Data representation

Advanced Compiler Construction

Michel Schinz – 2014-03-06

The problem

Data representation

A compiler must map the datatypes of the source language to the datatypes provided by the target language. This mapping must be as efficient as possible, in terms of both memory consumption and execution speed.

For simple languages like C, the mapping is trivial: most source datatypes (e.g. `int`, `float`, `double`, etc.) are directly mapped to the corresponding datatypes 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 data representation, 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)))
```

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 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 data representation

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

The idea is that *all* source datatypes 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 data, and the tag identifies the type of that data.

While simple, fully boxed data 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 data 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 data.
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 data 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**.

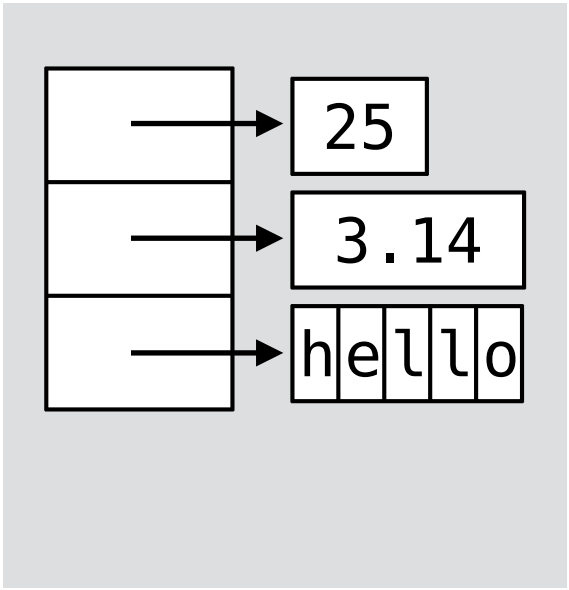
For example, since in Java all objects are manipulated by reference, there is no point in specializing for more than one kind of reference (e.g. `String` and `Object`).

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.

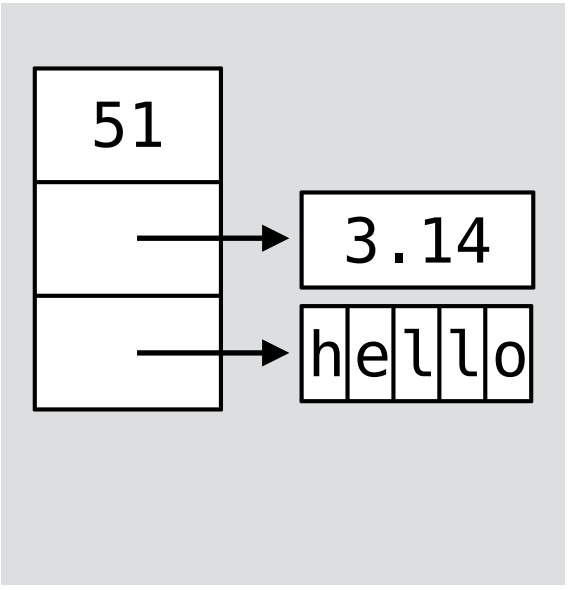
Comparison of solutions

The following drawings 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.

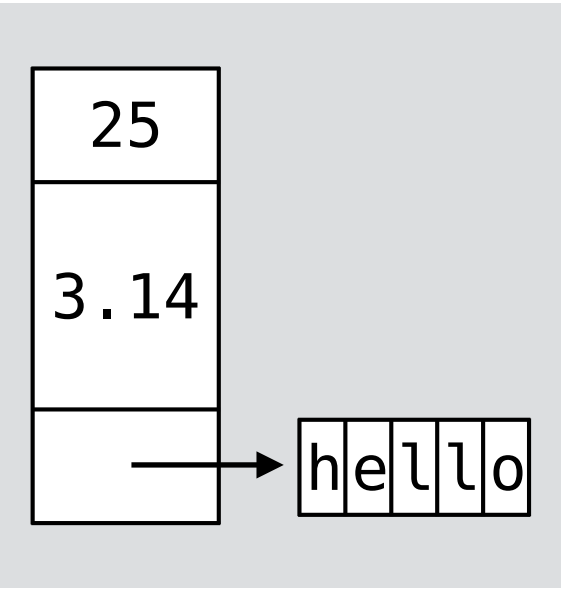
fully boxed



boxed with integer tagging



(fully) specialized



Translation of operations

When a source datatype is encoded to a target datatype, the operations on the source datatype 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 the four basic arithmetic primitives can be derived for tagged integers. Source terms are in red, target terms in blue.

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

L_3 data representation

L_3 datatypes

L_3 has the following datatypes: 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_3 tagging scheme

For L_3, 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_3 is given by the table below.

Datatype	LSBs
Integer	…1
Block (pointer)	…00
Character	…110
Boolean	…1010
Unit	…0010

Data representation phase

The data 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 $[\![\cdot]\!]$, which maps high-level CPS terms to their low-level equivalent.

Representing L_3 functions

L_3 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 data conversion phase, it will be presented separately in the next lecture.

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

$[\![(\texttt{let}_f\ ((f_1\ (\texttt{fun}\ (c_1\ n_{1,1}\ \ldots)\ e_1))\ \ldots)\ e)]\!] =$

$\quad (\texttt{let}_f\ ((f_1\ (\texttt{fun}\ (c_1\ n_{1,1}\ \ldots)\ [\![e_1]\!]))\ \ldots)\ [\![e]\!])$

$[\![(\texttt{app}_f\ n\ n_c\ n_1\ \ldots)]\!] =$

$\quad (\texttt{app}_f\ n\ n_c\ n_1\ \ldots)$

Representing L_3 continuations

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

Their body must still be transformed recursively:

$[\![(\texttt{let}_c\ ((c_1\ (\texttt{cnt}\ (n_{1,1}\ \ldots)\ e_1))\ \ldots)\ e)]\!] =$

$\quad(\texttt{let}_c\ ((c_1\ (\texttt{cnt}\ (n_{1,1}\ \ldots)\ [\![e_1]\!]))\ \ldots)\ [\![e]\!])$

$[\![(\texttt{app}_c\ n\ n_1\ \ldots)]\!] =$

$\quad(\texttt{app}_c\ n\ n_1\ \ldots)$

Representing L_3 integers (1)

⟦(let$_1$ ((n i)) e)⟧ *where* i *is an integer literal* =

```
  (let1 ((n 2i+1)) ⟦e⟧)
```

⟦(if (int? n) ct cf)⟧ =

```
  (let* ((c1 1)
         (t1 (& n c1)))
    (if (= t1 c1) ct cf))
```

Representing L_3 integers (2)

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

… other arithmetic primitives are similar (see slide about tagged integer arithmetic).

```
⟦(if (< n₁ n₂) ct cf)⟧ =
  (if (< n₁ n₂) ct cf)
```

… other integer comparison primitives are similar.

Representing L_3 integers (3)

```
⟦(letₚ ((n (block-alloc-k n₁))) e)⟧ =
  (let* ((c1 1)
         (t1 (>> n₁ c1))
         (n (block-alloc-k t1)))
    ⟦e⟧)
⟦(letₚ ((n (block-tag n₁))) e)⟧ =
  (let* ((c1 1)
         (t1 (block-tag n₁))
         (t2 (<< t1 c1))
         (n (+ t2 c1)))
    ⟦e⟧)
```

>> is right shift

<< is left shift

… other block primitives are similar.

Representing L_3 characters

⟦(let$_l$ ((n c)) e)⟧ *where* i *is an character literal* =

```
  (let_l ((n [(char→int(c) ≪ 3) | 110₂])) ⟦e⟧)
```

⟦(let$_p$ ((n (char-read))) e)⟧ =

```
  (let* ((c3 3)
         (ct 110₂)
         (t1 (char-read))
         (t2 (<< t1 c3))
         (n (| t2 ct)))
    ⟦e⟧)
```

| is bit-wise or

⟦(let$_p$ ((m (char-print n))) e)⟧ =

left as an exercise

Representing L_3 booleans

```
⟦(letι ((n #t)) e)⟧ =
  (letι ((n 11010₂)) ⟦e⟧)
⟦(letι ((n #f)) e)⟧ =
  (letι ((n 01010₂)) ⟦e⟧)
⟦(if (bool? n) ct cf)⟧ =
  (let* ((m 1111₂)
         (t 1010₂)
         (r (& n m)))
    (if (= r t) ct cf))
```

Representing L_3 unit, halt

⟦(let$_1$ ((n #u)) e)⟧ =

  (let$_1$ ((n 0010_2)) ⟦e⟧)

⟦(if (unit? n) ct cf)⟧ =

  left as an exercise

The halt statement is left untouched by the data representation transformation.

⟦(halt)⟧ =

  (halt)

Exercise

How does the data-representation phase translate the following CPS version of the successor function?

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