Malfunctional Programming

Stephen Dolan

September 14, 2016

Malfunction is an untyped program representation intended as a compilation target for functional languages, consisting of a thin wrapper around OCaml's `Lambda` *intermediate representation.*

Compilers targeting Malfunction convert programs to a simple s-expression-based syntax with clear semantics, which is then compiled to native code using OCaml's back-end, enjoying both the optimisations of OCaml's new `flambda` *pass, and its battle-tested runtime and garbage collector.*

1 Introduction

When a programming language researcher designs a new language to explore some particular aspect of programming (in my case, subtyping, in yours, perhaps dependent types, probabilistic programming, or `COMEFROM`-with-current-continuation), the first person it's shown to tends to rudely interject with the following question:

"Very nice, but can you make it *go*?"

Even if the researcher comes prepared for this, the follow-up is always:

"Can you make it go *fast*?"

Implementing a language from scratch is a daunting task. However, despite the many high-level differences between programming languages, their requirements of a compiler back-end tend to be very similar. It seems prudent, therefore, to make use of the many years of work put into making OCaml not just go, but go fast.

Malfunction is an untyped language allowing you to do just that. The syntax is loosely based on the OCaml compiler's `-dlambda` output (although Malfunction's syntax is simplified, to allow easy code generation), and the semantics are an untyped lambda calculus (stricter in most places than those of `Lambda`, to remain robust to future changes to OCaml). For example, this expression in OCaml:

```
List.iter print_string ["Hello"; "World"]
```

is equivalent to this expression in Malfunction:

```
(apply
  (global $List $iter)
  (global $Pervasives $print_string)
  (block "Hello" (block "World" 0)))
```

2 Why OCaml?

Why re-use OCaml's back-end specifically, when there are plenty of other compilers available? The central issues are efficiency and garbage collection.

C compilers and related projects like LLVM provide very efficient code generation, but it is tricky to integrate garbage collection. C compilers assume ownership of the stack layout, and so may introduce temporary stack references to heap objects. A *conservative* garbage collector can find these references (by assuming any pointer-like bit-pattern is in fact a heap pointer), but an efficient *moving* collector needs precise data about stack layout, so that heap objects can be safely relocated.

Higher-level virtual machines such as the Java Virtual Machine and the .NET Common Language Runtime come with state-of-the-art garbage collectors. However, their design is closely tied to the sort of languages they were built to run, and bytecode verification means they will refuse to run any program whose safety cannot be explained in terms of the Java or C# type system. So, it is difficult to efficiently compile programs proven to be safe by more advanced type systems (although not impossible, see e.g. OCaml-Java [3]), as the verifier must be pacified by redundant runtime type checks.

Dynamic languages, such as Scheme, Smalltalk or JavaScript, are easy to compile to and have reasonably fast implementations. However, when running statically typed functional programs, time is wasted on runtime type checks.

OCaml's `Lambda` is an easy compilation target (being essentially the untyped lambda calculus [4]), which loses no efficiency in runtime type tests: instead,

it assumes its input to already have been proven safe by a high-level type checker. This makes it an ideal target for statically-typed languages with advanced type systems, since the safety of programs need not be explained to the backend. The performance of `Lambda` is high, especially since the recent work on inlining [2].

3 Reusing `Lambda`

The next question is whether it's at all valid to reuse `Lambda`, which was designed solely as an intermediate representation for OCaml. In principle, optimisations made by the OCaml back-end need only be valid for the sort of programs that the OCaml front-end can generate. However, this is a surprisingly large set. For instance, consider the following strange function:

```
(lambda ($t)
  (let
    (if (== (apply (field 0 $t) 0)) 0
      (apply (field 1 $t) 42)
      (+ (field 1 $t) (field 2 $t)))))
```

This function takes a tuple whose first component is a function, which is called. If that function returns 0, then the tuple is a pair whose second field maps integers to integers, while if it returns 1 then the tuple is a triple whose second and third fields are both integers.

Odd as it seems, this function can be written in OCaml by abuse of GADTs[1]. In fact, the advanced features of OCaml's type system (GADTs, functors, first-class polymorphism, etc.) are erased before compilation to Lambda, so it would be remarkably difficult for the back-end to prove that a program could not have been compiled from OCaml. So, I conjecture that OCaml will not miscompile any Malfunction program, or at least that when it does, it will also miscompile a sufficiently contrived OCaml program.

4 Semantics of Malfunction

As with all low-level untyped languages, some behaviour is undefined. The snippet `(field 0 0)`, which attempts to dereference an integer, will cause a segmentation fault on a good day (but even that's not guaranteed).

A problem with compilation to C and related languages is that it's almost impossible to know whether a given snippet has defined behaviour, since many undefined operations tend to work fine, at least on certain versions of certain compilers at certain settings.

[1] This is left as an exercise to the reader

While Malfunction has plenty of undefined behaviour, it also comes with a pedantic interpreter, which defines a semantics and can be used to (slowly) evaluate any program to detect undefined behaviour. This semantics is stricter than that of `Lambda`: for instance, accessing a tuple using array primitives tends to work in `Lambda`, since they are represented the same way, but is banned by the Malfunction semantics. This stricter semantics makes Malfunction more robust to future changes in the OCaml backend.

5 Experimental Idris backend

An experimental Malfunction backend for the dependently typed programming language Idris [1] is available. This backend does not yet implement all Idris primitives, so cannot run all programs. Nonetheless, preliminary benchmarking of some simple programs (currently, Idris lacks a comprehensive benchmark suite) shows the Malfunction backend outperforming Idris's C backend by a factor between 3.2x and 14x.

6 Other applications

While Malfunction is intended primarily as a compilation target for statically-typed functional languages, it can also be used:

- To randomly generate programs with well-defined semantics (by checking them against the interpreter), in order to find bugs in the optimising compiler, in the style of Csmith [5].
- As an alternative to C/Assembly for implementing primitive operations, as code written in Malfunction gets inlined into code written in OCaml.

Malfunction is available from:

`http://github.com/stedolan/malfunction`

References

[1] E. Brady et al. Idris, a language with dependent types. *IFL*, 2008.

[2] P. Chambart. High level OCaml optimisations. *OCaml Workshop*, 2013.

[3] X. Clerc. OCaml-Java: OCaml on the JVM. In *Trends in Functional Programming*, pages 167–181. Springer, 2012.

[4] X. Leroy. Le système Caml Special Light: modules et compilation efficace en Caml. 1995.

[5] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. *PLDI*, 2011.