

# Systematic Compiler Construction

Michael Sperber      Peter Thiemann      Martin Gasbichler

January 30, 2004

Dieses Werk ist urheberrechtlich geschützt; alle Rechte mit Ausnahme der folgenden sind vorbehalten:

*Studenten der Informatik dürfen diese Dokument für ihre persönlichen Lehrzwecke verwenden.*

Ausdrücklich verboten wird jede andere Verwendung von Ausdrucken, Dateikopien oder Paperkopien. Insbesondere darf dieses Skriptum nicht in irgendeiner Weise vertrieben werden und es dürfen keine Kopien der Dateien auf öffentlich zugänglichen Servern angelegt werden.

Copyright © Michael Sperber, 1999; Peter Thiemann, 2000, 2002; Martin Gasbichler, 2004

# Chapter 6

## Data Representation

Before machine code generation is feasible, it is necessary to address a number of run-time issues that are independent of code generation per se, but are needed for code to run. Most of these issues did not become evident in the interpreter as they implicitly use the run-time environment of the metalanguage which is sufficiently powerful. With assembly language, however, the runtime environment is almost non-existent, so it is necessary to add a few ingredients.

### 6.1 Addressing Run Time Issues

The interpreters for the lambda calculus and the CPS language—despite being definitional—implicitly make use of a number of facilities of the metalanguage. They have gone unnoticed so far because they are not even part of the denotational semantics: the metalanguage of mathematics either provides these facilities as well or is simply not concerned with the operational consequences of one or the other implementation of them.

Among these issues are:

**Data representation** How does a value from a sum domain translate into the domain of bits and bytes?

**Memory management** In mathematics, “memory” is infinite—it does not matter how many objects a running program “creates.” Not so for real machines.

Of these issues, data representation and memory management are very closely related and pose important design constraints on the other two. Therefore, they are first.

#### 6.1.1 Data representation and memory management

From the interpreters, it is not obvious what the exact requirements concerning data representation on the actual machine are. A naive coding of the components of the `value` data type would look like this:

`Unit` The word 0.

`EmptyList` The word 0.

`Int` A word with the same value as the integer argument.

`Bool` The word 0 for `true`, the word 1 for `false`.

**Char** A byte with the ASCII of the character.

**String** An area in memory (the *heap*) where the first word contains the length of the string and remaining bytes contain the characters.

**Closure** An area in memory where the first word is the address of the closure environment, and the second word is the address of the code representing the body of the abstraction that created the closure.

**Cons** An area in memory which begins with the first (head) component of the pair, after which comes the second (tail) component of the pair.

**Wrong** The word 0.

Unfortunately, the naive coding is unsuitable for several reasons. For illustration, consider the following Mini-Caml program:

```
let pair x = [x]

let main () = begin pair 42; pair [42] end
```

The program applies `pair` once to a number and once to a list. Whereas the number fits in one word, the list—really a pair—does not. However, `pair` needs to handle its argument in some way to stuff it into a pair. Unfortunately, since it is polymorphic, it has no information about the type of the object and therefore cannot determine how to put it inside a pair. Intuitively, the type of the object does not matter since `pair` does not “look” at the object but only puts it somewhere else. However, it at least needs to know how *big* the object is.

The obvious solution is to make all objects have the same size—a single word comes to mind. Some objects already fit in one word. Those which do not are composite objects easily represented by a pointer to a memory area representing the object rather than the memory area itself. The process of replacing a composite object by a pointer to it is called *boxing*, the reverse—dereferencing the pointer—it called *unboxing*.

However, the resulting representation still has two problems:

```
let main () = fst 42
```

The semantics of the above program is obviously undefined—it contains a type error. However, the data representation does not reflect that fact: The representation for the number 42 looks just like a pointer to a pair, and the program, when run, will happily dereference it, leading to a more or less random result. Even though C programs routinely exhibit random behavior because of exactly such errors, this is just as clearly undesirable. If pointers were distinguishable from integers, the running program could check that the argument to `fst` is indeed a pointer to a pair, and, on failure, report the error in an orderly fashion which allows the programmer to fix the bug.

In the full Caml language, the type system prevents such errors. Therefore, run-time type checks are not necessary there. Conceivably, a static type system with a suitable checking machine could be added to the compiler. However, even with static type checking in place, another problem remains which is not as easily moved to the realm of the static:

```
let rec garbage count =
  if count = 0
  then 42
  else begin [42]; garbage (count - 1) end;

let main () = garbage 99999999
```

Apart from the fact that the program does not do anything useful: It creates 99999999 pairs when it is run, and each pair consumes two words of memory. Most realistic machines do not have that much memory. However, the pair [42] becomes “garbage” immediately after its creation: It is no longer accessible for the program (it is *dead*), and therefore the memory that it takes up can immediately be re-used for other purposes. However, Mini-Caml has no equivalent to Pascal’s `dispose` directive or C’s `free` function to make the memory available again. Also, even though it is clearly statically visible in the example when the pair dies, such a static analysis is usually much harder in realistic programs. (Even though it is possible.)

Therefore, most implementations of high-level languages provide a facility called *garbage collection* or just plain *GC*. When a program runs out of memory, it calls the GC which reclaims the memory of all objects which the running program may no longer access. GC techniques and algorithms fill a book. However, it is clear that the GC needs support from the language implementation to do its work:

1. It needs access to all objects that the running program has “immediate” access to. In our interpreter, this would be the values of the registers. These objects form the *root set*.
2. The GC needs to trace through all objects reachable—directly or indirectly—from the root set to form the set of *live objects*. To do this, it needs to distinguish *immediate* objects which fit in a word from pointers, and it needs to know what components of composite objects are reachable through the object.

The first point will be easy to satisfy. However, the second requirement really means that the objects have to carry a limited amount of type information for the tracing to be possible.

The next draft of the data representation could look like this:

- All values that the running program handles are pointers to objects in the heap.
- The first word of an object in the heap either represents a small immediate object (the empty list, unit, true, false, or a character) or the type of larger object.
- The second word of an object contains its size in bytes. This could be inferred from the type in most cases, but not for strings, for instance. Generally, it is a useful piece of information to have around.
- The remaining words contain the actual data representing the object.

This representation fulfills all the requirements. However, it is still inefficient: For every integer operation, a dereference becomes necessary because all integers reside in the heap. Moreover, the result must be stored in the heap, resulting in further overhead. However, on 32-bit architectures, pointers always have the following form (again reverting to conventional bit counting):

$$\begin{array}{c|c} \hline \text{Bits 31-2} & \text{Bits 1-0} \\ \hline x & | 00 \\ \hline \end{array}$$

The two least significant bits are always 0 and therefore contain no useful information. If 30 bits are sufficient, words of the following format (*descriptors*) could represent objects:

Type	Bits 31–2	Bits 1–0
Integer	integer value	00
Unit	00000000000000000000000000000000	01
Empty list	00000000000000000000000000000001	01
False	00000000000000000000000000000010	01
True	000000000000000000000000000000110	01
Character	000000000000000000000000xxxxxxx11	01
Heap object	upper bits of pointer	10

The lower two bits encoding partial type information are called the *tag* of the object. Note that choosing a tag of 00 for integers means that addition and subtraction work just as before, and only division and multiplications require shifts for the necessary adjustments. The fact that pointers no longer represent themselves is easily remedied by an offset of -2 in indirect addresses—something that modern processors provide anyway. One tag value is still available for miscellaneous purposes.

An object in the heap has the following layout—starting with the address pointed to by the descriptor:

Word 0	Word 1	...
Header	Data	...

The header has the following layout:

Bits 31–4	Bits 3–0
Size (bytes)	Type

The type tag in the header can have the following values:

Type	Tag
Pair	0000
Closure	0001
Continuation	0011
Environment	0100
String	1000

The assumption is that every heap object with a type tag that has its upper bit sets is actually a *bitmap* which does not contain further descriptors, but instead some opaque data of no interest to, say, the garbage collector. Again, the encoding is a matter of convention, not necessity.

Note also that no type tag has a trailing 10 bit pattern—this is reserved for representing broken hearts in a moving garbage collector.

### 6.1.2 Environments and continuations

For first-order heap objects such as strings or pairs, it is straightforward to find a suitable heap representation. For environments and continuations, slightly more care is needed because both carry identifiers—at least as represented in the interpreters. However, in Mini-Caml, identifier usage and binding follows the rules of *static scoping*. The name suggests that a compiler can leave identifiers in the static realm and drop them for execution purposes.

The first-order representation of environments indeed suggests that environment access is possible through a simple index. As the interpreter creates new bindings in a LIFO fashion, the index is really a reverse stack depth. A possible layout for environment objects is therefore the following:

Word 0	Word 1	Word 2	...
Header	previous environment	Entry 0	further entries

In this setup, the environment consists of linked *frames*, each of which can contain any number of bindings. Note that currently, binding works in such a way that bindings always happen one variable at a type; frames therefore typically contain only one binding each.

With environments out of the way, closures are easy:

Word 0	Word 1	Word 2	...
Header	Environment	Code	...

This would, however, destroy the uniformity of heap object layout: A heap object is supposed to consist of descriptors only. It is therefore required to store the code in an separate heap object, a *code vector* that contains bitmap data only. We introduce an additional layer, a *code template* which holds the code vector along with any data needed by the code. There is no such data in the current system but additional features of the language, such as a module system, will need this space. Code templates look like this:

Word 0	Word 1
Header	Code vector

Closures now simply hold an environment and a code template:

Word 0	Word 1	Word 2
Header	Environment	Code template

We also need to define the tags of code vectors and code templates. Code vectors have the most upper bit set to indicate that they are holding bitmap data. Code templates are ordinary descriptor vectors:

Type	Tag
Code vector	1001
Code template	0101

Continuations and exception handlers can use the same layout. They also store their code in code templates:

Word 0	Word 1	Word 2	Word 3	Word 4
Header	Environment	Continuation	Exception handler	Code template

### 6.1.3 Generating descriptors and headers

Now that we know how to our value domain is represented in the memory we have to provide the means to generate this representation. Depending on the nature of the values, either the compiler or the generated code produces the actual representation. This section describes a small library for generating descriptors and headers statically, i.e. within the compiler. The run-time library will be much smaller and will be presented during the presentation of the actual code generation.

First, a function to combine two integer values into a single value comes in handy:

```
let adjoin_bits high low width =
  Int32.logor (Int32.shift_left high width) low
```

The first generated object are descriptors. We need to define the width of the tag, the known values for the tag, and a function to build a descriptor from a tag and some data:

```

let tag_field_width = 2

let fixnum_tag = Int32.of_int 0
let immediate_tag = Int32.of_int 1
let heap_object_tag = Int32.of_int 2

let make_descriptor tag data =
  adjoin_bits data tag tag_field_width

```

Generating fixnums and the immediate values is now a simple exercise. The library does not include a function to generate descriptors for heap objects as the pointer is only known at run time.

Generating headers is next. The fundamental constants are the length of a header and the length of the field that is holding the type:

```

let bytes_per_word = 4

let header_length_in_bytes = Int32.of_int (bytes_per_word * 1)

let header_type_field_width = 4

```

The constructor for headers needs to know the type and the size of the header:

```

let make_header header_type size =
  adjoin_bits size header_type header_type_field_width

```

The miscellaneous tags for the types form a simple enumeration:

```

let header_pair_tag = Int32.of_int 0
let header_closure_tag = Int32.of_int 1
let header_continuation_tag = Int32.of_int 3
let header_environment_tag = Int32.of_int 4
let header_code_template_tag = Int32.of_int 5

let header_string_tag = Int32.of_int 8
let header_code_vector_tag = Int32.of_int 9

```

Two functions compute the size of a heap object. The first, `make_dsize`, works for descriptor vectors whereas the second, `make_bsize`, works for byte vectors:

```

let make_dsize no_ds =
  Int32.add (Int32.of_int (bytes_per_word * no_ds)) header_length_in_bytes
let make_bsize bytes =
  Int32.add (Int32.of_int bytes) header_length_in_bytes

```

Using these functions, the size of the heap objects can be defined to be either a constant, or a function of the size of the data:

```

let pair_size = make_dsize 2
let closure_size = make_dsize 2
let continuation_size = make_dsize 4
let make_environment_size no_bindings = make_dsize (1 + no_bindings)
let code_template_size = make_dsize 1
let string_size len = make_bsize len

```

Generating the headers is now trivial:

```

let pair_header    = make_header header_pair_tag pair_size
let closure_header = make_header header_closure_tag closure_size
let continuation_header = make_header header_continuation_tag continuation_size
let code_template_header =
  make_header header_code_template_tag code_template_size
let make_environment_header no_bindings =
  make_header header_environment_tag (make_environment_size no_bindings)

let make_code_vector_header len =
  make_header header_code_vector_tag (code_vector_size len)

let make_string_header len =
  make_header header_string_tag (string_size len)

```

The final task of our small library is the access to heap objects. Of course the code has to perform this access at run time, but for many cases, the offset within the heap object is known statically. Therefore, our library defines a few constants that hold offsets in bytes for the standard objects:

```

let header_offset = 0

let bytes_from_offset off =
  (Int32.to_int (header_length_in_bytes)) + (off * bytes_per_word)

let env_prev = bytes_from_offset 0
let make_env_bind offset_in_binding = bytes_from_offset (1 + offset_in_binding)

let clos_env = bytes_from_offset 0
let clos_ct  = bytes_from_offset 1

let ct_cv = bytes_from_offset 0

let cont_env  = bytes_from_offset 0
let cont_cont = bytes_from_offset 1
let cont_exh  = bytes_from_offset 2
let cont_ct   = bytes_from_offset 3

```

As pointers to heap objects are descriptors, access using such an offset needs to subtract the tag. The function `untag` performs this adjustment:

```

let untag o = o - (Int32.to_int heap_object_tag)

```

# Bibliography

- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [Boc76] G. V. Bochmann. Semantic evaluation from left to right. *Communications of the ACM*, 19:55–62, 1976.
- [Cha87] Nigel P. Chapman. *LR parsing: theory and practice*. Cambridge University Press, Cambridge, UK, 1987.
- [DeR69] Franklin L. DeRemer. *Practical Translators for LR(k) Parsers*. PhD thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Ma., 1969.
- [DeR71] Franklin L. DeRemer. Simple LR(k) grammars. *Communications of the ACM*, 14(7):453–460, 1971.
- [DF92] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.
- [DP82] Franklin DeRemer and Thomas Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems*, 4(4):615–649, October 1982.
- [DS95] Charles Donnelly and Richard Stallman. *Bison—The YACC-compatible Parser Generator*. Free Software Foundation, Boston, MA, November 1995. Part of the Bison distribution.
- [FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings, 1995.
- [FHP92] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code generator generator. *Letters on Programming Languages and Systems*, 1(3):213–226, 1992.
- [Fis93] Michael J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3/4):259–288, 1993.
- [Ive86] Fred Ives. Unifying view of recent LALR(1) lookahead set algorithms. *SIGPLAN Notices*, 21(7):131–135, July 1986. Proceedings of the SIGPLAN’86 Symposium on Compiler Construction.
- [Ive87a] Fred Ives. An LALR(1) lookahead set algorithm. Unpublished manuscript, 1987.
- [Ive87b] Fred Ives. Response to remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices*, 22(8):99–104, August 1987.

- [Joh75] S. C. Johnson. Yacc—yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.
- [Lee93] Rene Leermakers. *The Functional Treatment of Parsing*. Kluwer Academic Publishers, Boston, 1993.
- [PC87] Joseph C.H. Park and Kwang-Moo Choe. Remarks on recent algorithms for LALR lookahead sets. *SIGPLAN Notices*, 22(4):30–32, April 1987.
- [PCC85] Joseph C. H. Park, K. M. Choe, and C. H. Chang. A new analysis of LALR formalisms. *ACM Transactions on Programming Languages and Systems*, 7(1):159–175, January 1985.
- [Plo75] Gordon Plotkin. Call-by-name, call-by-value and the  $\lambda$ -calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- [Spe94] Michael Sperber. Attribute-directed functional LR parsing. Unpublished manuscript, October 1994.
- [SSS90] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing Theory*, volume II (LR(k) and LL(k) Parsing) of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1990.
- [ST95] Michael Sperber and Peter Thiemann. The essence of LR parsing. In William Scherlis, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 146–155, La Jolla, CA, June 1995. ACM Press.
- [ST00] Michael Sperber and Peter Thiemann. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems*, 22(2):224–264, March 2000.
- [WM92] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau — Theorie, Konstruktion, Generierung*. Lehrbuch. Springer-Verlag, Berlin, Heidelberg, 1992.