LittleSmalltalk – Object Representation

Author: Danny Reinhold
Version: 1.0
Status: Published
Date: April, 2nd, 2007

Summary:
This document explains in detail how the LittleSmalltalk system represents objects in memory.
This document is not really a good marketing document since it will show some of the ugly internals of the system. Please don't think that the system is too worse to be used because of this document. Other Smalltalk implementations are not that different at this level…

General object layout

All types of objects start with a header. It is important for the system that the headers of all objects are binary compatible.
It is not important if an object represents a small integer, a byte array or a complex Smalltalk object – you can always be sure that the object starts with exactly the same type of header.

The header contains information about the size and class of the object and some hints about the internal representation of the actual object data.

In ANSI-C the general object structure looks like this:

```
struct object
{
  int size;
  struct object *class;
  struct object *data[0];
};
```

The `size` and `class` fields form the header of an object. The `data` array represents the body of the object.

The exact interpretation of the header fields and the structure of the object body depends on the concrete type of object.

Ordinary objects

An ordinary object is a pure Smalltalk object.
In such an object the object body is a list of pointers to other objects.

The size field

The size field contains the number of bytes required to store the object body.
Since each item in the object body in an ordinary object is a pointer to another object, reach item takes 32 bit (4 bytes).
So the number of items in the object body always is (size / 4).

An the size is always (sizeof(struct *) * n + sizeof(header)) for an object with n items (don't forget to count the size of the header).

Please note that the size field of an ordinary object always holds a multiple of 4 (lower two bits are 0). This is important when looking at byte arrays.

The class field

The class field simply points to an object that represents the class of the current object.

The object body

The object body is a list of pointers to objects. The items in this list represent the member variables of the object. So an object with n member variables also has a body list with n object pointers.

SmallInt and Int objects

A SmallInt object can contain an integer value in the range [0,999].
An Int object contains an arbitrary 32 bit integer value. While SmallInt objects are used within calculations Int objects are meant to hold pointers.
The internal layout of both types of objects is identical and looks in ANSI C like this:

```
struct integerObject
{
  int size;
  struct object *class;
  unsigned int value;
};
```

Here the object body consists of the value that the object shall represent.
For such objects it is important that the class field points exactly to the SmallInt or to the Int class.
In a Int or SmallInt object the size field also contains the value of the object .

Byte Arrays

Byte arrays are another important type of objects.
In a byte array (for example a string) the body consists of an array of bytes (represented as unsigned characters).
In ANSI C this structure looks like this:

```
struct byteObject
{
     int size;
     struct object *class;
     unsigned char bytes[0];
};
```

The size field of a byte array is important.
Although even byte arrays always end at 4 byte boundaries (they are filled up with pad bytes if neccessary) bit 1 of the size field if always set!

Remember that in all other kinds of objects bits 0 and 1 of the size field are always 0!
The virtual machine detects if an object is a byte object simply by verifying if bit 1 is set or not. But first you have to ensure that it is not an integer object since bit 1 of the size field can be set there…

How to detect the type of an object

If the virtual machine shall decide which is the internal representation of an object, it first checks if the class field either points to the SmallInt or the Int class (then it is either a SmallInt or an Int object) or if bit 1 of the size field is set (then it is a byte array).
It is important to do the checks in this order.

Today those complicated encodings look strange and error prone to programmers – and in fact they are in most cases not really clever.
But when LittleSmalltalk was created resources were limited and it was good and common to save memory whenever this was possible.

It is also notable that the implementation currently only works on 32 bit archtectures, since there are lot of dependencies in the system that tell us that a pointer takes exactly 4 bytes.

Nowadays this object representation scheme and some implementation details look strange and are not really flexible. I think I will change some of this stuff until the final release of 5.0 or maybe later – but I think I will change it…