integer tagging example: represent the source integer *n* as the target integer $2n + 1$.

- distinguishable from (aligned) pointer by LSB,
- slightly reduced range (1 bit less).

Example: NaN tagging

IEEE 754 floating-point values (i.e. `double`) have special NaN values, returned on error, identified by top 12 bits:



NaN tagging:

- represent `double`s as themselves,
- use 52 lower bits of NaNs to store tagged values:
  - pointers,
  - integers,
  - etc.

On-demand boxing

(Un)boxing can be done **on-demand** for statically-typed languages:

- box when entering polymorphic context,
- unbox when returning to monomorphic context.

Pros and cons:

- no penalty for monomorphic code,
- can be expensive at runtime.

Also doable for dynamically-typed languages, but requires type inference.

Specialization

Specialization (or **monomorphization**): get back to simple case by translating polymorphism away.
For example, if `List[Int]` appears in a program, a class representing lists of integers is generated.
Pros and cons:

- avoids the cost of boxing and tagging,
- produces *lots* of code,
- can fail to terminate.

Partial specialization

Partial specialization:

- share specialized code as much as possible (e.g. specialize only once for all reference types), and/or
- allow the programmer to specify when to specialize, and box otherwise.

Pros and cons:

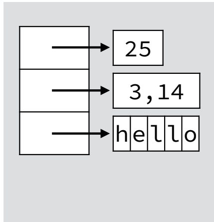
- can provide the performance of specialization for critical code without the cost.

Comparing solutions

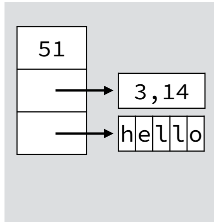
Three representations of an object containing:

- the integer 25,
- the double 3.14
- the string "hello".

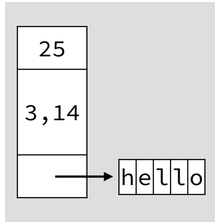
fully boxed



boxed with integer tagging



(fully) specialized



Translation of operations

Independently of the chosen solution, operations acting on source values must be adapted to the representation, e.g.:

- addition of boxed integers is done by:
  1. fetching the two integers from their box,
  2. adding them,
  3. allocating a new box, storing the result in it.
- addition of tagged integers is done by:
  1. untagging the two integers,
  2. adding them,
  3. tagging the result.

For tagging, one can do better though!

Tagged integer arithmetic

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

L_3 values representation

Representation of L_3 values

L_3 has the following kinds of values:

1. functions,
2. tagged blocks,
3. integers,
4. characters,
5. booleans,
6. unit.

For now, we assume (incorrectly!) that functions are simple code pointers.
Tagged blocks are represented as pointers to themselves.
Integers, characters, booleans and the unit value are tagged.

L_3 tagging scheme

In L_3, we require the two LSBs of pointers to be 0, in order to use the tagging scheme below:

Kind of value	LSBs
Integer	$\ldots 1_2$
Block (pointer)	$\ldots 00_2$
Character	$\ldots 110_2$
Boolean	$\ldots 1010_2$
Unit	$\ldots 0010_2$

Values representation phase

The **values representation** phase of the L_3 compiler:

- takes a "high-level" CPS program:
  - values: all L_3 values,
  - primitives: all L_3 primitives,
- produces an equivalent "low-level" CPS program:
  - values: bit vectors and pointers (both 32 bits),
  - primitives: instructions of the VM (similar to typical processor).

Specified as usual as a transformation function called $[\![\cdot]\!]$, mapping high-level CPS terms to their low-level equivalent.

Atoms

```
⟦n⟧ where n is a name =
  n
⟦i⟧ where i is an integer literal =
  2i+1
⟦c⟧ where c is a character literal =
  (code-point(c) ≪ 3) | 110_2
⟦#t⟧ =
  11010_2
⟦#f⟧ =
  01010_2
⟦#u⟧ =
  0010_2
```


Continuations & functions

Continuations are restricted enough that they don't need to be translated:

```
⟦(let_c ((c_1 (cnt (n_1,1 …) e_1)) …) e)⟧ =
  (let_c ((c_1 (cnt (n_1,1 …) ⟦e_1⟧)) …) ⟦e⟧)
⟦(app_c n v_1 …)⟧ =
  (app_c n ⟦v_1⟧ …)
```

Functions must be translated, but we ignore it for now (see next lecture) and assume the following *incorrect* translation:

```
⟦(let_f ((f_1 (fun (c_1 n_1,1 …) e_1)) …) e)⟧ =
  (let_f ((f_1 (fun (c_1 n_1,1 …) ⟦e_1⟧)) …) ⟦e⟧)
⟦(app_f v n_c v_1 …)⟧ =
  (app_f ⟦v⟧ n_c ⟦v_1⟧ …)
```


Integers (1)

```
⟦(if (int? v) c_t c_f)⟧ =
  (let_p ((t1 (& ⟦v⟧ 1)))
    (if (= t1 1) c_t c_f))
⟦(let_p ((n (+ v_1 v_2))) e)⟧ =
  (let* ((t1 (+ ⟦v_1⟧ ⟦v_2⟧))
         (n (- t1 1)))
    ⟦e⟧)
```

& is bit-wise and

… other arithmetic primitives are similar.

```
⟦(if (< v_1 v_2) c_t c_f)⟧ =
  (if (< ⟦v_1⟧ ⟦v_2⟧) c_t c_f)
```

… other integer comparison primitives are similar.

Integers (2)

```
⟦(let_p ((n (block-alloc-k v_1))) e)⟧ =
  (let* ((t1 (shift-right ⟦v_1⟧ 1))
         (n (block-alloc-k t1)))
    ⟦e⟧)
⟦(let_p ((n (block-tag v_1))) e)⟧ =
  (let* ((t1 (block-tag ⟦v_1⟧))
         (t2 (shift-left t1 1))
         (n (+ t2 1)))
    ⟦e⟧)
```

… other block primitives are similar.

Integers (3)

```
⟦(let_p ((n (byte-read))) e)⟧ =
  (let* ((t1 (byte-read))
         (t2 (shift-left t1 1))
         (n (+ t2 1)))
    ⟦e⟧)
⟦(let_p ((n (byte-write v))) e)⟧ =
```

left as an exercise

Characters

```
⟦(let_p ((n (char->int v_1))) e)⟧ =
  (let_p ((n (shift-right ⟦v_1⟧ 2)))
    ⟦e⟧)
⟦(let_p ((n (int->char v_1))) e)⟧ =
  (let* ((t1 (shift-left ⟦v_1⟧ 2))
         (n (+ t1 2)))
    ⟦e⟧)
⟦(if (char? v) c_t c_f)⟧ =
```

left as an exercise

Booleans, unit, etc.

```
⟦(if (bool? v) c_t c_f)⟧ =
  (let_p ((r (& ⟦v⟧ 1111_2)))
    (if (= r 1010_2) c_t c_f))
⟦(if (unit? v) c_t c_f)⟧ =
```

left as an exercise

```
⟦(halt v)⟧ =
```

left as an exercise

Exercise

How does the values representation phase translate the following CPS/L_3 version of the successor function?

```
(let_f ((succ (fun (c x)
                (let_p ((t1 (+ x 1)))
                  (app_c c t1)))))
  succ)
```